***Notater til 112***

Innholdsfortegnelse

[Funksjonelle og ikke-funksjonelle krav 2](#_Toc136939416)

[Brukerhistorier 3](#_Toc136939417)

[Akseptansekriterier 3](#_Toc136939418)

[Arbeidsoppgaver 4](#_Toc136939419)

[Minimum Viable Product (MVP) 6](#_Toc136939420)

[Maven 7](#_Toc136939421)

[CI og CD 7](#_Toc136939422)

[Prosjekt og arbeidsmetodikk 7](#_Toc136939423)

[Working, directory, staging area, local repository 8](#_Toc136939424)

[Versjonskontroll: 9](#_Toc136939425)

[Branching 9](#_Toc136939426)

[Programvarekvalitet 9](#_Toc136939427)

[Model-View-Controller (MVC) 10](#_Toc136939428)

[Testing 11](#_Toc136939429)

[Refaktorisering og testdrevet utvikling 11](#_Toc136939430)

[Immaterielle rettigheter 12](#_Toc136939431)

[Personvern (GDPR) 12](#_Toc136939432)

[Gode prinsipper for å skrive gjenbrukbar, vedlikeholdbar kode: 14](#_Toc136939433)

[SOLID 14](#_Toc136939434)

[GRASP 15](#_Toc136939435)

[KISS (keep it simple) 15](#_Toc136939436)

[DRY (don’t repeat yourself) 15](#_Toc136939437)

# Funksjonelle og ikke-funksjonelle krav

* **Funksjonelle og ikke-funksjonelle krav – «kan importere fra CSV» vs. «skal være robust mot feil i input-filer»**
* *Funksjonelle krav* beskriver hva systemet skal gjøre (men kan også beskrive hva systemet ikke skal gjøre). *Ikke-funksjonelle krav* definerer hvordan systemet skal innfri de funksjonelle kravene.
* **Funksjonelle:** Et funksjonskrav er en erklæring om hvordan et system må oppføre seg. Den definerer hva systemet skal gjøre for å møte brukerens behov eller forventninger. Funksjonelle krav kan betraktes som funksjoner som brukeren oppdager. De er forskjellige fra ikke-funksjonelle krav, som definerer hvordan systemet skal fungere internt (f.eks. ytelse, sikkerhet osv.).
* **Ikke-funksjonelle:** Ikke-funksjonelle krav (NFR) er begrensningene eller kravene som stilles til systemet. De spesifiserer kvalitetsattributtet til programvaren. Ikke-funksjonelle krav omhandler problemer som skalerbarhet, vedlikeholdbarhet, ytelse, portabilitet, sikkerhet, pålitelighet og mange flere. Ikke-funksjonelle krav tar opp viktige kvalitetsproblemer for programvaresystemer.

Ikke-funksjonelle krav kan deles inn i to kategorier:

1. **Kvalitetsattributter:** Dette er egenskapene til systemet som bestemmer dets generelle kvalitet. Eksempler på kvalitetsattributter inkluderer sikkerhet, ytelse og brukervennlighet.
2. **Begrensninger:** Dette er begrensningene som er pålagt systemet. Eksempler på begrensninger inkluderer tid, ressurser og miljø.

Funksjonelle krav, som navnet antyder, beskriv funksjonene til systemet som skal designes. Det er en beskrivelse av hva systemet skal være og hvordan det vil fungere for å tilfredsstille brukerbehov. De gir en klar beskrivelse av hvordan systemet skal svare på en bestemt kommando, funksjonene og hva brukerne forventer.

Ikke-funksjonelle krav forklare begrensningene og begrensningene til systemet som skal designes. Disse kravene har ingen innvirkning på funksjonaliteten til applikasjonen. Videre er det en vanlig praksis med å underklassifisere de ikke-funksjonelle kravene i ulike kategorier som:

* Brukergrensesnitt
* Pålitelighet
* Sikkerhet
* Ytelse
* Vedlikehold
* Standarder

# Brukerhistorier

* **Brukerhistorier – «As a *goat* I want *more bushes and shrubs* so that I can *ruminate about the quality of my code*.»**

Brukerhistorier er korte og enkle beskrivelser av hva en sluttbruker ønsker å oppnå ved å bruke et produkt eller en tjeneste. Funksjonaliteten beskrives i brukerens perspektiv og lar utviklingsteamet finne en teknisk løsning. Brukerhistoriene kan betraktes som byggeklosser som sammen danner en epic, et hovedelement i programvaren, som i sin tur utformer et tema eller en hovedfunksjon.

Brukerhistorier svarer på “Hvem”, “Hva”- og “Hvorfor”-spørsmål og følger et fast format, vanligvis: «Som en [rolle], ønsker jeg [noe/å gjøre noe], slik at jeg [oppnår en gevinst]». Et eksempel på en brukerhistorie kan være, “Som konsulent i Metier ønsker jeg å dele mine erfaringer slik at jeg bidrar til å øke kunnskapen om digitalisering”.

# Akseptansekriterier

* **Akseptansekriterier–** «Gitt X, så skjer Y», testbart (akseptansetesting) og klart definert – brukerhistorie handler om behov, akseptansekriterie handler om å definere hvordan vi måler at vi når behovet. Når akseptansekriteriene er oppnådd, er vi ferdig med brukerhistorien

**Hva er et akseptansekriterie?**

Et Akseptansekriterie er en tilstand programvaren må tilfredsstille for å bli akseptert av en bruker, kunde, eller en integrasjon.

Det brukes til å definere funksjonaliteten til en oppgave (normalt user story) for å tydeliggjøre forventet resultat på både funksjonelle og ikke-funksjonelle krav. Dette bør være en del av *Definition of Done*.

**Når defineres det?**

En vanlig felle er å begynne med definisjonen etter utviklingen er påstartet. Utfallet av dette blir normalt å bekrefte funksjonaliteten som er laget enn å ta hensyn til hva kunden eller brukeren faktisk trenger og ønsker.

Ved å definere akseptansekriteriene i forkant er vi sikre på å ta stilling til selve problemstillingen og hvorfor behovet er der.

**Hvordan komme i gang?**

Det er viktig å tenke *hva*, ikke *hvordan*. For å skrive gode akseptansekriterier skal vi dekke behovet til brukeren, ikke hvordan utviklerene skal lage løsningen.

Eksempelvis er dette et godt akseptansekriterie:  
 *Bruker kan godkjenne eller avvise en faktura*.  
Dette eksempelet er ikke et godt kriterie da det legger fokus på det tekniske:  
 *Brukeren kan klikke på en sjekkboks for å godkjenne en faktura.*

**Eksempler**

Brukerhistorie: Som en *student* vil jeg *se semesterkostnaden* så *jeg vet hvor mye balansen er på*.

Akseptansekriterier for dette kan være:

* Semesterkostnaden vises
* Semesterkostnaden kalkuleres
* Total balanse over alle semestre vises
* Ingen semesterkostnad vises hvis en ukjent innlogging er gitt

Brukerhistorie: Som en *bruker* vil jeg *logge inn på systemet* så *jeg kan benytte meg av [tjenesten].*

Akseptansekriterier kan være noe som dette:

* En vellykket innlogging skal navigere brukeren til hjemmesiden med en velkomstmelding (funksjonelt krav)
* En velkomstmelding skal vises øverst til høyre på hjemmesiden (ikke-funksjonelt krav)
* Hjemmesiden skal lastes i løpet av 1 sekund (ytelseskrav)

# Arbeidsoppgaver

* Arbeidsoppgaver – Hva må gjøres for å oppfylle akseptansekriteriene og kunne gjøre brukerhistorien til virkelighet?

**Eksempel: Brukerhistorier, Akseptansekriterier og arbeidsoppgaver**

1. **Som student ønsker jeg å kunne se oversikt over alle forelesningene i fagene jeg tar slik at jeg kan vite når og hvor jeg skal møte til forelesning.**

**Akseptansekriterier:**

* student kan se all informasjon om minst to fag
* student kan se når hver forelesning begynner og slutter
* student kan se hvor hver forelesning skal være

**Arbeidsoppgaver:**

* brukergrensesnitt for å vise informasjon om fag
* modellere fag og student, samt persistens av dette
* forretningslogikk for å hente ut informasjon om lagrede studenter og fag
* forretningslogikk for å legge inn informasjon om studenter og fag (ikke påkrevd, kan fakes i test)

1. **Som foreleser ønsker jeg å kunne sende beskjeder til alle studentene som tar faget mitt slak at studentene kan få viktig informasjon.**

**Akseptansekriterier:**

* foreleser får sende/gi beskjed til alle studentene
* studenter mottar beskjed
* studenter som ikke tar faget får ikke beskjed

**Arbeidsoppgaver:**

* modellere foreleser og studenter
* brukergrensesnitt for å kunne lage en beskjed.
* forretningslogikk for å sende melding til studentene, f.eks epost
* forretningslogikk for å vite hvem som skal ha beskjeden og ikke

1. **Som student ønsker jeg å kunne se hvilke obligatoriske oppgaver som er godkjent slik at jeg vet om jeg får gå opp til eksamen.**

**Akseptansekriterier:**

* student kan se status på obligatoriske oppgaver
* student kan se om oppgavene er godkjent eller ikke
* student kan se om alle de obligatoriske oppgavene er godkjent
* student kan se om ingen av de obligatoriske oppgavene ikke er godkjent

**Arbeidsoppgaver:**

* brukergrensesnitt for å vise status på oppgaver i et fag
* modellere fag, student og oppgaver i et fag
* forretningslogikk for å kunne hente oppgaver i et fag
* brukergrensesnitt som viser om oppgaven er godkjent eller ikke.

# Minimum Viable Product (MVP)

* Minimum Viable Product– Det minste som må være på plass for at vi skal kunne gå i produksjon. Viktige krav kan gjerne løses manuelt eller på andre måter enn å lage kode for det – det ABSOLUTTE minimum, ikke ferdig system.
* MVP skal ha med de delene av prosjektet der det er høyest risiko (både på det tekniske planet med også funksjonalitetsmessig) og det er viktig å få tilbakemelding fra brukere.

**Hensikt:**

* Hensikten med MVP er å få tilbakemeldinger fra brukerne. Da må produktet inneholde nok til at brukeren kan bruke det

**Hvordan oppgavene burde bli prioritert i et prosjekt for å lykkes:**

* De oppgavene som har høyest risiko skal prioriteres først. Risiko kan være begrunnet i flere ting, f.eks teknisk vanskelig å få til, virksomhetskritisk funksjonalitet.
* Ved å prioritere teknisk kompliserte oppgaver tidlig i kombinasjon med tidlig produksjonssetting, får vanskelige tekniske oppgaver tid til å modnes slik at riktig løsning kan velges før det er for sent.
* Tidlig erfaring med hvordan brukeren er vil også hjelpe med å avgjøre om riktig løsning er valgt.
* Virksomhetskritiske oppgaver skal også velges først, siden det er denne funksjonaliteten som er mest verdt for brukere eller for produkteiere.
* Ellers bør oppgaver inneholde hele stacken, slik at funksjonaliteten kan utvikles i sin helhet med en gang (så kommer reell bruk og tilbakemelding slik at justeringer kan gjøres.)
* Oppgaver som ikke gir reell verdi for kunden skal ikke prioriteres (kråkesølv)

**Eksempel:**

* Ha et spillbrett
* Vise en brikke
* Kunne flytte en brikke
* Flagg på brett
* Registrere at en robot har vært innom et flagg
* Vinne spill

# Maven

Maven er et populært verktøy for byggautomatisering og prosjektstyring, spesielt brukt i Java-baserte prosjekter. Det forenkler prosessen med å bygge, teste og distribuere programvare ved å tilby en standard struktur og en rekke konvensjoner.

* POM
  + Bruker en XML-fil kalt POM (Project Object Model) til å definere og konfiguere prosjekter.
  + POM-en inneholder prosjektmetadata, avhengigheter, bygginstruksjoner og annen konfigurasjonsinformasjon. Den fungerer som den sentrale konfigurasjonsfilen for et Maven-prosjekt
* Plugins
  + Maven støtter et bredt spekter av plugins som utvider funksjonaliteten. Plugins gir ekstra mål og kan brukes til å utføre ulike oppgaver, som kjøring av tester, generering av dokumentasjon, kjøring av statisk kodeanalyse eller distribusjon av artefakter.
  + Plugins konfigueres vanligvis i POM-filen.

# CI og CD

**Continuous Integration (CI):**

* Er en praksis der utviklere integrerer og bygger kodehyppig. Det innebærer at hver gang en utvikler fullfører en kodeendring, blir den automatisk bygget og integrert med resten av koden i et delt repository.
* Formålet med CI er å oppdage og rette feil tidlig, samt sikre at den samlede koden fungerer godt sammen.

**Continous Deployment/Delivery (CD)**

* Refererer til praksisen med å automatisere og kontimuerlig utplassere programvare til produksjonsmiljøet etter CI.
* Continous Deployment innebærer automatisk utplassering av hver godkjent kodeendring til produksjonen
* Continous Delivery betyr at programvaren er klar for utplassering når som helst, men selve utplasseringen blir manuelt aktivert.

Både CI og CD er en del av DevOps-filosofien, som fokuserer på automatisering, samarbeid og rask utvikling av programvare. Disse praksisene bidrar til å redusere risiko, forbedre kvaliteten på programvaren og muliggjør hyppere og mer pålitelig utrulling.

# Prosjekt og arbeidsmetodikk

**The Agile Manifesto (smidig programvareutvikling)**

* *Personer og samspill* fremfor prosesser og verktøy
* *Programvare som virker* fremfor omfattende dokumentasjon
* *Samarbeid med kunden* fremfor kontraktsforhandlinger
* *Å reagere på endringer* fremfor å følge en plan

**Extreme Programming (XP)**

* XP er bygget på verdier, prinsipper og praksis
* Målet er å la små til mellomstore team produsere høykvalitets programvare og tilpasse seg skiftende og skiftende krav
* Ekstrem programmering handler i et nøtteskall om god praksis tatt til det ekstreme. Siden parprogrammering er bra, la oss gjøre det hele tiden. Siden tidlig testing er bra, la oss teste før produksjonskoden i det hele tatt er skrevet.

**Scrum**

* Metodikken er designet for team med 3 til 10 medlemmer
* Bryter ned arbeidet i må (sprintkøer) og skal utføres innen tidsbokser kalt sprinter
* Sprintene har som regel varighet ikke lengre enn en måned oftest to uker.
* Scrumteamet vurderer fremgangen i daglige møter på 15 min eller mindre kalt daglige scrums.
* **Et bilde som inneholder tekst, skjermbilde, display, nummer

  Automatisk generert beskrivelse**På slutten av sprinten har teamet ytterligere to møter hvor man viser utført arbeid for å få tilbakemeldinger, og sprintevaluering hvor teamet kan reflektere og forbedre seg.

**Kanban**

* Er en måte for å beregne igangsatt arbeid i et produksjonsanlegg.
* En ny jobb må vente hvis antall påbegynte jobber har nådd en grense.
* Så snart en jobb er ferdig, kan neste ventede jobb settes i gang.

# Working, directory, staging area, local repository

***Working directory***:

* Er arbeidsområde ditt lokalt på disk der du har sjekket ut en gitt commit fra et repository.

***Staging area***:

* Er der git får oversikt over alle endringene som er gjort og som skal legges til i git.
* Basert på disse endringene beregner git en hash som er summen av endringene som gjøres.
* Når du gjør git add <filer> blir endringene i filene tatt med fra working diractory til stating area.

***Local repository:***

* Når du så gjør git commit vil alle endringene som er lagt til i staging area legges til i ditt lokale repository, hvor alle endringene du har gjort også er lagt til.

# Versjonskontroll:

* Gjør at du sporer endringene du har gjort. Du får oversikt over hvilke endringer som er gjort, når, og av hvem
* Du får backup hvis du har arbeidet på ulike enheter, du er garantert at det er samme versjon av koden så lenge alle har samme commit.

# Branching

**Trunk-based**

* Alle committer rett på master-branch

**Branch-based**

* Egne feature-branches som merges inn til master jevnlig

# Programvarekvalitet

* **Pålitelighet –** gir *korrekt* resultat (gyldig input gir gyldig resultat) – («jeg er ikke sikker på om dette vil funke» / «jeg må sjekke resultatet for hånd for å stole på det» → programmet er upålitelig)
* **Robusthet –** håndterer feil, ugyldig input oguforutsette situasjoner – både brukerfeil, programmeringsfeil og eksterne feil (hardware, natur). («editoren kræsjer av og til når jeg prøver å lagre» / «VIKTIG! ikke trykk på knappen mens det røde lyset lyser» → programmet er lite robust)
* **Fleksibilitet –** kan tilpasses / brukes annerledes når kravene endrer seg («jeg implementerte spill-logikken i Excel» → regneark er veldig fleksible!)
* **Vedlikeholdbarhet –** hvor lett det er å finne og rette feil, eller endre oppførselen («hver gang jeg fikser en bug kommer det to nye» – koden er *skjør* slik at én endring/bugfix lett fører til feil andre steder)
* **Sikkerhet –** nært knyttet til punktene over (er programmet så «fleksibelt» at det kan misbrukes? robusthetsfeil er ofte sikkerhetsproblemer, i det minste i form at *denial of service*)
* **Effektivitet –** «får jeg svar raskt nok til at det er vits i å vente?» / «denne appen spiser batteriet» – responstid og jevn respons er ofte viktig for brukere, ok å vente på svar så lenge du vet at det skjer noe
* ***Kodekvalitet* –** formattering, lesbarhet/navn osv

**Programmering**

* **Abstraksjon –** interfaces, klasser, metoder, generics – forenkle, se bort fra uvesentlige detaljer, fokusere på de essensielle detaljene
  + Abstraksjon i programmering er like viktig som kommunikasjon i teamarbeid – det gir deg et *språk* som gjør at du kan kommunisere tydelig med maskinen og med andre programmører!
* **Modellering** 
  + Diagrammer (klassediagrammer)
  + Poenget med modellering er å spesifisere, dokumentere/kommunisere, og/eller å utforske (få oversikt over et system, eller få inntrykk av hvordan det blir etter en endring.)
  + Diagrammer skal være abstrakte, altså fokusere på de tingene som er viktig i den situasjonen du ruker diagrammet. Unødvendige detaljer gjør diagrammet vanskeligere å forstå og dermed mindre informativt.
  + Relasjoner tegnes med streker mellom boksene – bruk pil for arv ( A -> B for A extends B) og pil med striplete linje for implementering (A ---> B for A implements B)
* **Seperation of concerns –** skill forskjellige aspekter av funksjonaliteten fra hverandre. F.eks., *model-view-controller* sørger for å skille modellen (informasjonen som prosesseres) fra viewet (hvordan det vises til brukeren) – kan f.eks. gjøre det enklere å lage flerspillerklient i et spill eller gjøre programvaren tilgjengelig for blinde eller på forskjellig plattformer eller forskjellige språk

# Model-View-Controller (MVC)

MVC er et designmønster som brukes til å organisere og strukturere programvareapplikasjoner.

**Model**

* Modellen representerer applikasjonens data og logikk.
* Den håndterer datahenting, oppdatering og validering
* Modellen er uavhengig av visningen og kontrolleren og gir grensesnittet for å hente og manipulere dataene.

**View**

* Visningen er ansvarlig for presentasjonen av dataen til brukeren.
* Den viser informasjonen fra modellen og mottar brukerens inndata.
* Visningen er passiv og reagerer på endringer i modellen ved hjelp av observatør– eller hendelsesmekanismer

**Controller**

* Kontrolleren er koblingen mellom modellen og visningen.
* Den håndterer brukerens interaksjon, mottar inndata fra visningen og oppdaterer modellen i henhold til handlingene
* Kontrolleren er ansvarlig for å tolke brukerens handlinger og oppdatere modellen og visningen deretter.

MVC-mønsteret bidrar til å oppnå en god seperasjon av ansvar, der hver komponent har sitt eget område og oppgaver. Dette gjør det enklere å vedlikeholde, teste og videreutvikle applikasjonen. Endringer i en komponent har minimal påvirkning på de andre komponentene, noe som gir fleksibilitet og gjenbrukbarhet.

# Testing

**JUnit**

Populært rammeverk for automatisert testing i Java. Det brukes til å skrive og kjøre enhetstester, som er tester som sjekker funksjonaliteten til individuelle enheter (som metoder, klasser eller komponenter)

**Mocking**

* En teknikk som brukes i enhetstesting for å erstatte eksterne avhengigheter med simulerte objekter kalt mocks.
* Målet er å isolere enheten som testes ved å simulere responsene fra eksterne avhengigheter og fokusere på å teste selve enheten uten å påvirke andre deler av systemet.
* Spesielt når det er utfordrende eller upraktisk å arbeide med de virkelige eksterne avhengighetene i testmiljøet.
* Det gir fleksibel og kontrollerbar måte å teste enheter og håndtere avhengigheter på, og bidrar til å forbedre enhetstestenes kvalitet og effektivitet.

**Headless**

* Refererer til utførelsen av automatiserte tester uten bruk av et grafisk brukergrensesnitt (GUI).
* Testene kjøres i stedenfor i en ikke-visuell kontekst, der programvaren testes gjennom API-er eller kommandolinjesnitt
* Spesielt egnet for backend-testing, der man ønsker å teste server-side funksjonalitet og API-er uten å måtte håndtere det visuelle aspektet av programvare. Det gjør det mulig å validere datahåndtering, logikk og respons fra serveren uten behov for et GUI
* Nyttig i situasjoner der man ønsker å fokusere på funksjonaliteten, ytelsen eller sikkerheten til programvaren uten å bekymre seg for det grafiske brukergrensesnittet
* Gir en effektiv og pålitelig metode for å automatisere tester og bidrar til å oppnå høy kvalitet i stabilitet i programvaren.

# Refaktorisering og testdrevet utvikling

***Testdrevet utvikling***:

* Er når man skriver testene før man lager selve programmet. Man skriver da kode slik at testene skal fungere.
* Testing fører til at koden blir brukt med en gang (må tenke brukeropplevelse på koden),

**Refaktorisering**:

* Når testen(e) kjører, er det rom for å forbedre kvaliteten på koden, dette kalles refaktorisering. Refaktorisering er å endre struktur i koden uten å endre resultatet utead. Alle testene skal fortsatt kjøre.
* Refaktorisering gjøres for å heve kvaliteten på koden, øke lesbarhet og øke vedlikeholdbarheten på koden, slik at det blir lettere å jobbe med koden over tid.
* Refaktorisering øker lesbarhet og vedlikeholdbarhet, kvalitetspåvirkning, trygghet for å gjøre endringer.

# Immaterielle rettigheter

Immaterielle rettigheter, også kjent som intellektuell eiendom, omfatter opphavsrett, varemerker og patenter.

**Opphavsrett (copyright)**

* Beskytter åndsverk som tekster, musikk, programvare, bilder og kunst. Beskyttelsen varer lenge, for eksempel 70 år etter forfatters død i Norge
* Kopiering eller gjenbruk av andres åndsverk uten tillatelse er straffbart
* Gjenbruk av Nintendo-spillgrafikk og karakterer som Mario vil bli rammet av opphavsretten.
* Bruk av biblioteker i programvare utgjør også kopiering av åndsverk og kan kreve tillatelse.

**Varemerker (trademark)**

* Handler om å beskytte produktidentitet, som bedriftsnavn, logoer, produktnavn og utforming.
* De må vanligvis registreres og forsvares for å unngå forvirring blant forbrukerne.
* Bruk av varemerker i domenenavn, logoer og programvaredesign er relevant.
* Hvis spillet «Kurt-Mario Kart» hadde blitt lansert, ville det sannsynligvis ført til et varemerkesøksmål fra Nintendo

**Pantenter**

* Beskytter ideer og oppfinnelser
* Etter søknad får man en tidsbegrenset enerett til å utnytte en spesifikk oppfinnelse.
* Patenter er mindre relevant for programvare i Norge, men er brukt mye i USA.
* Komprimeringsalgoritmen i GIF-bilder var patentert frem til 2004, noe som krevde lisens for implementering

Som programutvikler er man ansvarlig for å følge loven, selv om man ikke kan skylde på sjefen eller foreleseren for brudd på loven. Som student er man også ansvarlig for å følge universitetets regler om plagiat, som er separate fra opphavsretten og kan ha strenge konsekvenser.

# Personvern (GDPR)

* GDPR (General Data Protection Regulation) er den viktigste tingen å kjenne til. I Norge inngår GDPR i personopplysningsloven og Datatilsynet har ansvar for å følge opp regelbrudd.
* Generelle prinsipper for GDPR:
  + Du eier dine egne data
  + Gjelder så lenger den som samler opplysningene eller prosesserer dem, eller de opplysningene handler om er basert i EU/EØS
  + Persondata kan bare behandles hvis man har en gyldig grunn – f.eks. ved samtykke, ifm. med avtaler/kundeforhold, lovkrav, osv. Dvs. du må alltid ha en legitim grunn for å samle og behandle personopplysninger, og du kan ikke lagre opplysninger du ikke har bruk for. Hensynet/interessene til personen det gjelder veier (som regel) tyngre enn bedriftens interesser («trenger jeg å lagre dette for å gjøre applikasjonen bedre for personen? eller er dette bare for å gjøre applikasjonen bedre for reklame-kundene våre?»)
  + Samtykke må være informert, og man må eksplisitt velge å samtykke, man skal kunne trekke samtykke like enkelt som man ga det
  + Man har (som regel) rett til å vite hvilke opplysninger som er lagret, og få dem utlevert, og til å opplysningene korrigert eller slettet
  + Den som samler/behandler opplysningene er pålagt å ivarta sikkerhert og behandle opplysninger i henhold til reguleringen. Det er rimelig strenge straffer (store bøter for selskaper, evt. også politisak)
* Generelle krav, bla. for oss som programutviklere:
  + personopplysninger skal kun oppbevares så lenge det er nødvendig
  + personopplysninger skal kun brukes til sitt opprinnelige formål (det vil si at hvis du har samlet inn informasjon får du ikke lov å bruke informasjonen til noe annet enn det du først har bedt om lov til/informert om)
  + løsninger vi lager skal ha innebygd personvern
  + løsninger skal oppfylle krav til dokumentasjon
  + alle som behandler personopplysninger kalles databehandlere og skal ha databehandleravtale
  + en databehandleravtale er en avtale som regulerer hensikt med behandling av informasjon, varighet, formål, rettigheter og plikter osv. Skal finnes i alle ledd som behandler informasjon (alle underleverandører)
* Selv om det er databehandleren som har hovedansvaret for å ivareta personvernet, har du som programmere også et selvstendig moralsk ansvar for å ikke bidra til uetiske eller ulovlige systemer

# Gode prinsipper for å skrive gjenbrukbar, vedlikeholdbar kode:

## SOLID

Good practices for programming.

**Single Responsibility:** “A class should have only one reason to change” - Uncle Bob

I stedet for å ha alle metodene I en kjøretøysklasse kan du dele den opp slik at det er motor og hjul osv. Klasse også. Så hver av klassene har sitt eget ansvar.

**Open/Closed:** “A class should be open for extension, but closed for modification.” Promotes re-utilization

Når du deler ansvar for en klasse, bør du gjøre det på en måte som gjør at atferd kan utvides/erstattes.

Kjøretøysklasse, det kan være mange forskjellige typer kjørety, men de implementerer alle kjøretøyklassene. Dette gjør det slik at alle de individuelle kjøretøysklassene ikke trenger å ha metoden «akselerasjon», bare hovedvognklassen må ha det.

**Liskov substitution:** Definition: *You should be able to change an instance using a sub-type and your code should still work.*

Hvis kjøretøysklassen har en metode kalt shift() og elbilklassen utvider kjøretøysklassen, vil den ikke kunne implementere denne metoden siden elbiler ikke har shift. Dette vil da resultere i at denne metoden enten er tom i elbilklassen eller at den kaster et nytt unntak. I begge disse tilfellene er det et brudd på Liskovs prinsipp.

En klasse skal ikke implementere en klasse der noen av metodene ikke vil bli implementert for klassen.

**Interface segregation:** “Better to have too many interfaces than having too little”

Gjør det mulig å dele ansvaret til en klasse uten LSP-brudd.

Dette kan bidra til å fikse brudd på Liskovs prinsipp. Hvis vi deler kjøretøysklassen i to undertyper, forbrenning og elektrisk, og deretter utvidet disse til de andre klassene, vil vi ha unngått brudd på Liskovs prinsipp.

**Dependency inversion:** One should depend on abstraction and not concrete instances.

Kjøretøysklasse som er avhengig av en dekk-klasse i stedet for å ha dekk som en instans i kjøretøysklassen

## GRASP

* For å fremme høy sammenheng
* Minimere avhengighet
* Kontroller-klasse som styrer og koordinerer interaksjon mellom andre klasser.
* Sikre at hver klasse og objekt ha definerte ansvarsområder
* Oppnår modulær, fleksibel og vedlikeholdbar kodebase

## KISS (keep it simple)

* De fleste systemene fungerer best hvis de holdes enkle I stedet for kompliserte.

## DRY (don’t repeat yourself)

* Redusere repetisjon av informasjon som sannsynligvis vil endres, erstatte den med abstraksjoner som er mindre sannsynlig å endre.