Produksjonsmiljø	Å sette noe i produksjon. Betyr at applikasjonen er tilgjengelig for sluttbrukeren. Før den kommer hit har den typisk vært gjennom et utviklingsmiljø og stagingmiljø (som skal være så likt som mulig som produksjonsmiljøet) for å oppdage feil.
Domenekunnskap	Kunnskap som er spesialisert innenfor et gitt fagområde. «Agn» er domenekunnskap for folk som fisker og kan bli tatt for gitt av dem som har kjennskap til den, mens utenforstående ikke har samme kontekst for forståelse
Teknisk gjeld	«Dette skal jeg rydde opp i senere, men det fungerer nå». Raske løsninger som skaper problemer lenger ut i utviklingsfasen og livsløpet til en applikasjon.
Engineering	Engineering omhandler hvordan oppnå resultater av påkrevd kvalitet innenfor tidsplanen og budsjett.
Diagrammer	Diagrammer brukes for å visualisere sammenhenger og avhengigheter i et programvaresystem. Eksempler: UML, Flytdiagram, Aktivitetsdiagarm, Tilstandsdiagram. Sekvensdiagram
Prosessmodeller	Hvilke faser og aktiviteter består prosjektet av, hvilke avhengigheter er det mellom disse og når gjennomføres de? Dette må tas hensyn til når man skal velge en prosessmodell. Prosessmodeller er et spekter og elementer brukes på tvers av modelltyper. Forskjellige prosessmodeller passer til ulike typer programvare som skal utvikles, og tilpasses prosjektet som skal gjennomføres. Resultatet er det viktige, ikke hvorvidt en spesifikk modell har blitt brukt. **OppsummeringsPP:** Planbaserte modeller (typisk waterfall) og inkrementelle og iterative modeller (agile etc) I planbaserte modeller legges planen mest mulig på forhånd, mens inkrementelle og iterative metoder bygges kontinuerlig og revideres etter hvert som virkeligheten endrer seg. Ulike krav gir forskjellige metoder. Forskjellige deler av prosjektet kan benytte ulik grad av metodene. Modellene er utgangspunkt og vi henter inn det vi trenger.
Faser i en prosessmodell	- Avdekking av krav - Spesifisering av krav - Validering av krav - Beskrivelse av systemet - Utvikling - Testing - Integrasjon - Drift - Vedlikehold
Planbasert prosessmodell	- Mest mulig/alt planlegges før de første linjene med kode skrives - Programvare implementeres etter en overordnet, ferdig oppsatt plan - Rammebetingelsene og grensesnittet for koden er satt før utviklingsfasen - Gjerne rigide akseptansekriterier

	- Waterfall er en slik metode - Benyttes ofte hvor det er kritisk at alt skal fungere, som innenfor helsesektor eller luftfart.
Inkrementell/iterativ	- Hver fase kan opptre samtidig, der forholdene mellom dem implementeres og evalueres fortløpende - Programvare bygges inkrementelt - Deler av et prosjekt kan være planbasert, mens andre deler følger andre modeller - Utvikleren veksler mellom analyse, design, implementering og testing kontinuerlig - Agile ligger ca her. **OppsummeringPP:** Programvareutvikling går i sykluser, der krav, utvikling, testing går fortløpende og ofte samtidig underveis i prosjektgjennomførelsen. - Nærmere slik vi løser problemer selv - Realisering av at ikke alle deler av et system er like viktig. - Håndterer endringer i prosjektbehov mer fleksibelt (krav, avhengigheter etc.) - Kan levere en operativ versjon av prosjektet så tidlig som mulig. - Tydeligere for prosjekteier hva som er tilstanden på prosjektet fortløpende.
Integrasjon og konfigurasjon	- Forsøk på å gjenbruke eksisterende kode i størst mulig grad, og eventuelt utvide biblioteket med konfigurasjon for å oppfylle nye krav - Ny applikasjonsløsning «konfigureres» til å løse nytt problem i større grad - Kan være både planbasert og flytende - Sees ofte i sammenheng med rammeverk I stedet for å utvikle en applikasjon «fra bunnen av» finner vi noe som allerede gjør det vi skal-ish og konfigurerer denne appen til å løse problemet. Dette krever god kunnskap om rammeverket/applikasjonen som er nødvendig for å styre krav i riktig retning – ellers ender du opp med å bruke mye tid på å få til noe som applikasjonen ikke er lagd for Ofte bruker vi eksterne komponenter som kan passe inn under «integrasjon og konfigurering» som en del av et prosjekt, f.eks database eller søkemotor.
Waterfall (planbasert)	Hvert steg i prosessen kommer etter hverandre – etter at det første steget er gjennomført beveger man seg videre til neste steg. Det er fem steg i Waterfall 1. Utarbeiding av krav 2. System- og softwaredesign 3. Implementasjon og enhetstesting 4. Integrasjon- og systemtesting 5. Drift og vedlikehold Kun når man er ferdig med et steg går man videre til neste. Waterfallmodellen i seg selv er planbasert, men innenfor hvert nivå

	kan man benytte andre modeller, for eksempel kan man jobbe agilt på implementasjon og enhetstesting. De er låste, rigide prosesser som krever mye planlegging og jobb med spesifikasjoner. Den rigide formen binder hvert ledd til avgjørelsen i leddet foran og gjør at etterfølgende ledd må arbeide rundt begrensingene gitt fra foregående steg
Fordel Waterfall	- Modellen er enkel å bruke og å forstå - Den er enkel å administrere ettersom hver fase skal ha et spesifikt resultat - Bra for kritiske systemer som krever detaljert sikkerhets- og trygghetsanalyse av spesifikasjon og design av programvaren, som fly og medisinsk utstyr.
Ulempe Waterfall	- Det er ikke rom for refleksjon, er man ferdig med en fase så skal man over til neste fase - Det er vanskelig å gå tilbake og endre på ting - Fra et prosjekt er beskrevet som idé til det er gjennomført er det ikke sikkert at det er det samme problemet som skal løses lenger, ettersom ting endrer seg over tid. - Kunden vet ikke hva behovet er ved oppstart og hva som er det egentlige problemet senere i prosessen. - Under utvikling oppdages andre utfordringer eller behov som kan forenkle prosjektet vesentlig ved å justere på krav eller programvaredesign, som ikke er mulig i Waterfall uten å starte fra begynnelsen. - Krever god erfaring og trygg arkitektur for å unngå teknisk gjeld.
Inkrementell utvikling.	(Inkrementell og iterativ – brukes om hverandre og ofte beskriver de samme tingene avhengig av kontekst) I stedet for separate adskilte prosess-steg så foregår aktivitetene gjennom hele prosjektet – i varierende grad. Tidlig i prosjektet vil spesifikasjon og krav være en større andel enn senere, men spesifikasjonen er ikke «satt i stein». Passer bedre til måten vi selv løser problemer på – vi evaluerer ikke alle mulige måter å angripe problemet på, men finner en vi tror kan fungere – og tester den. Man er klar over at ikke alle deler av et programvaresystem er like viktig, og at et programvaresystem kan ha nytteverdi før det oppfyller alle kravene til det. En funksjonell versjon av produktet kan være operativt så tidlig som mulig.
Fordeler Inkrementell utvikling	- Gjør det mulig for alle involverte å se hvilken vei utviklingen går - Tydeliggjør hva som mangler og hvordan delene henger sammen med virkeligheten. - Det blir tydelig for kunde hvordan de enkelte delene av prosjektet er ment å fungere - Tilbakemelding fra produkteier kan integreres i systemet tidlig - Misforståelse på grunn av manglende domenekunnskap eller uklare beskrivelser avsløres tidlig.

Ulemper inkrementell utvikling	- Store organisasjoner kan ha interne begrensinger, slik som byråkrati og grensedragninger mellom avdelinger og team - Kan være vanskelig å håndtere oppgaver som krever byråkrati - for eksempel validering og godkjenning, mer sentral lovgivning der spesifikasjonene er låst og en spesifikk versjon av programvaren godkjennes før bruk - Prosessen er ikke synlig på samme måte – leveranser og kjørende kode blir viktig for å kunne måle fremgang.
Målbarhet for prosessmodeller	Prosessmodeller kan måles: «Alt planlagt» = 0 og «Anarki» = 10 0 ---------- 5 ---------- 10 - Waterfall ligger ved 0. - Inkrementell utvikling ligger ved 5 - Agile ligger nærmere 10.
Forretningssystemer (Visma, Lindbak POS, regnskap, salg, innkjøp, grunndatahåndtering)	- Iterativ planlegging - Uformell prosess - Uformell endring - Design og implementasjon om hverandre - Uformell kildekodekontroll - Utviklere utvikler egne tester - Uformell utrulling Det betyr ikke at du kan slurve med koden!
Driftkritiske systemer (Jernbanenettverkets sanntid, kraftnettverket i Norge, pakker blir sendt til riktig adresse, forholdet mellom drift og kunde)	- Litt planlegging - Formell prosess - Overordnet design up-front - Løpende implementasjon og design-review - (U)formell kildekodekontroll - Utviklere utvikler egne tester - Manuell testing i QA - Formell utrulling
Innebygde livskritiske systemer (Luftfart, medisinsk utstyr, atomkraftverk, der liv kan gå tapt)	- Full risikoanalyse - Full kravinnsamling - Komplett arkitektur og design - Løpende inspeksjon og review av kode - Utviklere utvikler egne tester - Test-team utvikler integrasjonstester - Formell utrulling
Prototype	En prototype er en testversjon av funksjonalitet, av produkter eller et konsept for å se hvordan og om de fungerer i praksis før vi utvikler den videre. Prototyper er bruk og kast, men nyttige ting kan tas med i prosjektet ved behov. Kan brukes ved kontraktinngåelse for å se om det «er noe slikt dere (kunde) tenker på». Prototyper brukes når vi… - Vil vite om noe fungerer før vi går videre med planen vår - Vil teste hvilken av løsningene våre som gir utbytte (typisk UX(???)) - Trenger bedre og mer nøyaktige krav til systemet - Vil ha et mer effektivt brukergrensesnitt - Trenger enklere vedlikehold (fordi vi får en bedre forståelse av problemet før den «ordentlige» koden skrives) - Ønsker raskere utvikling

Agile	Agile-modellen leverer resultater til kunde tidlig og ofte for å hele tiden vise hva som blir gjort, og man får tilbakemeldinger kontinuerlig av kunde. Kunden blir fornøyd fordi de er med på hele prosessen. - Lever fungerende software ofte (hver uke > hver måned) - Ønsk endringer i krav velkommen, selv sent i utviklingen. Agile/smidige prosesser bruker endringer til å skape konkurransefortrinn for kunden - Forretningssiden og utviklerne må jobbe sammen daglig gjennom hele prosjektet - Bygg prosjektet rundt motiverte personer. Gi dem miljøet og støtten de trenger og stol på at de får jobben gjort - Den meste effektive måten å formidle infomrasjon inn til og innad i et utviklingsteam er å snakke ansikt til ansikt - Fungerende programvare er det primære målet på fremdrift - Smidige metoder fremmer bærekraftig programvareutvikling. Sponsorer, utviklere og brukerne bør kunne opprettholde et jevnt tempo hele tiden. - Kontinuerlig fokus på fremragende teknisk kvalitet og godt design fremmer smidighet - Teknisk kvalitet og godt design betyr ikke rammeverk, språk eller «edge technology» - Teknisk kvalitet og godt design betyr kode som **overlever** rammeverk, språk eller «edge technology» - Enkelhet – kunsten å maksimere mengden arbeid som ikke blir gjort – er essensielt - Koden du skriver gjør deg til en programmerer, koden du sletter gjør deg til en bedre en, og koden du ikke trenger å skrive gjør deg til en god en. Fra video: - Agile er iterativ - Korte, rett på sak dokumenter i stedet for store dokumentasjoner - Time-boxed – vi setter av 3 uker og gjør det vi rekker. Rekker vi å implementere funksjon a, b og c på 2 uker, så legger vi på funksjon d også, siden vi har en uke igjen å jobbe på. - Ønsk endringer velkommen - For at Agile skal funke trenger vi ærlige, samarbeidsvillige utviklere, og engasjerte kunder
Agile Manifest	- **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
Agile Prinsippene	**1.** Vår høyeste prioritet er å tilfredsstille kunden gjennom tidlige og kontinuerlige leveranser av programvare som har verdi **2.** Ønsk endringer i krav velkommen, selv sent i utviklingen. Smidige prosesser bruker endringer til å skape konkurransefortrinn for kunden **3.** Lever fungerende programvare hyppig, med et par ukers til et par måneders mellomrom. Jo oftere, desto bedre

	4. Forretningssiden og utviklerne må arbeide sammen daglig gjennom hele prosjektet. **5.** Bygg prosjektet rundt motiverte personer. Gi dem miljøet og støtten de trenger, og stol på at de får jobben gjort. **6.** Den mest effektive måten å formidle informasjon inn til og innad i et utviklingsteam, er å snakke ansikt til ansikt. **7.** Fungerende programvare er det primære målet på fremdrift. **8.** Smidige metoder fremmer bærekraftig programvareutvikling. Sponsorene, utviklerne og brukerne bør kunne opprettholde et jevnt **9.** Kontinuerlig fokus på fremragende teknisk kvalitet og godt design fremmer smidighet. **10.** Enkelhet – kunsten å maksimere mengden arbeid som **ikke** blir gjort – er essensielt. **11.** De beste arkitekturer, krav og design vokser frem fra selvstyrte team. **12.** Med jevne mellomrom reflekterer teamet over hvordan det kan bli mer effektivt og så justerer det adferden sin deretter.

Utfordringer med Agile	- Kunnskap om smidig arkitektur er avgjørende
	- Automatisert testing er avgjørende
	- Løsningen kan være for kritisk til å ikke gjøre full analyse og planlegging
	- Kunde ønsker låst spesifikasjon, fast pris og fast deadline
	- Ved tvist om avtalt leveranseinnhold kan det være vanskelig å «holde på sitt» så det er avgjørende med gode avtaler og skriftlig kommunikasjon
	- Avgjørende med tillitt mellom partene
	- Forutsetter brennende engasjement hos alle medarbeidere
	- Man kan kun velge to av tre
	  o Billig, fort eller høy kvalitet
	  o Fast pris, fast leveransedato eller fast funksjonalitet

MVP – Minimum Viable Product	Minste versjon av produktet vårt som skaper verdi, slik at vi kan validere og få satt produktet i produksjon så tidlig som mulig. Kvalitet og verdi først.
Extreme Programming (XP)	Extreme Programming ligger under Agile-paraplyen. Software er for viktig for å bruke tid på tingene som ikke betyr noe. Vi vil ha verdi og resultater! Om man begynner fra bunn, hva må vi ha? - Kode. Har vi ikke et program som kjører og skaper verdi for en kunde, så har vi ikke gjort noe. Derfor lager vi MVP først. Vi programmerer også sammen, for man tenker bedre ved samarbeid enne alene. Derfor ansvaliggjøres også alle, og ikke én person. - Testing. Vi må vite når vi er ferdig, og testene våre forteller oss dette. Skriv tester først så når de blir godkjente så er vi klar over når vi er ferdig. Skriver vi ikke tester først så kan vi tro at vi har gjort det vi skal, men så har vi egentlig ikke det (:pPpPpP) Når tester er godkjent så refactorer vi så lesbarheten i koden blir bedre. - Lytte. Vi må vite hva problemet vi skal løse faktisk er, så du vet hva du skal lage tester for. Dette vet du mest sannsynlig ikke selv, derfor er det viktig at du lytter til kunden, brukeren

	og de som har domenekunnskapen. Vi har hyppig kontakt med kunde, og har de gjerne «on-site» så de hele tiden kan vurdere det vi gjør. - Design. Hva forteller programmet deg om hvordan det skal være strukturert? Bruk dette og som grunnlag til å lage et enkelt design. Vi trenger bare nok design til å møte inneværende behov. Keep it simple. *Vi er eksperter på å bygge et fort. En kunde er interessert i å bygge et fort, og har en idé for sitt eget fort og ønsker at vi skal bygge det. Så, for at kunden skal være fornøyd må vi **snakke** masse med de og vise frem det vi bygger for å være sikker på at det matcher deres idé. Derfor vil vi gjerne at de er med oss når vi bygger så de kan si hva de liker og ikke liker, og vi kan utføre det de ønsker. Vi vil gjerne at fortet vi bygger for kunden skal være sterkt, derfor **tester** vi alle spikere og skruer som holder fortet sammen og at de er på de best plasserte stedene. Vi vil bygge et bra fort for kunden, så det er viktig at vi ikke blir utbrente, derfor jobber vi ikke for mye med det hver dag, og forteller de etter endt arbeidsdag alt vi har gjort og statusen på prosjektet så de vet hvor vi ligger og ca hvor lang tid det er før vi er ferdig. Til slutt bruker vi tid på å bygge de viktigste delene først, og gjør disse ferdig i stedet for å bygge alt samtidig.* **OppsummeringPP:** - Små lanseringer ofte – rask feedback - Parprogrammering – løpende kompetansedeling - Test Driven Development – Verdi og resultat først - Refactoring – bedre vedlikeholdbarhet - Continuous Integration – Kontinuerlig testing av helheten i prosjektet
LEAN	- Eliminer waste o Færre møter, ikke dokumentasjon for dokumentasjonen skyld - Amplify learning - Decide as alte as possible - Empower the team - Build integrity in o Alle tar ansvar - See the whole o Vurder årsak/verdiønske over funksjonsforslag
Spiralmodellen	En runde i modellen representerer en fase, denne består av fire kvadranter: - Definere mål og identifisere risiko - Vurdere og redusere risiko - Utvikle og vurdere - Planlegging Disse fire kjøres på en loop for hver iterasjon i spiralen. Spiralmodellen ligger ca. på 3-4, mellom Waterfall og Inkrementell
SCRUM	SCRUM ligger under agile-paraplyen og er et **rammeverk** for å utvikle informasjonssystemer. Det brukes i hovedsak til å utvikle programvarebaserte systemer. Scrum-prinsippet bygger på

	samarbeid innenfor team på tre til ni medlemmer som bryter ned oppgaver til prosjekter som kan ferdigstilles innen gitte tidsrom, såkalte «sprints» som vanligvis strekker seg fra to uker til en måned. Metoden omfatter også daglige vurderings- og planleggingsmøter som ikke skal vare lenger enn et kvarter.
Scrum-master	Et team rullerer på rollen som Scrum-master f.eks hver sprint. Scrum-master har ansvar for: - Å organisere stand-up - Å organisere backlog - Måle fremdrift - Teamets kommunikasjon med interessehaverne
Stand-up	En del av SCRUM, et daglig møte med teamet som ikke skal ta mer enn 10-15 minutter – derfor står man på disse møtene. Mål er å identifisere eventuelle flaskehalser med inneværende arbeid - Nye utfordringer - Tekniske hindringer - Uklare spesifikasjoner Stand-up er ikke for diskusjoner om dype temaer, start da heller et nytt møte med de det gjelder rett etter stand-up.
Sprint	Sprint er en tidsperiode på 2 uker til 1 mnd hvor et ferdig, brukbart og potensielt produksjonsklart produkt er skapt. En ny sprint starter rett etter konklusjonen av den forrige sprinten. Fremdrift i en sprint vises gjerne med en Kanban-tavle. Under en sprint: - Ingen endringer er skapt som vil påføre skade på sprintmålet - Kvalitetsmål skal ikke minke og scope kan bli avklart og diskutert mellom produkteier og utviklerteam ettersom man lærer mer om produktet.
Backlog (PBI, Product Backlog Items).	En backlog er en liste som inneholder alt man kjenner til av ting som trengs i produktet. «It is the single source of requirements for any changes to be made to the product". En backlog blir aldri ferdig og utivder seg så lenge produktet utvikler seg. Tenk litt på den første kolonnen i Trello, som inneholder alt vi vet om som må gjøres, inkludert bug-fixes. PBI burde ikke overskride et dagsverk. Vi grupperer gjerne PBIer i «Epics» (Tittelen på en kolonne i Trello) for å ha større knagger å henge de på. **OppsummeringsPP:** - Inneholder mange «Product Backlog Items» - Samling av gjennomførbare tiltak for å ferdigstille produktet - Backloggen er en levende liste og den er IKKE ferdig når prosjektet starter.
Kost + Verdi = Priorirtet	Når vi definerer krav benytter vi gjerne en matrise med kost og verdi som ved hjelp av formelen Kost+Verdi=Prioritet, som gjør at vi kan sortere krav etter prioritet.
Story/Effort points (brukes ved estimering)	Hver oppgave i backloggen gjettes til et gitt antall story/effort points. En oppgave med dobbelt så mange poeng som en annen antas å ta dobbelt så lang tid. Et team kan plukke de oppgavene de ønsker å jobbe med i inneværende sprint. Ut i fra hva som gjennomføres i sprinten får vi et

	bilde av antall points et team klarer å håndtere, dette er teamets VELOCITY. Så hvis teamet klarer en oppgave med 4 poeng og en oppgave med 8 poeng så har de en velocity på 12. Dette kan brukes til estimering senere, man kan anta at teamet klarer 12 poeng hver sprint. - Forenkler estimering - T-skjortestørrelser: XS, S, M, L ,XL - Halve/hele dager
Burndown chart	En tabell som starter med dag 0 og den sammenlagte summen med effort points, og strekker seg med tid til høyre. For hver dag/iterasjon trekker vi fra de poengene vi har klart å gjøre unna. Ta hensyn til at bugs kan dukke opp underveis og gjøre at man må legge til poeng,
Produkteier	- Den i organisasjonen som har endelig ansvar for backloggen - Samarbeider med kunde for å prioritere - Optimaliserer verdien utviklerteamet skaper - Sikre at backloggen er tydelig, transparent og forstått av alle - Kan delegere deler av arbeidet til teamet, men har sluttansvaret - Scrum virker ikke uten denne rollen
Tester (rolle, en som…)	- Egen stilling i større organisasjoner/produktteam - Representeres ellers av en eller annen utvikler - Jobber gjerne med å utvikle automatiserte integrasjonstester
Reviewer/retrospective	Hver sprint avsluttes med en review, og består av… - Gjennomgang av kode (code review) - Hva gjorde vi bra? (Turn up the good) - Hva gjorde vi dårlig? (Eliminate waste) NB: Scrum dreier seg om erfaringsbasert ledelse og uten læring fungerer det ikke
Three Amigos	Produkteier, utvikler og tester. Gjennomgår materiale fra samtaler med kunder og konkretiserer brukerhistorier for ytterligere formalisering og kvalitetssikring. Kilden til de levende kravene som hele tiden fylles i backloggen og kompletterer den levende kvalifikasjonen og dokumentasjonen til systemet.
Spike (F.eks i XP)	Hopp ut av testdrevet- eller parprogrammering, hopp ut av alle gode vaner ved programmeringen. Lag en prototyp for å lære. Stabiliser etter dette. En Spike varer aldri mer enn 1-2 dager.
Behavior Driven Development - BDD	Hovedfokus på dialog med kunde og sluttbruker for å identifisere interessante historier. Three Amigos er sentrale sammen med kunden. Identifiserer «ubiquitous language» (Ubiquitous language is a model that acts as a universal language to help communication between software developers and domain experts). Har som optimal konsekvens at scenarier (brukerhistorier) kan testes automatisk. Oppgaver/testscenarioer kan formuleres med mal fra Dan North: - Gitt at (forutsetninger) - Når (handling) - Så (skjer konsekvens)
Cynefin	Verktøy for å velge prosesstyngde eller tilnærming. Delt opp i fire kvadranter:

	1. **Obvious**: alle skjønner hvordan det skal gjøres, det er lett å planlegge og estimere. Typ: Skrelle en banan 2. **Complicated**: Alle skjønner hva, men må ha ekspert for å utføre. Lett å planlegge og estimere. Typ: Reparere en bil 3. **Complex**: Hva og hvordan er ikke kjent, må utforske. Risiko for bieffekter. Typ: Hydrogendrevet bil? Robot som kurerer kreft 4. **Det haster, folk dør, handle nå**! Typ: katastrofer, data på avveie. 5. **(Ekstra) Disorder**: Kan ikke velge kvadrant, må dykke dypere i problemet Prioriteringer under **disorder**: - Chaos/complex hvis ingen har gjort det før - Complex hvis noen har gjort det før men i annen sammenheng - Complex/complicated hvis noen i vår organisasjon har gjort det før, eller vi har tilsvarende ekspertise - Complicated hvis noen i vårt team har gjort det før - Obvious hvis alle vet hvordan vi gjør det.
Storymapping	- Oppgaver – korte fraser med verb som beskriver hva folk gjør - Oppgaver har forskjellige mål og utfall - Arrangeres fra venstre til høyre i en narrativ flyt - Dybden viser variasjoner og alternative oppgaver - Oppgaver organiseres i aktiviteter på toppen - Aktivitetene danner ryggraden - Aktivitetene kan deles opp i mindre skriver for å identifisere oppgavene som skal til for å nå et gitt mål eller utfall - Mål og utfall dikterer lanseringsrekkefølge. Dette vi gjorde i timen med Lars-Erik med post-it lapper om morgenrutiner. **OppsummeringPP:** - Kolonner – aktiviteter brukerne foretar i systemet - Rader – lanseringer
User Stories	User stories er korte historier som overordnet forteller hva en «actor» skal gjøre i et system. User stories skal alltid skrives med et simpelt språk uten domenespesifikke termer, dette gjør at de blir forståelige for folk utenfor domenet. Vi beskriver det også med navn og «actors» for at vi enklere skal kunne sette oss inn i situasjonen og føler at det er ekte. *Anne er en doktor som skal skrive resept til en pasient i klinikken. Hun har allerede journalen til pasienten, og kan velge mellom «gjeldende resept», «ny resept». Velger hun nåværende medisin må hun sjekke om dosen er riktig. Velger hun ny medisin antar vi at Anne vet hva slags medisin som nå skal brukes og vi lar henne søke i databasen. Når hun taster inn noen bokstaver, viser vi all medisin med matchende navn fortløpende. Anne velger den medisinen hun*

	skulle ha og velger riktig dose. Til slutt spør vi om resepten er OK eller om hun vil endre på den. Når user storien er forstått kan et utviklerteam bryte historien ned til detaljerte user stories og deretter bryte det videre ned til de funksjonene som trengs i systemet. Dette brukes også til å estimere tid og kostnad for et prosjekt. User stories er ikke beregnet for detaljerte og komplekse oppgaver eller for å forklare hvordan detaljerte krav om f.eks et komplekst regnestykke skal være
Krav	Kravene til et system er funksjonene et system skal tilby, sammen med begrensningene av systemet. Kravene beskriver behovene brukerne har til systemet for å gjennomføre spesifikke oppgaver. Grupper krav som handler om de samme modulene/tingene sammen, slik at det er lett å finne krav som hører sammen. Identifikatorer for krav må være unike og stabile, og skal ikke endre seg etter at det er definert. Bruk gjerne teksttagger i stedet da disse kan uttrykke både hierarki og funksjonalitet: Arrangement.Opprette.Tittel istedet for 2.4.1 Link også krav sammen dersom de er gruppert på forskjellige steder men allikevel henger sammen, det gjør det lettere å navigere og man kan finne muligheter for å implementere felles funksjonalitet på kryss av moduler: Arrangement.Opprette.Tittel: Tittel skal ha min. 5 max 30 bokstaver (henger sammen med Arrangement.Sorter.Tittel) Start med Use-Case for å finne krav
Funksjonelle krav	- De synlige egenskapene til systemet som gjør at systemet løser den oppgaven det er designet for - Beskriver funksjonene i systemet – hva systemet skal gjøre og ikke gjøre 1. En bruker skal kunne betale for ordren sin med Vipps 2. En bruker skal kunne endre epostadressen knyttet til kontoen sin I en ideell situasjon er listen over funksjonelle krav komplett – men i virkeligheten så endrer ting seg. Man kan glemme selvsagte krav og det er kun når brukeren ser programmet kjøre at de selvsagte kravene dukker opp, derfor er det viktig med en prototype. (Bruker må kunne melde seg på et arrangement, men har vi husket kravet om at bruker også må kunne melde seg av et arrangement?) Krav dukker også opp når programvare er tatt i bruk og brukerne oppdager at «om vi hadde gjort slik… så hadde vi spart» og tilsvarende, og disse kravendringene er nesten alltid funksjonelle.

Domenespesifikke krav	Er fra et fagfelt du ikke er kjent med og må passes ekstra godt på! Disse kravene kan styre andre funksjonelle krav slik som hvilke autentiseringsmetoder som skal være tilgjengelig - Bruker må autentiseres med BankID, fordi alle som jobber innenfor denne bransjen må ha to faktor autentisering
Ikke-funksjonelle krav	Dette er krav til systemet som ikke direkte reflekterer funksjonalitet, men som fortsatt er viktige egenskaper som systemet kan ha. Kan beskrives som de «usynlige» delene av et system og inneholder begrensinger som er viktige for prosjektgjennomførelse. Beskriver egenskaper ved systemet – ikke funksjonalitet ved systemet. 1. Må skrives i [programmeringsspråk] 2. Må følge [UU/GDPR, andre lover] 3. Må kjøre hos [Cloud-leverandør] Krav 1 og 3 er ikke-funksjonelle krav som setter begrensinger. 1 fordi det sier noe om at du må bruke et spesifikt språk, og 3 fordi kunden kanskje allerede har all sin infrastruktur hos Amazon, Google Cloud, Microsoft Azure o.l. Unngå at ikke-funksjonelle krav beskrives på en generell, ikke testbar måte. - Systemet skal være lett å bruke (IKKE BRA) - Alle funksjoner i systemet skal kunne brukes etter to timer med opplæring (BESKRIVES HELLER SLIK) Ikke-funksjonelle krav endrer seg sjeldnere enn funksjonelle, og ofte kommer slike endringer utenfra organisasjonen, f.eks. GDPR eller UU.
Spesifikke krav	Et spesifikt krav sier noe om hvordan ting skal løses, hvordan flyten kan være og hvilke hensyn som må tar for å oppfylle andre krav i systemet.
Åpent krav	Går ikke i detalj om hvordan noe skal løses, bare at systemet skal tilby XYZ. Typisk brukt for å be om innspill fra leverandør på hvordan noe best kan løses.
Beskrive krav	Hvordan krav formuleres avhenger av kontekst og rollen til den som skal lese dokumentasjonen. Ikke alle krav spesifiseres på samme måte og heller ikke med like mye detaljer underveis i et prosjekt
Beskrive krav - MAL	The Chemist (User class or actor name) shall be able to reorder (Do something) any chemical he has (to some object) ordered in the past (Qualifying conditions) by retrieving the order details (response time, or quality statement)
Finne krav	To hovedgrupper: **Intervju**, spørre noen - Forskjellige former for dialog med brukerne og eierne av et system – både de som skal bruke systemet, de som skal teste det og de som skal drifte det -

	Observasjon, se på noen - Se hvordan brukerne utfører jobben sin i dag, uten å gå direkte i en intervjusituasjon for å spørre om hvordan de gjør noe, eller ved å spørre de overordnede om hvordan de ansatte skal gjøre noe Man kan også - Sjekke lovverk, hvordan regulerer lovverk forskjellige bransjer? Hvis det er noe vi må følge her burde det være med som krav? - Tilsvarende programvare, se hva som finnes som kan ligne på det som skal utvikles for å løse lignende problemstilling som finnes hos deg. Krav som du kan ha glemt kan fremstilles/avsløres. **OppsummeringsPP:** Andre metoder - Workshops - Fokusgrupper - Spørreskjema - Observasjon - Tilsvarende programvare
Use-Case	En use-case beskriver en overordnet oppgave som personen ønsker å utføre i systemet. Det er ingen sekvens av steg, men selve oppgaven som personen ønsker å utføre er i fokus. «Som bruker ønsker jeg å …» Start med use-case om det er vanskelig å se hvilke krav som finnes til et system.

Kontrollere krav	- **Validity**
	  - Reflekterer kravet det faktiske behovet som skal løses? Verden endrer seg og det er ikke lenger sikkert at kravet henger sammen med virkeligheten slik den var ved prosjektoppstart
	- **Consistency**
	  - Spesifiserte krav skal ikke være i konflikt med hverandre. Hvis et krav krever at det ikke lagres personnummer om brukerne, mens et annet krav krever at brukerne skal indentifiseres med personnummer så må ett av kravene endres.
	- **Completeness**
	  - Kravene skal dekke alle funksjonene som systemet skal gjennomføre (selv om kravene er det som definerer hva systemet skal gjøre, men dekker det hva det forventes å gjøre fra produkteier?)
	- **Realistic**
	  - Er kravene praktisk gjennomførbare innenfor prosjektets begrensinger, deadlines og budsjett?
	- **Verifiability**
	  - Kravene må beskrives på en måte som gjør de testbare, det betyr at du bør kunne skrive et sett med tester som bekrefter at det leverte systemet

	oppfyller kravet slik det er beskrevet. Hvis et krav ikke er testbart er dette et tegn på et problem med hvordan kravet er beskrevet. Kravet er gjerne for diffust til å kunne implementeres på en fornuftig måte. Det å kunne skrive tester for funksjonelle krav før koden blir skrevet er en fundamental del av testdrevet utvikling.
Pålitelighetskrav (Usikker på om forklaring er riktig på dette)	Krav som tar høyde for at feil kan skje. - Kunde skal kunne betale for en ordre Så kan man spesifisere at 1 av 1000 bestillinger vil feile, så kan man beskrive det som at sannsynligheten for dette blir 0.001. Det betyr ikke at 1 ut av 1000 bestillinger kommer til å feile, men om man sjekker 1000 bestillinger så skal antall feil ligge på ca 1.
Tilgjengelighetskrav (Usikker på om forklaring er riktig på dette)	Krav som beskriver hvor lang oppetid et system skal kunne ha. Evnt krav som spesifiserer hvor lite nedetid et system skal ha. «Availability is the probability that a system will be opretaional when a demand is made for the service. Therefore, an availability of 0.9999 means that, on average, the system will be available 99.99% of the operating time.
Watchdogs og beskyttelsessystemer	Egne enheter og rutiner som introduseres i maskinvare eller programvare for å oppdage om et system har feilet, og gjør en handling for å stenge ned eller starte systemet på nytt. Watchdogs – passer på om et system er operativt og starter det eventuelt på nytt. Beskyttelsessytemer – dersom system arbeider utenfor standarden den er satt til å arbeide innenfor, så stenges systemer ned eller det nekter å fungere. Åpen dør på mikrobølgeovn.
Pålitelighetskrav, typer.	**Checking requirements** - Krav til sjekk av input og outputverdier i et system **Recovery requirements** - Krav til hvordan håndtere en feilsituasjon, for eksempel lagre data til flere steder, og hvordan få systemet tilbake til kjørbar tilstand. **Redundancy requirements** - Egenskaper til et system som gjør at man kan kjøre flere instanser av samme funksjonalitet – slik at om en går ned så tar en annen over, og at feil i en enkelt komponent ikke tar ned hele systemet **Process requirements** - Krav til utviklingsmetoder og andre prosessmetoder som skal benyttes for å unngå at feilsituasjonene oppstår i det hele tatt.
Kravhåndtering	Prioritering av krav og funksjoner er noe som gjøres fortløpende – det som var viktig i starten er kanskje ikke viktig foran en senere sprint En backlog krever vedlikehold – noen krav eller funksjoner er ikke lenger relevante senere i prosjektet Det eneste sikre i et prosjekt er at krav og prioriteringer vil endre seg - Kjørbare prototyper sørger for at endringene fanges så tidlig som mulig - Vær åpen for kravendringer i alle ledd

Modellering (diagram)	En del av systemet eller prosessen visualisert. Modeller er abstraksjoner der større linjer trekkes. Bruk modeller for å skape diskusjon og avdekke krav, modellen kan ha verdi selv om den bare ble tegnet på en tavle og visket ut. Modeller er ikke fasiten, modeller er verktøy for å kommunisere, avdekke krav og features og bidra til at vi har samme forståelse av en problemstilling. Modellene kan også tydeliggjøre informasjon som ligger begravet i teksten, eller gi mer informasjon om hvordan noe er tenkt å fungere som kanskje ikke er beskrevet tydelig nok. **Hvorfor modeller som kildekode ikke fungerer** - Ved endringer i modellene vil det være vanskelig å integrere endringene inn i eksisterende kodebase - Passer dårlig sammen med små, raske iterasjoner som vanligvis benyttes for Agile utvikling. - Modellen vedlikeholdes utenfor der kode vanligvis vedlikeholdes - Hvis modellspråket representerer hele flyten i programmet, så vil det være tilsvarende språk som programmeringsspråket selv **Hvorfor modeller som visualisering av kode fungerer** - Illustrerer tydelig avhengigheter mellom modulene i koden som allerede er skrevet og er fasiten - Lever sammen med resten av prosjektet og er alltid et oppdatert bilde av hvordan applikasjonen henger sammen - Utfordring: Modellen beskriver ikke overordnet arkitektur (utenfor systemet), og kan være vanskelig å følge på kryss av systemer (slik som mikrotjenester og eksterne tjenester). **OppsummeringPP:** Modellering som representasjon av idéer - Stimulerer og holde fokus i diskusjonen rundt et produkt - Modellene brukes uformelt, og beskriver hva tanken var der og da - Uformelle modeller brukes ofte i Agile-kontekst Oppnå enighet om hva vi snakker om - Sørge for at vi diskuterer de samme tingene og har den samme forståelsen av hva vi jobber med - Felles forståelse av problemområdet og forklaringsverktøy ovenfor andre Brukes for å kommunisere viktige elementer - Modellering brukes for å avklare uenigheter og sørge for felles forståelse - Det er ikke noe poeng i å beskrive alt i et system i detalj – og ting modelleres ved behov – ikke i en stor bolk ettertid. Modeller er et verktøy - Det er ikke noe vi er NØDT til å ha med.

	- Modeller som ikke gir bedre forståelse eller bedre kvalitet i produktet har ingen verdi - Modeller som er gale – dvs. at de ikke lenger beskriver hvordan noe fungerer kan ha motsatt effekt – negativ verdi.
Modellperspektiver	**Eksternt**: - Systemet sett utenfra, hva er prosessene, konteksten og miljøet som systemet skal fungere i? **Interaksjon** - Hvordan forholder systemet seg til miljøet sitt, eller hvordan forholder delene av systemet seg til hverande? **Struktur** - Hvordan er systemet organisert, eller hvordan er datastrukturene i systemet lagt opp? **Oppførsel** - Modellering av den dynamiske oppførselen til systemet og hvordan det responderer til hendelser.
Modellering, nyutvikling og videreutvikling	Forslag til hvordan et nytt system, modul eller funksjonalitet skal henge sammen. Typisk for diskusjonen som dukker opp underveis i utviklingsprosessen. - Ofte uformelt - Større «fare» for endringer - Fasiten er ikke alltid kjent - Brukt i diskusjoner og som diskusjonsgrunnlag - Nyttig senere i prosjektgjennomførelsen for recap og om det tilføres nye prosjektmedarbeidere.
Modellering, dokumentasjon i ettertid	Dokumentasjon av eksisterende system, modul eller funksjon for å gjøre det enklere å utvide og forstå konseptene om hvordan noe henger sammen - Ofte mer formell i notasjon - Beskriver hvordan noe faktisk er (fasiten er kjent) - Mer stabil – eksisterende systemer og moduler endrer seg saktere - Nyttig dersom den blir lest.
Kontekstmodeller	- Etablerer en felles kontekst for systemet og omgivelsene rundt systemet, slik at alle involverte har samme forståelse av miljøet og begrensninger rundt og i systemet - Hvilken deler består eller avhenger systemet av – og hvilke avgrensinger er gjort for systemet som helhet Prosessmodell - Kan brukes til å vise hvordan en prosess fungerer i den «virkelige verden»
Interaksjonsmodeller	- Hvordan interagerer de forskjellige delene av systemet med hverandre, og med brukerne av systemet Use-case - Forklarer hovedfunksjonene – oppgavene – til en gitt actor i modulen. Del opp i flere modeller dersom antallet use-cases blir stort - Use-cases må beskrives skriftlig for t de skal ha verdi i en kravspesifikasjon – den enkle grafiske fremstillingen kan være

	nyttig for første forsøk på å finne funksjonalitet og brukere av et system. Sekvensdiagram - Går dypere inn i hvordan en funksjon er implementert – hvilke systemer er involvert, hvilke kall går mellom dem og hvordan er flyten gjennom systemet som helhet for å løse oppgaven som brukerne ber om å få løst. Viser eksterne avhengigheter mellom moduler i et prosjekt og hva det forventes at andre team/selskap håndterer
Strukturmodeller	- Beskriver struktur internt i programmet eller på data som programmet jobber med, både overordnet og detaljert Klassediagram - Beskriver datamodeller – enten som faktiske klasser i et objektorientert språk, eller som databasetabeller i en database. - Verktøy for å avdekke hvilke data som henger sammen og hva vi trenger å ta vare på for hvert objekt og hvilke objekter/tabeller som krever referanser mellom hverandre
Oppførselsmodeller	- Hva skjer med data gjennom en enhet fra input til output, og hva skjer når noen gjør ___? Data-flow-diagram (flytdiagram?) - Beskriver hvordan data flyter gjennom de forskjellige delene av et system, og hvordan data blir behandlet i hvert steg av systemet. Representeres gjerne som en aktivitetsmodell i UML Tilstandsdiagram - Praktisk til å forklare hvordan tilstanden til et system endrer seg som respons til eksterne stimulans. - Mest nyttig for kontinuerlig kontrollsystemer, typisk applikasjoner i form av spesialisert og embedded programvare - Forutsetter at systemet har et diskret (og helst lite) sett med tilstander og at tilstanden endrer seg basert på hendelser
Aktivitetsdiagram	Beskriver arbeidsflyten i en prosess/oppgave. Vanlig for å beskrive hvordan en funksjon eller oppgave løses i «virkeligheten». Viser hva som skjer parallelt og hvilke avgjørelser som tas.
Prosjekttidslinje	- Hvilke oppgaver avhenger av andre oppgaver, når er det leveranse og hvilke ressurser er opptatt til hvilke tidspunkt - Når man følger agile metodikk hjelper det lite å planlegge mest mulig på forhånd, når poenget er å tilpasse seg endringer for krav og kundeønsker.
Manuelle regresjonstester	Ett eller flere mennesker utfører samtlige handlinger i systemet. Både happy path og feilsituasjoner testes. - Test case 1.1: Sjekk resultater ved bruk av gyldig brukernavn og passord - Test case 1.2: Sjekk resultater ved bruk av ugyldig brukernavn og passord - Test case 1.3: Sjekk resultater ved bruk av tomt brukernavn og passord Dette tar lang tid og tester ofte ikke de banale tingene

Manuelle utforskende tester (exploratory tests)	- Forsøke å benytte systemet på uventede måter for å avdekke svakheter - Forsøke å finne bruksområder som kan forbedres eller til og med ny funksjonalitet

Enhetstester	- Automatisert kjøring av samtlige funksjoner eller klasser i systemet hver for seg
	- Dekke samtlige feilsituasjoner i tillegg til «happy paths»
	- En enhet i sin reneste form er en funksjon
	- En klasse
	- To eller flere klasser som spiller sammen
	- En hel pakke/modul
	- En hel applikasjon
	- Som hovedregel ett inputpunkt og ett output punkt
	- En handling en bruker gjør i systemet
	- Gjerne også mindre deler av større handlinger
	  o Validere e-postadresse
	  o Summere ordretall
	- Tester mindre enheter i koden – slik som enkle metoder uten avhengigheter
	- Perfekt for bibliotekskode og isolerte metoder i kodebasen
	- Kan enklere teste alle «hvis dette skjer så»- scenarioer i isolerte metoder.

Integrasjonstester	- Automatisert kjøring av f.eks to og to deler sammen. Ordreprosess og nettbetaling. - Tester mer «komplette» deler av systemet og overordnet funksjonalitet, der delene av systemet har blitt **integrert** med hverandre - De viktigste testene, ettersom de automatisk tester hovedfunksjonaliteten i systemet
Systemtester	- Automatisert versjon av manuelle regresjonstester - Alt integrert, fra GUI til database og tredjepartstjenester (nettbetaling).
Test Driven Development (TDD)	Hvordan vet du om koden du har skrevet virker, i dag og om en måned? Jo, du begynner med tester. Design systemet ditt ved å skrive tester. Et produkt av dette er at systemet blir enkelt å bruke, enkelt å teste. Når vi skriver tester først så skriver vi de for de minste enhetene vi skal ha, derfor tar det ikke lang tid for hver test/implementasjon. På denne måten får vi automatiserte enhetstester som er enkle å bruke for regresjonstesting (teste at endringer/nye funksjoner i systemet ikke brekker tidligere skrevet kode) og lar oss fikse bugs med en gang de er introdusert. Jo lenger en bug er i systemet jo vanskeligere er det å fjerne de fordi ny kode kan avhenge av koden som inneholder buggen. Derfor er det bra at vi får fjernet de med en gang. Vi får også «confident coders», for man får beskjed dersom noe har gått galt og derfor er det enkelt å fikse. Det man får er bare enhetstester og ikke systemtester, men sannsynligheten for at systemet fungerer er stor når alle enhetstester passeres, men man burde allikevel ha tester for hele systemet.

	Syklus i TDD. 1. Skriv en test 2. Kjør test og se at den feiler 3. Implementer koden 4. Kjør testene a. Repeter fra punkt 3 til testen godkjennes 5. Refactor/rydd opp i koden, endre navn og fjern duplikater 6. Repeter
Red – Green - Refactor	En teknikk for å restrukturere eksisterende kode, endre den struktur uten å endre oppførsel **Red** – skriv testen slik at du bekrefter at den fremtidige metoden returnerer riktig resultat. Resultatet foreløpig er at testen feiler og markeres som rød **Green** – skriv koden i metoden slik at metoden returnerer dataene som er korrekte i henhold til testen din. Testen skal nå passere og markeres som grønn **Refactor** – Rydd opp i koden din slik at den blir vedlikeholdbar og strukturert på en fornuftig måte for fremtiden. For gjenbruk, økt forståelse, dokumentasjon og mindre teknisk gjeld.
Refactor	Refactoring av eksisterende kode for å lære, teste bedre og enklere å kunne utvide koden ved hjelp av løst koblet kode. Alltid forla tkoden renere enn du fant den. Utviklere har like lett for å slurve, rote og forsøple som mennesker som besøker en park. Tenk at det kommer noen etter det, eller hjelp til ved å rydde etter de forrige som ikke gjorde det.
Suite	Sett av testcase (typisk for en pakke)
Fixture	Et sett av tester med sammenheng (typisk testklasse)
Test	En isolert test av en gitt adferd (typisk testfunksjon)
SUT	System under test, den enheten du tester (typisk klasse)
Unit	Enhet, en gitt atferd i systemet. - Ikke nødvendigvis en funksjon eller klasse, men gjerne en gruppe sammenhengende instanser med SUT som startpunkt
Akseptansetest	- «Er systemet klart for levering til produkteier?» - «Er alle de nødvendige kravene oppfylt for at leveransen er komplett?» - Enkelte av kravene kan være kontraktsbundet -der det typisk er inngått en skriftlig avtale om hva som skal leveres Disse testene må utformes av **prosjektleder og prosjekteier** sammen, slik at både kunde og leverandør er sikre på at produktet som leveres tilfredsstiller kravene til systemet Nærmere deadline vil dette også være i form av alpha/beta-testing internt hos kunden.
Navngivning	Unngå [klassenavn]test og [funksjonsnavn]test, slike tester kan lure deg til å bare lage en test pr funksjon. Bruk gjerne _ til å separere ord for enklere lesbarhet. Forsøk å beskrive handlingen som utføres.
Avhengigheter som vanskeliggjør testing	- Typisk IO - Database - Disk - GUI

	- Framworks - Libraries
Fordeler testing	- Testing gir bedre kvalitet og løst koblet kode - Testing hjelper til å finne riktig algoritme ved å utforske og gjøre refactoring - Husk Red-Green-Refactor - Hva som er en enhet/suit er opp til teamet å definere - Pass opp for skjøre tester

Testing og løst koblet kode	- Unngå å instansiere klasser som benytter IO eller tredjepartskomponenter
	  - IO kan være alt fra GUI til database til nett-tjenester og annet hardware
	  - Tredjepartskomponenter er typisk ting du finner på github, maven.
	  - Alle disse kalles for ustabile avhengigheter
	- Ved å gjøre dette vil software vårt få økt tilpasningsdyktighet til endringer hos både kunde og i teknologi
	- Som en bonus blir software vi skriver mye enklere å teste. I noen tilfeller er det den eneste muligheten til å kunne lage automatiserte tester

Hvorfor testing	For å oppnå stabilitet, trygghet og selvtillit til/i koden vår. Gjør koden det den skal når den blir skrevet? Enklere debugging. Trygghet med at koden gjør det den skal også i fremtiden. Som dokumentasjon. Setter krav til arkitektur. Begrenser behov for manuell testing og QA Fjerner frykt for endring Studier viser høyere effektivetet.

Abstrakt vs konkret klasse, testing.	- En konkret klasse eller funksjon er en som har kjørbar og kallbar kode i seg.
	  - Slike kan (nesten) alltid instansieres (new Person())
	- En abstrakt klasse er en klasse som kun fungerer som mal og kontrakt for andre klasser
	  - En abstrakt klasse kan allikevel inneholde kode som implementerende klasse kan kalle eller eksponere
	  - Abstrakt kan heller ikke instansieres.
	- Et interface er en abstrakt klasse uten kode, altså kun en kontrakt og kan derfor heller ikke instansieres

Design patterns	- Design pattern er et paraplybegrep for programmeringsmønstre som går igjen i mange organisasjoner og kodebaser - Et pattern er en beviselig kostbar løsning på et gjentagende problem
Repository	- Skjuler den konkrete lagringsmekanismen som blir benyttet fra resten av den verdifulle koden - Gir en uniform og fleksibel måte å aksessere data på - Minner om list - Får gjerne ytterligere metoder for å hente forskjellige (og mindre) utvalg data.

Main og ustabile klasser	All software har et utgangsunkt. Som ofte er main, men i mange web-rammeverk kalles en annen spesialfunksjon. Dette er det eneste stedet det er lov å nevne konkrete klasser som vi kan kunne ønske å bytte ut. Slike klasser kalles gjerne «volatile», altså ustabile. Ustabile klasser er klasser som har: o Avhengigheter vi kan ønske å bytte ut. o Kode som endrer seg ofte o Kode som endrer på grunn av tredjepartsendringer for eksempel Maven pakker - Det er disse vi ønsker å snu pilene bort i fra. - Derfor er individuelle main() per platform et «nødvendig onde», men en vidunderlig løsning på utallige andre problemer.
Command pattern	Totals report er en fin klasse som representerer en use case. Use case forbindes ofte med command pattern. Rapportering er en undertype av commands der det bes om informasjon-spørring «query». - Ved å skrive resultater direkte til System.Out.Print binder vi klassen hardt til terminal GUI. - Ved å returnere en strukturert data i stedet gjør vi algoritmen helt gjenbrukbar på tvers av plattformer og rammeverk. - Hver App main implementerer sitt eget grensesnitt for å presentere dataen.
Test doubles/fakes	Ettersom vi abstraherer et repository kan vi varaiere hva slags konkret implementasjon vi vil bruke i testene Slike varianter som blir brukt i tester blir kalt «test doubles» Varianter vi skriver selv til test kalles fakes En fake er alltid dummere enn en ordentlig klasse Les en state eller bekrefte funksjonskall - Etter å ha matet data inn i et FakeRepository kan vi lese de ut igjen og sjekke at de har riktige verdier - En annen måte er å benytte en Mock **OppsummeringPP:** For eksterne avhengigheter og for å kunne teste kode med sideeffekter (som f.eks det blir sendt en mail som følge av at koden kjører) brukes test doubles. Dette er kode som later som de er den eksterne tjenesten (fakes), som sjekker at en metode kalles (eller ikke kalles) i eksisterende klasse (mocks) eller som inneholder en uferdig versjon av en klasse utelukkende for testing (stubs).
Mock (behavioral testing)	Tar opp funksjonskall og kan verifisere eller bekrefte bruk i etterkant og det forventes ikke at det skal returneres noe verdi, man vil kun se at metodekallene faktisk er blitt gjort. - Mock er også en type test double Kommer til god nytte når man for eksempel skal sende en beskjed over nettverk til en annen tjeneste - Ikke mulig å kalle en livetjeneste og se om det er kommet inn testdata i tide og utide - Hva om tjenesten sender epost f.eks Mocks brukes for å verifisere at en funksjon blir kalt

	Man kan velge å kvalitetssikre så mange av parameterne som fornuftig for use-caset.
Stubs (State testing)	Stubs tester logikken i koden vår der hvor vi ellers ville kalt på tredjeparter f.eks en database. Har du en database over alle varene i en nettbutikk til en hver tid, og har logikk for å kalkulere noe basert på antall varer i nettbutikken så vil du i stedet for å iterere og telle hver enkelt vare i databasen hver gang du kjører koden heller skrive en Stub som har et hardkodet antall som ville vært reellt f.eks 2300 varer. Testen tester dermed logikken i koden vår med 2300 og ser at den ville fungert med dette antallet, så vil det mest sannsynlig også funke når varantallet er 1800, 2000, 2650 osv.
Fordeler med løst koblet kode (DTO, data transfer object)	1. Med løst koblet kode og mulighet til å variere lagrings og prosesseringsmetode kan vi teste og distribuere koden vår uten å endre kjernekoden 2. Med løst koblet kode er det lettere og ryddigere å utvide softwaret 3. Med løst koblet kode som utveksler DTOer kan forskjellige team jobbe på forskjellige use-caseer samtidig 4. Med løst koblet kode kan GUI-rammeverk og lagringsmedium oppgraderes annenhvert år uten å endre kjernekoden 5. Med alle individuelle komponenter testet vil man aldri være i tvil om at eventuelle feil ligger i integrasjonspunkter eller tredjepartskomponenter.
Risiko med løst koblet kod	1. Komponenter som burde vært testet sammen blir ikke testet sammen 2. For mye må konfigureres i main() – gjør abre ustabilde avhengigheter konfigurerbare 3. Stubs og Mocks misbrukes og det kommer mye «arrange-kode» i testene 4. Mulig mental overhead til man blir vant med abstraksjoner
Versjonskonroll, kildekodetre	Inneholder alt som er relevant for prosjektet. Hjelper deg med å ha oversikt over de forskjellige versjonene og kunne se endringene mellom dem. Et versjonskontrollsystem gir oss et sett med verktøy som støtter opp under funksjonalitet knyttet til versjonering av kode og informasjon. Versjonskontroll er ikke bare for kode, også bra for dokumentasjon og konfigurasjon

Hvorfor versjonskontroll?	Fordi vi sjeldent jobber alene på et prosjekt
	- Mange mennesker jobber med det samme prosjektet, de samme filene og det samme innholdet.
	- Mange ganger endrer vi det samme innholdet samtidig og jobber med samme fil (men kanskje på forskjellige steder)
	- For eksempel hvis du har slettet feil fil eller vil gå tilbake til det du jobbet med for noen timer siden så går du bare tilbake til tidligere versjon
	- Versjonskontroll lar oss spørre
	  o Hva som ble endret
	  o Når det ble endret
	  o Hvem som endret det
	  o Hvorfor det ble endret (commit-melding)
	  o Vi gjør feil

	○ Versjonskontroll gir oss backup med tidsdimensjon
Versjonskontroll generasjoner	Tidligste: RCS, en person jobber med filen av gangen. Felles lagringssted på felles disk Neste: CVS, SVN, Sentral server. Bedre støtte for at mange kan jobbe med samme fil samtidig Nå: Git, Mercurial. Distribuert – hver utsjekket versjon er hele historikken.
Commit	En endring i tilstanden i kildekodetreet - En eller flere filer har blitt endret eller lagt til, og dette utgjør en enhet med arbeid - Endringene committes sammen med en beskrivelse til andre og deg selv om hva endringen er - Andre som lurer på hvorfor noe er gjort har nå en innledende forklaring på grunnen, hvorfor og hva som eventuelt kan være følgende ved endring som rører samme kode - Commit ofte!
Diffs	For å unnga at vi må lagre hver eneste kopi av en fil, så lagrer versjonskontrollsystemet vanligvis bare forskjellene mellom hver fil, og viser forskjellene mellom versjoner. (Git – grønt felt)
Git	Distribuert versjonskontroll – ditt eget Git-repo på din egen maskin er like «riktig som git-repoet et annet sted – så hva som er nåværende referanseversjon er konvensjon, ikke noe som er innebygd i systemet NB: Ettersom git er distribuert, så er det ingen «sentral kilde» som kan si hva som er hvilken versjon. I stedet for versjoner tar Git og identifiserer hver commit med en egen commit-ID. Denne bygger på Iden til forrige commit, innholdet i repoet når commiten blir gjort (innholdet i filen), hvem som har committet og når det ble gjort. Det er først etter at du har gjort en commit at endringene er en del av kildekodetreet i versjonskontroll
Branching	Sjekker ut en egen versjon av f.eks versjon 1 i et repo. Endringene i denne nye versjonen (branchen) gjøres helt separert fra hovedtreet, og påvirker ikke treet i noe grad. Når bruker som lagde en branch er fornøyd og det han skulle gjøre fungerer så kan han merge det inn i hovedtreet igjen. Eventuelt om alt feilet, kan branchen bare slettes.
Hvorfor branche?	Fordi en branch påvirker ikke resten av kildekodetreet, men er akkurat det samme som kildekodetreet i det du lager en ny branch. Er nyttig dersom du skal lage features som ikke er klare for produksjon, fikse bugs eller annet uten at vi stopper arbeid med andre ting. NB: Hovedbranchen skal alltid være produksjonsklar!
Konflikter, versjonskontroll	Oppstår når to personer har endret på samme linje mellom to versjoner, så vil det oppstå en merge konflikt. Dette er et tilfelle det versjonskontroll-systemet ikke greier å si hva den rette løsningen er. Dette må løses manuelt ved å velge hvilken del av koden som er riktig
Gitignore	En liste med filer som ikke skal legges til i Git som Git skal overse selv om du gjør git add. Typsik vil du ikke ha med .idea mappen siden denne er personlig. Viktig å gitignore hemmeligheter som API-nøkler, passord o.l.
Feature branches	En arbeidsmetode som tilsier at alle nye features som skal implementeres i en stabil programvare skal utvikles i en separat branch fram til featuren er helt klar og deretter merges inn i master

	Dette gjør det meget enkelt å identifisere alt som hører til den nye featuren. (Kode, dokumentasjon og tester). Det forhindrer også at andre ting som utvikles samtidig på samme kodebase forurenser ditt arbeid. Hvis du fikser en bug til den stabile versjonen så lager du en ny branch, gjør fixen og merger inn igjen, dette påvirker ikke andre sitt arbeid.
Git tag	Markerer hva kildekodetreet på et gitt tidspunkt utgjør. Uavhengig av branches – en tag refererer til en spesifikk commit, og hvordan kildekodetreet så ut på det tidspunktet. Typisk ved produksjon (når ny kode gjøres tilgjengelig for brukere), tag kildekodetreet med «release», slik at det er enkelt å finne tilbake til «den versjonen vi lanserte».
GitHub Pull Request	En pull request er en beskjed til prosjekteier/teamet om at «jeg vil ha disse endringene inn i hovedbranchen». Pull requesten lages fra en branch.
GitHub Fork	En fork er en komplett, selvstendig versjon av det gamle prosjektet. I de fleste tilfellene gjøres forking på GitHub for å ha en egen versjon å jobbe på (ettersom du nødvendigvis ikke har tilgang til å endre andres repositories). Tradisjonelt er forking brukt ved open-source prosjekter hvor man vil lage en egen versjon, ofte på grunn av uenigheter rundt prosjektet eller at det orginale prosjektet mer eller mindre er dødt. MySQL er forket som MariaDB og Percona som utvikles separat fra MySQL som egne produkter.
Forking	Ettersom Git beholder historikken via commit-Ider så kan vi fortsatt merge endringer mellom prosjektene, så lenge endringene er kompatible med tilstanden der de skal merges inn. Dette er grunnen til at Pull Requester kan baseres på en branch i en fork.
Continuous integration	Hver gang kode pushes/lastes opp til en branch, så kjøres alle testene som hører til koden. Hvis testene feiler så varsles utviklerne og status på branches settes til at den feiler, og derfor ikke klar for master. Dette gir oss direkte tilbakemelding på om vi har brukket noe eller koden vår ikke fungerer som forventet

SOLID/Arkitektur	- **SRP**
	  ○ En funksjon, klasse eller modul skal bare ha én årsak/kilde til endring
	  ○ Dette gjør dem enklere å teste, vedlikeholde og utvide
	- **Open Closed Principle**
	  ○ Koden skal være enkel å utvide uten å måtte endre eksisterende kode
	- **Liskov Substitution Principleb**
	  ○ Ingen skjulte effekter eller behov utover det som er synlig på- og logisk for interface/abstrakt klasse
	- **Interface Segrefation principle**
	  ○ La konsumentet forholde seg til bare de funksjonene som er nødvendig
	  ○ Forenkler bruk av komponenter som Inneholder Mye
	- **Dependecy Inversion principle**
	  ○ Muliggjør løst koblet kode, konfigurasjon, testing og resten av SOLID prinsippene

	○ Hold kontrollen på ustabilde avhengigher så nærme main() som mulig ○ Injiser i constructor. Løst koblet og SOLID arkitektur er - Lett å teste - Lett å skalere - Lett å oppgradere - Gjør livet lettere
S – Single Responsibility Principle «*A class should only have one reason to change*» Tenk på lommekniven med tusen ting	En klasse skal bare ha en årsak til å endre seg. Flere ansvarsområder gjør det mer utfordrende, om ikke umulig å gjenbruke de avhengighetsløse delene. Klassene vi burde hatt finnes ofte i lange funksjoner med mange variabler. Variabelene kunne vær felter(fields) i en ny klasse i stedet. Kall heller på metoden i den nye klassen fra der den lange funksjonen var. Eksempel: - En klasse både tegner et rektangel og beregner arealet på det. Skill heller dette ut i to, så vi har en klasse som tegner og en klasse som beregner.
O – Open/closed Principle «*Software entities (classes, modules, functions etc.) should be open for extension but closed for modification*» Bildet av to dokker med ribbein og lunger ute.	- Det er om å gjøre å endre minst mulig kode når nye features introduseres - Mange algoritmer er gjenbrukbare, men deler av innholdet varierer - Prioriter å gjøre koden *utvidbar* og *fleksibel* fremfor *gjenbrukbar* - Dette bidrar til å gjøre det lettere å introdusere ny funksjonalitet - Ved å holde hver klasse liten og utvidbar (som i SRP^) blir det også lettere å endre eksisterende funksjonalitet - Åpen for utvidelse, men lukket for endring. - **Template method pattern** ○ Nyttig der hvor en generell algoritme varierer i forskjellige sammenhenger ○ Lar en baseklasse kalle funksjonalitet i arvede klasser før og etter sin egen kode ○ Benyttes gjerne sammen med Strategy pattern - **Strategy pattern** ○ Benytter *polymorphism* for å tillate å variere indre algoritmer i like prosesser ○ F.eks ønsker vi å varire om vi henter priser fra disk eller en online tjeneste ○ Ofte nyttig der man ellers ville brukt switch/case eller mange *if*er i tur og orden ○ Gjerne hjulpet av *Factory Pattern* for å instansiere riktig strategi. - **Factory Pattern** ○ Brukes for å kunne endre hva slags implementasjoner som er i bruk, uten at koden vet om dem.

	○ Muliggjør bl.a. å legge til nye pakker med ny funksjonalitet etter kompilering. ○ Dette er også metoden som brukes for å "modde" spill, f.eks counter strike moddet fra half-life. ○ Med reflection kan man flytte avhengigheter til konfigurasjon ▪ Det går an å nøye seg emd ett "factory" pr. Type i kjernen. ▪ Men en slik metode hører bare hjemme i en konkret factory-klasse! **OppsummeringsPP:** Ved å legge til rette for *utvidelse* og *ny* funksjonalitet uten å måtte redigere eksisterende kode forenkler man vedlikehold og videreutvikling
L – Liskovs Substitution Principle «*If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction*»	Når en klasse eller en funksjon benytter en abstraksjon må ALL funksjonalitet på den konkrete implementasjonen oppføre seg som forventet Konkrete implementasjoner skal kunne endres og byttes uten å påvirke kode som benytter abstraksjonene Ved brudd på dette prinsippet mister abstraksjonene nytteverdien sin og koden kunne like gjerne vært 100% tett koblet. **OppsummeringsPP:** «Et kvadrat er ikke et rektangel!» Når en arvet klasse må modifisere funksjonaliteten i baseklassen er den ikke lenger en konkretisering, men en «annen ting». Den bør derfor heller ikke arve baseklassen, men heller være søsken og eventuelt duplisere noe kode. Skal man gjøre en beregning på et kvadrat så er alle fire sider like, og derfor har den en funksjon for å beregne areal hvor den bare tar inn ett input på en side sin lengde. Arver rektangel-klassen denne kvadratklassen så kan man ikke beregne arealet fordi rektangelet har to og to like sider. Derfor kan de ikke arve selv om de er nesten det samme.
I – Interface Segregation Principle	- Konsumenter av abstraksjoner bør kun måtte forholde seg til funksjonalitet som er relevant. - Altså bør abstraksjoner i seg selv være sett av sammenhengende funksjonalitet. - En konkret klasse som sender fra seg data har f.eks en .send() funksjon. Denne tilfredsstiller kanskje bastraksjonen Imessaging.send() - Dersom den konkrete klassen også krever et kall til .open() og .close() bør den få et annet interface som indikerer at den krever det. F.eks IConnectedMessaging.open() og IConnectedMessaging.close() - Koden som kaller .send(), eller kanskje koden som injiserer IMessaging kan vite om- og teste etter IConnectedMessaging. For så *eventuelt* kalle .open() og .close() - Ref. LSP **OppsummeringPP:**

	En klasse skal ikke måtte implementere metoder fra interfacet om de ikke har behov for de. Flere metoder betyr flere potensielle avhengigheter, mental overhead og kompatibilitetsbrudd. Ved å servere lette konkrete abstraksjoner forblir klientkoden enkel å vedlikeholde.
D – Dependency Inversion Principle «*High level modules shpuld not depend on low-level modules. Both shpuld depend on abstracions.*»	1. Muliggjør løst koblet kode - Ved å invertere avhengigheter med abstraskjoner muliggjør vi utvidelse og endring av avhengighetene uten å påvirke koden som bruker dem. - Kode blir testbar fordi vi kan bytte ut lagrings- og kommunikasjonsmekanismer med test doubles - Abstraksjonene skal ikke peke på noen konkrete ustabile avhengigheter - Vår kjernekode (domain model) skal kun avhageneg av våre abstraksjoner - Det er en ekstra fordel å kun benytte dataoverføringsobjekter (DTO'er) i abstraksjonene for ytterligere å skille kjernekoden fra omverdenen - Store softwarekontrakter forutsetter levetid på 10-30år! - Rammerverk og platformer kommer og går. Ren og uavhengig kode lever evig. - **Dependency Injection** ○ Anti Pattern: Control Freak ▪ Når man gjør new X() på ustabile avhengigheter ▪ Hindrer utbytting og oppgradering av rammeverk og forretningsregler ○ Pattern: Constructor injection (Denne vi helst vil bruke) ▪ Send med ting som implementerer abstraksjonene du avhenger av til constructor. ▪ Samle hvilke- og hvordan klasser instansierers så nære main() som mulig. ▪ I web-prosjekter o.l er ofte "main()" en annen oppstarts-klasse. Det er alltid godt dokumenteret. Gjerne ifm. "Application Startup" ○ Anti Pattern: Service Locator ▪ Når man har ett factory for alt - som brukes over alt. ▪ Lager "Temporal Couplings". ▪ Blir vedlikeholdsmareritt, og vanskelig å vite når man trenger et ▪ *(Men det er lov fra main())* **OppsummeringPP:** Ved å sørge for a tkode som har eksterne avhengigheter jobber mot en abstraksjon (vanligvis et interface) i stedet for en implementasjon (selve klassen) blir kjernefunksjonaliteten i klassen utvidbar.

	Koden kan nå fortsette å brukes selv om lagringsmetoden, grensesnittet eller andre elementer endres – så lenge disse passer til abstraksjonen (interfacet) som er beskrevet.
Containerization	- I stedet for å få tilgang til en virtuell maskin som vi setter opp og installerer programvaren vår på, bygger vi en egen, liten versjon av «maskinen» vår og sier «kjør denne»! - Vi stapper alt som applikasjonen trenger inn i samme boks. - Hvordan «maskinen» skal se ut beskrives i et eget format og med en applikasjon som lager «maskinen» vår slik vi ber om. - Det du får er en container som inneholder applikasjonen, plattformen og avhengighetene – en komplett versjon av alt som må til for å starte opp applikasjonen. - Containere er ikke fullstendige virtuelle maskiner (teknisk sett) men man kan se på de som det. - «Men det funker på min maskin» er ideen bak containerization. Man skriver en programmatisk beskrivelse som sier hvordan maskinen skal sette sammen. Man ber om at maskinen blir bygd etter oppskriften: kopier inn disse filene, kjør disse kommandoene, kjør denne kommandoen for å starte applikasjonen. Man ender opp med det miljøet som applikasjonen er laget for å kunne kjøre i. **OppsummeringsPP:** Pakketering av programvare og avhengigheter sammen. I stedet for å forholde oss til applikasjonen vår, avhengighetene og driftsmiljøet til applikasjonen, så legger vi alt sammen i en enkelt pakke – en container. Automatisering av byggeprosessen. Vi automatiserer hvordan applikasjonen og miljøet rundt applikasjonen settes sammen ved hjelp av en mal.
Isolasjon, containerization	Applikasjonen vår lever for seg selv, ingen andre vet om den eller får endre på filer/innhold etc. som er en del av containeren og applikasjonen år. - SuperApp og MegaApp vet ikke om hverandre, selv om de kjører på samme fysiske maskin så lenge de kjører i ulike containere. De kan fortsatt prate sammen via nettet, men de lever i sitt eget miljø – isolert fra andre containere. Avhengigheter og miljøet i hver container er utelukkende tilgjengelig for applikasjonen i containeren – ingen andre containere benytter seg av eller vet hva som finnes inne i andre containere.
Stabilitet, containerization	Containeren vår er statisk og kan ikke endres utenfra, men vi kan laste opp en ny versjon og starte den i stedet om vi vil endre noe Endringer i en annen container påvirker ikke innholdet i vår egen, og applikasjonen kjører i «sin egen verden» som ikke påvirkes av hva andre gjør. Ved problemer kan containeren vår restartes med samme innhold som den hadde da vi startet den for første gang – som å reboote en maskin uten at noen endringer har skjedd på den fra slik den var da vi satte den opp første gang

Estimering	Estimater, kvalifisert gjetting. Estimater krever erfaring, både med teamet som skal jobbe på prosjektet og med problemet som skal løses. Husk at estimater krever tid og du har *ikke* tenkt på alt og kommer heller ikke til å gjøre det! Spor dine egne estimater – den eneste måten du kan bli bedre på å estimere er ved å se på hva du trodde før prosjektet og hvor feil du tok etter på **og** hvorfor du tok feil. Om du ikke evaluerer i ettertid så vet du ikke hva du bommet med eller hvor mye du bommet med. Om du alltid bommer med en omtrent faktor på to, doble alle estimatene dine med en gang. Basisregel: gang med pi. Noen kunder vil ha ALT estimert, men har liten forståelse for at et estimat tar også opp tiden din, og har liten forståelse for at et estimat er et estimat og ikke en fasit. To typer: - **Erfaringsbasert estimering** Personlig kunnskap om hvor lang tid det vil ta å gjennomføre et prosjekt eller en feature - krever inngående kjennskap til problemet, eget team og mulige problemer underveis. - Story/effort points - Planning poker - Grovestimering og klassifisering (tshirt-sizing) - Dekomposisjon og rekomposisjon - Bygg en prototype. Hjelper deg med å finne skjulte kompleksiteter i systemet. - **Algoritmisk estimering** Ved å gi en nøkkelfaktor (f.eks antall linjer kode, antall krav etc) og benytte formler for hva dette tilsier i utviklingstid for denne typen prosjekter, genereres et estimat for utviklingstid. Vanskelig å gjennomføre i praksis – nøkkelfaktoren er kanskje bare kjent i ettertid (hvor stort prosjektet blir), utover at man kan gjøre et forsøk på å estimere hvor stor nøkkelfaktoren blir (så estimatet baserer seg på et estimat). Påvirkes i stor grad av endringer i utviklingsmetodikk, språkvalg og andre faktorer. **Brukes mest innenfor tungt planbaserte miljøer, slik som luftfart og militære utviklingsprosesser**.
Grovestimering og klassifisering	T-shirt sizing. Plasser alle estimater i en av tre eller fire grupper: Small, Medium, Large, (X-Large). Størrelse på utvikling estimeres til ett av disse nivåene, og tilsvarende gjøres med størrelse på forretningsnytte/verdi. Har noe da utviklingstid Large og verdi Small så prioriteres ikke denne dersom det er andre punkter som har høyere verdi og lavere utviklingstid. Grovestimering gir alle et tidlig bilde av hva som er viktig og avslører oppgaver som er store men som ikke har samme utbytte forretningsmessig. Grovestimering tydeliggjør også ovenfor produkteier hva som er vanskelig, slik at de kan prioritere hva som må gjøres. Dette gjøres tidlig i prosessen slik at det aktivt filtrerer ut krav. Hvilke krav skal vi

	implementere først – de featurene som gir oss størst nytte tilbake burde prioriteres.
Planning poker	Verktøy for å få et bedre estimat. Alle deltakere har et sett med kort med verdier fra en og oppover (etter valgt skala). For hver feature legger alle ut et kort som representerer hvor lang tid / mange story points featuren krever. Dersom alle er innenfor ca samme verdi så brukes denne verdien. Hvis det er stort spenn i kortene som er valgt så diskuterer vi hvorfor vi tror det er så stor forskjell på hvor lang tid det tar. Vi tar ikke snittet mellom kortene som er spilt ut, vi bli enige om hva vi tror er mest riktig etter diskusjon. **Målet** med dette er å avsløre informasjon som bare enkelte på teamet kjenner til, og på den måten få et mer riktig estimat.
Dekomposisjon	**Mange små estimater** framfor ett stort. Sørg for å dele opp større oppgaver i enkeltstående features og krav til systemet.
Rekomposisjon	**Bygg det større estimatet** fra de mindre.
#NoEstimates	I stedet for *hvor lang tid tar det å bygge xyz, hvor mange kroner koster det å bygge xyz* så snus modellen til å være *bygg det beste produktet før dato x, bygg det beste produktet for x kroner*.
Cone of Uncertainty	Nøyaktigheten til estimatene våre blir bedre når resterende arbeidsmengde minker og vi har mer kunnskap om problemet vi løser
Avhengigheter	Alt vi ikke lager selv som en del av prosjektet, men benytter av ferdigutviklede bibliotek og programvare. Avhengigheter kan styres av ikke-tekniske grunner så vel som tekniske grunner. F.eks, en kunde har bare linsens for en versjon som bare støtter et biliotek frem til versjon 2, alt etter versjon krever at kunde kjøper ny linses… det har de ikke tenkt til. **Naive/implisitte avhengigheter**, avhengigheter vi kanskje ikke tenker over, som bare er en del av plattformen vår – operativsystem, versjon av programmeringsspråk, et. Dette er vanligvis ikke-funksjonelle krav til systemet - Java - .NET - Google Cloud - MySQL - Apache HTTPd - nginx
Eksplisitte avhengigheter	Avhengigheter vi kan definere – og dermed verifisere – i oppsettet/installasjonen/kompileringen av prosjektet vårt. Ting som vi eksplisitt tar med inn i prosjektet vårt av programvare som andre har utviklet. Databasebiblioteker, malspråk, webrammeverk etc.
Pakke- og avhengighetssystemer	Java: Maven, Gradle. Python: pip, pypi. Pakkesystemene lar oss si «jeg vil ha dette biblioteket i denne versjonen tilgjengelig i prosjektet mitt» også ordnes resten automatisk. (Gradle og Maven i eksamensprosjektet). Vi slipper å ordne med installasjon og plassering av biblioteker osv, dette fikses automagisk. Pakkesystemene brukes på akkurat samme måte internt

	i organisasjonen også, slik at vi kan bruke interne biblioteker på tvers av prosjekter og team. Vi slipper manuell håndtering av bibliotek, hvor ting skal legges inn, manuelt oppsett av hvilke bibliotek/kataloger som skal være tilgjengelig i prosjektet.
Låsefil	En låsefil sier akkurat hvilken versjon av et bibliotek som ble installert, slik at vi kan gjenskape samme miljø (det vil se – de samme avhengighetene) både der prosjektet utvikles og der det skal kjøre. Dette gjelder også alle avhengighetene til biblioteket vi installerte, slik at vi i praksis skal ha full oversikt over avhengighetene som ble installert (og kan gjenskape dette oppsettet).
Semantisk versjonering	2.1.0. 2 = Major 1 = Minor 0 = Patch Patch, denne skal økes når du gjør backwards-compatible bug-fixer. Ingen nye features er lagt til. Minor, denne skal økes når du legger til funksjonalitet som er bakover-kompatibel. Major, denne skal økes når du gjør endringer i selve API. Hvis offentlige metoder blir fjernet eller definisjonen av eksisterende metoder endres.
Faser i et softwares liv	- Utvikling - Videreutvikling - Vedlikehold (Det er her vi er lengst) - Pensjonering
Pakkesystem	Programvare som holder orden på avhengighetene i prosjektet vårt og hvilke versjoner vi avhenger av.
Legacy Code	- Enhver kode uten automatiserte tester - Du vet aldri om det du endrer kommer til å påvirke noe du ikke vet om - Nesten garantert avhengigheter på uheldige plasser - Som regel like vanskelig å bruke som vanskeligste avhengighet - Høy risiko for at koden er kompleks og vanskelig å forstå
Legacy hardware og programvare	- Programvare budnet til hardware som ikke lenger eksisterer - Programvare som bruker rammeverk som ikke lenger er støttet eller ikke lenger er en del av plattformen det ble utviklet på - Avhengigheter i form av annen programvare som benyttes av applikasjonen er ikke lenger støttet og kjører ikke på moderne plattformer - Data som ikke har blitt vedlikeholdt og håndtert dårlig i applikasjonen - Utgåtte utviklingsverktøy og utviklingsmiljø – det bli eksperimentert med å bruke containere for utviklingsmiljøet for langtlevende prosjekter - Eksterne kommunikasjonsprotokoller endrer seg (COM-porter, printer-porter)
Orchestrazion	Koordinering – handler om hvordan vi sprer containerne våre på kryss av servere. Hvor mange, hvor mye og hvilke ressurser til hver container?

Videreutvikling av programvare, hvorfor?	- Det kommer endringer i lover og regler - Endringer i forretningsprosess - Feil i programvare - Konkurrenter
Utfordringer med videreutvikling	- Mange systemer i bruk - Integrasjonen mellom systemene - Leverandører som forsvinner - Gamle systemer fortsatt i bruk - Utskifting av plattformene som systemene kjører på