2020

Systemutvikling

NOTATER

HERMANN OWREN ELTON; OLAF ROSENDAHL

Innholdsfortegnelse

MMI

HCI – Human Computer interaction

MMI – Menneske maskin interaksjon

Et fagfelt som handler om å undersøke og forbedre samhandling mellom mennesker og datateknologi. Involverer utformingen av både maskinvare og programvare.

Brukskvalitet

Brukskvalitet handler om hvor enkelt det er å bruke et menneskeskapt objekt. Et system har høy brukskvalitet når det er:

- Lett å lære
- Effektivt
- Lett å huske
- Feilfritt og feiltolerant
- Behagelig og tilfredsstillende å bruke

Brukergrensesnitt

Er det grensesnittet som gjør at en bruker kan kommunisere med maskiner. Det er viktig at man tenker over hvordan dette grensesnittet er laget, fordi det er dette kundene eller brukerne av systemet kommer til å se. Det er mange grunner til å lage systemer som er gode for mennesker å samhandle med. Den kanskje mest konkrete og åpenbare grunnen er økonomi:

- Det øker sannsynligheten for at produktet ditt lykkes i markedet
- Det gjør at dine ansatte kan være mer effektive i arbeidsoppgavene sine, og trives bedre på jobb. Dette fører til høyere produktivitet og mindre sykefravær.
- Det gjør at du kan bruke mindre tid og ressurser på brukerstøtte.

Interaksjonsdesign (IxD)

Betegnelsen på et stort fagfelt som handler om å designe samhandling mellom mennesker og digitale systemer, objekter, tjenester og miljøer. Handler om å designe systemer og objekter som er enkle, effektive, morsomme og behagelige å bruke. Interaksjons design er å lage samtalen mellom produkt eller system og dens bruker. De viktigste elementene involverer utformingen av både maskinvare og programvare. Resulterer i et produkt hvor estetikk, brukervennlighet, ergonomi, kognitiv teknologi, design, pyskologi og sosiologi spiller en stor rolle for hvordan sluttbrukeren oppfatter samhandlingen. Det handler om å forstå menneskets behov og, å kunne designe verktøy som kan hjelpe oss å nå målene enten de er å kjøpe en bussbillett eler å handle på nettet. Jobben til en interaksjonsdesigner er å gjøre teknologi funksjonell og brukervennlig.

De fem dimensjonene i IxD

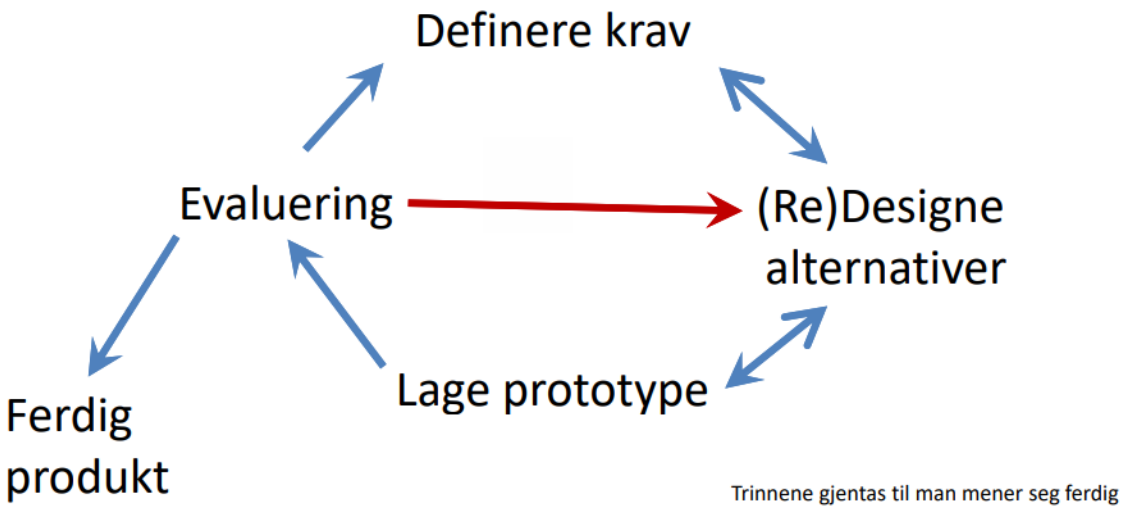
1. Ord - Burde være lett å forstå, og skrevet på en måte slik at viktig informasjon blir lett tilgjengelig for slutt brukeren.
2. Visuell representasjon – Dette handler om de grafiske elementene som bilder typografi som brukeren samhandler med
3. Fysiske objekter og rom – referer til fysiske maskinvare, alt fra mus og keyboard til mobilen som brukeren bruker til å samhandle med produktet.
4. Tid – er tiden som brukeren bruker med de første tre dimensjonene.
5. Oppførsel – er følelsene og reaksjonene som brukeren får når de samhandler med produktet

   https://www.interaction-design.org/literature/article/what-is-interaction-design

   usability: https://www.nngroup.com/articles/usability-101-introduction-to-usability/

   meget tydelig pil som peker på hvor man skal trykke så det burde egentlig ikke vært

Når man jobber med IxD så ser en typisk modell sånn ut:



Don Normans prinsipper om interaksjons design:

- Synlighet
- Tilbakemeldinger
- Begrensinger
- Tilordne/avbilde
- Konsistens

- Tilbydelser – Hvilke handlinger det er naturlig for brukeren å tro at man kna gjøre med systemet.

IxD er en brukerorientert prosess, den er iterativ. Man kan bevege seg mellom faser om det trengs, stadig målinger og evaluering er viktig for å forstå hvordan vi ligger an.

Prinsipper for interaksjonsdesign

Synlighet: Det er viktig at funksjoner ved et system er synlige, både ved å vise at de er tilstede og å vise hvordan de skal brukes. Har du noen gang stått foran en fancy vask og rett og slett ikke ant hvordan du får ut vann av krana? Det har jeg. Første gang jeg traff på den typen blandebatteri du ser til høyre, fikk jeg problemer. Ikke 1 hjalp det å prøve å vri på håndtakene, ikke hjalp det å presse dem opp eller dra dem ned, og da jeg prøvde å trekke et av dem ut til siden, løsnet det faktisk. Når de i tillegg var litt trege og jeg var nervøs for å ta i etter allerede å ha delvis demontert det ved et uhell, tok det litt tid før jeg endelig fikk vann ut – ved å skyve dem fram og tilbake horisontalt. Det er noen prikker på håndtakene som gir et hint om bevegelsesretningen, men det var åpenbart ikke tydelig nok for meg. Et mer eksplisitt design (for eksempel en diskret pil) kunne kanskje spart meg noen minutter og en god porsjon tvil om egne evner.

Feedback: Feedback handler om at systemet signaliserer til brukeren hva som foregår. Trykker du på Send på et skjema på en nettside, er det godt design å gi brukeren en tilbakemelding om at skjemaet nå faktisk er sendt. Trykker du på en knapp på en trykkfølsom skjerm, er det bra hvis lyd og/eller fargeendringer på skjermen indikerer at trykket ditt faktisk ble registrert.

Begrensninger: Noen ganger trenger vi å legge begrensninger på hvilke former for interaksjon som er mulig i et gitt øyeblikk, slik at det ikke oppstår feil. Det er for eksempel dette operativsystemet gjør når det gråer ut dialogbokser du ikke får bruke på et gitt tidspunkt. Å legge inn fysiske eller visuelle begrensninger gjør det lettere å bruke et system korrekt. Hvis en automat ikke aksepterer andre mynter enn kronestykker, kan det være lurt å designe åpningen man putter mynter inn i slik at det ikke går an å putte andre

mynter inn – da har vi laget en begrensning som hindrer brukerne i å legge på feil type mynter.

Konsistens: Dette handler om å lage ting slik at brukere gjør lignende ting på lignende måter. Det sparer tid og tankeaktivitet. Hadde krana jeg nevnte over hatt et mer konvensjonelt design, ville jeg kunne brukt kunnskap fra bruk av andre kraner og sluppet å føle meg så dum.

Tilbydelser: Dette er et forsøk på å oversette ordet affordances . Det handler om hvilke 2 handlinger en bruker opplever at hun kan gjøre med et system. Denne opplevelse av hvilke handlinger systemet tilbyr oss vil henge sammen både med hvordan systemet er laget og hvilke erfaringer vi har gjort oss tidligere med andre systemer. For eksempel vil de fleste automatisk anta at en knapp kan trykkes på, at håndtak kan dras i eller vris på (litt ettersom hvordan de er utformet), og at man skal putte noe med to lange og tynne utstikkere inn i stikkontakter. Det svenske firmaet Prisma Teknik har designet mange av boksene man må 3 trykke på hvis man vil krysse lysregulerte fotgjengerfelt i Trondheim, og jeg ble forvirret av designet deres i starten. Boksen har en meget tydelig pil som peker på hvor man skal trykke så det burde egentlig ikke vært noe problem, men siden jeg ikke så så nøye på boksen var det mer logisk for meg å trykke på den delen som stakk ut – nemlig lyset nederst. Middels store ting som stikker ut på denne typen bokser er jo som regel knapper som skal trykkes på, ikke sant? Det ga heller dårlig resultat så jeg sluttet med det etterhvert. Ellers har den god feedback – det kommer et høyt pip når et trykk er registrert, og lyset slår seg på. Dermed kan både synshemmede og hørselshemmede få det med seg. I tillegg forteller lyder når det blir grønt for de som ikke ser trafikklysene. Man kan også argumentere for at mine feiltrykk var et resultat av manglende konsistens – på de gamle boksene skulle man trykke på en knapp nederst, så kanskje det var rett og slett dette jeg prøvde på. Hadde boksen vært flat hele veien, hadde jeg muligens sett nøyere på merkingen i stedet for bare å trykke i vei.

Datainnsamling

Hvem er brukarene og hva ønsker de fra produktet?

Ordnet plan for datainnsamling:

1. Sette klare mål for datainnsamlingen
    a. Hvordan skal dataene brukes og analyseres
2. Relasjon med deltakerne
    a. Profesjonell og tydelig, samtykke om nødvendig
3. Triangulering
    a. Bruk av flere teknikker for å oppnå flere perspektiver
4. Pilotstudier (test ut)

Varsomhet

Etiske spørsmål, hvem «eier» dataen, samtykke/taushetserklæring.

Å definere krav

Vi har alltid et utgangspunkt for kravene til grensesnittet, nemlig ideene som gjorde at prosjektet ble startet i det hele tatt. Vær eksplisitte om disse, fra et brukergrensesnittperspektiv: Hvorfor trenger verden det dere lager, hvorfor er det bedre enn eksisterende løsninger.

Hvem er brukeren?

Det er ofte ikke veldig mye hjelp i ¨spørre en bruker dirkete hva hun eller han ønsker fra et nytt produkt. En god forståelse av hvem brukeren er og hvilke mål de generelt har i situasjoner der produktet er tenkt brukt, er derimot et mye bedre utgangspunkt. Dette finner vi ut ved å samle inn data om brukeren. Involver brukeren tidlig.

Personaer – er en typisk bruker som man finner opp og tester programmet med hypotetisk. Her er det viktig å være konkret.

Brukerhandlinger

I kravene bør vi forklare hvordan brukere skal samhandle med systemet vanlig metoder er Use Case(brukermønster) og Scenarioer.

Use Case - Beskriver hva systemet skal gjøre for en type bruker. Beskriver et funksjonelt krav til systemet i et eksternt perspektiv.

Scenarioer (subjektivt)

Scenarier er bilder av fremtiden- Riktig utformet, gir scenarier støtet til nye ideer om hva vi bør gjøre i framtiden og hva vi bør unngå. Scenarioer er et godt utgangspunkt for diskusjon med brukere og systemeiere.

Hva er prototyping

Prototypen handler om et første inntrykk eller en liten smakebit på produktet, på noen områder er det en liten skala modell av produktet. En prototype for et nettsted er en interaktiv demo av nettstedet. Prototyping handler om å utforske ulike løsningsforslag, men det handler også om å kontrollere. For oss er en prototype en presentasjon av et konsept/ produkt, utarbeidet med tanke på formidling testing og evaluering. Prototyping og design av alternativer går hånd i hånd, og det er gjenre en gildene overgang mellom disse fasene. Prototyper egner seg både for utforskning av ideer og brukertesting.

Hvorfor prototype

- Test idene
    - Det gir oss mulighet til å teste ideene våre og vise de frem
- Tilbakemelding
    - Får tilbakemelding og evaluering fra bruker, som til syvende og sist er den som skal benytte seg av produktet eller ideen
- Forståelse
    - Brukerne kan interagere med prototypen lettere enn gjennom en skriftlig beskrivelse, og det vil være lettere å forstå og oppfatte bruksområdene.
- Kommunikasjon
    - Det vil være hensiktsmessig også fordi det gjør kommunikasjonen mellom bruker og utvikler enda enklere og beriket

Dimensjoner ved prototyping

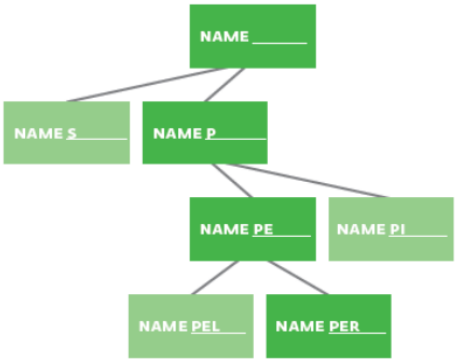
- Utseende
    - Størrelse, farge, form, fasong, tekstur, proporsjon, hardhet, gjennomsiktighet, haptisk, lyd
- Innhold (Data)
    - Type, bruk, personvern, hierarki og organisering
- Funksjonalitet
    - Systemets funksjoner, brukerens behov
- Interaktivitet
    - Input, output, feedback og informasjon
- Romlig struktur (fysikk)

- Sammensettingen av grensesnitt og informasjonssystemer, relasjon mellom elementer, 2D/3D, tangible eller intangible.

Prototyper - kompromisser

Horisontal prototype: mange funksjoner lite dybde, gir overblikk brukes til høynivåpresentasjoner.

Vertikal prototype: få funksjoner, med mye dybde. Er utformet for å besvare spesifikke spørsmål i brukertester.



Prototyper – Utvikling

Evolusjonære prototyper – Utvikles gradvis igjennom itereringer der siste versjon er endelig produkt. Beslutninger på et stadium for konsekvenser for senere versjoner.

Bruk og kast prototyper – kastes etter bruk og man starter med en ny prototype ved neste iterasjon. Det er mulighet for å eksperimentere ved hver integrering uten bekymringer for konsekvenser.

Evolusjonær prototyping vs. Bruk og kast

De fleste protyper lages kun for testing og utforskning underveis i design prosessen, så de kastes når man ikke trenger dem lenger (bruk og kast prototyper). Noen ganger velger man å la prototyper utvikle seg gradvis og til slutt bli det endelige produktet (evolusjonære prototyper).

Lavoppløslige prototyper – lite eller ingen interaksjon, ingen eller delvis fungerende funksjonalitet, krever lite innsats å lage, kan lages av papir, uforpliktende og lett å kaste, kan ikke forveksles med endelig produkt-

Høyoppløslige prototyper – Mye interaksjon, fungerende funksjonalitet, krever mye innsats å lage, forpliktende – vanskelige å kaste, kan forveksles med endelig produkt.

Hvordan prototype?

To muligheter: Du kan starte fra scratch eller modifisere en eksisterende ide.

Prototypen kan være: En serie med skisser, Et storyboard, Tegneserie, Lysark, Video, Fysisk modell, Pappmodell, Programkode osv, og wireframes.

Oppsummering

Informasjonsarkitektur

Har fokus på organisering, strukturering og merking av innhold på en effektiv og bærekraftig måte. Målet er å hjelpe brukerne å finne informasjon og fullføre oppgaver.

Brukeropplevelse

Brukeropplevelsen handler om hvordan brukerne opplever det å bruke et system, og da gjerne med spesielt fokus på hvilke følelser systemet framkaller hos brukeren.

Evaluering

Er å finne verdien av noe. En systematisk datainnsamling, analyse og vurdering av en planlagt, pågående eller avsluttet aktivitet, en virksomhet og et virkemiddel eller en sektor.

Evaluering vs. testing

Testing og evaluering er ikke det samme. Evaluering er det beredt begrep som inkluderer: Tester, formelle modeller, ekspergjennomgang, gjennomgang av sjekklister over typiske feil, prototyper, osv. Alt som kan gi brukeren en ide om endelig løsning kan evalueres.

Hvorfor evaluere?

Evaluering avslører feil, Gir deg innsikter du ikke hadde fått ellers, viser hva brukerne faktisk gjør, og ikke hva du eller de tror de kommer til å gjøre. Evaluering hjelper det å holde fokus på brukerne.

Evalueringsformer

Formativ -man gjør en evaluering for å påvirke utviklingen og sikre at designene man jobber med oppfyller kravene, utføres underveis.

Summativ - en evaluering av sluttproduktet.

Evaluering som involverer brukere

Her finnes det mange metoder, inkludert de klassiske bruker testene. De fleste metodene kan justeres og tilpasses de enkelte prosjektene. Metodene kan både brukes til å finne brukskvalitetsfeil, og å ta designavgjørelser.

- Brukertester
    - Brukere gjør definerte oppgaver i kontrollerte omgivelser. Det kan være observasjoner, målinger og feil, som rapporteres.
- A/B-tester
    - Kvantitativ testmetode der man lager to (eller flere) varianter av et design, presentere brukere for en tilfeldig variant og rett og slett ser hvilken variant som har best resultat.
- Feltstudier i naturlige omgivelse
    - Brukere utfører oppgaver i «naturlige» omgivelser. Det kan vare observasjon og beskrivelser som blir rapportert.
- Opportunistiske evalueringer
    - Dette er enkle tester som typisk gjøres tidlig i utviklingen, der man bare huker tak i noen få brukere og ber dem gi rask tilbakemelding på en konkret ide

Evaluering som ikke involverer brukere

- Heuristiske gjennomganger
    - Dette handler om p gå gjennom velkjente tommelfingerregler og retningslinjer for brukerkvalitet
- Pluralistiske gjennomganger
    - Her møtes brukere, utviklere og brukerkvalitetseksperter for å jobbe seg gjennom viktige scenarioer.
- Prediktive modeller
    - MMI-feltet byr på mange modeller som prøver å forutsi hvordan det vil være å bruke et gitt bruker grensesnitt.

Metoder og teknikker

- Brukbarhetstester (usability)
    - Observasjon
    - Spørre brukere
    - Testing
- Feltstudier
    - Observasjon
    - Spørre brukere
    - Spørre eksperter (UX/ domene)
- Analytisk evaluering
    - Spørre eksperter (HCI eller domene)
    - Modellere

Evalueringsmetoder og teknikker

Flere tilnærminger brukes samtidig og ulike metoder og teknikker kan brukes i flere av tilnærmingene

Metode/teknikk	Kontrollerte omgivelser	Naturlige omgivelser	Uten brukere
Observasjon	X	X	
Spørre brukere	X	X	
Spørre eksperter		X	X
Testing	X		
Modellering			X

Metode/teknikk	Kontrollerte omgivelser	Naturlige omgivelser	Uten brukere
Observasjon	Video og interaksjonslogg	Etnografiske teknikker: skygging, flue-på-veggen	
Spørre brukere	Pre- og posttestingsspørsmål, strukturerte intervjuer	Intervjuer og diskusjoner	
Spørre eksperter		Heuristisk evaluering, diskusjoner	Heuristisk evaluering
Testing	Testing av typiske oppgaver (brukbarhetstesting)		
Modellering			HTA, GOMS og annen teori

Evalueringsmetoder spesielt relevante for nettsteder

- Kodevalidering
- Testing i forskjellige nettleser og på forskjellige plattformer
- Testing av fleksibiliteten til et design

- Testing av grad av universell utforming og bruker kvalitet
- Testing av feilmeldinger
- Skjermavalidering

Brukertesting

Bruker testing er det nærmeste du kommer en trylleformel for å forbedre brukervennlighet. Det er den raskeste måten for å tilegne seg økt kompetanse som designer.

Bruker testing – Mål

Målet med bruker testing er å få den innsikten man trenger for å gjøre produktet som it-systemer og nettsteder enklere å bruke

En bruker test: Simulerer en reell situasjon, inneholder konkrete oppgaver Er en test hvor du observer brukeren Brukes for å evaluere brukervennligheten til et system. Resultat av testen brukes for å forbedre det som ikke fungerte bra.

Definering av mål

En brukertest bær ha en klart definer og begrenset hensikt. «Evaluere brukskvaliteten til et system» er ikke et godt mål. «Evaluere menyene», evaluere fargevalget», eller «evaluere navigasjon strukturen til nettstedet» er bedre mål.

Planlegging av brukertest

1. Hva er formålet?
    a. En brukertest må alltid ha et formål

Eksempler:

Problem	Formål med brukertest
Kundesenteret mottar mange henvendelser om ting som kundene bør kunne finne på nettstedet	Vi ønsker å vite om hvorfor kundene ikke finner informasjon om selvbetjening
Tekstene på nettstedet er mange, lange og uinteressante	Overbevise beslutningstakeren om at det er verdt bryet å sette inn ekstra ressurs på å rydde opp og skrive om tekstene
Du har laget et helt nytt konsept med ny navigator for nettstedet, men du aner ikke hvordan dette vil være for brukerne	Finne ut om brukerne finner frem på den nye siden og om de skjønner konseptet

2. Hvilken funksjonalitet skal testes
    a. Liste over relevant funksjonalitet som skal testes.
3. Hvilket system?

    a. Du må av klare hvilken versjon av systemet som skal testes. Systemet bør inneholde opplysningene som trengs for å gjennomføre testene.
4. Hvem er testbrukere
    a. Lag en liste over de brukergruppen som skal teste funksjonaliteten
5. Hvor skal testen foregå?
    a. Skal brukarene komme til dere eller skal dere gå til brukerne?
6. Hvilke utstyr skal benyttes?
    a. Kartlegge behov for testutstyr utover at brukeren må ha en PC
    b. Hva sags system har brukerne
        i. Skjerm oppløsning
        ii. Operativt system
7. Hvilke oppgaver skal brukeren få
    a. Finn ut hvilke typer oppgaver testbrukerne bør gjennomføre
    b. ut fra listen over mulige oppgaver
        i. finn en prioritetsrekkefølge
        ii. Definer suksesskriterier, For hver oppgave
8. Hva er timeplanen?

    Brukertest 1
    Dato: 12.10.2018

Tid	Navn/E-post	Tlf	Stilling	Mål-gruppe	Alder	Kommentar
09:00-09:45	Bruker 1					
10:15-11:00	Bruker 2					
11:00-12:00	LUNSJ					
12:00-12:45	Bruker 3					
13:15-14:00	Bruker 4					
14:30-15:15	Bruker 5					

    a.
9. Hvilke spørsmål skal du stille brukeren?
    a. innledende spørsmål og avsluttende intervju må beskrives

10. Hvem er i testteamet
    a. Dette avhenger av testens omfang
    b. De rollene man kan ha i en bruker test:

Rolle	Ansvarlig for
Testleder	Overordnet ansvar for planlegging og gjennomføring
Hoved observatør	Den av observatørene som følger med på samtlige testbrukere, ansvarlig for gjennomføring av ulike oppgaver, …
Observatør	Observerer testbruker og noterer
Maskinstyrer	Ved bruk av prototype (uferdig), fjernstyre ulike tilstander for å simulere et fungerende system. På papir prototype, byttes ark
Prosjektleder	Ansvarlig for godkjenning av testplan som omfatter tidsforbruk og bemanning
Systemutvikler	Ansvaret for test data, systemet er tilgjengelig for brukerne, …

11. Hvordan skal funnene formidles?
    a. Resultater fra brukertest skal sammenstilles i en rapport
    b. Dette danner grunnlaget for endringer evt godkjenning av systemet
12. Når skal vi mætes for å ta funnene over i designendringer

Universell Utforming

Prinsipper:

1. Enkel og intuitiv i bruk. Utformingen skal være lett å forstå uten hensyn til brukerens erfaring, kunnskap, språkferdigheter eller konsentrasjonsnivå.
2. Forståelig informasjon. Utformingen skal kommunisere nødvendig informasjon til brukeren på en effektiv måte.
3. Toleranse for feil. Utformingen skal minimalisere farer og skader som kan gi ugunstige konsekvenser, eller minimalisere utilsiktede handlinger.
4. Like muligheter for alle. Utformingen skal være brukbar og tilgjengelig for personer med ulike ferdigheter.
5. Fleksibel i bruk. Uansett individuelle preferanser og ferdigheter. Den synshemmede skal kunne høre, den hørselhemmede se og så videre.
6. Lav fysisk anstrengelse. Uformingen skal kunne brukes effektivt og bekvemt med minimum besvær.
7. Størrelse og plass for tilgang og bruk. Hensiktsmessig størrelse og plass skal muliggjøre tilgang, rekkevidde, betjening og bruk, uavhengig av brukerens kroppsstørrelse, kroppsstilling og mobilitet.

Med dette menes utforming eller tilrettelegging av hoved løsningen i de fysiske forholdene, herunder informasjon s og kommunikasjonsteknologi, slik at virksomhetens alminnelige funksjon kan benyttes av fles mulig. Dette er alt som går under hvordan produktet kan lett brukes av alle typer brukere så alt som går inni i hensyn til vansker og kunnskaps nivåer. I praksis er det vanskelig å lage et system som alltid oppfyller alle prinsippene innen universell utforming. Det er likevel veldig mye vi kan gjøre, og i motsetning til det mange tror, er det ofte ikke mer tidkrevende ¨designe universelt enn å designe vanlig, det handler først og fremst bare om å ha med seg denne måten p tenke på helt fra starten.

Lovkrav og tilsyn

Diskriminerings og tilgjengelighetsloven (§9 & 11) stiller krav om universell utforming til offentlige og private virksomheter

URL: https://lovdata.no/dokument/LTI/lov/2008-06-20-42

Krav til nettløsninger

WCAG = Web Content Accessibility Guidelines

- Retningslinjer for tilgjengelig webinnhold

Oppbygging av WCAG 2.0

Prinsipper - På øverste nivå er det fire prinsipper som danner grunnlaget for tilgjengelighet på nett:

- mulig å oppfatte
- mulig å betjene
- forståelig
- robust

Retningslinjer - På nivået under prinsippene er det 12 retningslinjer som utgjør de grunnleggende målene. Retningslinjene er ikke testbare, men utgjør rammene og de overordnede målsetningene

Suksesskriterier - For de enkelte retningslinjene er det testbare suksesskriterier som gjør det mulig å bruke WCAG 2.0 til å teste at krav overholdes, blant annet i forbindelse med designspesifikasjoner, innkjøp, lovregulering og avtaler

Prinsipper (Center for Universal Design)

1. Enkel og intuitiv i bruk
2. Forståelig informasjon
3. Toleranse for feil
4. Like muligheter for alle
5. Fleksibel i bruk
6. Lav fysisk anstrengelse
7. Størrelse og plass for tilgang og bruk

Gode argumenter for Universell utforming

Universell utforming gjør nettsidene dine:

- mer inkluderende
- enklere å bruke
- mer mobiltilpasset
- synlige for Google og andre søkemotorer
- mer lønnsomt for deg

UML

Hvorfor modellerer vi?

- Kommunikasjon
    - Der og da
    - Noen ganger trenger vi å illustrere konsepter visuelt i tillegg til verbalt
    - Tekniske konsepter lar seg godt beskrive med modeller av ulike slag.
    - Vi modellerer gjerne hurtig og udetaljert sammen på et whiteboard (model storming).
- Dokumentasjon
    - For ettertiden
    - En modell inneholder mye komprimert informasjon
    - Uttrykker ofte mer med en modell enn med mange sider i et dokument
    - En modell er mer

Model Storming
Vi kan modellere sammen i møter eller små arbeidsøkter for å løse et problem der og da. Det perfekte verktøyet er da et White Board. Vi lager enkle, overordnete modeller hurtig. Digitale verktøy kan brukes, men det kan ta litt mye tid og derfor ta fokus bort fra den gode diskusjonen. Modellen blir ikke perfekt og må ikke nødvendigvis tas vare på. Ofte tar man likevel bilde av tavla og arkiverer det. Model stroming er vanlig praksis i agile prosjekter.

Dokumentasjon
Vi kan også lage mer detaljerte modeller som dokumenterer hvordan ting er gjort i et prosjekt. Dette gjøres gjerne for å dokumentere for ettertiden. Dette gjerne for at andre skal kunne ta over et prosjekt eller gjære vedlikehold. Her vil gjerne en utvikler eller arkitekt modellere ved hjelp av et avansert verktøy. Dette er vanlig praksis ved bruk a «unified process».

UML - Diagrammer

Stereotyper
Stereotyper er gjennomgående for alle typer objekter i alle diagrammer i UML. Det sier noe om type objekt, altså klassifisering. Det er mer eller mindre det samme som arv, men brukes gjerne når det blir for omstendelig å bruke arv- En entitet kan for eksempel representeres med «Entitiy» - stereotypen. Man kan også legge til egen definerte stereotyper for å klassifisere objektene ytterligere.

Use case (Brukstilfelle)
Et use case beskriver en typisk spesifikk brukssituasjon av systemet. I modellen representeres den med en oval sirkel med en tittel som alltid inneholder verb. I tillegg ønsker vi å si hvem som er assosiert med et use case. En aktør er typisk en bruker av et system, gjerne gruppert etter roller, men kan i noen tilfeller også være et eksternt system som er involvert. En aktør assosieres med et use-case ved hjelp av en enkel linje uten piler.

««Include»»
Vi kan øke detaljeringsgraden ved å bruke inkluderte use case. Vi sier da at et use case er en obligatorisk del av en annen use case. På denne måten kan vi også gjenbruke use case. Typisk kan søkefunksjonalitet brukes i flere forskjellige andre

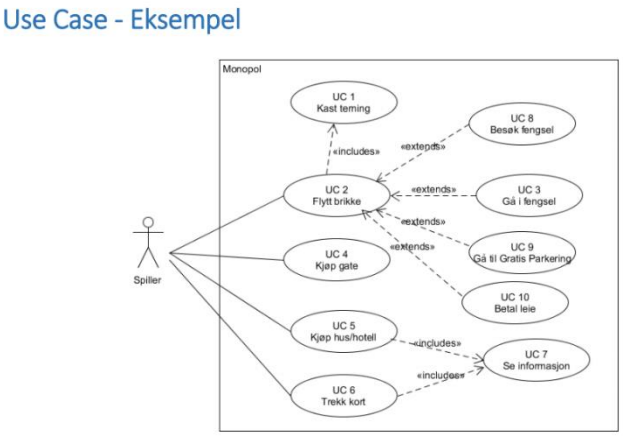
use case. Merk at dette ikke sier noe om rekkefølge, kun at for å kunne registrere et slag må vi også finne en kunde.

««Extend»»

Vi kan også bruke utvidelser, Use case relatert ved bruk av ««extend»». Nå er ikke det relaterte use caset en obligatorisk del av et annet, men noe som blir utført i spesielle tilfeller. Utvidelsen har derfor gjerne en betingelse assosiert med seg. Hvis denne oppfylles, skal også tillegget utføres. Merk at pila går motsatt vei. Tilleggs-caset utvider base -caset. Vi kan for eksempel se for oss en situasjon der en betaling ikke går igjennom og en kunde må gjennomføre en kontantbetaling.

Systemgrense

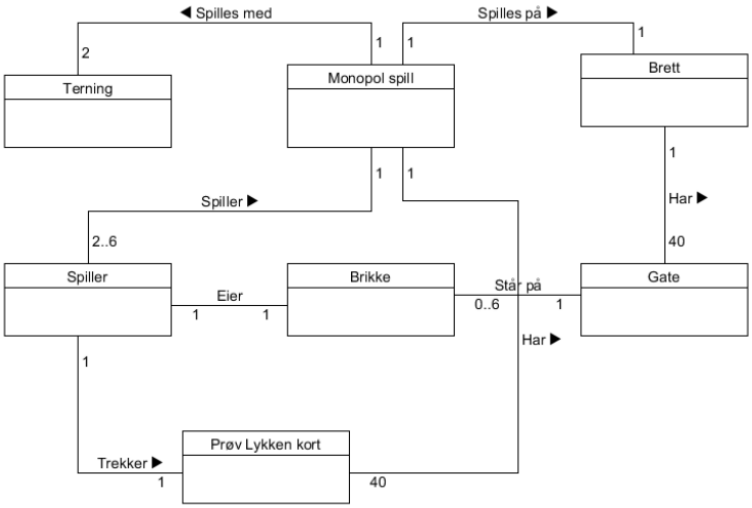
Vi kan lage en systemgrense som forteller hvilke use case som tilhører et system. Dette vil typisk være kun et system og nettopp det systemet vi skal lage, men det kan i mer komplekse tilfeller være snakk om flere systemer.



Domenemodell

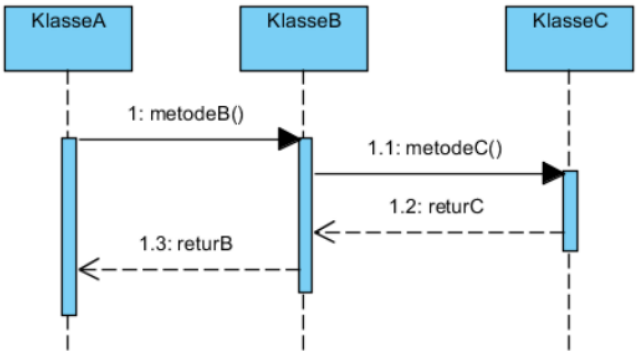
Domenemodellens funksjon er å få utviklere og domeneeksperter til å enes om en felles forståelse av problemdomenet. Utviklerne må forstå domenet de skal jobbe med og alle intersenter må snakke om det samme når man bruker begreper fra domenet vi trenger et felles språk for å unngå misforstpelser i utviklingsprosessen. Navngiving av elementer i modellen er derfor veldig viktig. En domenemodell skal ikke være detaljert. Den skal vise konseptene i et domene på en oversiktlig måte. Blir modellen for stor mister den sin funksjon, som er kommunikasjon. Husk at den skal kunne benyttes i samtale med ikke tekniske interessenter.

Domenemodell - Eksempel



Sekvensdiagram

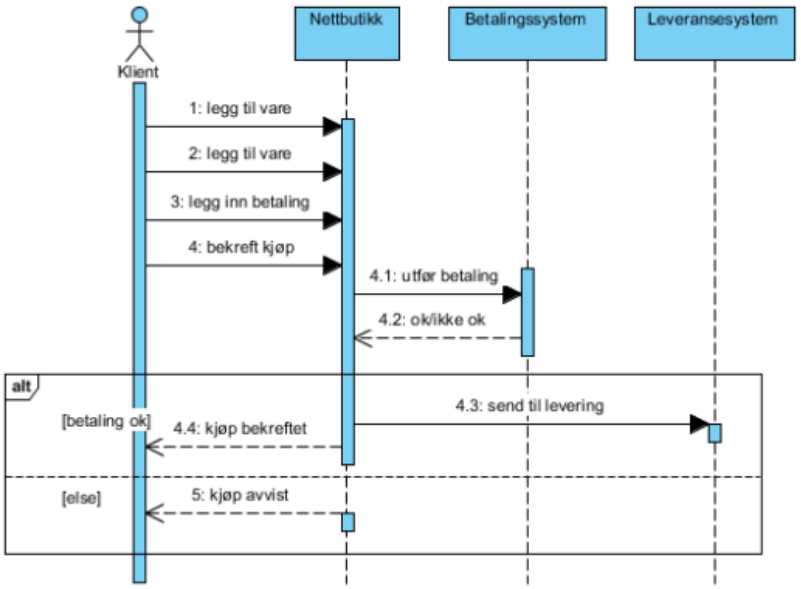
Sekvensdiagram




```java
public class KlasseA {

  private KlasseB b = new KlasseB();

  public String metodeA() {
    String returB = b.metodeB();
    ...
    ...
    return returB;
  }
}
```

```java
public class KlasseB {

  private KlasseC c = new KlasseC();

  public String metodeB() {
    String returC = c.metodeC();
    ...
    ...
    return returC;
  }
}
```

Et sekvensdiagram viser kontrollflyt mellom klasser inne i et system, som i eksemplet over, men kan også vise flytt mellom komponenter/subsystemer

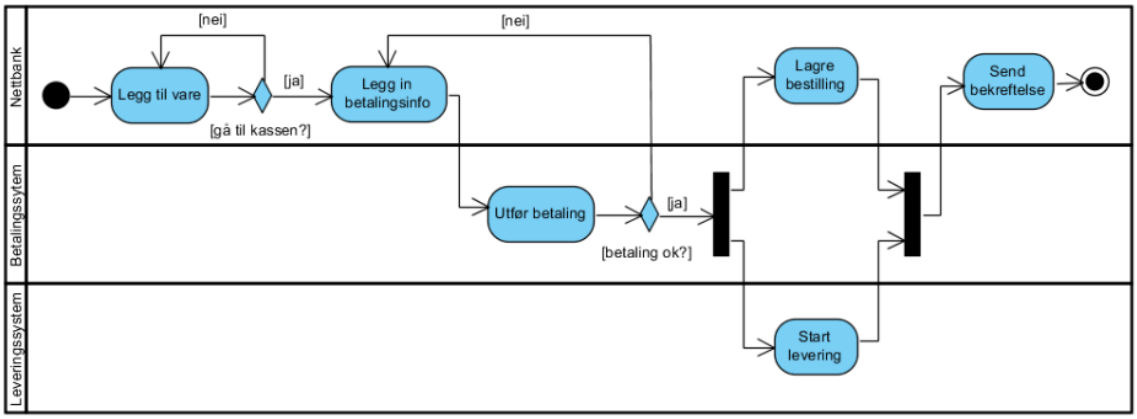


Objekter eller klasser representeres med bokser. Det er vanlig å bruke «: klassenavn» eller kun klassenavnet. Tid vises med en tidslinje fra topp til bunn og rektanglet viser kontrollflyt. Meldinger representeres med heltrukken linje og tekst. Hvis vi er på klassenivå er dette synonymt med metode retur meldingen representeres med stiplet linje og tekst. Noen ganger kan det være nyttig å være tydelig på når objekter kreeres og destrueres. Hvis dette ikke er vesentlig, trenger man ikke ta det med. Objekter som blir kreert kan plasseres der de blir kreert på tidslinjen. Eksplisitt destruksjon av objekter kan evt vises med et kryss på tidslinja. Løkker kan representeres med en ramme med en betingelse for å iterere. Man kan bruke flere typer rammer; **opt** for if, **alt** for if/else og andre som kan si noe om parallellitet.

Aktivitetsdiagram

Et aktivitetsdiagram kan vise aktiviteter i en prosess hvilken rekkefølge de må utføres i, hvilken som kan gjøres i parallell og evt betingelser for at de skal utføres. Aktivitets diagrammer blir ikke så mye brukt i dag fordi vi programmerer mer abstrakt enn før. Men for komplekse forretningsprosesser og arbeidsflyt er de kjempenyttige. Her har vi gjerne mange deloppgaver hvor noen er avhengig av at andre er utført for andre kan begynne. Dette er det mye lettere å visualisere enn å forklare med ord. Hvis man lager komplekse lavnivå algoritmer er de også nyttige. Aksjoner representeres med bokser og inneholder derfor verb. Start og slutt representeres med en svart dot og en svart dot med ring rundt seg. Overganger mellom aksjonene representeres med

enkle piler. For å skjøre flere aksjoner parallelt, kan vi «forke» slik at vi får flere eksekverende tråder. Så slår vi sammen trådene i etterkant med en «join». Representeres med svar tykk strekk for star og slutt. For å velge en av to eller flere aksjoner kan vi lage en decision node med en diamant. Vi legger til betingelser i parenteser. Swim lanes er hvordan vi representerer hvilke komponenter som gjør hva-



Programvarearkitektur

Er mange bilder og eksempler, veldig lite tekst, bedre å se bildene.

Handler om hvordan hele programmet skal være strukturert, alt fra hva slags kode språk til hva salgs, kvalitets standard på alt som blir laget.

Definisjoner

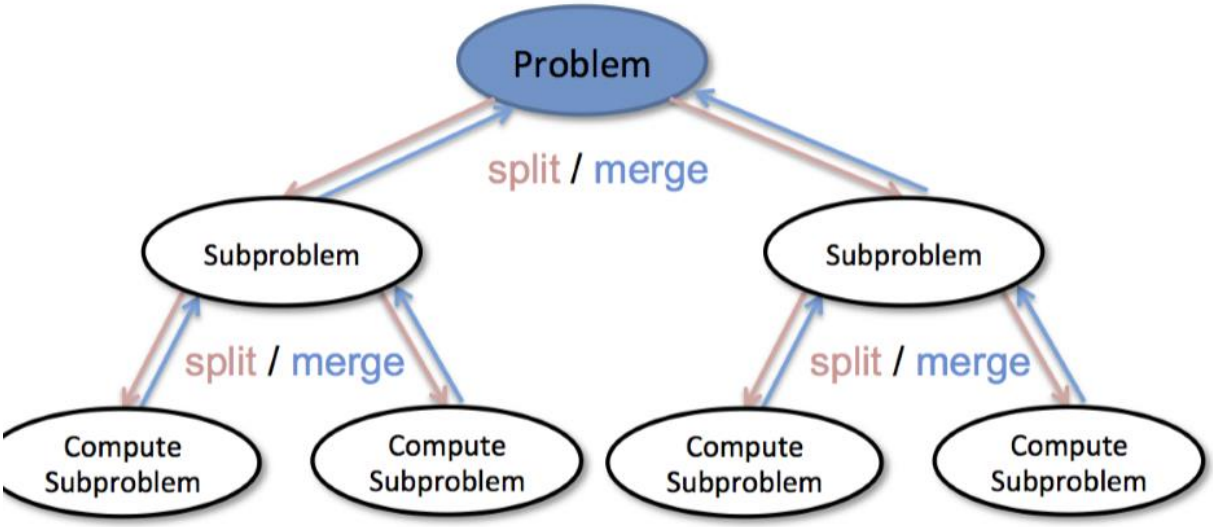
De grunnleggende strukturene av programvaresystemer, og hvordan vi på beste måte kan gjøre gode valg av de grunnleggende strukturene. Erfaringer viser at det blir dyrt å gjøre om på slike valg etter at utviklingen har kommet godt i gang. Hovedhensikten med arkitekturen er å redusere risikoen i et utviklingsprosjekt. Det er sammenheng mellom maskinvare-arkitektur og programvare-arkitektur

Hvorfor skal vi jobbe med arkitekturen til et system

Grunnlag for analyse av systemets oppførsel før det har blitt bygd. Grunnlag for beslutninger om elementer som kan gjenbrukes. Støtte for tidlige beslutninger om utvikling, implementasjon og vedlikehold. Grunnlag for kommunikasjon med oppdragsgiver og/eller andre interessenter. Hjelpemiddel for risikovurderinger. Mulighet for kostnadsreduksjoner.

Hvordan håndtere kompleksitet?

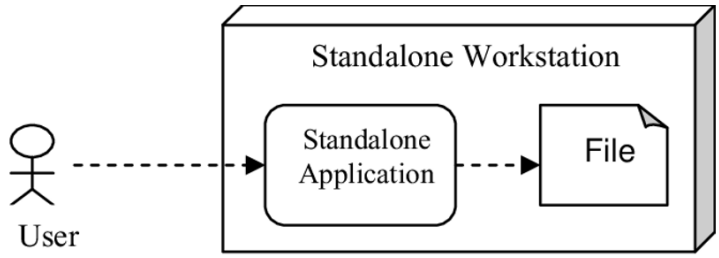
- Den mest vanlige strategien er å dele opp i mindre deler
- Splitt og hersk (Divide and conquer)



Vanlige modeller

Standalone Application:

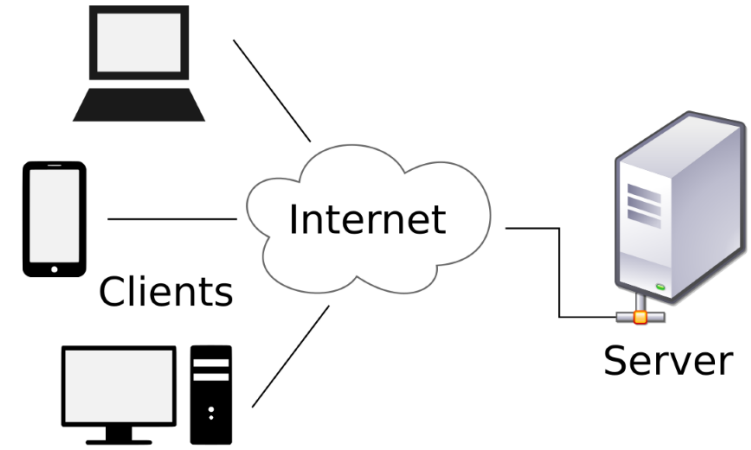
Standalone Application



Klient – tjener arkitektur:

Også kjent som klient-server arkitektur. Denne arkitekturen er bygg opp av to parter en server og flere klienter. Serveren utfører oppgaver for de forskjellige klientene. Og klientene spør om forskjellige tjenester som de trenger. Blir brukt i online apper som email, dokument deling, og bank.

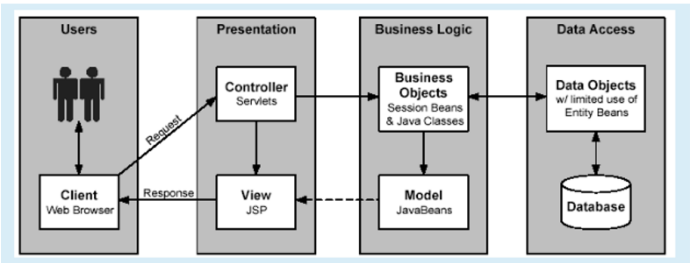
Klient - tjener arkitektur



Tre-lags-arkitektur:

Denne strukturen går ut på at programmet er de strukturert til forskjellige lag, som utfører forskjellige oppgaver. Hvert lag utfører oppgaver for et lag høyere opp kjeden. Blir brukt i generelle desktop apper og butikk web apper.

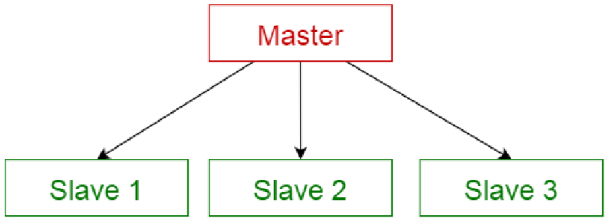
tre-lags-arkitektur



Master-slave pattern:

Denne pattern'et er bygd opp av to partier, master og slaves. Master er komponenten som delegerer arbeid til de forskjellige identiske slave komponentene, og returnerer ferdig resultater tilbake til master. Blir brukt i database replikasjon, hvor master her autorativ Source og salvene er databaser som er i sync med denne. Blir også brukt i prinsipper koblet til bus i et data system.

Master-slave pattern



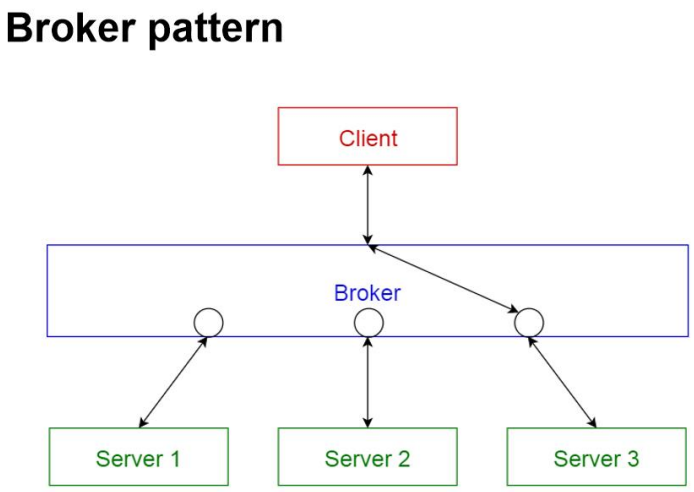
Pipefilter pattern:

Denne pattern kan bli brukt til å strukturere systemer som produserer og prosesserer strømmer av data. Hvert prosesserings steg utføres i en filter komponent. Data som skal bli prosessert blir sent gjennom pipes, pipsene kan bli brukt for bueføring eller for synkronisering hensikter. Blir brukt i kompilatorer. De forskjellige filtrene utfører lexical analyse, parsing, semantic analyse og kode generering.

Pipe-filter pattern

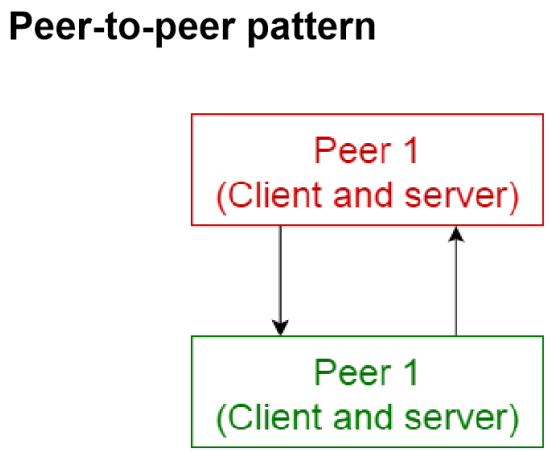
Broker pattern:

Denne pattern er brukt til å strukturere systemer med uparete komponenter. Disse komponentene kan kommunisere gjennom remote service invokasjons. En broker komponent har ansvar for koordinasjon og kommunikasjon mellomkomponentene. Serveren publiserer sine kapasiteter til en broker. Clienten spør etter en service fra brokeren, og brokeren sender clienten til et passende service. Brukt i meldings broker software som Apache ActiveMQ RabbitMQ og JBoss Messaging



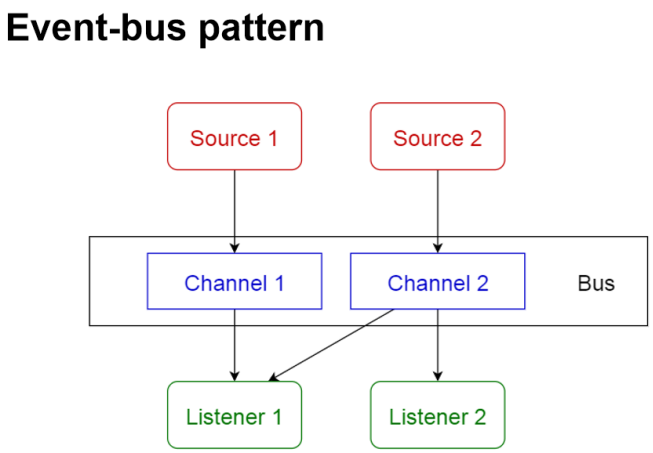
Peer to Peer pattern:

I denne pattern er individuelle komponenter kalt peers. Peers fungerer både som en client og en server. Denne rollen kan dynamisk endres når som helst. Blir brukt i fil delings neverk som Gnutella og G2. Multimedia protools som P2PTC OG PDTP.



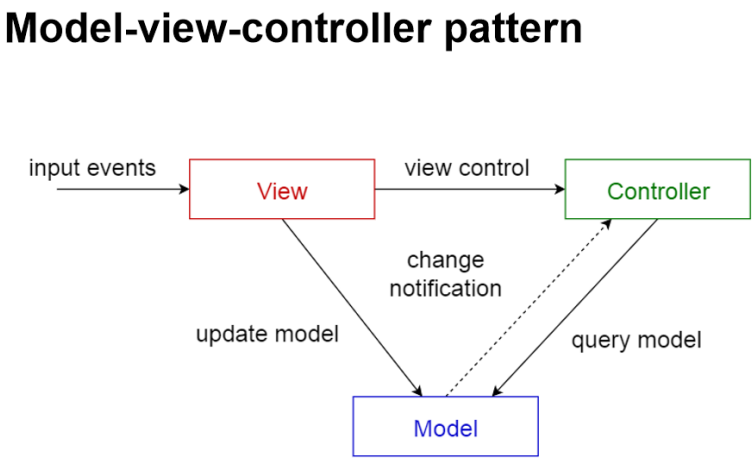
Event-bus pattern:

Denne pattern driver primært med events og har fire hoved komponenter. Event source, event listner, channel og event bus. Kilden publiserer en melding til en spesifikk kanal på event bussen. Listern abonnerer til denne kanalen, og listner sier ifra når en melding blir publisert på en kanal de abonnerer på. Blir brukt i Android Development og notifikasjon tjenester.



Model-view-controll pattern

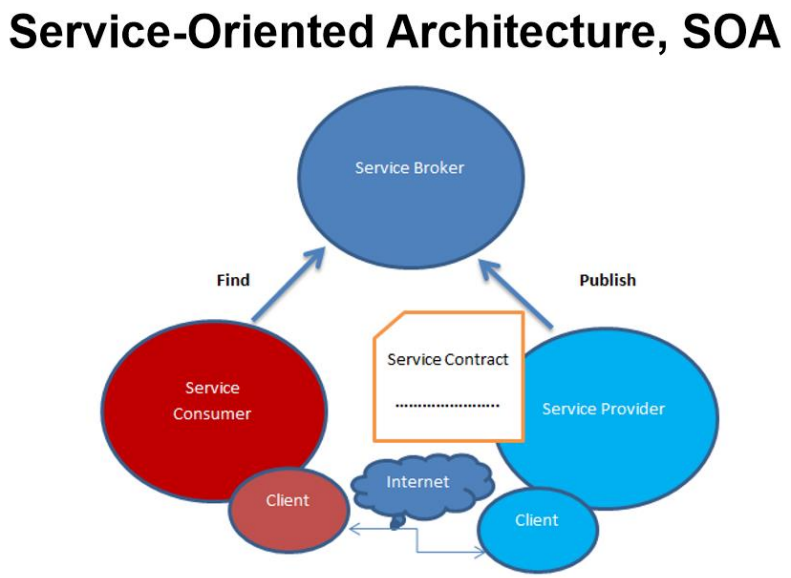
Denne pattern, også kjent som MVC patter, deler en interaktiv app inni tre deler kjent som, Model som har hoved funksjonaliteten og data. View som viser informasjon til brukeren. Controller som håndterer input fra brukeren. Dette blir gjort for å separere den interne representasjonen av informasjon fra hvordan den er presentert til, og akseptert fram brukeren. Den deler opp komponenter og gjør rede for effektiv gjenbruk av kode. Blir brukt i arkitektur for World wide Web applikasjoner i store programmerings språk, og blir brukt i Web frameworks som Django og Rails.

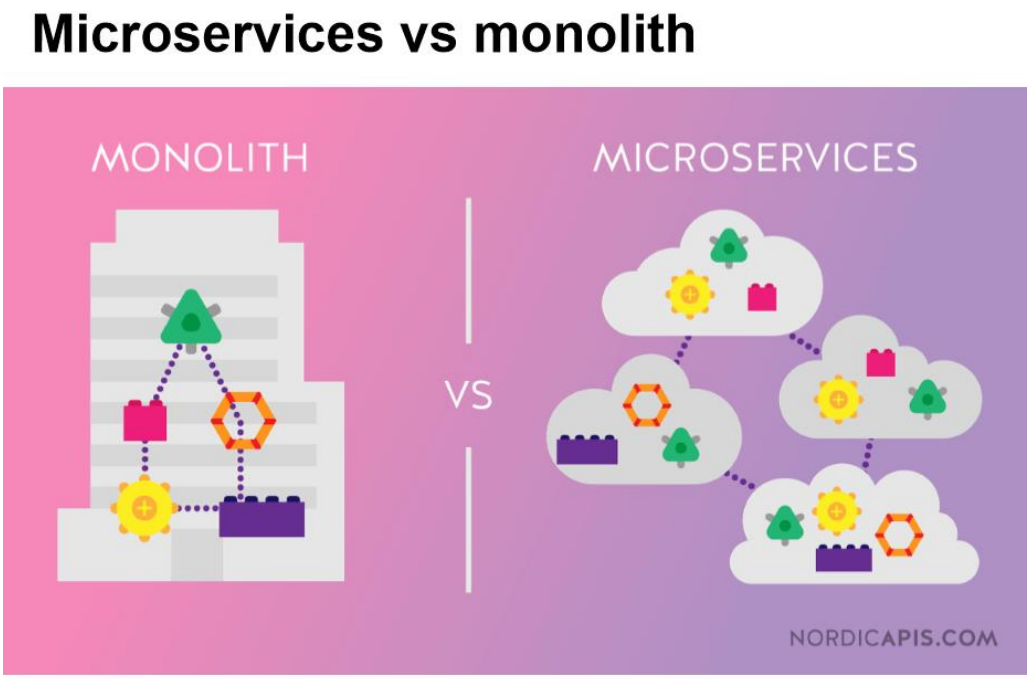


Ulemper, fordeler og kilde:

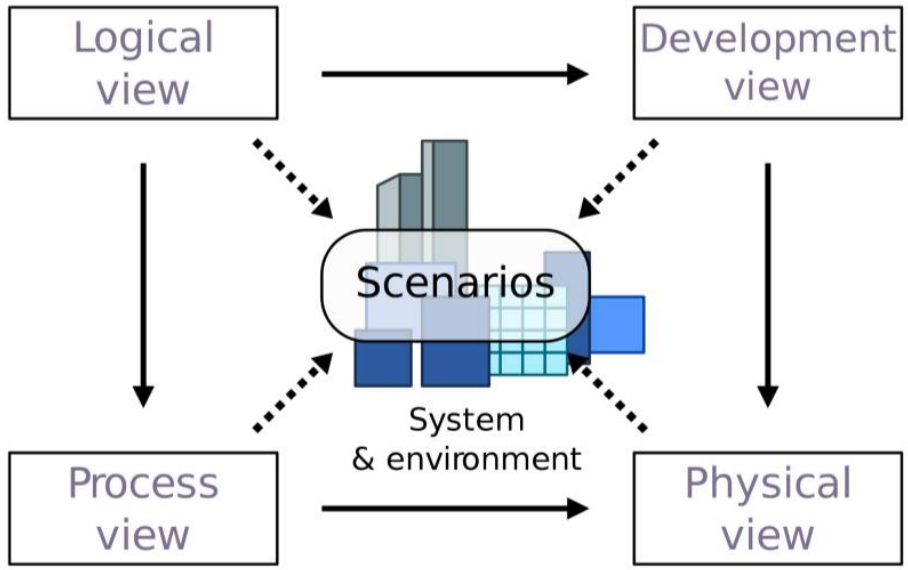
https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013

Service-oriented Architecture:





Arkitektur i systemutviklingen



Smidige Metoder

Metodikk

Fra ca. 1970 var det vannfallsmodellen som var gjeldende. Rundt år 2000 kom både UP og smidige metoder. Up ble mest utbredt i årene rett etter år 2000 (det lignet med på det vante) Smidige metoder er det som er «teller» nå, med XP TDD Scrum, Kanban, Domain-driven design.

Vannfallsmodellen

Er en tungmodell En og en fase, med krav først så design osv. Forskjellige folk i hver fase. Overlevering av mye dokumentasjon fra fase til fase. Endringer underveis ble vanskelig. Kunne ikke gå tilbake å endre uten at det ble fryktelig dyrt. Mange prosjekter feilet.

UP (Unified Process)

Up innførte iterasjoner for å takle endringer underveis. Et firma som heter Rational konkretiserte UP med et sett med maler for dokumentasjon som ofte brukes i dag. UML-modellering er svært sentralt i UP. Dokumentmalene definerer hvilke modeller som skal brukes. Er use-case-drevet. Vi starter med use-case'r og detaljerer mer og mer med andre typer modeller etter hvert. Bruker faser.

Smidig utvikling

Smidig, og raskt. Smidig utvikling oppsto på 1990-tallet med 2001 er likevel et merkeår siden The Agile Manifesto ble publisert da. Oppsto som en reaksjon på de tunge prosessene. Det var vanskelig å gjøre endringer på krav underveis. Folk følte at de ikke fikk gjort nok nyttig jobb pga. krav om store mengder dokumentasjon.

Extreme Programming

Kanskje den første smidige metoden. Veldig nyskapende og har prinsipper i seg som er gjeldende i dag. Nesten ingen dokumentasjon og modellering Har 29 regler.

XP har 29 (kanskje i overkant rigide) regler:

- Programmer ALLTID parvis (par-programmering, kontinuerlig kode-review)
- Lag ALLTID en enhetstest for det du skal lage FØR du skriver koden.
- Ha en kort feedback-loop, hyppige tester og tilbakemeldinger fra kunde skal resultere i kontinuerlige endringer. • Kjør ofte akseptansetester av større deler av systemet. Disse lages fra user stories og bør helst være automatisert.
- Ha korte iterasjoner med leveranser i hver av dem.
- Ha fokus på enkel og tydelig kode. Refaktorer med en gang koden begynner å lukte. • Skriv aldri kode du tror kan bli kjekk senere. Dette er i kontrast til up-front-modellering (UP) hvor man gjerne prøver a lage noe flott og generelt som gjør at utvidelser skal bli enklere.
- +22

TDD (test-Driven develompent)

Viderefører av XP, men ikke like rigid. Fokus på enhetstesting og akseptansetesting. Fokus på refaktorering, omskrivning av kode når den begynner å lukte. Tester og krav er en og samme ting:

- Enhetstester
    - setter krav til kode
    - validerer kode
- User stories
    - definerer kundekrav
    - er testcase for akseptansetester

Scrum

En smidig prosessmetode som ikke forteller oss noe om hvordan vi skal utvikle koden eller modellere krav, men hvordan vi skal jobbe og få fremdrift. Team på ca. 7-8 pers, stand up møter hver morgen 15 min. sprinter med klare mål hver sprint og demo når hver sprint er ferdig. I etterkant av en sprint utfører man en sprint-review. Her presenterer teamet resultatet av sprinten for produktseriene slik at de kan få tilbakemeldinger på dert som er gjort. På denne måten sikrer man at man lager det produktet produktseriene egentlig vil ha. I tillegg gjennomfører man et retrospektiv. Her er det kun teamet som deltar og her evaluerer man hva som har fungert i siste sprint og hva som eventuelt kan gjøres bedre. På den måten kan også kontinuerlig forbedre hvordan man jobber, og ikke bare som produseres.

Scrum-board

For hver sprint starter man med sprint-planlegging, et møte hvor hele teamet deltar. Man finner oppgaver, deler dem eventuelt opp i mindre oppgaver og plasserer dem på et scrum-board. Disse flyttes underveis i sprinten slik at man alltid har oppdatert status.

Lean

Lean er et begrep/tankegang/metode på samme måte som agil- Har veldig mange likheter men har annen opprinnelse, nemlig fra «lean manufacturing» og Toyotas produksjonssystem. Det er vanskelig å se forskjellen på smidig og Lean. Lean legger hovedvekt på å unngå å gjøre/lage ting uten verdi eller «waste».

Prinsipper i Lean development

- Eliminate waste: Alt som produserer må ha verdi. Unngå funksjonalitet man tror kan bli kjekk senere, men kunden egentlig ikke trenger, dokumentasjon ingen leser, for mye byråkrati osv.
- Build quality in: Ha fokus på kvalitet på kode og systemet som helhet. Dette kan oppnås med god forståelse av problemdomenet og god kommunikasjon med kunden, og bruk av teknikker som refaktorering og testing på mange nivåer.

- Create knowledge; Kort feedback-loop mellom kunde og utvikler slik at flere ting kan testes og man kan gjøre endringer raskt. Utviklere bør også reflekterer over det de gjør slik at forbedringer kan gjøres.
- Defer commitment: Vent med irreversible avgjørelser så lenge som mulig, til vi har lært mest mulig.
- Deliver fast: Tenk enkelt, unngå over-engineering og få ting u ti produksjon så tidlig som mulig. Få den viktigste funksjonaliteten ut i produksjon først. Sannsynligheten for at det er bare de fleste brukarene bruker er likevel høy. Bruk tilbakemeldinger fra brukere til å gjøre forbedringer og de riktige utvidelsene senere.
- Respect people: La utviklerene få ansvar. Blir man redusert til en «code-monkey» mister man motivasjon og man er ikke effektiv. Ledere bør motivere folk i stedet for å kontrollere dem.
- Optimize the whole: for å lykkes må man forstå helheten i et system, forstå problemdomenet og forstå avhengigheter til andre systemer.

Kanban

Stammer fra lean-verdenen Er en enkel metode for å unngå flaskehalser i produksjon eller ved utvikling av it-systemer. Man bruker et kanban-board som har en spesifikk egenskap, nemlig et tall som begrenser antall oppgaver som kan havne i hver kolonne. Dette er for å unngå eller motsatt. Hvis du blir arbeidsledig siden det ikke er lov å putte flere lapper i en kolonne skal du heller hjelpe de som har ting å gjøre. Tenk at du jobber på et samlebånd med flere stasjoner. Når du mangler oppgaver på din stasjon springer du bort å for å hjelpe andre slik at produksjonen aldri stopper.

Testing

Testing av programvare

Vi tester programvare for å validere at systemet:

- Responderer riktig på alle typer input
- Ikke har bugs
- Yter godt nok, både ved stort trykk og over tid
- Er bruker vennlig nok
- Møter kravene satt av interessenter

- Kan installeres i riktig miljø
- Ikke brekker ved endringer.

Tradisjonelt testet man i gitte faser av prosjektet, mens med smidig metodikk orøver man å teste kontinuerlig og mer automatisk. Mer av ansvaret ligger nå hos utviklerne, mens man før i større grad benyttet seg av rene testere.

Tradisjonell vs. Smidig testing

Før var utvikling og testing separat og man overleverte produktet, og ansvaret, i forskjellige faser av utviklingen. Nå er kvalitet hele teamets ansvar hele tiden. Mange roller samlokaliseres, og fungerer som ETT team, ikke som «oss og dem».

Kontinuerlig integrasjon

Handler om å automatisere mest mulig av systemutviklingsprossesen. Bygge kode, kjøre tester, installere bygd versjon i testmiljøet, evt. produksjonsmiljøet, når en utvikler sjekker inn kode på GitHub kan man i teorien automatisk legge ny versjon i produksjon i løpet av minutter. Men de fleste ønsker å ha noen manuelle tester i tillegg. Men det koser lite å lage nye versjoner, og derfor kan man lage nye versjoner ofte, kanskje flere om dagen. Uten CL er det nesten ikke mulig å jobbe smidig.

Å teste akkurat nok

Når man bruker Cl og lager nye versjoner hver dag blir forskjellen på ny og gammel versjon veldig liten. Sannsynligheten for feil blir derfor mindre og sannsynligheten for at en eventuell feil er stor er også mindre. Når man også kan rulle tilbake til forrige versjon veldig raskt hvis man oppdager feil i produksjon, vil kanskje ikke konsekvensen av dette våre så stor? Det gjelder å finne passe nivå på testingen, og å prioritere testing i forhold til risiko. Vi kan benytte oss av risk-poker.

Risk-poker

Alle på teamet «scorer» sannsynligheten og impact samtidig med en kortstokk, så bruker vi snittet. Med denne tilnærmingen er det ikke en eller noen få i teamet som avgjør, men hele teamet. Da blir treffsikkerheten mye større.

Test-data

Når vi skal kjøre tester er vi avhengig av et forutsigbart test-miljø. Test-dataene må derfor være lik hver gang vi kjører en test. Vi kan for eksempel lagre SQL-skript i versjon kontrollsystemet som oppretter ønsket tilstand før vi kjører en test. Viktig prinsipper er repeter barhet og determinisme. Når vi kjører tester automatisk må

testen kunne forutsette at den jobber på lik data hver gang den kjører. Da kan testen kjøres et vilkårlig antall ganger og den vil produserer samme resultat hver gang og derfor være deterministisk.

Mocking

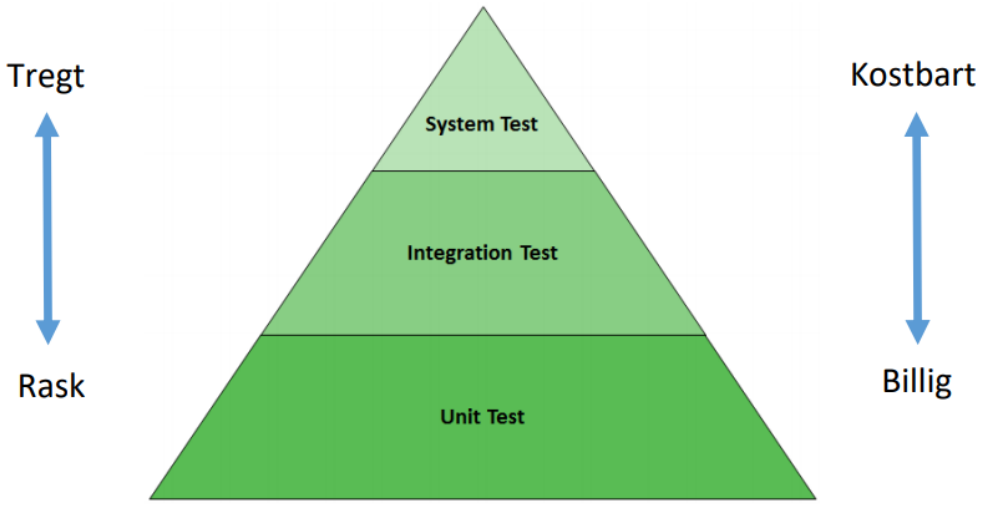
Noen ganger kan vi ikke jobbe med reelle data. Da kan vi «mocke»/«fake»/»simulere». Vi kan bytte ut en eller flere komponenter eller klasseinstanser med «mock»-objekter, også kalt «dummy». Vi benytter f.eks. aldri en produksjons database i en test. Vi bruker heller en helt egen test database som vi oppretter før og kaster etter bruk. Når man tester krasj-egenskaper til biler bruker man heller ikke ekte mennesker, men heller en «crash-test dummy».

Testfaser

En funksjonell test består av tre (eller fire) faser:

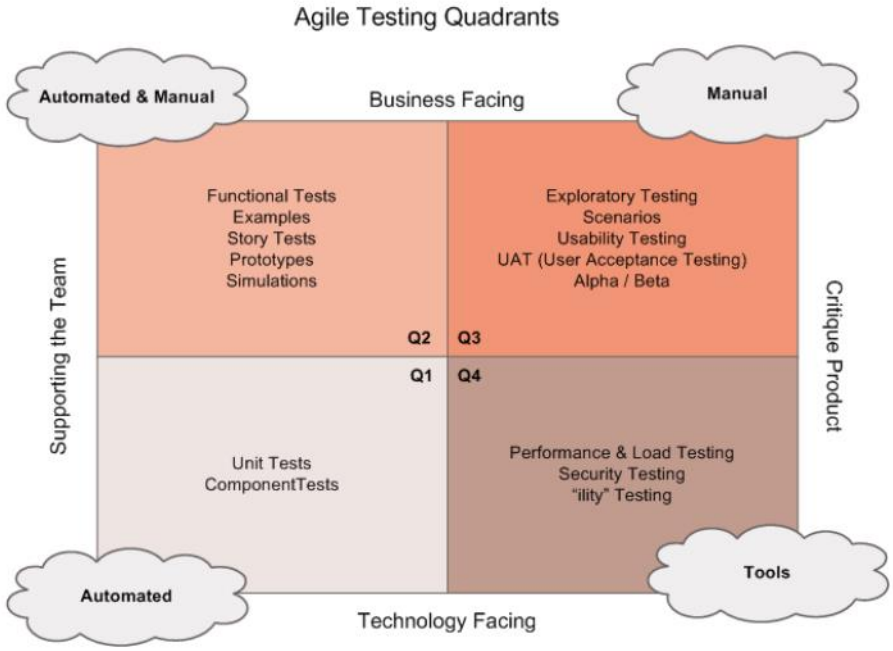
Oppsett	I denne fasen skal man sikre at testdataen som benyttes blir lik hver eneste gang testen kjøres. Vi oppretter altså en fast starttlstand, også kalt en «**test fixture**».
Gjennomføring	Vi utfører operasjonene som er spesifisert i testen.
Verifisering	Vi må «asserte»/ «bekrefte» at den nye tilstanden er slik vi forventer det.
(Opprydding)	Dette er strengt tatt ikke en del av testen, men en mer teknisk fase hvor man eventuelt rydder opp etter seg. Det kan være å lukke eller skrote testressurser.

Testnivåer



Test-typer

Test-typer – Agile Testing Quadrants



Enhetstesting

Utviklertesting av små enheter, gjerne individuelle klasser. Bruker gjerne Junit (Java), PyUnit (Python). Alltid automatisert. Testen må være repeterbar. Vi kjører alltid testen som en del av byggeprosessen.

Integrasjonstesting

Tester at større komponenter i systemet spiller sammen. Man kan sette sammen noen få eller flere komponenter for å finne ut om det eventuelt er noen problemer med grensesnitt mellom dem. Kan og bør automatiseres.

Systemtesting

Tester er fult integrert system, ende til ende, med alle komponenter og avhengigheter, for å sjekke at systemet møter kravene. Kan automatiseres med verktøy som Selenium men siden vi er avhengig av alle ressurser, som databaser og eventuelle eksterne systemer, kan dette bli vanskelig å gjennomføre siden det er vanskeligere å oppnå repeterbare tester.

Akseptansetesting

En type systemtest som typisk utføres av kunden for å verifisere at systemet møter de kravene som ble avtalt i kontrakten. Disse er da manuelle. Akseptansetesting i TDD handler om automatisk testing av «user stories». Akseptansetester i TDD kan gjerne være utviklet i samarbeid med kunden og har derfor en lignende funksjon.

Regresjonstesting

Skal sikre at systemet ikke brekker ved endringer. Testen må da kjøres for og etter en endring. Med kontinuerlig integrasjon skjer dette implisitt siden testene kjøres ved innsjekk av kode. Enhetstester fungerer derfor som regresjonstester. I tillegg kan man lage regresjonstester på integrasjonsnivå og akseptansenivå. Regresjonstester er gode kandidater for automatisering siden man skal utføre nøyaktig samme test for og etter en endring.

Smoke-testing/Sanity-testing

En uformell, ad hoc type test hvor man raskt prøver ut de viktigste delene av et system for å sjekke at alt henger på greip. Kan være greit at utviklere gjennomfører dette før systemet sendes videre til mere testing. Smoke-testing fokuserer på kritisk funksjonalitet i systemet mens sanity-testing fokuserer på ny funksjonalitet.

Utforskende testing

Testeren bruker sin kunnskap og erfaring med testing, sammen med kunnskap om systemet til å utføre kreativ testing. Kan åpenbart ikke automatiseres.

Destruktiv testing

Dette er type ad hoc, kreativ, utforskende testing hvor man aktivt prøver å få systemet til å feile. Kan ikke utføre automatisk. Vi må bruke all den menneskelige kreativitet og erfaring vi har.

Usability testing

Har vi laget det brukeren egentlig ønsker? Fungerer brukergrensesnittet.

Ytelsestesting

- Hvor mye last tåler systemet før det bryter sammen?
- Hvor mye plutselig last tåler systemet?
- Degraderes ytelse etter hvert?
- Har vi lekkasjer?

Hvordan måler vi ytelse

- Antall samtidige brukere
- Antall transaksjoner per tidsenhet
- Throughput
- Responstider

Flaskehalser

Vi ønsker å finne flaskehalsen i systemet. Da vet vi hvor systemet er tregest/svakest og hvor vi må sette inn ressurser for å gjøre forbedringer. Disse kan vi ofte finne hvis vi måler responstider på flere steder og på flere nivåer. Når vi har funnet flaskehalsen må vi finne årsaken til den. Veldig ofte er årsaken knyttet til begrensede ressierser som f.eks. Minne, CPU, Tråder; Databasekoblinger, gjerne via en ConnectionPool, Nettverk.

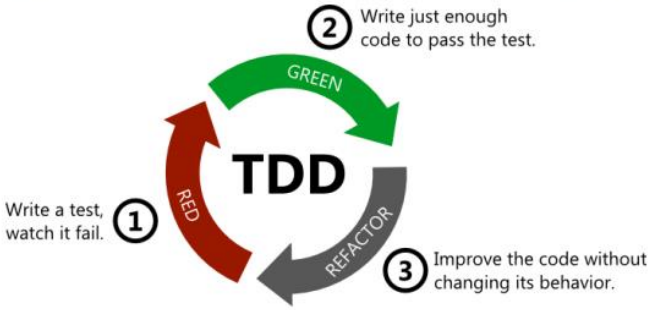
Typer ytelsestester

- Last-Test: Hvordan oppfører systemet seg ved forventet last? Hvor mye last tåler systemet før ytelsen degraderes vesentlig? Hva er det som gjør at ytelsen da degraderes?
- Stress-test: Finne systemets øvre limit. Når det bryter sammen? Hva er det som forårsaker at det bryter sammen?
- Spike-test: Hvordan oppfører systemet seg ved plutselig høy last? Dette ligner på stress-testing, men se for deg Birkebeiner-påmelding to minutter med ekstrem-last.

- Utholdenhetstest: Også kalt stabilitetstest. Hva skjer når systemet utsettes for jevn last over lang tid? Er ytelsen konstant, eller blir den dårligere? Er minnebruken konstant eller har vi minnelekkasjer? Er CPU bruken konstant eller er den økende? Er antall tråder konstant, eller er den økende?

TDD og BDD

Testdrevet utvikling handlet først mest om enhetstesting og «**test-first**»



Nå snakker vi også om BDD (Behavior Driven Development) eller ATDD (Acceptance Test Driven Development) som handler mye om automatiserte akseptansetester.

Profesjonsetikk

Mye av dette kapittelet er repetisjon fra i fjor, og mye er selv forklarende. Men en rask oppsummering er. Man må tenke over hva man lager og hvordan dette påvirker andre, og hva slags problemer som kan oppstå ved bruk av produktet man lager.

Ditt etiske ansvar – Programvare utvikling

- Etisk ansvarlig for programvare du utvikler.
- Handler ikke bare om å følge lover og regler – Moralske prinsipper må også følges.
- Programvare brukes på tvers av landegrenser.
    - Utviklere ukjent med brukeres kulturelle identitet.
    - Budsjett kan begrense mulighet for brukertester eller konsultasjoner med eksperter.
    - Dette kan føre til potensialet for brudd på personvern, kulturelle krenkelser og andre typer slike skader.

Etiske problemstillinger

- Ingeniører har et stort samfunnsansvar gjennom teknologien de forvalter og utvikler.
- Mye spennende innovasjon som kan gi positive effekter for samfunnet. Men - ny teknologi kan potensielt forårsake skade.
- I ingeniøryrket er det en økende forståelse for at etiske hensyn må anerkjennes i en stadig mer global verden.
- Ansvaret kan ikke ligge på teknologien alene, og derfor er vi avhengig av å ha regulering og styringsrammer.
- Et par eksempler:
    - Autonome/ selvkjørende biler. Ingeniører og teknologer utvikler bilen og dens «hjerne». Stanser ansvaret deres når bilen rulles ut fra fabrikken?
    - En ingeniørbedrift utvikler en robot som er i stand til å ta livet av et menneske. Hvilken rolle spiller ingeniøren knyttet til denne problemstillingen?

Ingeniørens ansvar, AI

- Hva med droner som er drapsroboter med kunstig intelligens og ladede våpen, klare for å drepe. Om få år kan disse være en realitet.
- Det tas til orde for at det er behov for raskt internasjonalt forbud mot slike våpen. Men har ikke også utviklerne et ansvar?
- NITO: uklart hvem sin oppgave det er å stille spørsmål ved om grunnleggende etiske hensyn er vurdert.
    - Den enkelte arbeidstaker har et ansvar for etikk, men vi savner verktøy for å arbeide med det.
    - Det finnes mange aktører som har utviklet etiske retningslinjer innen kunstig intelligens. Slike rapporter gir mange gode råd, men hvordan skal de settes ut i praksis?
- NITO mener at etiske vurderinger ikke kan overlates til den enkelte arbeidstaker. Etikk og bevisstgjøring formes gjennom offentlig diskusjon.
- Strategien peker både på Datatilsynet, Teknologirådet, forskningsmiljøene og de nasjonale forskningsetiske komiteene som aktører med et ansvar for å skape bevissthet om ansvarlig kunstig intelligens.

Etiske dilemmaer

- Du blir bedt om å gjøre noe ulovlig på jobb
    - Få det skriftlig, hør med sjefen, diskuter med kolleger, fagforening, advokat.

    - Før nøyaktig logg på det du gjør
    - Nekt – begrunn hvorfor
- Personvern og logging av aktivitet
    - Eks. Feilsendt epost med intime betroelser
    - Eks: Sjefen ønsker oversikt over ansattes nettrafikk
- Utvikling og bruk av IT har et etisk aspekt
    - Er den planlagte bruken av IT-systemet etisk forsvarlig:
        - I forhold til ansattes jobbsituasjon?
        - I forhold til personer som systemet har informasjon om?
        - Er systemet tilstrekkelig sikkert, slik at det ikke kan utnyttes av ondsinnede angripere?

GDPR

GDPR - General Data Protection Regulation.

Gir regler for elektronisk behandling av opplysninger om enkeltpersoner

Gjelder på de fleste samfunnsområder – enkelte unntak for eksempel

- Politi, forsvar, nasjonal sikkerhet
- Telekommunikasjon
- Journalistikk og kunstnerisk virksomhet
- Rent privat bruk av elektroniske hjelpemidler

Personvern og vern av personopplysninger

Personers rett til å være i fred. Som privatliv og familieliv, i hjemmet, personlig integritet, privat kommunikasjon og overvåkning.

Vern av personopplysninger: personlige opplysninger ikke anvendes til krenkelser av personvernet. Individuell innflytelse og kontroll med bruken av egne opplysninger.

Personopplysninger

Trenger ikke å være veldig personlige, men omfatter alle typer opplysninger eller informasjon som kan knyttes til en bestemt (identifiserbar) enkeltperson.

Personopplysninger kan være opplysninger om

- Navn, adresse, alder, kjønn, sivilstatus, bosted, utdanning, yrke, telefonnummer, bankkontonummer, fødselsnummer og portrettbilde
- Karakterer, fravær, faglig progresjon, spesielle behov, atferdsmønstre og sosiale ferdigheter.
- Eksamensbesvarelser, e-postkommunikasjon, innholdet i elektroniske meldinger bruker profiler.

Behandling av personopplysninger

Behandling- er et abstrakt begrep, omfatter alt du kan gjøre med opplysningene, gjelder primært elektronisk bruk av opplysningene.

Eksempler på behandling er: innsamling, registering, lagring, overføring, oppdatering, retting, sletting, sammenstilling, publisering, kryptering, osv.

Typer personopplysninger

Særlige kategorier (sensitive) opplysninger her inngår – helse og helserelaterte forhold, etnisk eller rasemessig bakgrunn, livssyn, politisk eller religiøs oppfatning, seksuell legning, strafferettslige, forhold, fagforeningsmedlemskap, genetiske opplysninger, biometriske opplysninger (når disse benyttes til personidentifisering).

Alminnelige opplysninger er alle typer opplysninger om enkeltpersoner som ikke regnes om særlige kategorier

Pseudonyme opplysninger skjuler den riktige identiteten til den som opplysningene handler om.

Anonyme opplysninger regnes ikke som personopplysninger er opplysninger som det ikke er mulig å finne ut at hvem det handler om.

Hovedaktør – den registrerte

Den enkeltpersonen som opplysningene handler om. En virksomhet eller bedrift er ikke den registrerte (virksomhets eller bedriftsopplysninger vil vanligvis ikke være personopplysninger). Den registrerte har visse rettigheter når andre gjør bruk av hans/hennes opplysninger.

Hovedaktør – behandlingsansvarlig

Ikke noe di er, men noe du blir. Du blir behandlingsansvarlig ved å fatte to typer beslutninger. Avgjør hva opplysninger skal brukes til, eller avgjør hvilke tekniske virkemidler som skal brukes. Behandlingsansvarlig har ansvaret for at personopplysninger håndteres på riktig måte. Kan bli ilagt sanksjoner, for eksempel gebyr, ved brudd på reglene i GDPR.

Hovedaktør – databehandler

Leie av datatjenester hos eksterne leverandører. Behandlingsansvarlig, for eksempel NTNU, er ansvarlig for de databehandlerne for eksempel inspera, som benyttes. Både behandlingsansvarlig og databehandler kan bli ilagt sanksjoner, for eksempel gebyr, dersom databehandleren bryter reglene i GDPR.

Hovedaktør – Datatilsynet

Hovedoppgavene til datatilsynet er å informere om personvernspørsmål, veilede om personvernregleverket, gi råd om praktisering av regelverket, kontrollere at regelverket etterleves. Kan blant annet kreve retting av regelfeil eller ilegge gebyr ved alvorlige regelbrudd.

Grunnleggende Prinsipper

#1

Når du behandler opplysninger om andre, så skal det skje etter visse prinsipper. Lovlighet, rettferdighet og åpenhet.

- Du må ha lov til å behandle opplysningene,
- du må ha være åpen om og informere godt om hva du gjør,
- du må gjøre det du lover og opptre ærlig og redelig.

Lovlighet kan innebære,

- at du må be om samtykke til å samle inn og bruke personopplysninger,
- at du måp finnes en lov eller forskrift som gir det anledning til å bruke opplysningene,
- at du må behandle visse opplysninger for å overholde en bindende avtale,
- at behandlingen er viktig og ikke altfor inngripende.

#2

Formålsbergrensing

- du må ha en spesifikk, tydelig og rimelig grunn til å samle inn og bruke opplysningene
- du kan ikke uten videre benytte opplysningene til noe helt annet enn hva de i utgangspunktet ble innsamlet for

Dataminimering

- du skal ikke samle inn flere opplysninger enn hva du strent tatt har behov for
- alle opplysningene du samler inn må være relevante for det du har tenkt å bruke dem til

#3

Riktighet

- opplysningene må være korrekte og oppdaterte
- ukorrekte eller utdaterte opplysninger må rettes eller slettes

Lagringsbegrensing

- Opplysningene skal ikke lagres lengre enn nødvendig
- Når du ikke lenger har bruk for opplysningene, skal de enten slettes eller anonymiseres
- Opplysningene kan oppbevares lenger dersom du har en rettslig plikt til å ta vare på dem

#4

Informasjonssikkerhet

- Opplysningene skal være tilstrekkelig sikret mot uautorisert tilgang, endring letting, skade eller ødeleggelse.

Ansvarliggjøring

- Du må kunne dokumentere at du overholde de nevnte prinsippene og reglene i GDPR.

Rettigheter – hva du har krav på?

Rettighetene gjelder for de registrerte, for eksempel alle studenter ved NTNU. Rettighetene skal ivaretas av de behandlingsansvarlige, for eksempel NTNU, Instagram gram, Snapchat, osv.

De viktigste rettighetene

- Informasjon
- Innsyn
- Retting
- Sletting
- Begrensning
- Dataportabilitet

Særskilte rettigheter ved hel-automatiserte avgjørelser og profilering. Algoritmer som fatter viktige vedtak, for eksempel skatteetaten, bank og finans.

Plikter – informasjonssikkerhet

Gjelder for behandlingsansvarlig og databehandler.

Definisjon av informasjonssikkerhet

- Konfidensialitet
- Integritet
- Tilgjengelighet
- Robusthet

Krav til informasjonssikkerhet avhenger av risiko, iverksette tiltak for å refusere risiko til et akseptabelt nivå.

Ledelsessystem, informasjonssikkerhet

Toppledelsen er ansvarlig for informasjonssikkerheten. Ved NTNU vil dette være rektor eller universitetsdirektøren i kommunen vil det typisk være rådmannen. I en privat bedrift vil det vanligvis være daglig leder. Toppledelsen skal stille krav til og følge opp arbeidet med informasjonssikkerhet.

De tre delene

1 Styrende del – virkeområde, mål, strategi, akseptkriterier, organisering, ledelse gjennomgang.

2 Gjennomførende del – Planer og rutiner for det partiske sikkerhetsarbeidet, arbeidsverktøy for eksempel maler for gjennomføring av risikovurdering.

3 Kontrollerende del – revisjoner og hendelseshåndtering

Personvern- og sikkerhetsfunksjonalitet integreres i digitale løsninger

•Krav i GDPR om at innebygd personvern og informasjonssikkerhet vektlegges ved systemutvikling eller anskaffelser

• Hvilke typer personvern og sikkerhetsfunksjonalitet kan bygges inn i digitale løsninger?

• Noen eksempler og begrensninger på hvilke opplysninger som kan registreres og pseudonymisering av opplysninger (der hvor det er aktuelt) o
gmodul for innsyn i egne opplysninger og automatisk sletting etter en viss tid og krypteringsmuligheter.