



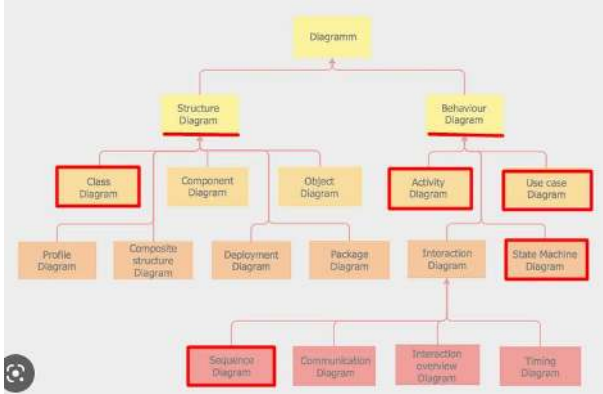
# Udviklingsmetoder til IT Systemer Guld Noter

Udviklingsmetoder til IT-systemer (Danmarks Tekniske Universitet)

# Udviklingsmetoder til IT Systemer Guld Noter

Udarbejdet af Moiz H. Khalil

## Begrebliste:

Begreb	Betydning
Test	<p>" a procedure intended to establish the quality, performance, or reliability of something, especially before it is taken into widespread use."</p> <p>Disse elementer kan man teste:</p> <ul style="list-style-type: none"> <li>- Prototype</li> <li>- Applikation</li> </ul> <p>Og disse elementer kan man inspicere</p> <ul style="list-style-type: none"> <li>- Kravspecifikation</li> <li>- Prototype</li> <li>- Arkitektur</li> <li>- Database skemaer</li> <li>- Applikation</li> </ul>
Vision	En vision er en langsigtet plan der er bredt og ambitiøs, ofte mange år i fremtiden.
Diagram	<p>Diagram er en form for arbejdstegning, i det her kursus bruger vi Adfærdsdiagrammer (dynamiske), og statiske diagrammer.</p> <p>Nedenunder står der de forskellige slags diagrammer der tilhører under hvad.</p> 
Mission	Nuværende problem, er små trin for at opnå Visionen.
OOAD (Objekt-Orienteret Analyse og Design)	En række delmetoder til gennemførelse af analysedelen af et udviklingsforløb.
UML (Unified Modelling Language )	<p>UML er det standardized udseende for diagram. Brugt for Objekt orienteret softwaresystemer. Der findes enten struktur diagrammer, eller adfærdsdiagrammer</p> <p>UML-diagrammer bruges til:</p> <ul style="list-style-type: none"> <li>- Kommunikation</li> <li>- Implementering</li> <li>- Dokumentation</li> <li>- Hardware</li> </ul>
Aktør	Indgår i et use case, ofte en bruger.
Iterativ metode	Formålet med iterativ metode, er at bruge noget en ad gangen, det her en af formålene med unified process.

	<p>Formålet med iterativ metode er ogs:</p> <ul style="list-style-type: none"> <li>- Kontinuert justering af krav &amp; prioritering</li> <li>- Minimering af risici</li> <li>- Vedvarende feedback</li> </ul>
Vandfaldsmetode	Vandfald metode er ikke brugbart, og er ikke godt, forklares mere i dybde i nedenstående kapitler.
<b>Use Case</b>	
Use Case	<p>Use case er en metode, der bruges til at analysere, identificere, afklare og organisere systemkrav.</p> <p><b>!VIGTIGT!</b></p> <ul style="list-style-type: none"> <li>- <b>Use cases skal være defineret ved et udsagnsord eller navneord, ofte 2 ords udsagnsord. (Må ikke være for stort et scope, f.eks. overtag herredømmet)</b></li> <li>- <b>Use case Beskrivelse, viser IKKE use case diagram.</b></li> <li>- <b>Use case diagram viser IKKE forløbet i en use case.</b></li> </ul>
Use Case Drevet	Vigtigste egenskaber som kunden prioriterer
Use case modeller	<p>De grundlæggende krav beskrives</p> <p>Generelt modeller med aktører.</p> <ul style="list-style-type: none"> <li>- Use case model</li> <li>- Sekvens diagram</li> </ul> <p>Implementations diagram</p>
Use case beskrivelse	<p>Der forskellige detaljeringsniveauer til use case beskrivelser:</p> <ul style="list-style-type: none"> <li>- Brief <ul style="list-style-type: none"> <li>o Kort tekst om primært scenarie</li> </ul> </li> <li>- Casual <ul style="list-style-type: none"> <li>o Flere afsnit evt. med flere forløb.</li> </ul> </li> <li>- Fully dressed <ul style="list-style-type: none"> <li>o Alle detaljer, pre- og postconditions.</li> </ul> </li> </ul>
Funktionelle Krav	<b>Funktionelle krav er at:</b>
Ikke funktionelle krav	<p>Hvad systemet kunne, f.eks. verificerer pinkode.</p> <ul style="list-style-type: none"> <li>- At programmet fungerer, og gør dens funktion.</li> </ul> <p><b>Ikke funktionelle krav er:</b></p> <p>Hvordan gør den det?</p> <ul style="list-style-type: none"> <li>- Usability</li> <li>- Reliability</li> <li>- Performance</li> <li>- Supportability</li> <li>- Lovkrav</li> <li>- Pålidelighed</li> </ul>
Alternativ Flow	Hvis ens standard use case ikke virker, opretter man en alternativ flow.
Branching	<p>Udgrening af hvad der forekommer i ens programmering, f.eks. alternativ flows, eller main flow, hvor det går op i switch statements.</p> <p>Det vigtigste med branching er:</p> <ul style="list-style-type: none"> <li>- Dokumenter kun de vigtigste</li> </ul>

	- Fejl, Forgreninger, og interrupts (afbrydelser)
Forretningsmodel	Grundlæggende model for forretningen
Supplerende specifikationer	Beskrivelse af ikke funktionelle krav Supplerende efter man har fået de funktionelle krav. F.eks.: - Krav
Ordbog	Ordliste over firmaets forkortet systemer eller ord, også kaldt terminologi for nøgleområder.
Prototyper	Tidligere produkter, til formål for at opstille krav
Faseplan og projektplan	Overordnede mål estimeringer defineret udefra alle iterationer i planen.
Iterationsplan	d. 1 fase i elaboration i unified process, her skal alt planlægges helt præcist, så man ved hver eneste detalje, ift. pris og tidsplan.
Procesudviklingsplan	Beskrivelse af unified process faserne ( Inception, elaboration, construction, transition )

## Risiko

Risikoliste og risikoplan	Risiko identifikation og forebyggelsestiltag, herunder Risk management, risk håndtering, risk analyse, risiko montering, risiko planlægning
Risiko analyse ( Risk Rating)	En table defineret ved <b>probability x impact</b> , hvori man sorterer dem efter score, og betegner dem i farveklasser, ift. Hvad frekvensen og katastrofisk en risiko kan være.
Risk Management	Projekt risks - effect schedule or ressources Product risks - effect the quality or performance of the software Business risks - effect the organisation
Risk håndtering (risk management process)	Processen I hvordan man håndterer risk management: <b>Risk identifikation</b> - Identify project, product and business risks. <b>Risk analysis</b> - Assess the likelihood and consequences of these risks. <b>Risk planning</b> - Draw up plans to avoid or minimise the effects of the risk. <b>Risk monitoring</b> - Monitor the risks throughout the project.
Risiko montering	Indenunder Risk montering, I Risk håndtering, kan man kigge på om der er en forandret risiko løbet af softwareprojektet. Dette gør man ved at: Vurder risiko - Fast Interval - Ændret probality? - Ændret impact? Formålet er at diskuter og reprioriter

pg. 4

	<ul style="list-style-type: none"> <li>- B aggregerer A</li> <li>- B indeholder A</li> <li>- B benytter A</li> <li>- B har initialiserende data for A</li> </ul>
Low Coupling ( Lav kobling )	<ul style="list-style-type: none"> <li>- En klasse skal være så uafhængig som muligt, den skal tildeles så lidt ansvar som muligt.</li> <li>- En klasse burde kun dække hvad den absolut skal dække, ikke for meget</li> </ul> <p>Formålet med lav Coupling er at man vil hellere vil ha for mange klasser end ik, da det skaber stor egenskab af modularitet.</p>
High Cohesion ( Høj samhörighed )	<p>En klasse skal have et veldefineret ansvarsområde, klasse har ansvar for nogle andre subklasser. Ligesom en form for stor kategori af klasser.</p> <ul style="list-style-type: none"> <li>- En klasse skal have et veldefineret ansvarsområde</li> <li>- En klasse skal ha et sæt operationer, der understøtter ansvarsområdet.</li> </ul>
Protected Variations	<p>Protected variations er at man har skabt klasser som ikke er alt for tilknyttet eller afhængig til andre klasser, dette er underbygget af low coupling, og sørger for stor moduralitet, og lettere vedligeholde af kode, hvis der går noget galt. Protected variations gør også:</p> <p>Beskyttelse af objekter mod ændringer i andre objekter</p> <ul style="list-style-type: none"> <li>- Hvis der er noget der kan ændre sig</li> <li>- Introducerer stabil grænseflade</li> <li>- Dette er ofte adaptorer, fordi de stadig protected selv hvis der sker noget</li> </ul>
Indirection	<p>“en form for mellemmand kommunikations-objekt”</p> <p>Hvis flere komponenter reagerer gennem hinanden, kan man lede dem gennem en hoved klasse, dette kalder man indirection</p> <ul style="list-style-type: none"> <li>- Formålet er at lave en lav kobling mellem klasser.</li> <li>- Database-adaptore, så det er lettere at skifte database.</li> </ul>
Polymorfi	<p>Polymorfi er at der sker noget ensartet i koden, at man arver en funktion, dette kan læses yderligere nedenunder, men det er generelt i form af arv det forekommer.</p> <p>Polymorfi kan også være ved:</p> <ul style="list-style-type: none"> <li>- Mange former</li> <li>- "Når flere typer af objekter kan noget ensartet"</li> </ul>
Controller	<p>Controller er hvad der uddeler ansvar rundt, og generalt hvad der uddeler kodeansvaret.</p> <p>Controller svarer til:</p> <ul style="list-style-type: none"> <li>- System klasse</li> <li>- Klasse som repræsenterer use case scenarie</li> <li>- Eksempelvis, "ProcessSaleHandler"</li> </ul>

	Controller er ofte ikke i domænemodellen. Generelt skal man passe på med ikke at have for store controllers.
Pure fabrication	<p>Pure fabrication er når man opretter noget der ikke er behov for, såsom hvis man har en klasse kun med getter eller setter, eller hvis man har en form for klasse, kun for transaction, eller noget som kun er til formål for at spille en rolle der ikke burde være der, dette kan være udviklet udefra lav kobling.</p> <p>Pure fabrikation kan også være:</p> <ul style="list-style-type: none"> <li>- Introduktion af en klasse der ikke findes i problem-domænet</li> <li>- Ofte for at opnå low coupling and high cohesion</li> </ul>
<b>Design elementer</b>	
Sekvensdiagram	<p>Beskeder mellem objekter, er baseret på metodekald på objekter.</p> <p>“Sequence diagrams are organized according to time. <b>The time progresses as you go down the page.</b> The objects involved in the operation are listed from left to right according to when they take part in the message sequence.”</p> <ul style="list-style-type: none"> <li>- Dynamisk diagram type (adfærd / Behaviour)</li> </ul>
klasse diagram	<p>Struktur af klasser, består af attributer, operates, associationer mellem klasserne.</p> <ul style="list-style-type: none"> <li>- Statisk diagram type</li> </ul>
Design klasse diagram	<p>Det er hvad Moiz betegner som en klasse diagram, det er hvad der består af:</p> <ul style="list-style-type: none"> <li>- Klassenavn</li> <li>- Attributer</li> <li>- Operatives (Metoder)</li> </ul>
Analyse klasse diagram	<p>Analyse klasse diagram er en domænemodel, det er overfladisk, og er beskrevet meget simpelt, der er henvist eksempler ved nedenstående kapitel.</p>
aktivitetsdiagram	<p>Aktivitetsdiagram, viser flowet i en process.</p> <ul style="list-style-type: none"> <li>- <b>Kodenært</b></li> <li>- Ofte noder med if-else statements, eksempler på dette kan være: <ul style="list-style-type: none"> <li>• Yes &amp; No</li> <li>• Read / Write &amp; Read only</li> <li>• Listen &amp; Not listen</li> </ul> </li> <li>- adfærds diagram type (dynamisk)</li> <li>- Aktivitetsdiagrammer har <b>ALTID</b> swimlanes</li> <li>- noder</li> </ul>
Tilstandsdiagram ( state machine diagram )	<p>Overgange mellem tilstande, f.eks. en dinosaur, som har tilstandene, æg, dinosaur, fossil, olie.</p> <p>Et tilstandsdiagram er ogs:</p> <ul style="list-style-type: none"> <li>- adfærds diagram type (dynamisk)</li> <li>- Viser et forløb</li> </ul>

	- Har ikke decision noder, og merge noder
Singleton Pattern	Singleton pattern: static, global sansynlighed:
Navneordsanalyse	<p>Navneordsanalyse, er hvor du tager en kundes bedrifter, udformet i normalt sprog, og bruger navneordene til at udstille ens analyse klassediagram. (samme som domænemodel)</p> <p>"Vi vil gerne have et system der kan administrere vores <b>badmintonbaner</b>. Vi har <b>spillere</b> og <b>trænere</b> der kan <b>booke</b> banerne. Der er <b>faste intervaller</b> man kan booke baner i. Trænerne kan booke alle baner i en <b>hal</b>. Trænerne booker først. Spillerne kan booke de resterende baner."...</p> <p>Så vil de røde hhv. Blive brugt til domæneklasser, også efter kan det gule blive muligvis brugt til en usecase.</p>
miscellaneous	
Sprint	Hvad vi forventer indtil næste møde
Scrum	Arbejde i sprint af 30 dags varighed
Abstrakt Klasse	<b>!VIGTIGT! Abstrakt klasse er altid i kursiv, og KAN ikke intiasteres</b>
Stakeholder / Interresant	<p>Hvem der er interreseret i produktet:</p> <ul style="list-style-type: none"> <li>- En person der er påvirket af projektet</li> <li>- En person der påvirker projektet</li> <li>- En person der bruger systemet</li> <li>- En person der betaler for systemet</li> </ul>
Use relation	Vist ved en stiplet linje med <<USE>>
Visibility	Alt der indeholder protected, public, private
Visibility : Protected ( # )	At den kan nedarves
Visibility : Static / statisk ( linje nedenunder / ___ )	Beskrives ved at den kun kan bruges i klassen.,
Extend	<p>Extend er uafhængigt, skrives i kode som A extender B.</p> <ul style="list-style-type: none"> <li>• <b>"The extending use case is dependent on the extended (base) use case.</b> In the below diagram the "Calculate Bonus" use case doesn't make much sense without the "Deposit Funds" use case."</li> <li>• <b>"The extending use case is usually optional</b> and can be triggered conditionally. In the diagram, you can see that the extending use case is triggered only for deposits over 10,000 or when the age is over 55."</li> <li>• <b>"The extended (base) use case must be meaningful on its own.</b> This means it should be independent and must not rely on the behavior of the extending use case."</li> </ul> <p>Kilde: <a href="https://create.ly.com/blog/diagrams/use-case-diagram-relationships/">https://create.ly.com/blog/diagrams/use-case-diagram-relationships/</a></p>
Include	Er afhængig af hvad den peger på.



	<ul style="list-style-type: none"><li>• “The base use case is incomplete without the included use case.”</li><li>• “The included use case is mandatory and not optional.”</li></ul> <p>Kilde: <a href="https://creately.com/blog/diagrams/use-case-diagram-relationships/">https://creately.com/blog/diagrams/use-case-diagram-relationships/</a></p>
Brugerkrav	F.eks. at systemet skal printe en rapport ud med oversigt af salg hver måned. <b>Læst af:</b> End-users (brugere), client managers, mm
Systemkrav	Hvad kravene er for systemet, <b>Læst af:</b> Software engineers, system architects.
Usability krav	Brugervenlighed i fokus, man kan teste det her med en brugertest.
Reliability Krav	Pålidelighed, hvor lang tid kan systemet holde.
Performance Krav	Response times, ressourceforbrug.
Supportability krav	Supportability, hvilken programmer eller OS skal du have kørende.
Hardware krav	Er CPU, GPU, RAM, Lager godt nok til at køre programmet.

## Unified process

Unified Process er baseret ude fra en tilgang til at angribe software projekter an, det består af forskellige trin.

Unified process er use case drevet. Use case drevet betyder de vigtigste egenskaber som kunden prioriterer.

Formålet med unified process er at bygge det iterativt, altså trinvis, for at formindske spildt kode.

### !VIGTIGT! – Der kan defineres krav i alle faser

### Grundlæggende principper for Unified Process:

De grundlæggende principper for Unified process er:

- Iterativ og inkrementel
- Use Case Drevet
- Risiko Drevet
- Arkitektur Centret

### Faserne i Unified process er

Inception, elaboration, construction, transition.

### Disciplinerne i Unified Proces

Business Modeling, Requirements, Analysis and Design, Implementation, Test, Deployment, Configuration and Change Management, Project Management, and Environment.

Der findes Iterativ development og waterfall method, iterativ er generelt bedst, og er derfor man bruger unified process.

### Inception

Inception er det korteste forløb, og indeholde:

- En vision på softwareprojektet
- En Forretningscase, hvori der defineres omfanget og omkostningen

- Projektplanlægning
- Kan softwareprojektet gennemføres?
- Skal arbejdet fortsættes?

## Elaboration

Formålet med elaboration, er at det her man opdækker den størstedel af krav specifikationen, det primære mål for elaboration er:

- Risk faktorer
- Godkende system arkitektur
- Use case diagrammer
- Conceptuel diagrammer ( klasse diagrammer, med de vigtigste ting)

Det sidste i elaboration fasen burde gi en fastsat plan for hvor mange penge og tid der skal bruges i Construction fasen.

På det her tidspunkt i faserne i unified process, burde planen være helt præcist, fordi inception er baseret udefra tidligere erfaring, eller projekter, herunder de største risikofaktorer som kan ødelægge tidsplanen. Hvorimod elaboration, inkorporerer de mindste risikofaktorer.

## Construction

Construction er den største fase, det her alle system features burde implementeres i små time-boxed iterationer, det her man skriver full-text use cases.

Her bruger man UML-diagrammerne:

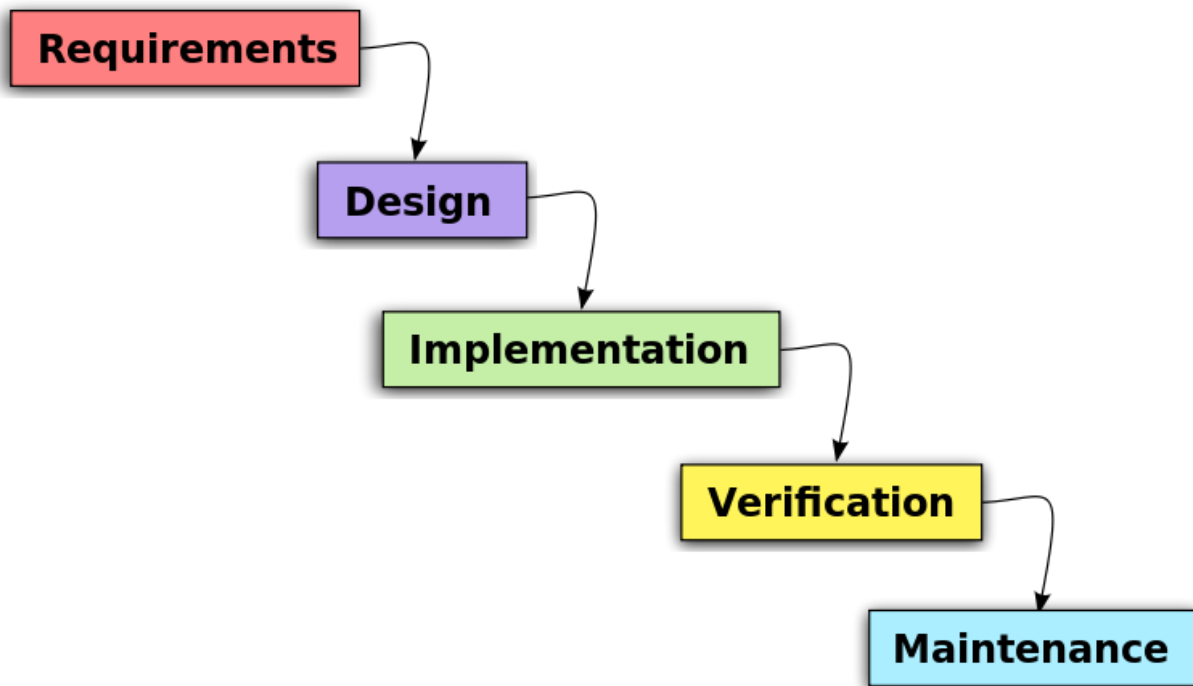
- Aktivitets diagrammer ( activity diagrams )
- Sekvens diagrammer ( sequence diagrams )
- Tilstands diagrammer ( state transition diagrams )
- Interaktions diagrammer ( interaction overview diagrams )

## Transition

Her udleveres det endelig software projekt, til slutbrugerne. Her får man feedback fra ens versioner, i denne fase raffinerer man ens arbejde, og laver finjusteringer.

## Modellering for Vandfaldsmodellen

Nedenunder kan der ses



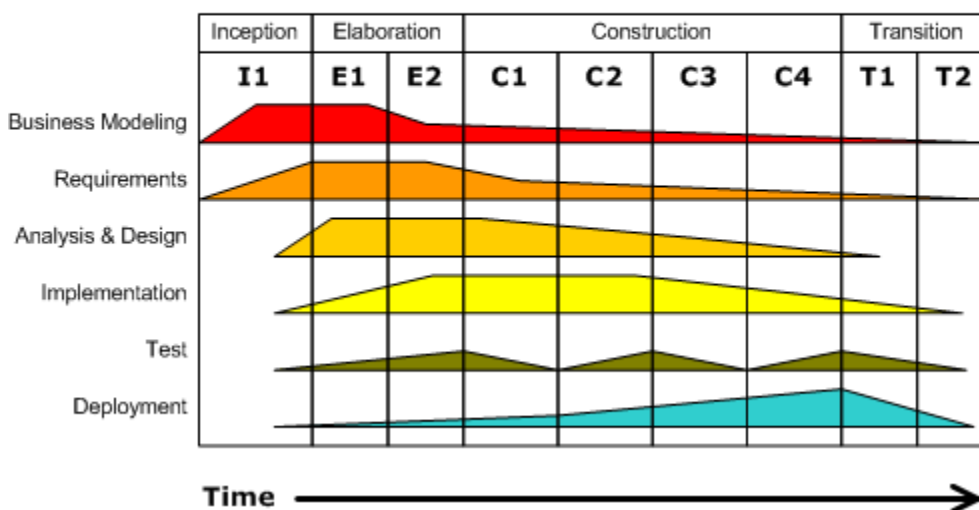
### Modellen for Iterativ Development (Iterativ udvikling)

I hver iteration, i hvert af faserne i unified process, består hovedsageligt af:

- Krav, Analyse, Design, Implementering, Test

#### **Iterative Development**

Business value is delivered incrementally in time-boxed cross-discipline iterations.



- Iterationerne er opdelt i faste timeslots
  - o de er timeboxed (2-6 uger)
  - o delopgaverne er fleksible
- Alle discipliner i hver iteration
- Hver ender fungerede program

### Rational Unified Process ( RUP )

4 Faser

- Projektets modenhed, (som vist ovenstående, er faserne Inception, Elaboration, Construction, Transition)

- 6 ingeniører
- 3 support
  - o Configuration and change management
  - o Project management
  - o Environment

## Modeller

Der findes forskellige slags modeller, de forskellige modeller bruges til at vise modelleringen af et software projekt.

Forskellige modeller er f.eks.:

- Domæne
- Test
- Brug

## Diagrammer

Diagrammer bruges til at få et indblik i modellen.

Diagrammer kan bl.a. hjælpe med at:

- Systemet ses fra et bestemt synspunkt
- Systemet på et bestemt detaljeringsniveau
  - o Generelt er tommelfingerreglen 50 elementer i et diagram.

## Analyse og Design

Analyse

- Undersøgelse af problem domæne og krav
- Hvad?

Design

- Konceptuel Løsning
- Hvordan?

## Krav-specifikation

## FURPS+

## Gruppering af krav - FURPS+

- Functionality -
  - Capability (Size & Generality of Feature Set), Reusability (Compatibility, Interoperability, Portability), Security (Safety & Exploitability)
- Usability (UX) -
  - Human Factors, Aesthetics, Consistency, Documentation, Responsiveness
- Reliability -
  - Availability (Failure Frequency (Robustness/Durability/Resilience), Failure Extent & Time-Length (Recoverability/Survivability)), Predictability (Stability), Accuracy (Frequency/Severity of Error)
- Performance -
  - Speed, Efficiency, Resource Consumption (power, ram, cache, etc.), Throughput, Capacity, Scalability
- Supportability
  - (Serviceability, Maintainability, Sustainability, Repair Speed) - Testability, Flexibility (Modifiability, Configurability, Adaptability, Extensibility, Modularity), Installability,
- + (Constraints)
  - Design, Implementation, Interface, Physical, Legal

## Prioritering af krav - MosCoW:

## Prioritering af krav - MosCoW

- Must Have
  - Flyet skal kunne lande
- Should have
  - Flyet bør have sæder
- Could have
  - Flyet kan have servering
- Would be nice to have
  - Flyet kunne have virtual reality til alle passagerer

Angiver Implementerings-rækkefølge!

- (Tit kommer man alligevel til at implementere Nice to have's)

## Use Cases

Use case er en metode, der bruges til at analysere, identificere, afklare og organisere systemkrav.

- Use cases skal være defineret ved et udsagnsord eller navneord, ofte 2 ords udsagnsord.

Grunden til man bruger use cases, er fordi:

- Brugernes perspektiv er indspillet
- Nemmere for kunden at sætte sig ind i hvad der bliver lavet i softwareprojektet
- Dette medfører, kundedeltagelse, som sikrer færre fejl.
- Kunder kan selv sikre use cases, da det er forståeligt.
- Formindsker risikoen for at miste fokus

- Krav beskrives i en sammenhæng, der skaber en forståelse rød tråd.

**!Vigtigt! – Use case diagram, indeholder ingen forløb.**

## Use case beskrivelser

Der forskellige detaljeringsniveauer til use case beskrivelser:

- Brief
  - o Kort tekst om primært scenarie
- Casual
  - o Flere afsnit evt. med flere forløb.
- Fully dressed
  - o Alle detaljer, pre- og postconditions.

Eksempel på et use case beskrivelse:

Nedenstående er en "casual" use case beskrivelse.

Use case: FindProduct
ID: 3
Brief description: The system finds some products based on Customer search criteria and displays them to the Customer.
Actors: Customer
Preconditions: None.
Main flow: <ol style="list-style-type: none"><li>1. The use case starts when the Customer selects "find product".</li><li>2. The system asks the Customer for search criteria.</li><li>3. The Customer enters the requested criteria.</li><li>4. The system searches for products that match the Customer's criteria.</li><li>5. If the system finds some matching products then<ol style="list-style-type: none"><li>5.1 For each product found<ol style="list-style-type: none"><li>5.1.1. The system displays a thumbnail sketch of the product.</li><li>5.1.2. The system displays a summary of the product details.</li><li>5.1.3. The system displays the product price.</li></ol></li></ol></li><li>6. Else<ol style="list-style-type: none"><li>6.1. The system tells the Customer that no matching products could be found.</li></ol></li></ol>
Postconditions: None.
Alternative flows: None.

Fully dressed use case, indeholder postconditions, pre conditions og alternative flows.

## Identifikation af aktører og use cases

Use case er at sige hvilke primære funktioner aktøren handler udefra?

- Formålet med use casen er at involvere kunden i ens programmering, så de forstår processen, Og så der kan ske mindre fejl i formålet med programmet.

Aktører:

- Hvem har brug for supporten i deres daglig dag?
- Hvem bruger use cases?
- Hvilke eksterne systemer, kommunikerer ogs med use casesne?
- Hvem er interreret i resultatet af software projektet?

*Alternativ flow*

Alternativ flow, er hvis man skal vælge en separat flow fra den standard use case ikke fungere.

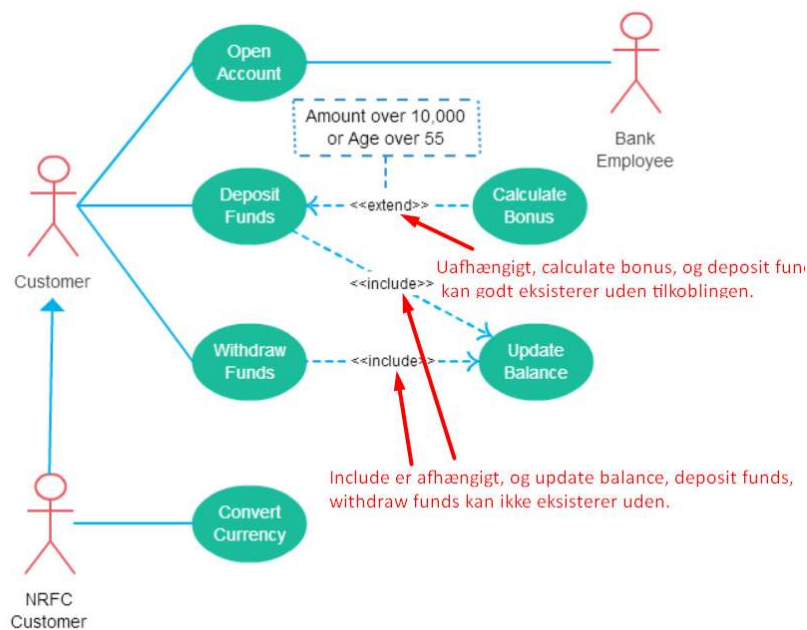
**Interresanter / Stakeholders**

- Interresanter / Stakeholders
  - Påvirker eller Påvirkes af systemet
  - En der enten bruger eller betaler for systemet

**Relation mellem use cases:**

Relation mellem use cases:

- Include
- Extend
- Arv



*Includes is usually used to model common behavior*

**Include:**

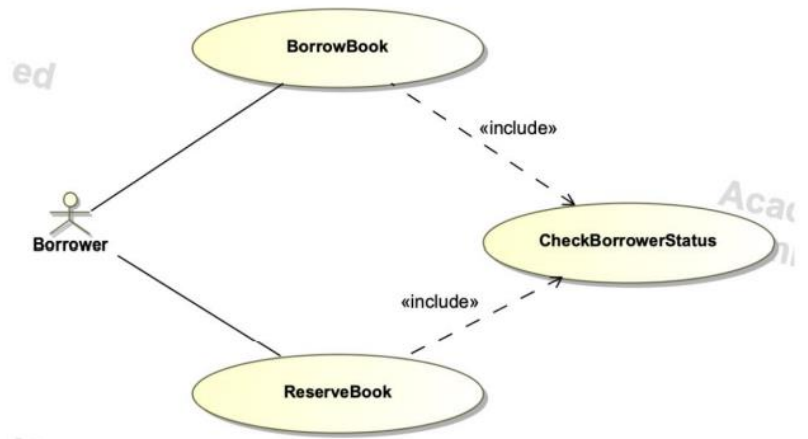
Er afhængig af hvad den peger på.

- "The base use case is incomplete without the included use case."
- "The included use case is mandatory and not optional."

Kilde: <https://creately.com/blog/diagrams/use-case-diagram-relationships/>

# Include

- Use casen indeholder en 'sub-use case'
- Kan være delt med andre use cases
- Kan introduceres ved komplekse use cases
- Er altid med i use casen
- noteres med stiplet pil TIL sub-use casen
- <<include>>



## Extend:

Extend er uafhængigt, skrives i kode som A extender B.

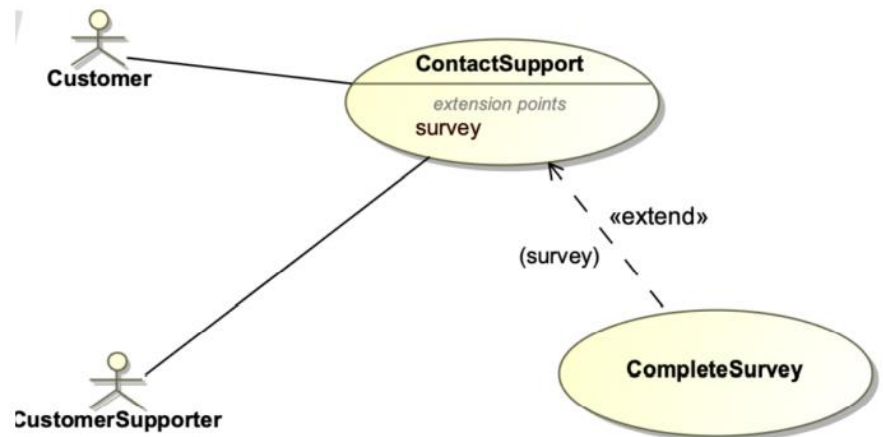
- **“The extending use case is dependent on the extended (base) use case.** In the below diagram the “Calculate Bonus” use case doesn’t make much sense without the “Deposit Funds” use case.”
- **“The extending use case is usually optional** and can be triggered conditionally. In the diagram, you can see that the extending use case is triggered only for deposits over 10,000 or when the age is over 55.”
- **“The extended (base) use case must be meaningful on its own.** This means it should be independent and must not rely on the behavior of the extending use case.”

Kilde: <https://creately.com/blog/diagrams/use-case-diagram-relationships/>



# Extend

- Use casen KAN indeholde en 'sub-use case'
- Kan være delt med andre use cases
- Kan introduceres ved komplekse use cases
- Kan være med i use casen
- extension point
- noteres med pil FRA sub-use casen



## Risikoanalyse

Herunder vises der forskellige former for risikoanalyser, herunder hvad det gavner.

**!VIGTIGT! Husk at rationel Unified process ( RUP ) er risikodrevet.**

## Risk Management

Risk amnagement er til at vise de forskellige former for risk der forefindes i management.

### Projekt risks

- effect schedule or ressources

### Product risks

- effect the quality or performance of the software

### Business risks

- effect the organisation

## Risk Håndtering ( Risk Management process )

Risk håndtering, er hvordan man håndterer de forskellige former for risici.

### Risk identifikation

- Identify project, product and business risks.

### Risk analysis

- Assess the likelihood and consequences of these risks.

### Risk planning

- Draw up plans to avoid or minimise the effects of the risk.

### Risk monitoring

- Monitor the risks throughout the project.

### Risk analyse ( Herunder Risk Rating )

I Risk analysen, kigger man på risk rating, som indeholder en tabel over probability x impact

Hvori man sorterer efter score, og laver en rating udefra frekvens og katastrofi.

Modellen for risk rating:

## Risk Rating = Likelihood x Severity

<b>S e v e r i t y</b>	Catastrophic	5	5	10	15	20	25
	Significant	4	4	8	12	16	20
	Moderate	3	3	6	9	12	15
	Low	2	2	4	6	8	10
	Negligible	1	1	2	3	4	5
			1	2	3	4	5
			Improbable	Remote	Occasional	Probable	Frequent
			<b>Likelihood</b>				

Catastrophic	STOP
Unacceptable	URGENT ACTION
Undesirable	ACTION
Acceptable	MONITOR
Desirable	NO ACTION

### Risiko Montering

Risiko montering er en gren af risk rating, altså hvor man vurderer risk rating.

Vurder risiko:

- Fast Interval
- Ændret probability?
- Ændret impact?

Diskuter og reprioriter

Nedenstående er liste over ting der kan ske når man risiko monterer.

Risk type	Potential indicators
Estimation	Failure to meet agreed schedule; failure to clear reported defects.
Organizational	Organizational gossip; lack of action by senior management.
People	Poor staff morale; poor relationships among team members; high staff turnover.
Requirements	Many requirements change requests; customer complaints.
Technology	Late delivery of hardware or support software; many reported technology problems.
Tools	Reluctance by team members to use tools; complaints about software tools; requests for faster computers/more memory, and so on.

## Risiko Strategier ( Risiko planlægning )

Transfer

- Videregiv ansvaret til en der har styr på det

Accept

- Tag den givne risici

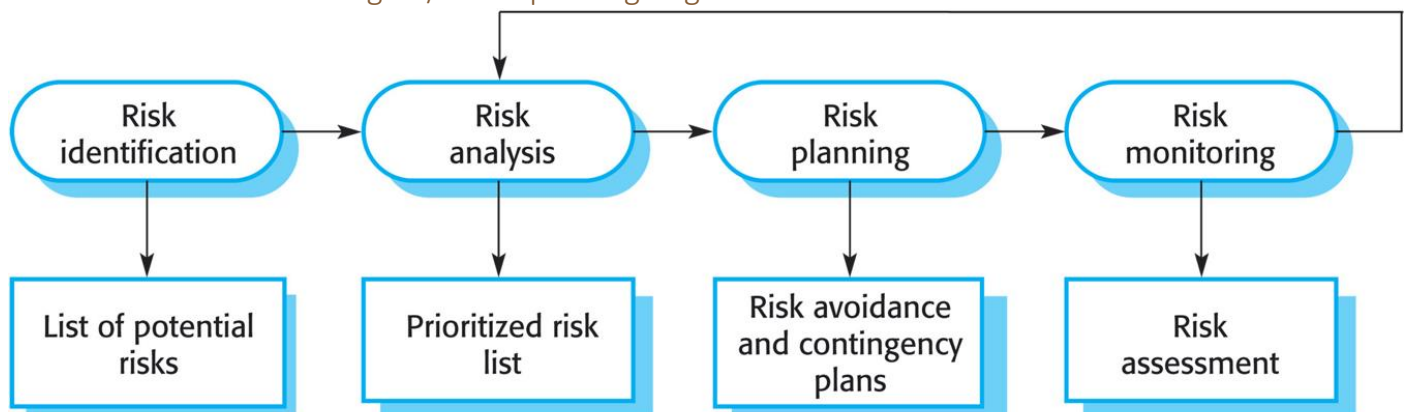
Mitigate

- Minimer konsekvenserne af risici

Eliminate

- Fjern problemet

Modellen for Risiko strategier / risiko planlægning:



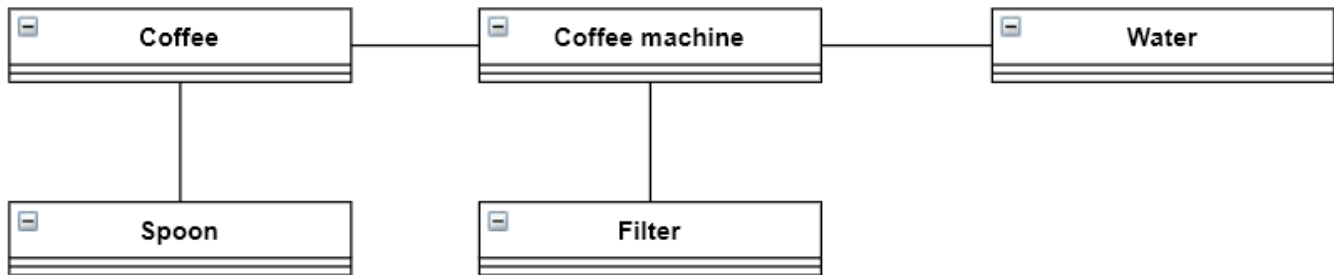
Copyright ©2016 Pearson Education, All Rights Reserved

## Domænemodeller ( analyse klassediagram )

I UML, bruges domæne modeller til at modellere på objekt model niveau. Altså hvordan et system fungerer, såsom hvis du skal lave en kaffe.

## Eksempel på simplificeret domænemodel

Nedenunder er der vist et eksempel på en simplificeret domænemodel. (uden attributter, associationer, kun domæner)



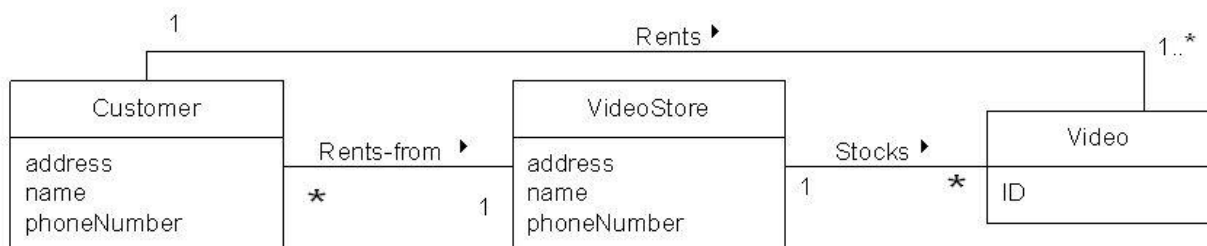
Domænemodeller er kendetegnet ved:

- Domæner, bestående af kasser, med underkasser hvor der står attributer
- Attributer, datatyper
- Associationer, herunder hvad associationen mellem to domæner er
- Multipliciteter, hvad og hvor meget der interageres mellem to domæner, to sidet.

## Eksempel på partiel domænemodel (Partiel Domain model)

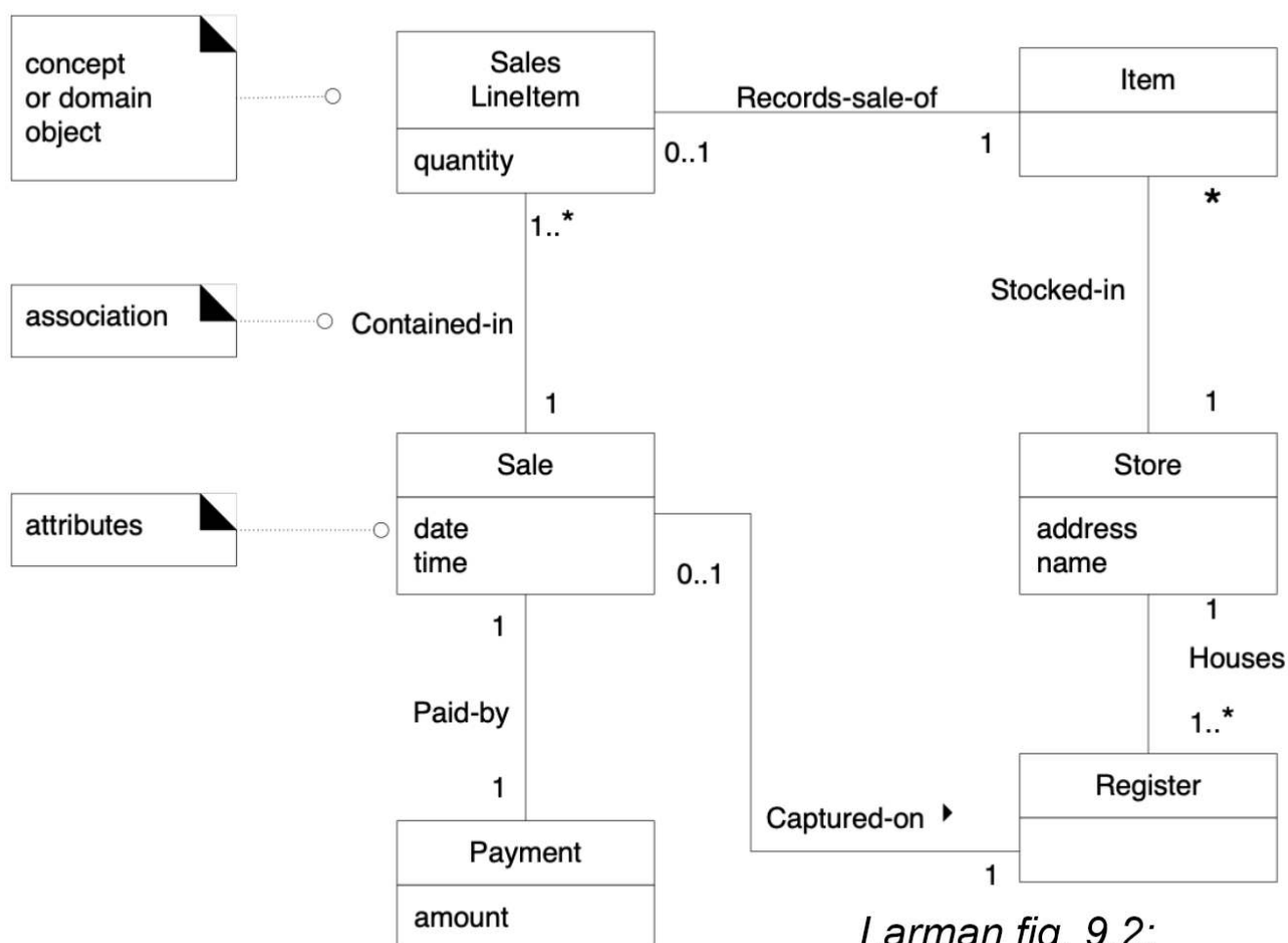
# EXAMPLE:

## • Partial Domain model



## Eksempel på udvidet domænemodel:

Nedenstående er der et eksempel på en domænemodel af en butik:



## Sekvensdiagrammer ( design sekvensdiagram )

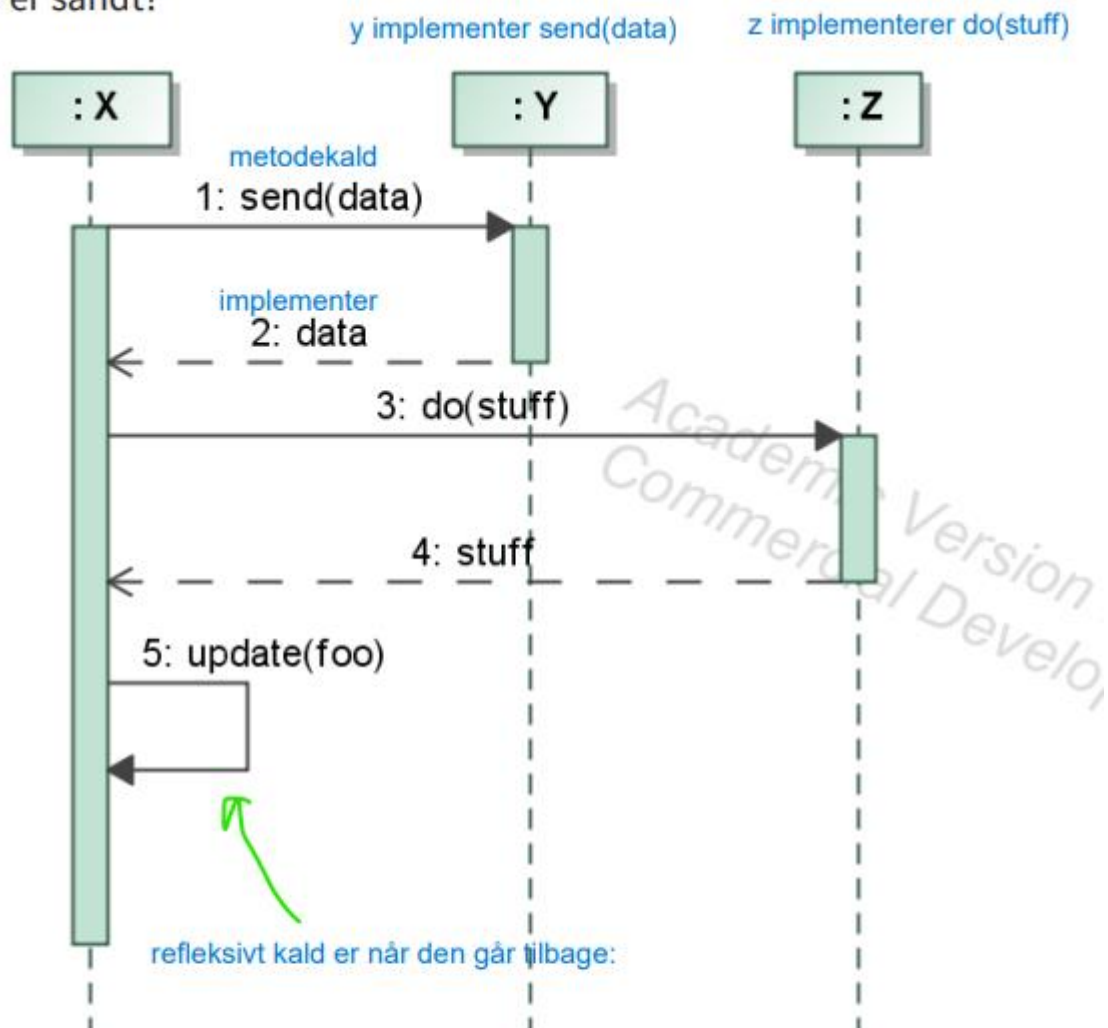
Er det samme som et design sekvensdiagram

Sekvensdiagrammer er en dynamisk model, viser proces interaktioner mellem to klasser, kunne også beskrives ved "modeller et specifikt scenarie"

Ofte kun del af en handling.

## Eksamensspørgsmål:

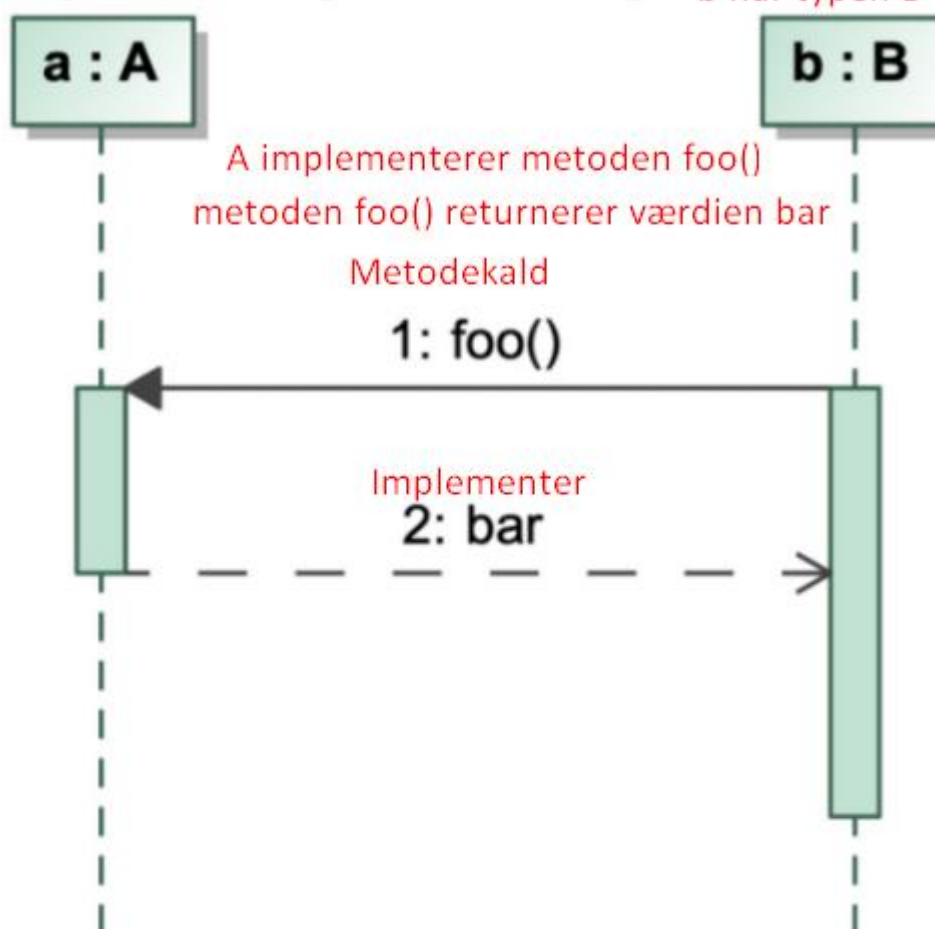
Hvad er sandt?



Choose one answer

- ☐ Objektet af typen Y laver et refleksivt kald
- ☐ Objektet af typen Y kalder objektet af typen Z
- ☒ Z implementerer do(stuff)
- ☐ Objektet af typen X kalder objektet af typen Y asynkront
- ☐ X implementerer send(data)

Sandt eller falsk?



Select the correct answers

sandt

falsk

A implementerer metoden foo()



b har typen B



Objektet a kalder metoden bar på objektet b



metoden foo() returnerer værdien bar



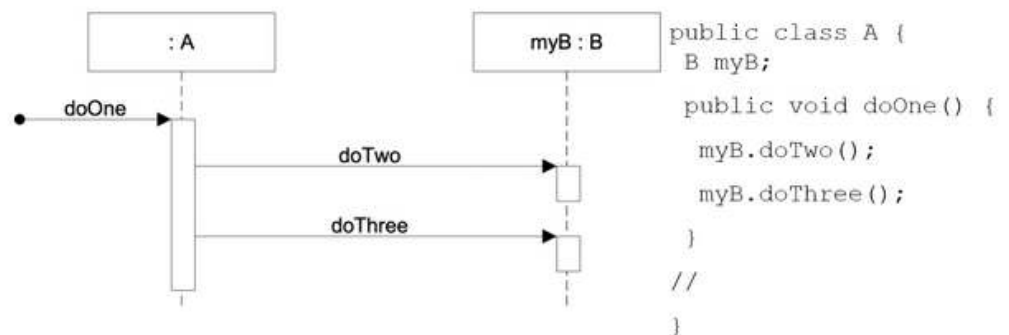
Klassen B kalder metoden foo() på klassen A



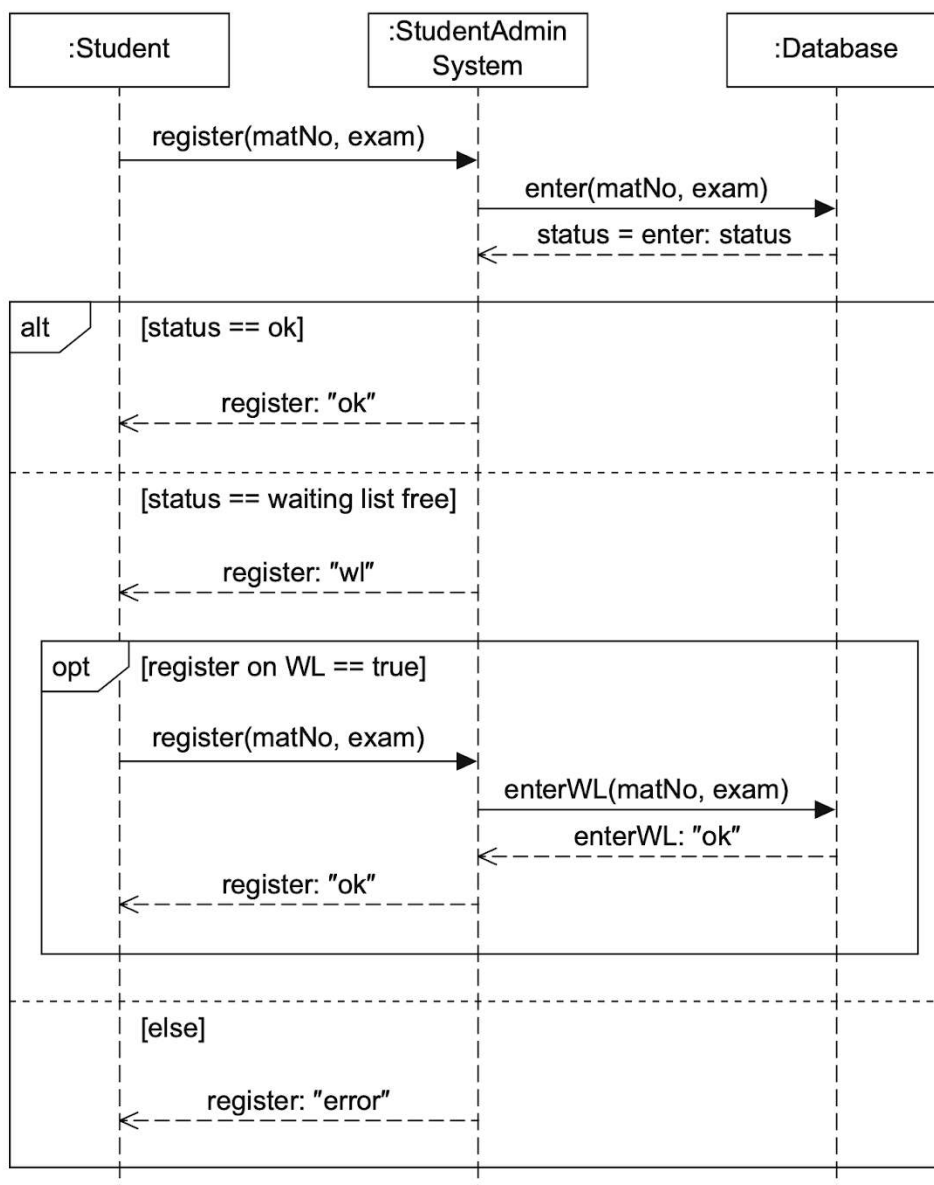


Design Sekvensdiagrammer oversat til kode:

## Design sekvensdiagrammer



Kompliceret eksempel af systemsekvensdiagram:  
(skriv forklarende tekst)



**Figure 6.10**  
Example of an alt and an  
opt fragment



## Design Klasse

Design klassen er hvad man bruger til at designe klasse i programmeringssprog, og forklarer hvilken association de har med hinanden, og hvordan klasserne består.

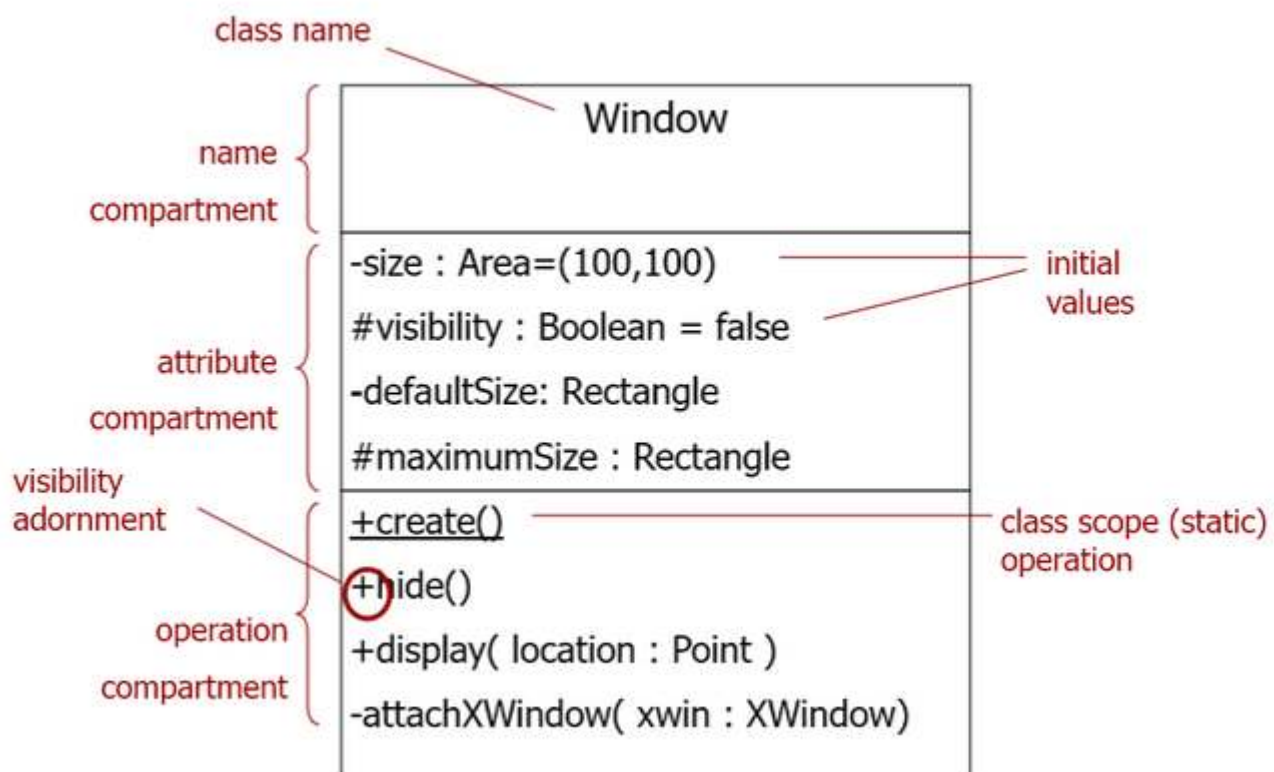
### Opstilling af design klasse

En design klasse består således:

Class name
<ul style="list-style-type: none"> <li>Attributter (Attributter er variablenavne på klasseniveau) Hvad der skal implementeres programmeringsmæssigt</li> <li>Typer Int, strings, doubles osv</li> <li>Synlighed Public, public, private</li> </ul>
<ul style="list-style-type: none"> <li>Metoder</li> </ul>

### Eksempel på design klasse:

Nedenstående kan der ses et eksempel på en design klasse:



**!VIGTIGT!** Husk alt ikke skal være inkluderet, kun det væsentlige.

### access modifiers

nedenunder kan der ses forskellige access modifier af hvad en værdi eller klasse kan være.

<b>public</b>	<b>+</b>	anywhere in the program and may be called by any object within the system
<b>private</b>	<b>-</b>	the class that defines it
<b>protected</b>	<b>#</b>	(a) the class that defines it or (b) a subclass of that class
<b>package</b>	<b>~</b>	instances of other classes within the same package

## Analyse

### Associationer

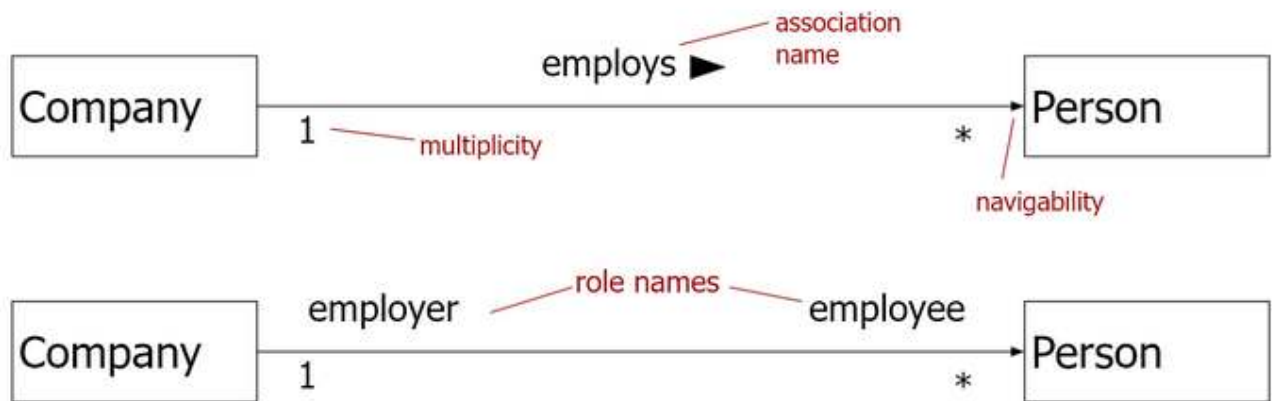
- Associationer er forbindelser mellem klasser
- Relation mellem objekter af klassens type

Associationen angives ved:

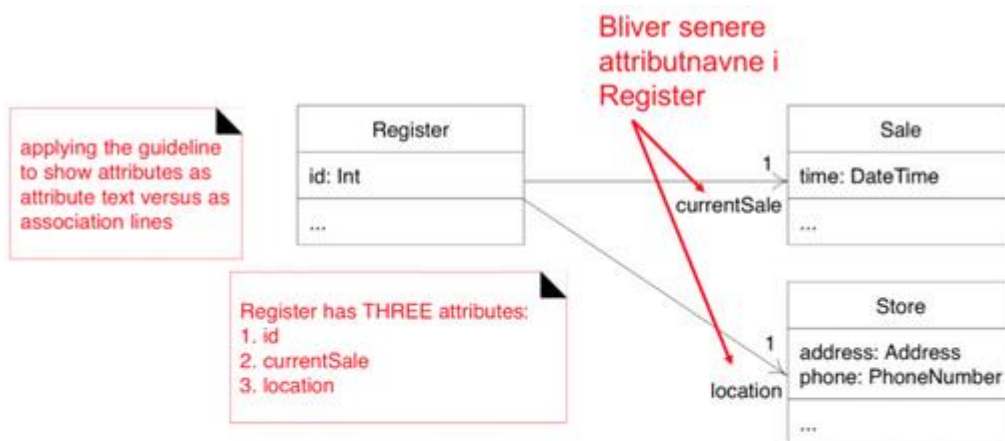
- Linje
- Navn
- Multiplicitet
- Evt. retning

### Eksempel på Associationer

Nedenstående er et eksempel over associationer, multiplicitet, navigability, association navn.



**!VIGTIGT!** Nedenstående kan man se at der er tilknyttet 3 attributes til register, hvad enden der står ovenover eller nedenunder multiplicitet er et attributnavn. Dette er en tommelfingerregel.



Brug:

- attributnavn ved simple datatyper
- associeringslinie til klasser

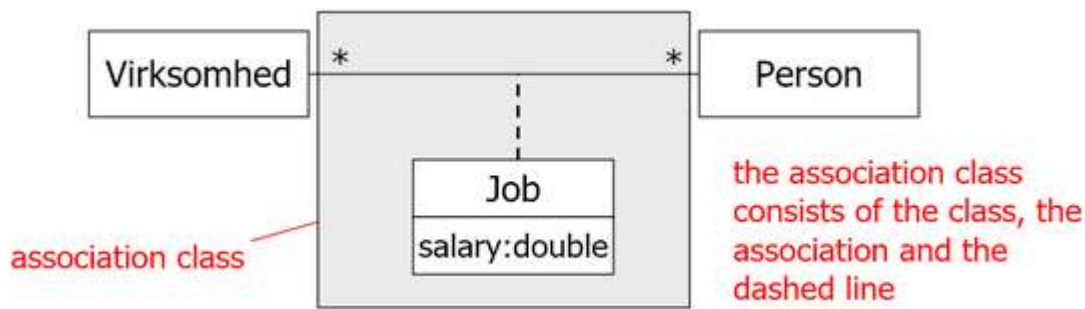
Larman fig. 16.5

## Associationsklasse

Associationsklasse, er når der er en klasse decideret kun for associationen.

Det er en instans for hver relation.

Det markeres med en stiplede linje, eksempel på associationsklasse, kan ses nedenunder:



## Multiplicitet

Multiplicitet er hvad beskriver relationen mellem klasserne.

Et eksempel på en multiplicitet her er:



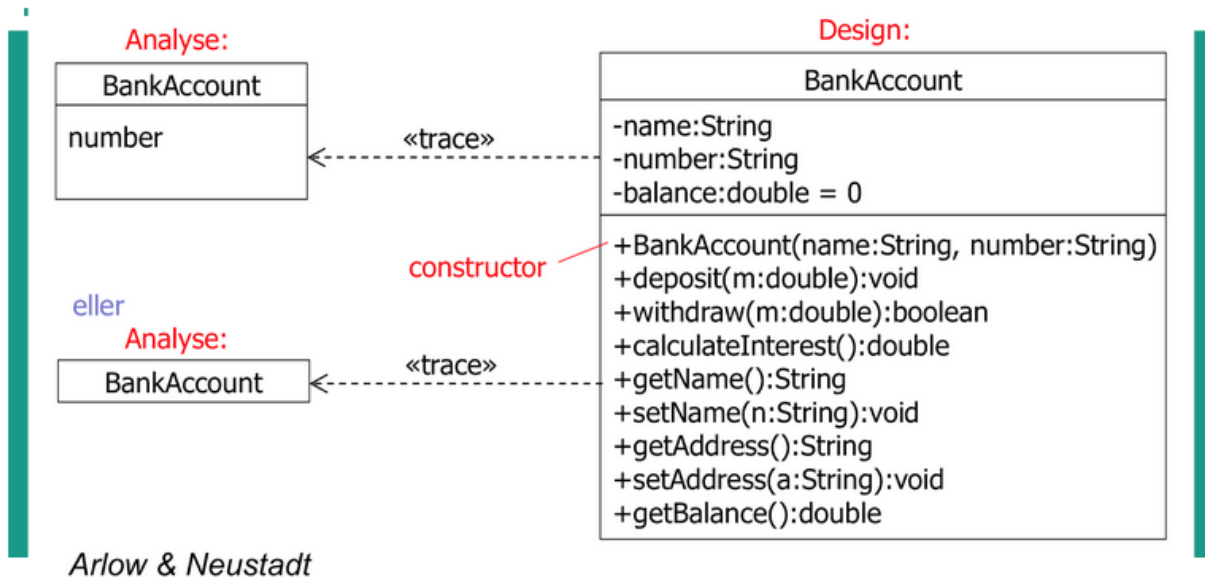
Her står der at, en company kan have uendelig mange personer, men at en person kun kan have en company, titlen over multipliciteten beskriver rollen der iagttages.

Multiplicitet syntax	
0..1	Zero to 1
1	Præcist 1
0..*	Zero til uendelig
*	Zero til uendelig
1..*	1 til uendelig
1..6	1 til 6

## Analyse vs Design

Analysen består af en kort boks, hvori design klassen er specificeret.

# Analyse vs Design klasser



Statiske og dynamiske modeller:

## Statiske og dynamiske modeller

- Dynamisk (interaktionsdiagrammer dvs. sekvens og kommunikations diagrammer)
  - Viser logik, og opførelse i koden
  - Ofte dynamiske diagrammer der tager længst tid at implementere, da det kræver en del tænkning)
  - Benyttelse af grasp i dynamiske diagrammer
  - Viser noget over tid og i bevægelse
- Statisk (klasse og pakke diagrammer)
  - Klasse diagrammer og pakke diagram
  - Benyttes til at designe strukturen / arkitekturen af programmet
    - Klasse navne
    - Attributter
    - Metoder
  - Ikke noget logik

## Klassediagrammer

Klassediagrammer benyttes til statisk modellering, altså ikke dynamisk.

Analyse:

- Klassediagrammer består af domænemodeller, klassenavn og evt. attributter.

Design:

- Designmodel
- Klassenavn, attributter og operationer.

En design model kan indeholde 10-100 gange flere klasser end analysemodellen.

## Konvention for klasse & Objekt:

En klasse er en arbejdstegning, hvori et objekt er en instans af en klasse.

En klasse skal indeholde følgende:

- Stort forbogstav
- Navneord
- Ental

Eksempelvis: `Public class Student{}`

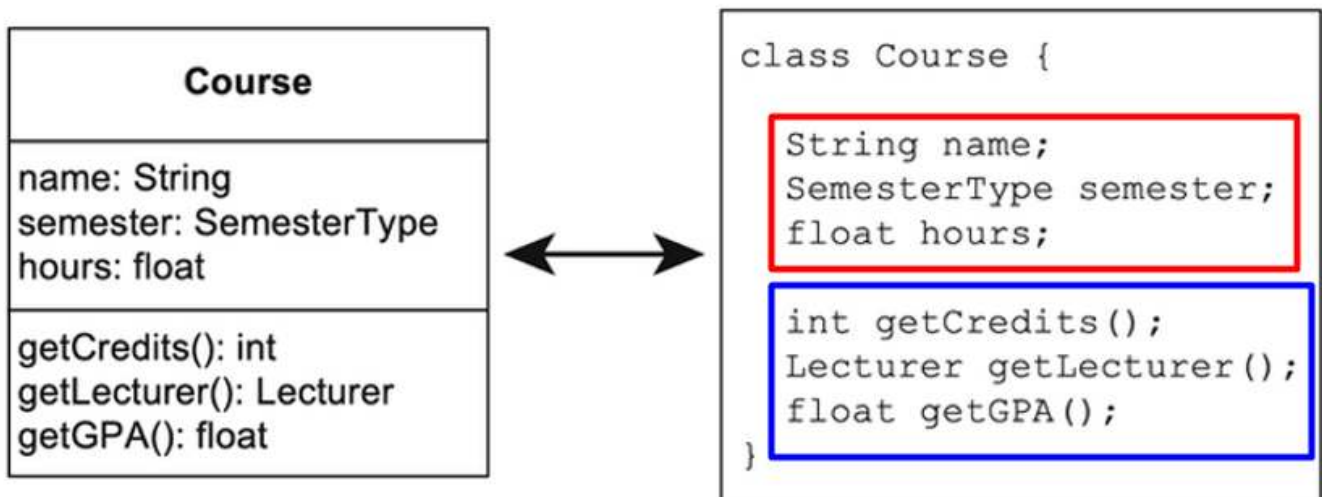
Objekt:

- Lille bogstav, ligesom andre variable

## Design klasse til kode eksempel:

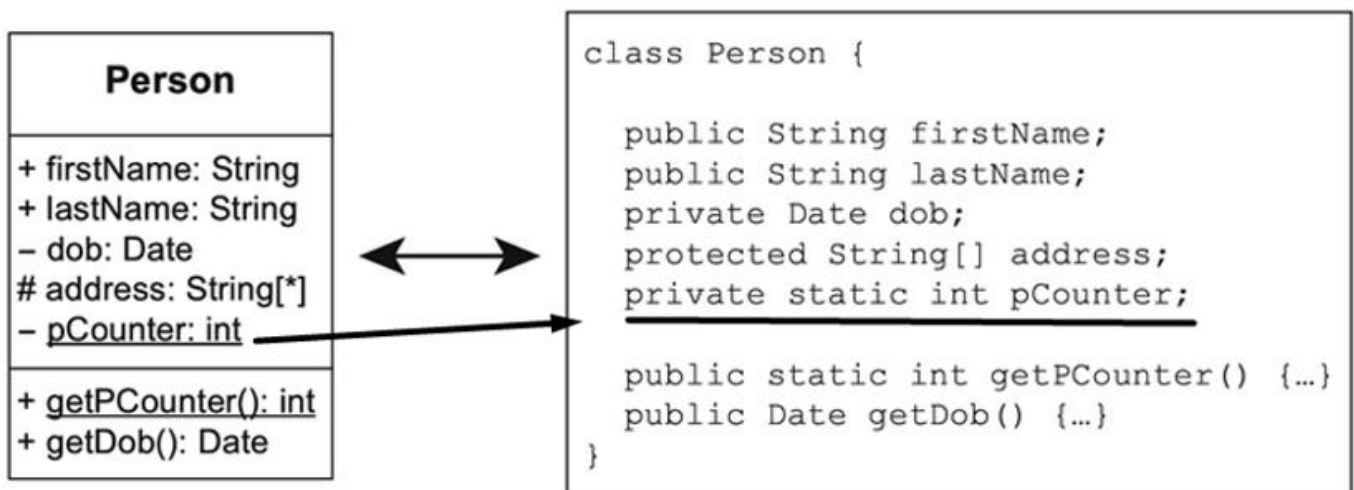
### Eksempel 1:

Når design implementeres, ser det således ud, man kan se at course er klassen, der er attributter, i det røde felt. Hvori getcredits er i form af (), med det blå felt.



### Eksempel 2:

Linjen under ordet betyder static



Attributter:

## Attributter

- **Navn**, **Type**, evt. multiplicitet, **default værdi**, **visibility**
  - **visibility** **name** : **type** multiplicity = **default** {property string}
- Eks:
  - - passengerList : String [0..\*] = checkedInList {ordered}
  - - age : int
  - + GRAVITY : double = 9.81 {readOnly}

## MVC (Model View Controller-lagdeling)

- Design Pattern

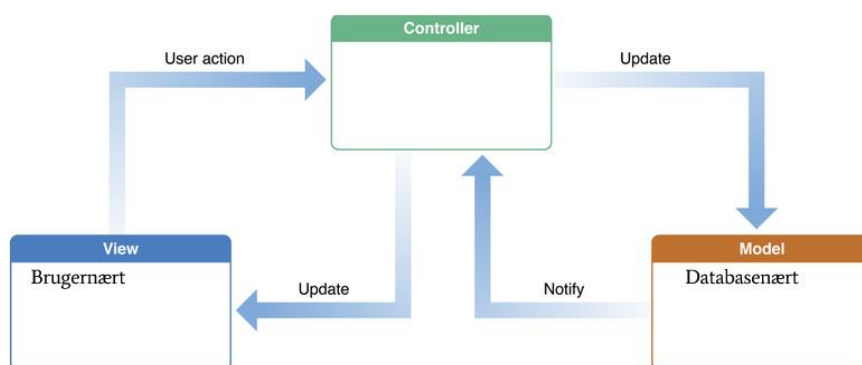
- Objekter har roller

- Et design, som er let at forstå, let at teste, samt let at vedligeholde og udvikle.

### Formål med MVC :

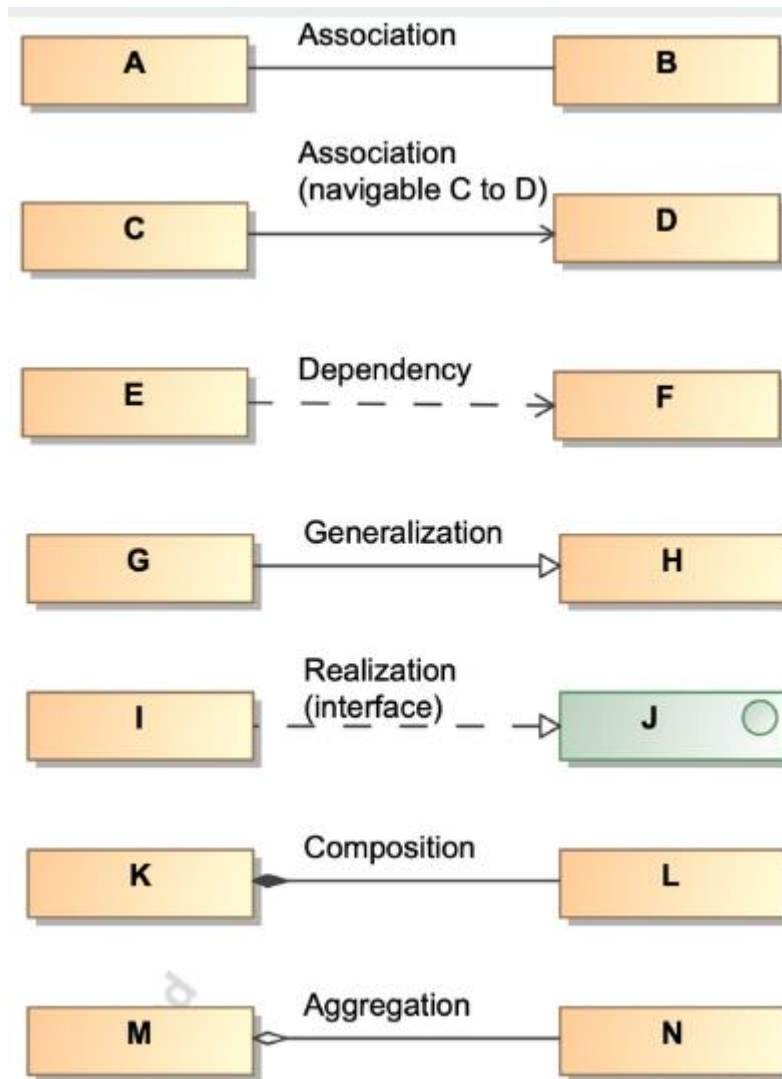
Formålet med MVC er at:

- Objekter er allokeret en rolle (enten model, view eller controller), med et defineret svar.
- Opnår lav kobling og høj samhørighed
- Opnå et design som er
- Et at forstå
- Let at teste
- Let at vedligeholde



## Relationer i klassediagram

Forskellige former for relationer mellem bokse i klassediagrammer:



Disse ting bliver henvist i nedenstående underkapitler.

## Association

Hvis to klasser skal kommunikere med hinanden, så skal der være et link mellem dem, dette kaldes en association.

- Associationer er forbindelser mellem klasser
- Relation mellem objekter af klassens type

Associationen angives ved:

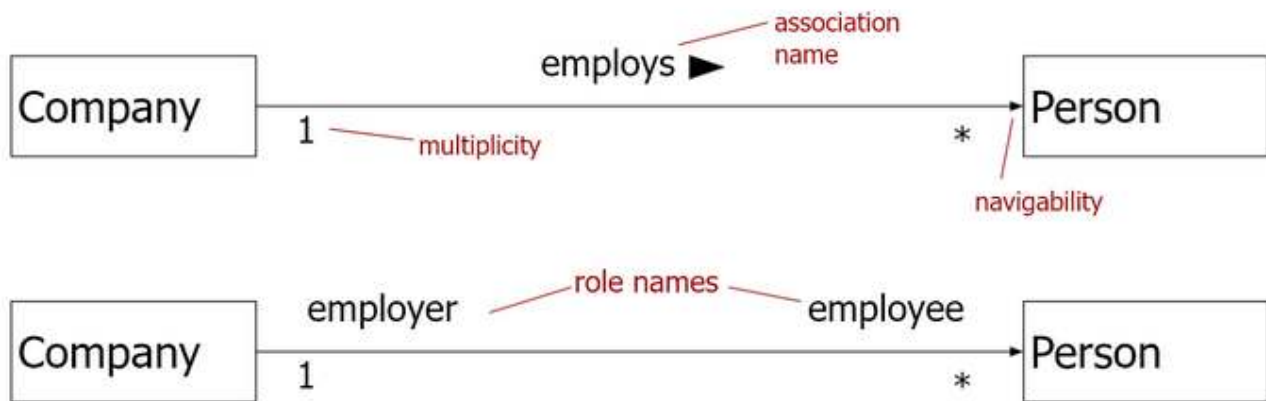
- Linje
- Navn
- Multiplicitet
- Evt. retning

## Association (navigable C to D)

Når der en pil i association.

Associationer med navigability



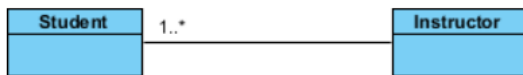


Godt Eksempel på Association:

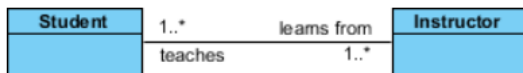
A single student can associate with multiple teachers:



The example indicates that every Instructor has one or more Students:

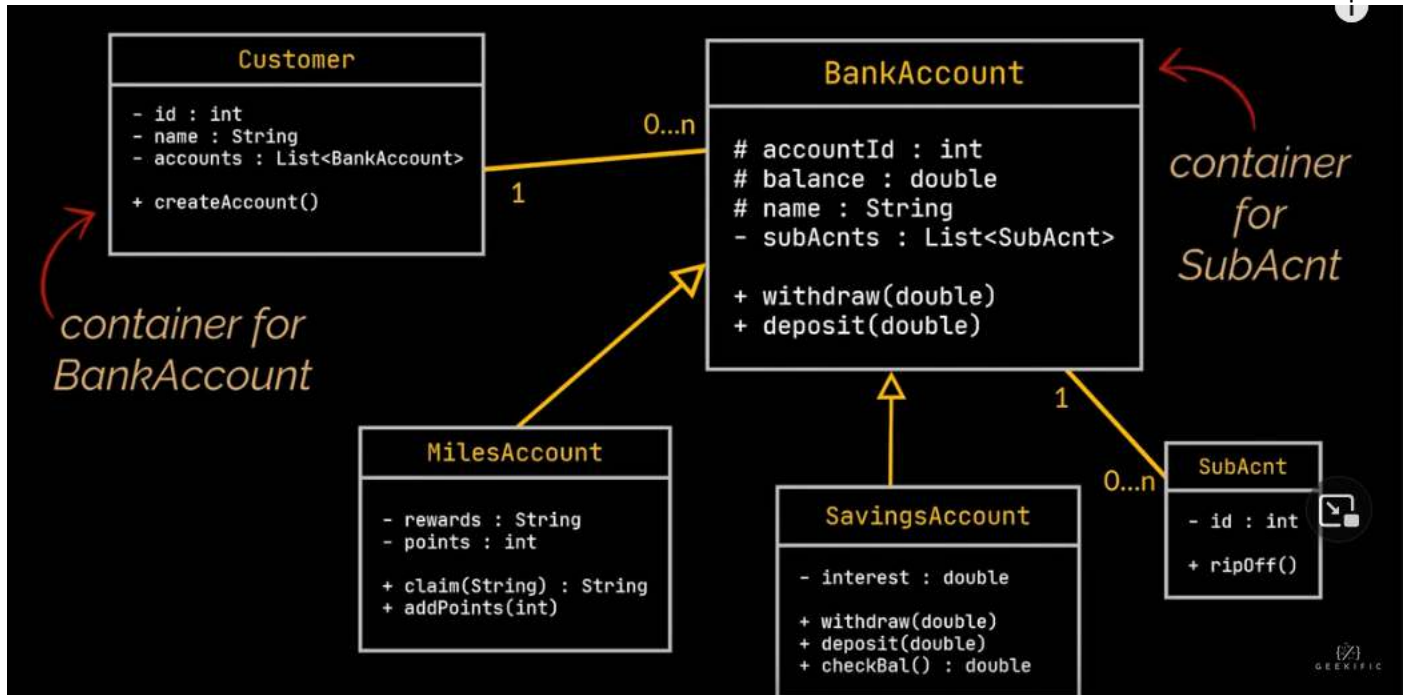


We can also indicate the behavior of an object in an association (i.e., the role of an object) using role names.



Endnu bedre eksempel på association:

I nedenstående eksempel, kan der ses at MilesAccount, Savingsaccount, er generaliseret (også kaldt nedarvet) fra bankaccount, herunder kan der ses at en bank account kan have en customer, og en customer kan have 0 til uendelig mange bankkontoer. Samt at bank account kan have 0 til uendelig mange sub accounts, og at en sub account kun kan have en bank account.



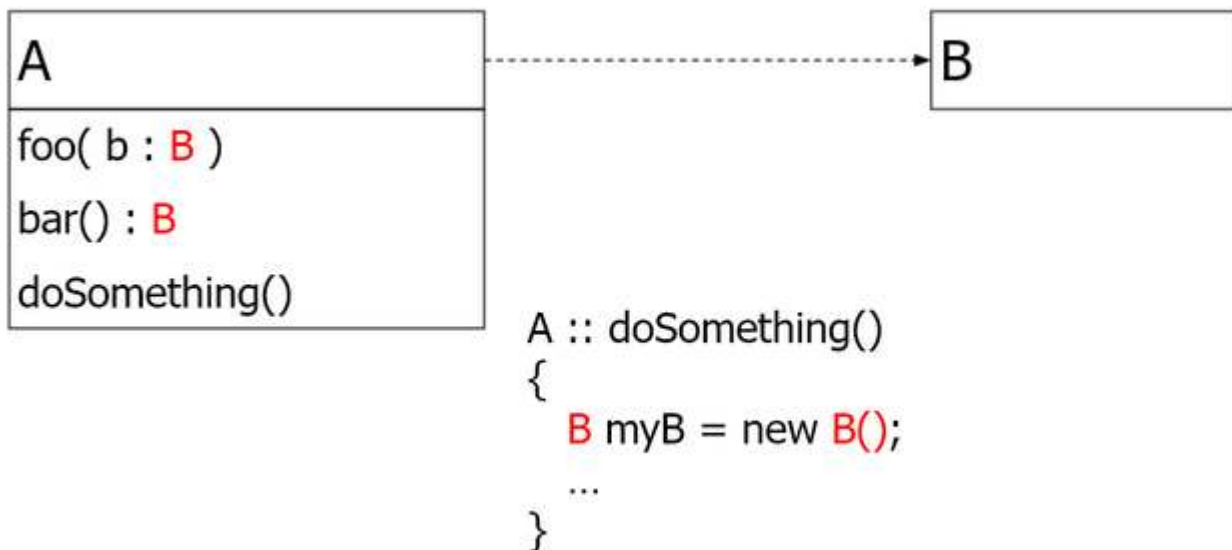
## Dependency (afhængighed)

Dependency kaldes også light-association.

- Dependency er en midlertidig reference.
- Ændring i en klasse påvirker en anden

Dependency pilen er defineret ved stiplede linje, og betyder afhængig af, altså ligesom at den har en new statement af en attribute. Som henviset ved nedenunder eksempel:

Her kan der ses hvordan en dependency bliver oversat i koden:



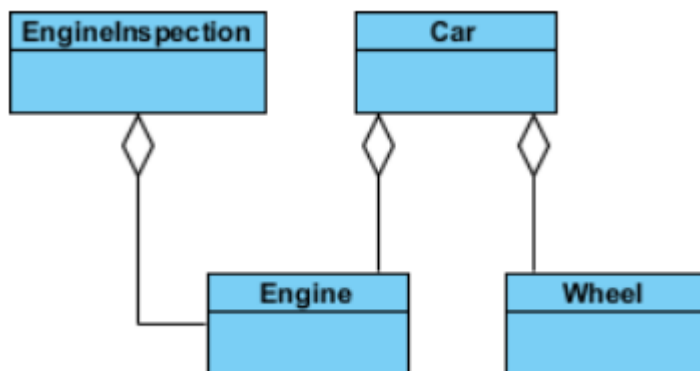
## Aggregation

Aggregation er når der er en relation mellem to klasser stadig kan eksistere stadig hvis hvis klassen med aggregationen på negligeres.

Aggregation er når der er dele til et fuldt system, der er masser af eksempler nedenunder:

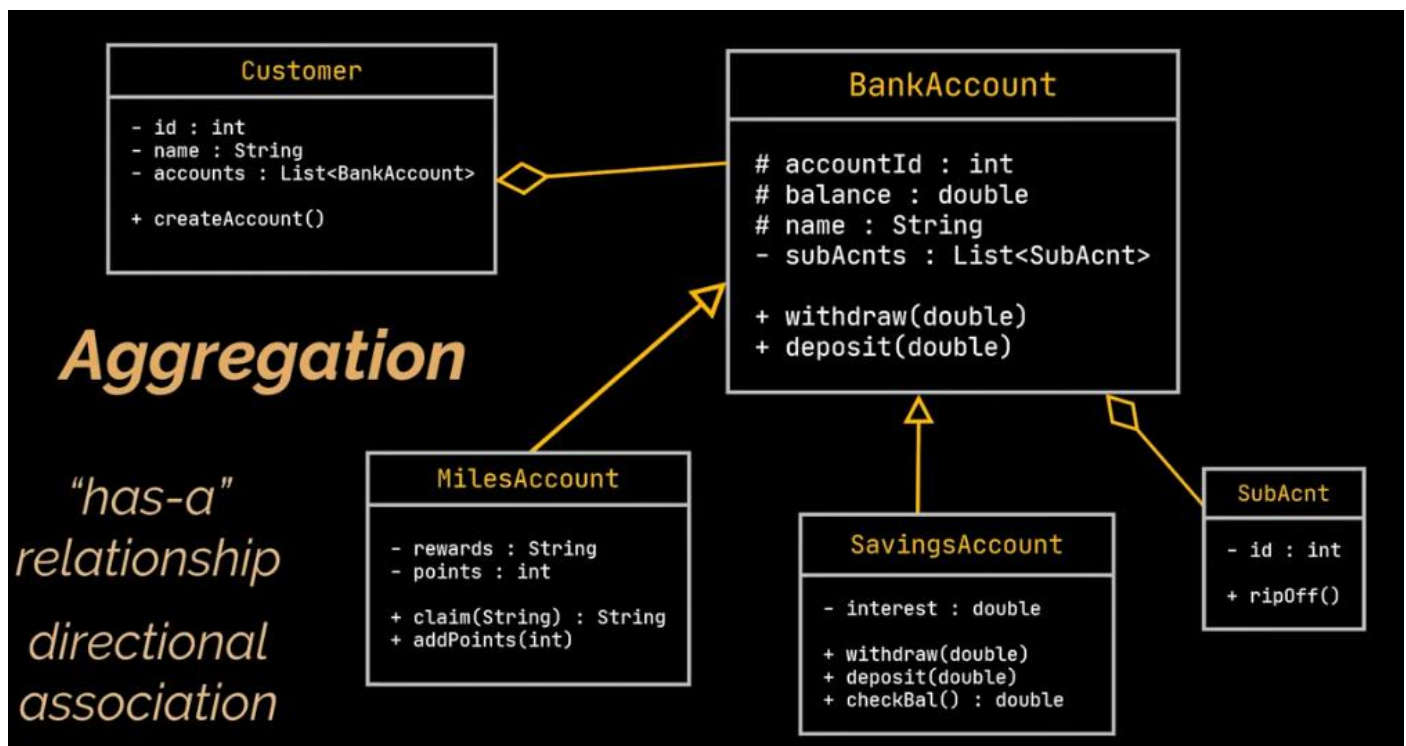
Læg mærke til aggregation lægger på systemboksen, og ikke på delene. Delene kan godt eksistere uden hinanden.

F.eks. hvis du sletter klassen car, eksisterer wheel stadig, samt gøre engine.



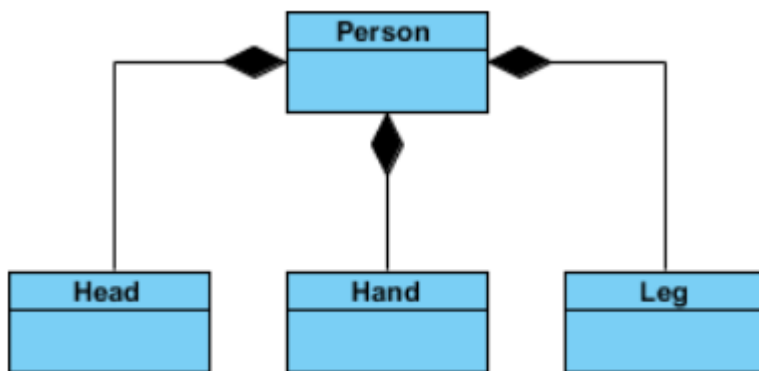
Godt eksempel på Aggregation:

I nedenstående eksempel kan der ses, at hvis Customer klassen slettes, forefindes BankAccount og dens nedarvet systemer stadig.



## Composition

Composition ligesom Aggregation er når der er dele til et fuldt, men de er dependant, så hvis man sletter en klasse, bliver hele de forbundet klasser også slettet.



Virkelig godt eksempel på composition og aggregation forskel:



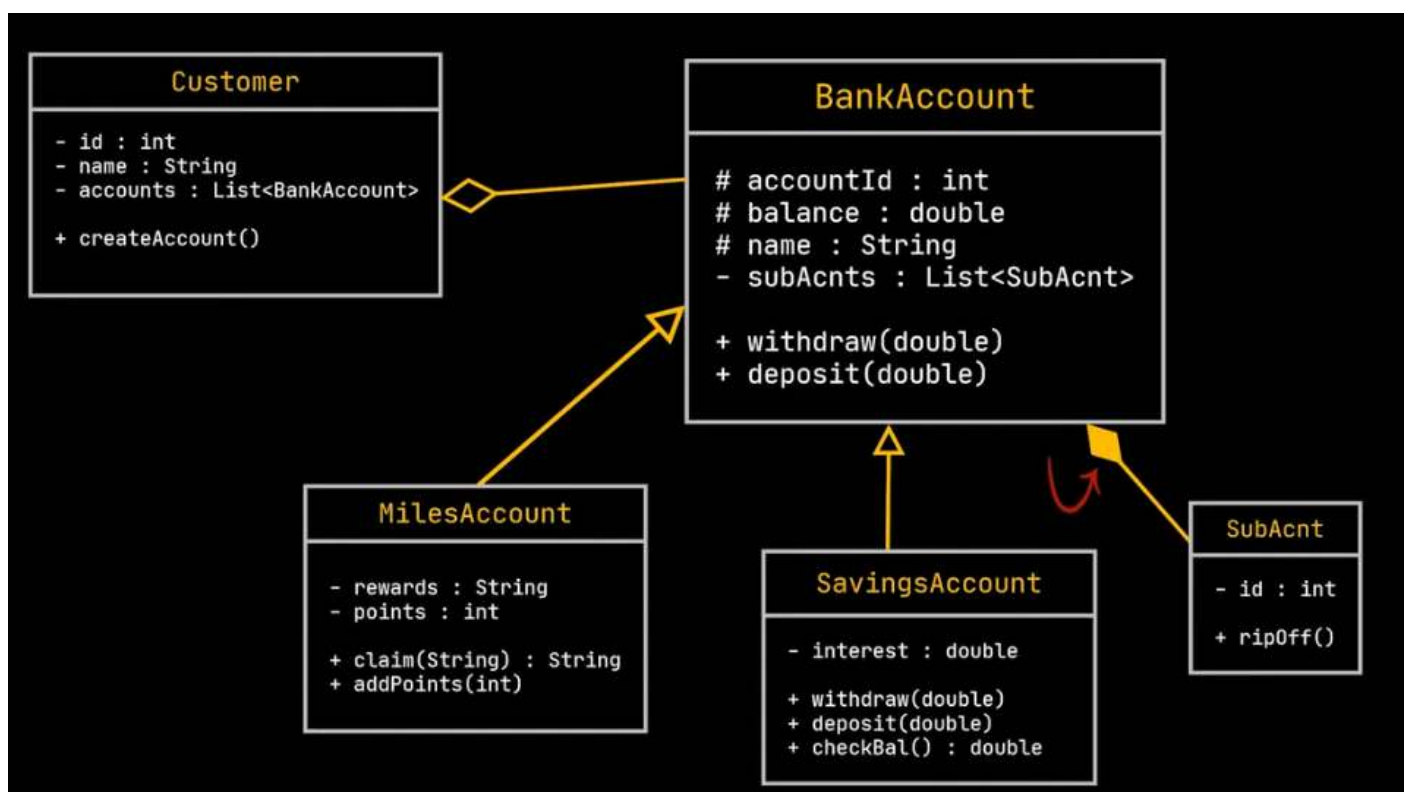
Composition: every car has an engine.



Aggregation: cars may have passengers, they come and go

Godt eksempel på composition:

Nedenstående vises der at SubAcnt ikke kan eksisterer uden bank account, teknisk set er en composition det samme som aggregation, men hvis hvad compositionen er ved, stopper med at eksisterer, er der intet formål med den enden enden at eksisterer, i denne kontekst SubAccount, nemlig fordi så har den ingen brug.



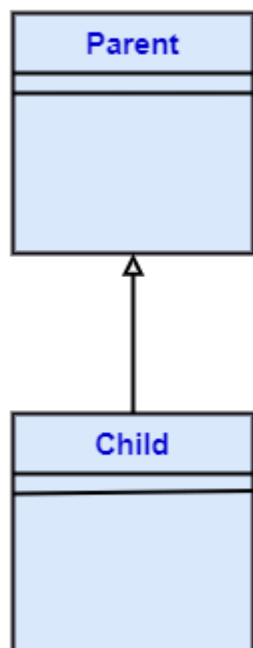
Her kan man se at head, hand, og leg er en del af person, men hvis du sletter leg går person også, samt hand og head.

### Generalization ( generalisering)

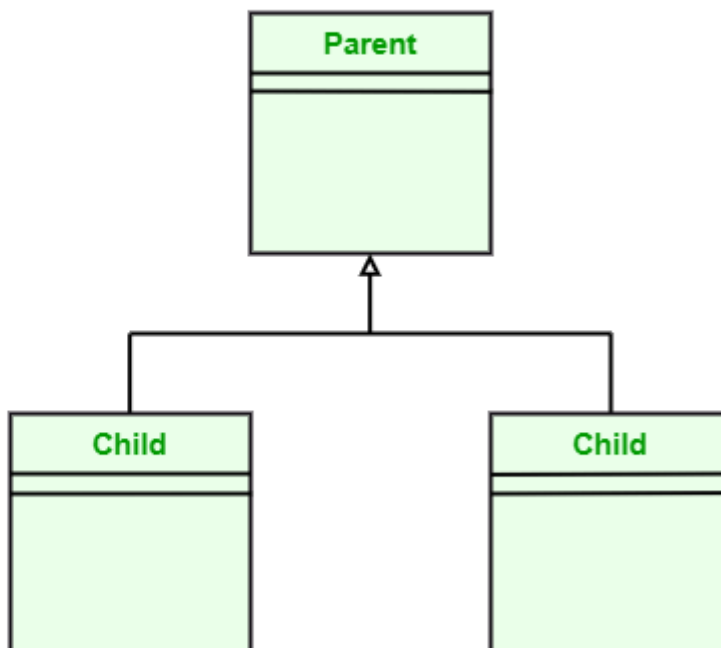
Generalization er når der nedarves noget

f.eks. kan en child ikke eksisterer uden en parent. Det basically nedarvelse.

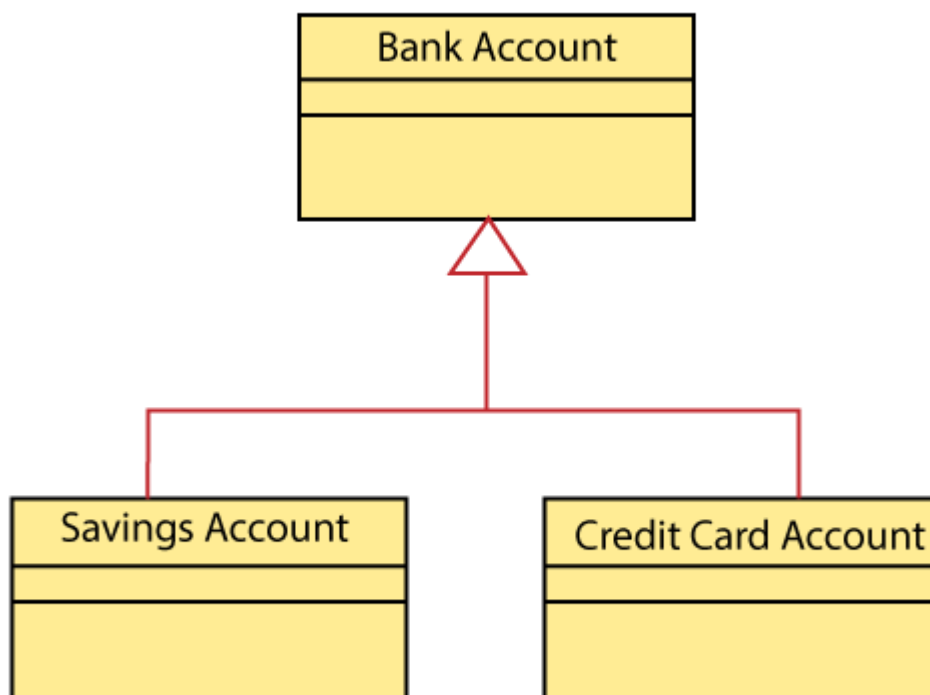
## Single Inheritance



## Multiple Inheritance



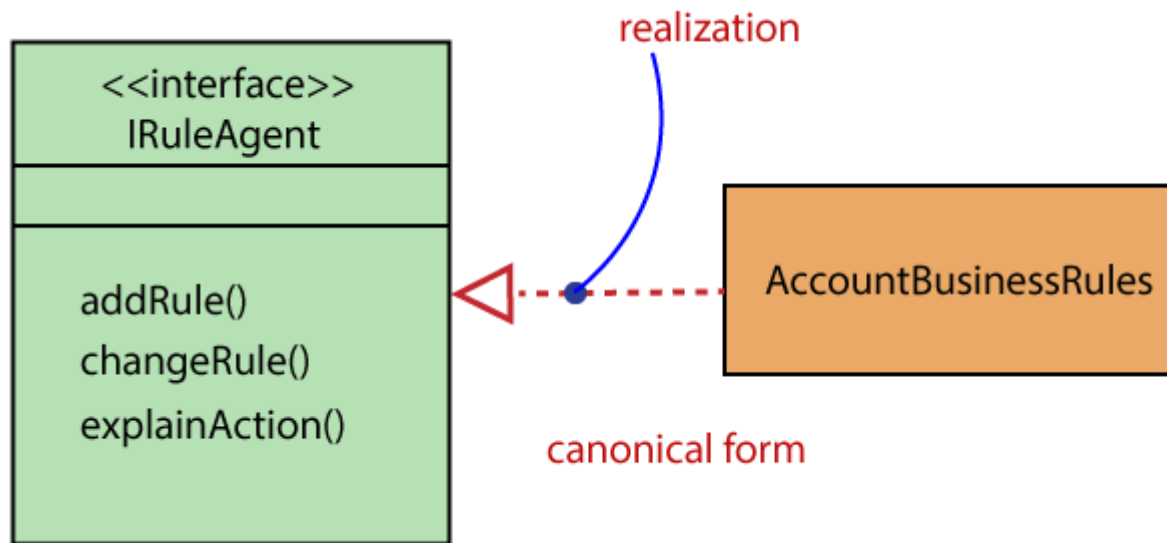
I nedenstående eksempel kan savings account og credit card account nedarves fra bank account



## Realization (interface)

Er et interface

Nedenstående kan der ses et eksempel, realization er ofte et andet program, eksempelvis et userinterface.0



## Abstrakte Klasser

Abstrakte klasser kan ikke instatiseret, og ser ofte sådan her ud, de er super klasser:



**!VIGTIGT!** Abstrakt klasse er altid i kursiv, og KAN ikke intiasteres

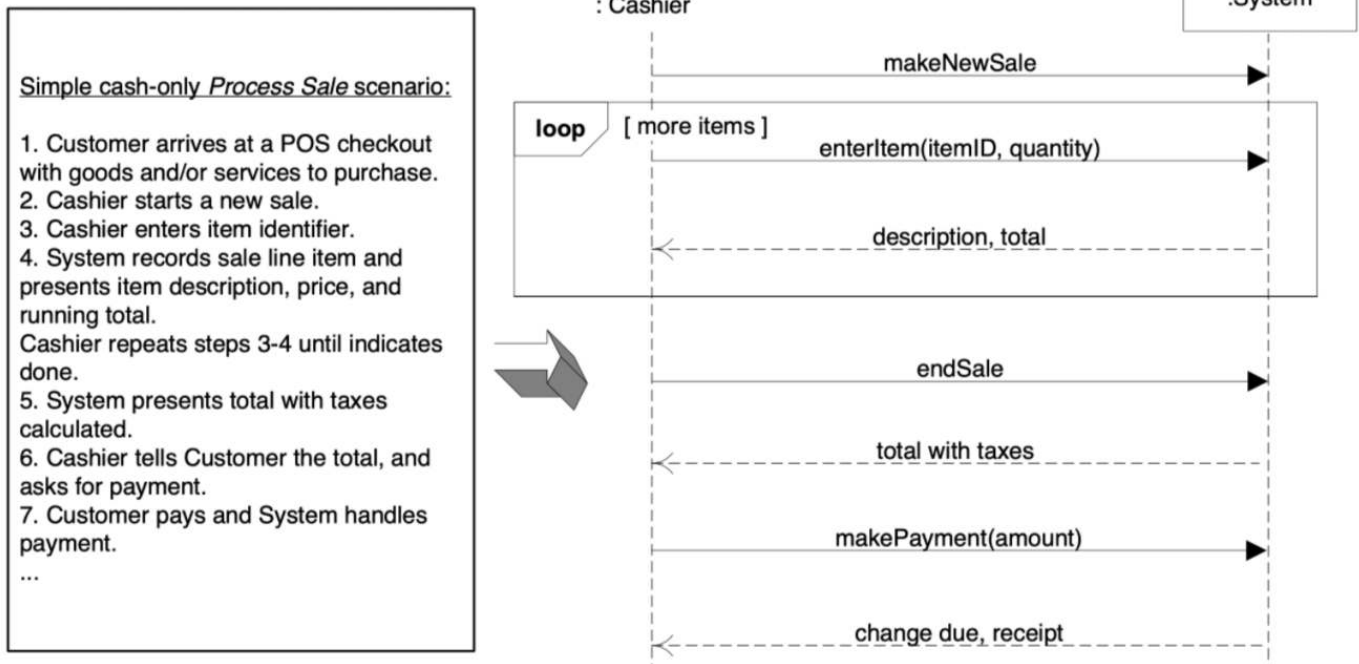
## System-crmer

Et system sekvens diagram er når der er en interaktion mellem

- Aktør
- System
- Visuel repræsentation, f.eks. et interface.

Nedenstående kan der ses et eksempel på system-sekvensdiagram.

- Larman Fig. 10.3



## Aktivitetsdiagram

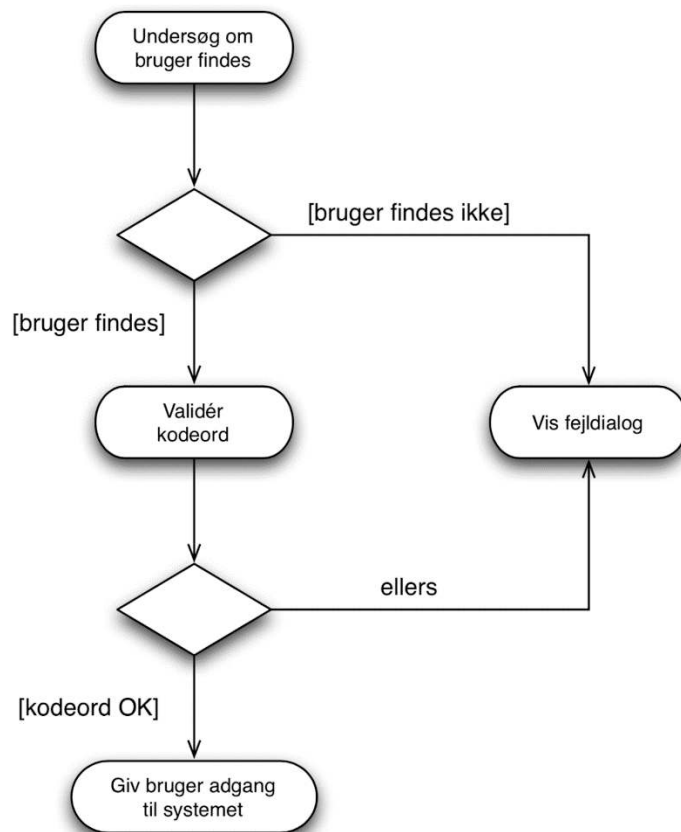
aktivitetsdiagrammer er UML udgaven af et flowchart, aktivitetsdiagrammer er en del af den dynamiske modellering, det viser altså et flow i en process.

Hvad aktivitetsdiagrammer kendetegnes ved ofte:

- Noget et objekt udfører
- Uddeles i simple handler (actions)
- Har en forgrening (agtig if-statement)

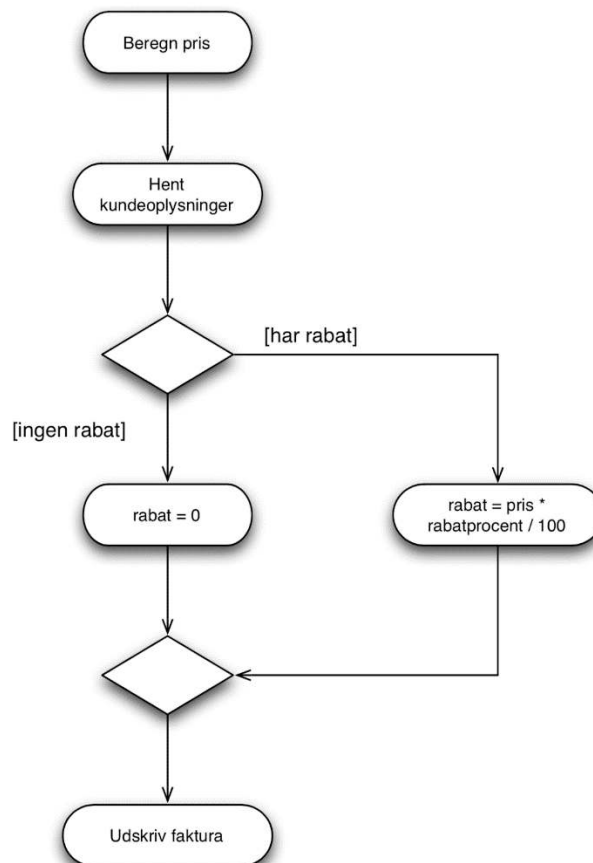
## Godt eksempel på aktivitetsdiagrammer:

Nedenstående er der et eksempel på et aktivitetsdiagram, der kan ses at man undersøger om en bruger findes, og at hvis bruger findes, går man to forskellige retninger, også ender de i samme retning til sidst hvor man får adgang til systemet. Aktivitetsdiagrammet er kendetegnet ved forgrening, altså agtig if-statement ved [bruger findes ikke] og [bruger findes]



### Et andet godt eksempel på aktivitetsdiagram:

I det næste eksempel kan der ses at at det starter ud med en handling / aktion beregn pris, til næste aktion som er hent kundeoplysninger, hvori der forgrenes i to retninger [Har rabat] & [Ingen rabat] også kommer i en sammenfletning, før handlingen udskriv faktura.





## Elementer indenfor aktivitetsdiagrammer

### Handling

Ovenstående er hver boks en handling.

Handling herunder er når der er et trin i aktiviteten, en handling tager ofte meget kort tid.

Her er der eksempler på handlinger.

FID = findes(ID)

visBruger()

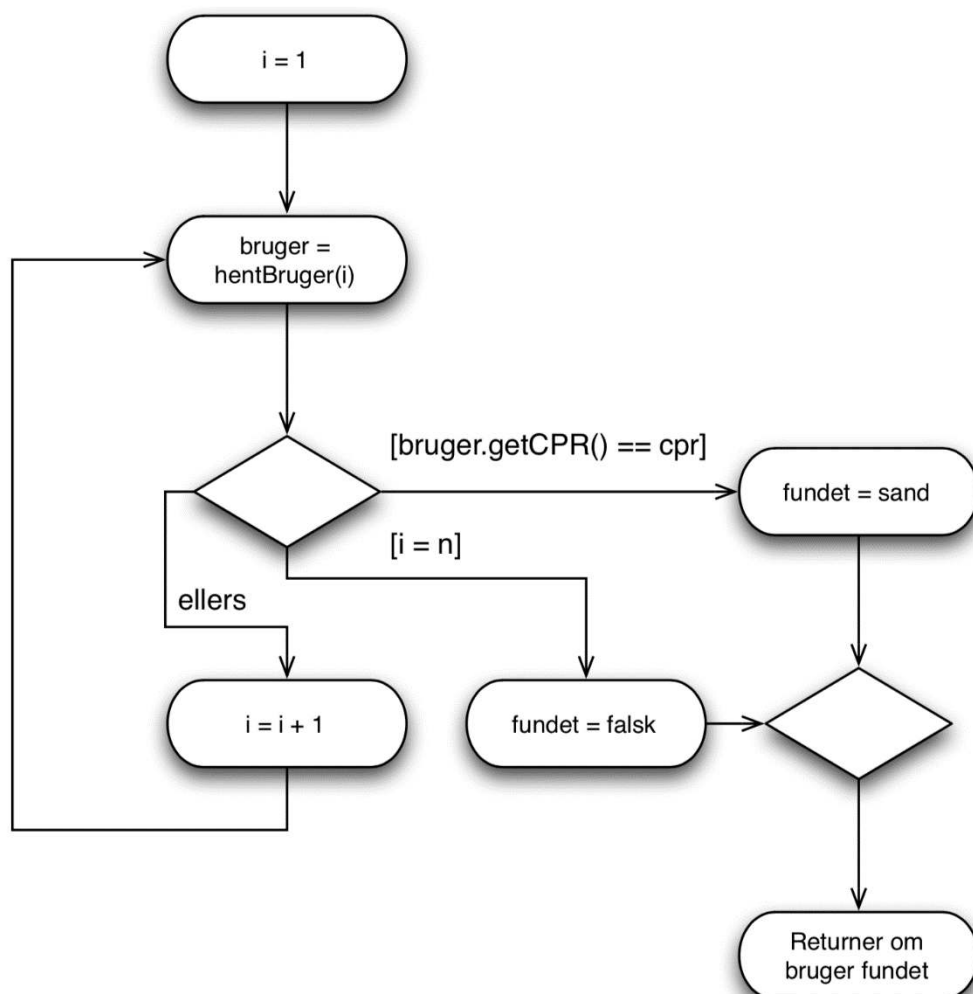
saldo = saldo +  
beløb

saldo = saldo +  
k1.hæv(beløb)

k2.indsæt(k1.hæv(beløb))

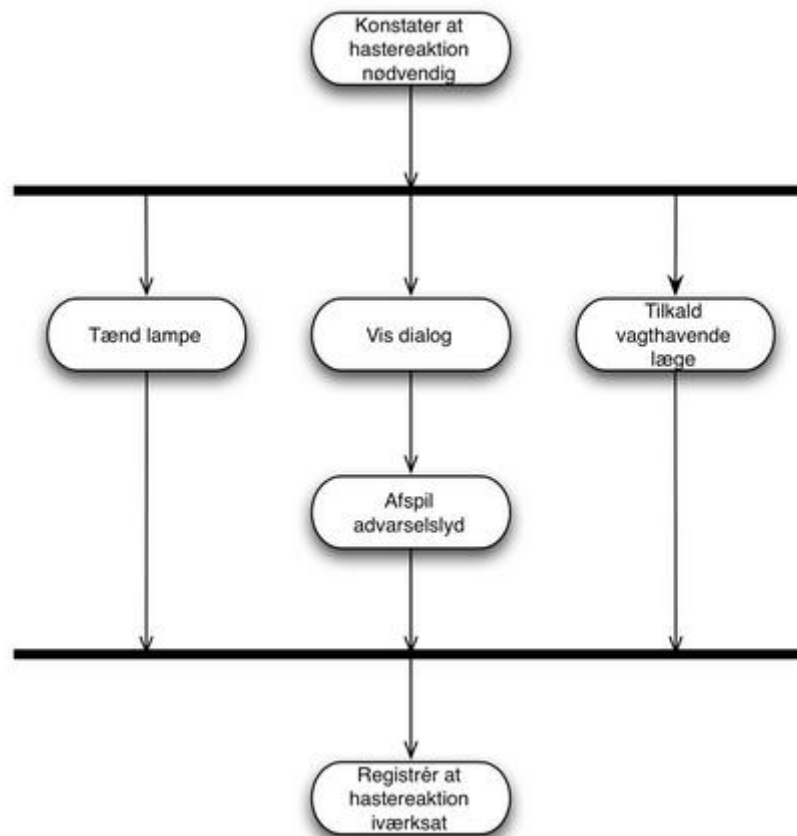
### Løkker/iterationer

Der kan også være løkker eller iterationer i aktivitetsdiagrammer, dette kan ses hvis der er en loop, eller en iteration såsom  $i++$ ,  $i=i+1$ , dette kan ses ved nedenstående eksempel:

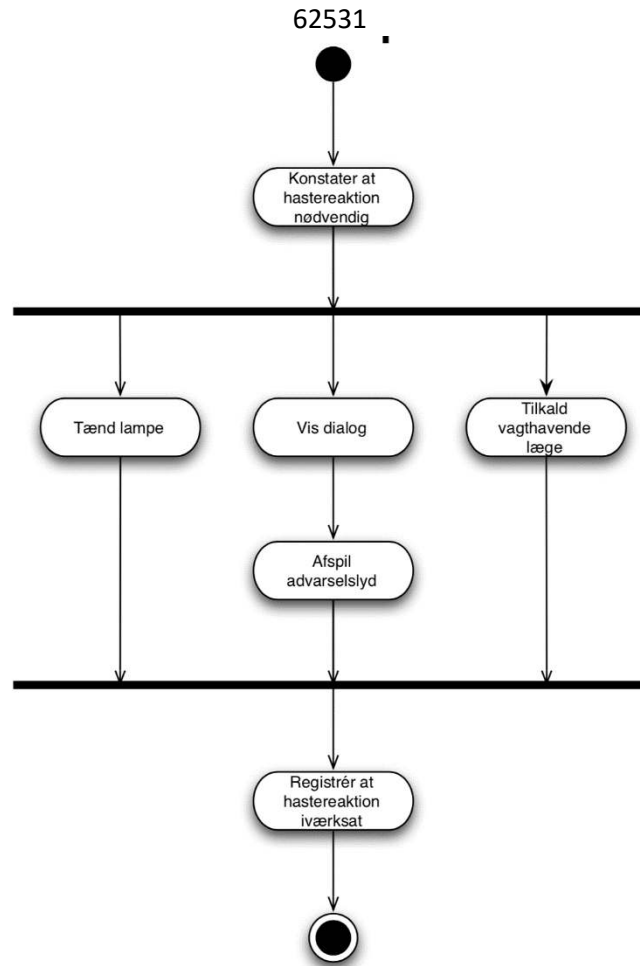


## Forks &amp; Joins i aktivitetsdiagrammer:

Forks i aktivitetsdiagrammer, er når flere ting løber parallelt med hinanden, dette beskrives ved en tyk streg.

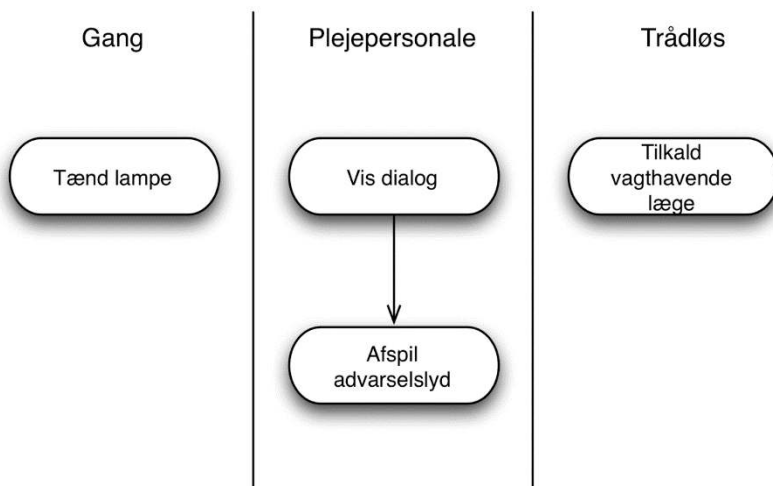


I aktivitetsdiagrammer kan der samt også forefindes start og sluttilstand, som er kendetegnet ved disse to knapper, der kan godt være mere end 2 knapper:



### Swim Lanes

Swim lanes er når der er adskilte områder, der arbejder anstændigt om en aktivitet, de adskilt af lodrette streger, der er et eksempel nedenunder. **!VIGTIGT! Swim lanes er altid aktivitetsdiagrammer.**



### Guards ( Guard Condition)


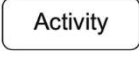



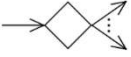
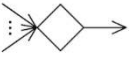


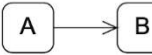
Guards er en form for guard før du kan komme videre, det er altid et boolsk udtryk kan besvares med false/true eller ja/nej

### Node ( Trekant )

Hvis der er et node, er det næsten altid et aktivitetsdiagram, og det betyder det foregøres.

## Elementer (indenfor aktivitetsdiagrammer)

Der er også disse andre ting

Name	Notation	Description
Action node		Actions are atomic, i.e., they cannot be broken down further
Activity node		Activities can be broken down further
Initial node		Start of the execution of an activity
Activity final node		End of ALL execution paths of an activity
Flow final node		End of ONE execution path of an activity
Decision node		Splitting of one execution path into two or more alternative execution paths
Merge node		Merging of two or more alternative execution paths into one execution path
Parallelization node		Splitting of one execution path into two or more concurrent execution paths
Synchronization node		Merging of two or more concurrent execution paths into one execution path
Edge		Connection between the nodes of an activity

**Table 7.1**

Notation elements for the activity diagram

## Generalt om Dokumentation af kode

Selvdokumenterede elementer – dette indebærer

- Klassenavne – svarer til ansvarsområde
- Metodenavn – svarer til funktionalitet
- Variabelnavne – svarer til indholdet

### Layout

Layout burde opdeles så det er læseligt, dette kan ske gennem:

- flotte curly brackets, med same curly bracket opsætning gennem hele softwareprojektet.
- Ekstra linje mellem metoder
- Separat fil til separate klasser

### Kommentarer

- Bare husk kommentarer til alt.

## Separation of concerns

Man burde have separation of concerns, i ens opgave.

Separation of concerns er når designprincip til at opdele en applikation i moduler, lag og indkapslinger, hvis roller er uafhængige af hinanden.

Dette består af:

- Hyppigt forekommende designprincip
- Opdeling af et system i separate områder
- Modularitet i koden
- Hvorfor modularitet?
  - o Opdeling af udvikling
  - o Læsbarhed
  - o Nemmere isolering af fejl
  - o Nemmere tests
  - o Genbrug
  - o Nemmere udskiftning

## Patterns (Herunder, GRASP)

### Design Pattern

Design patterns er ofte et design man har skabt, baseret på at løse ofte forekommende problemer, det er baseret udefra generaliserbarhed.

Design patterns er:

- Skalerbare
- Baseret fra en skabelon
- Hensigtsmæssigt for at løse problemer

### Singleton Pattern:

Singleton pattern, er når der er en streg nedenunder, dette oversættes til en static, altså global synlighed på kodesprog.

Singleton er en form for måde at simplificere kode, så man ikke behøver at instianserer flere gange.

Et eksempel er hvis man har en hjemmeside med en kurve med 15 baner, så i stedet for at have flere instanser igangværende, har man en som bruges i alle interfaces.

Godt eksempel på design klasse singleton, oversat til kode:

## Singleton

- singleton : Singleton
- Singleton()
- + getInstance() : Singleton

```
Public Class Singleton{  
Private static Singleton singleton;  
  
Public static Singleton getInstance(){  
    return singleton;  
}  
}
```

### GRASP

Formålet med Grasp, er at have lav kobling, høj samhørighed, også kaldt low coupling, high cohesion. Dette er hvad de gør:

GRASP står for: General Responsibility Assignment Software Patterns

#### - Creator

Hvem er ansvarlig for oprettelse af klassen?

#### - Information Expert

Hvem er ansvarlig for objekter?

#### - Low Coupling

Hvordan der sikres lav afhængighed mellem klasser og dermed f.eks.

- isolering af ændringer
- nemmere at forstå designet
- nemmere at genbruge

### - High Cohesion

- Ensartet ansvarsområde
- Understøtter Low Coupling

Hvis der er flere ansvarsområder i en klasse opstår der hurtigt flere koblinger

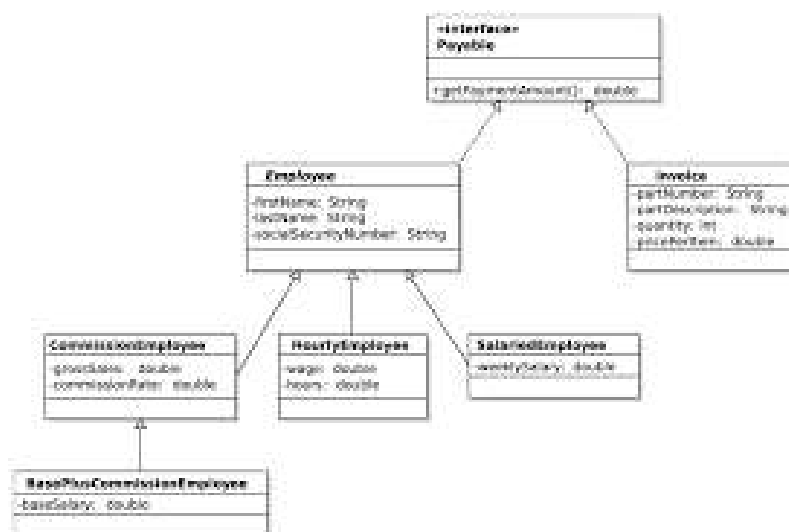
- Bedre forståelse
- Næmmere vedligeholde
- Næmmere Tests
- Næmmere genbrug

### - Controller

Systemklasse, hvad der sørger for at uddele ansvaret ud til view og model, såsom i MVC Modellen.

### - Polymorphism

Polymorphism er ikke arv, men ofte ligesom arv, altså forældre barn forhold, det ene kan ikke fungere uden det andet. Her kan et eksempel på Polymorphism ske, som er henvist ved dependency pilen.

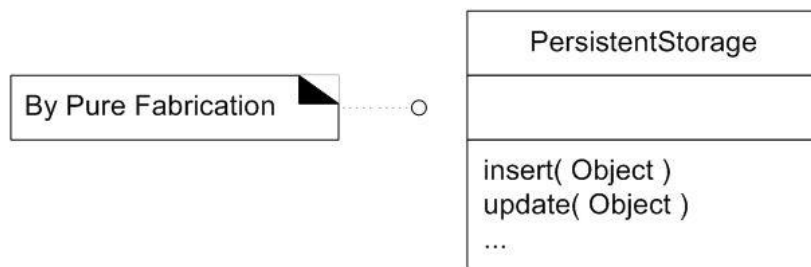


### - Pure Fabrication

Pure fabrication, er når man opfinder en klasse, som ikke burde findes. Noget som ofte sker hvis man gerne vil ha high cohesion low coupling, så begynder man at lave klasser som der i virkeligheden er på baggrund af noget der ikke burde være der. Eksempelvis:

# Pure Fabrication

- **Example:** Suppose we need to save **Sale** object in a relational DB.
- Information Expert or Expert says Sale should do it, because Sale knows its total.
- But it violates Low Coupling and High Cohesion because Sale will be coupled with JDBC etc.



This class is a Pure Fabrication – a figment of the imagination

Hvis en klasse kun også er der som en getter og setter

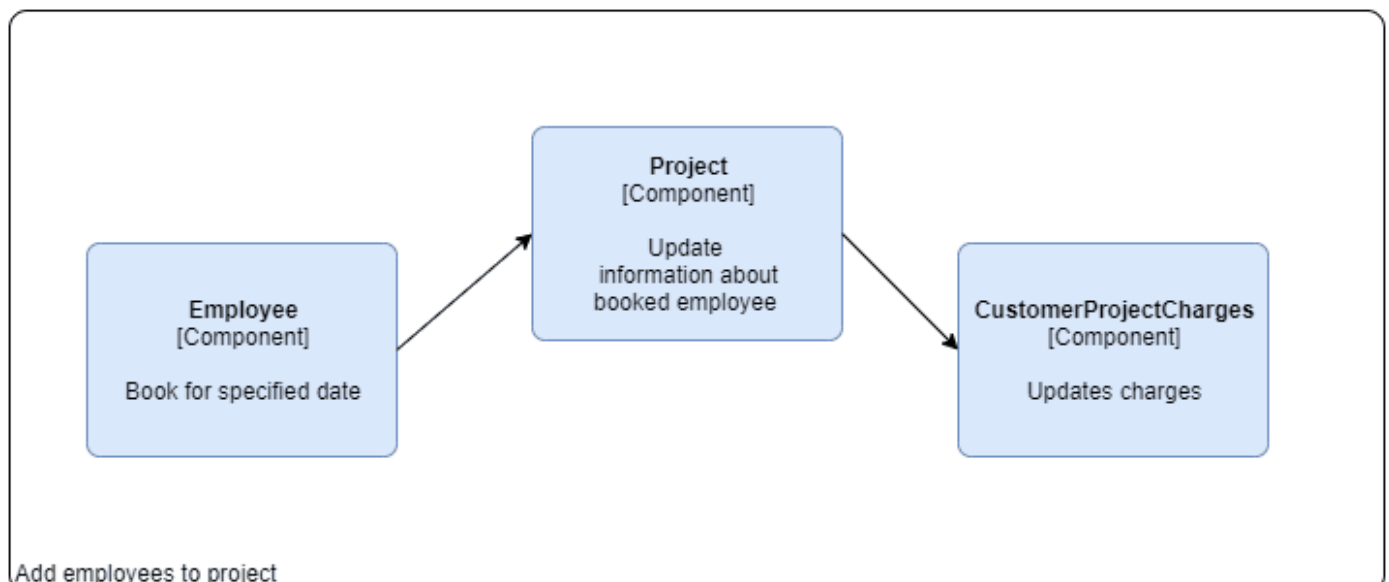
## - Indirection

“en form for mellemmand kommunikations-objekt”

Hvis flere komponenter reagerer gennem hinanden, kan man lede dem gennem en main klasse, dette kalder man indirection, her er et eksempel før og efter indirection:

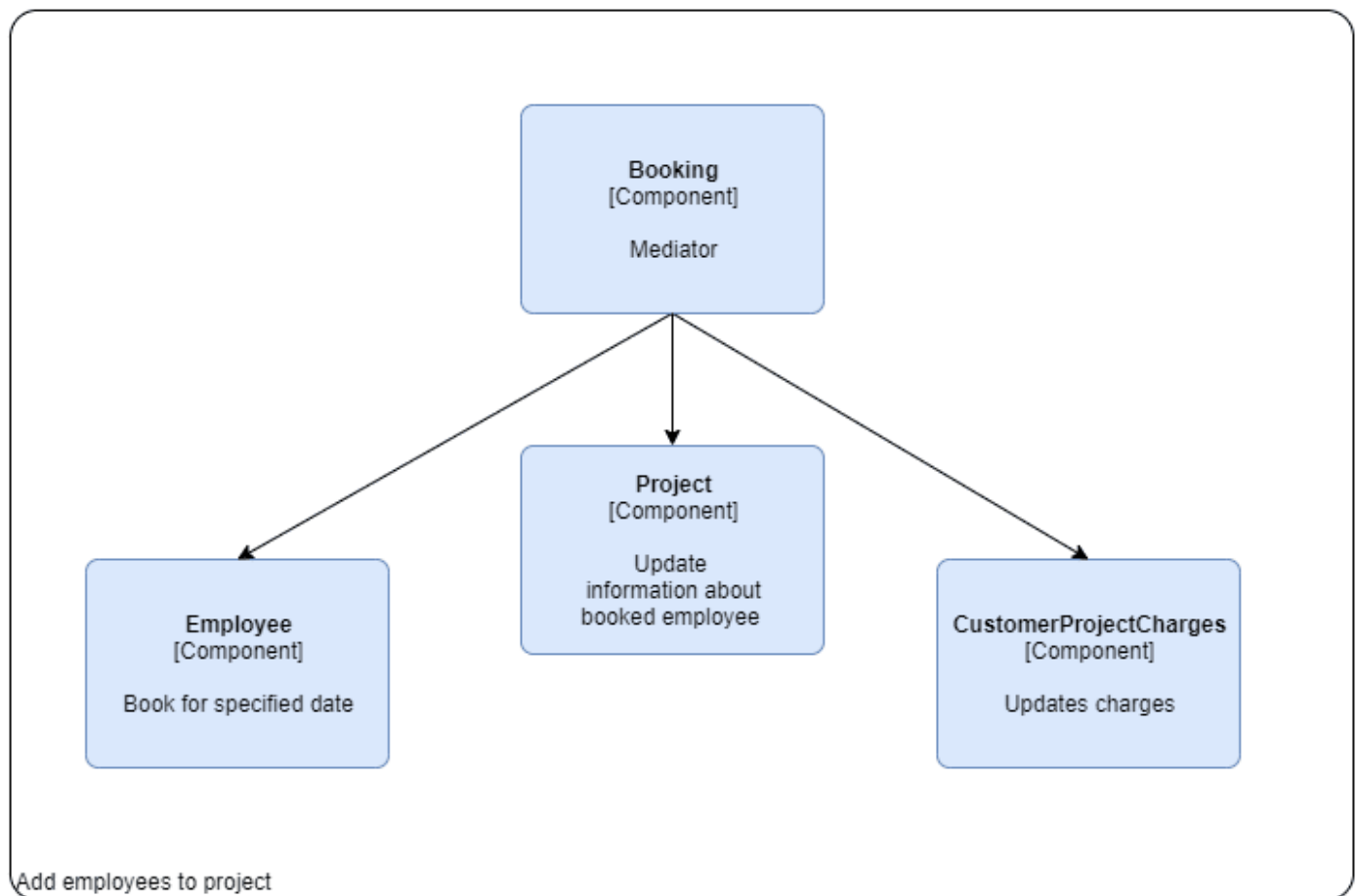
Det ligesom en form for flersidet adaptor. Formålet er at have en lav kobling mellem objekter.

Formålet med det er så de ikke er afhængige af hinanden.





Her et eksempel med indirection:



### - Variations

Protected variations er et underlag af low coupling, low coupling omhandler om at man har mange klasser, der dækker små bider af funktioner som en klasse kan, f.eks hvis du har en butik funktion, som indenunder har salg, køb, returner, osv. Protected variations er er til formål hvis man har ødelagt en lille del af et software projekt, så sker der ikke en riplende effekt gennem hele software programmerings systemet.

### Arv og Polymorphi

Polymorphi er hvis der er noget der extender, eller arver videre, som gør noget ensartet.

Eksempel på polymorfi:

I nedenstående eksempel, kan der ses at cat, dog, extender abstrakt klassen animal, hermed nedarves den fra den. Dette er polymorfi, da det skaber ensartet funktion.

```
abstract class Animal {
    abstract String talk();
}

class Cat extends Animal {
    String talk() {
        return "Meow!";
    }
}

class Dog extends Animal {
    String talk() {
        return "Woof!";
    }
}

static void letsHear(final Animal a) {
    println(a.talk());
}

static void main(String[] args) {
    letsHear(new Cat());
    letsHear(new Dog());
}
```

## GoF Creational Patterns

Ligesom Grasp, 23 patterns, som er opdelt i 3 kategorier.

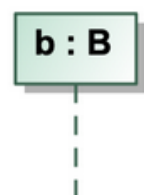
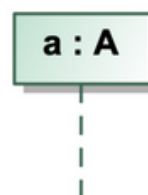
- Creational
- Structurel
- Behaviovural

## Objekt kendskab / reference

# Objekt kendskab/reference

- Attributter
  - B er en attribut i A
- Parametre
  - B er en parameter til en Metode i A
- Lokale variable
  - B er en lokal variabel i en metode i A
- Global variabel
  - B er en global variabel

Hvordan koder man det i java?



## Tilstandsdiagrammer

En applikation har altid en tilstand, applikations tilstand er baseret udefra objektets tilstand.

Et godt eksempel er en knap skaber nummerplader i motorregisteret.

Fra et punkt hvor man kan udfører noget.

- Hændelser som objektet kan reagere på

Mulige resultater

- Mulige overgang
- Finite state machine

At der er uendelig mange stater objekter kan foregå i.

- Deterministisk.

Alle hændelser sker i en proces. Dette er ofte hvad der sker i software.

- Non-deterministisk

Nogle hændelser kan ske i flere overgange, f.eks. når du trykker på en knap, og den kan udgøre sig to forskellige hændelser.

- Software bliver kaldt Deterministiske FSM'er

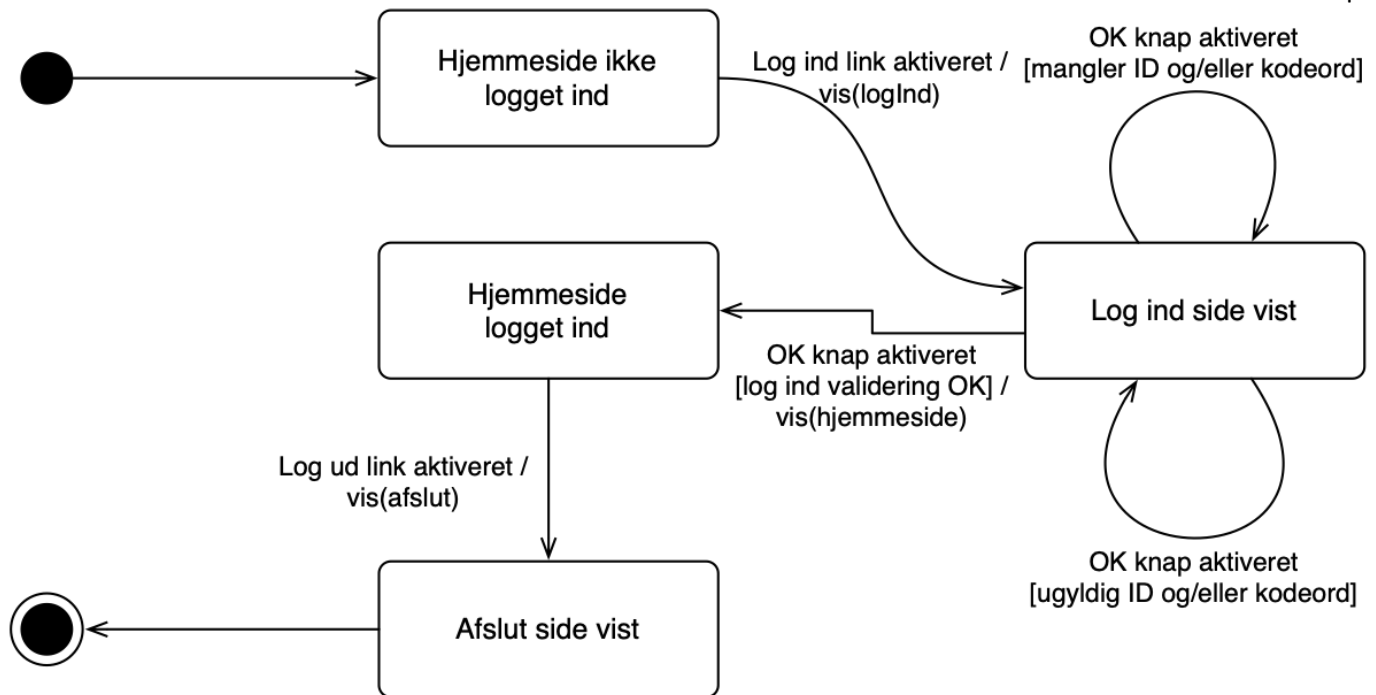
Objekt kan være alt fra:

- Bruger
- Bil
- Microbølgeovn
- Applikation
  - Eksempelvis, navigation, login tilstand
- Ordre
- Aflevering
- Eksamen
- Beskæftigelse
- Single

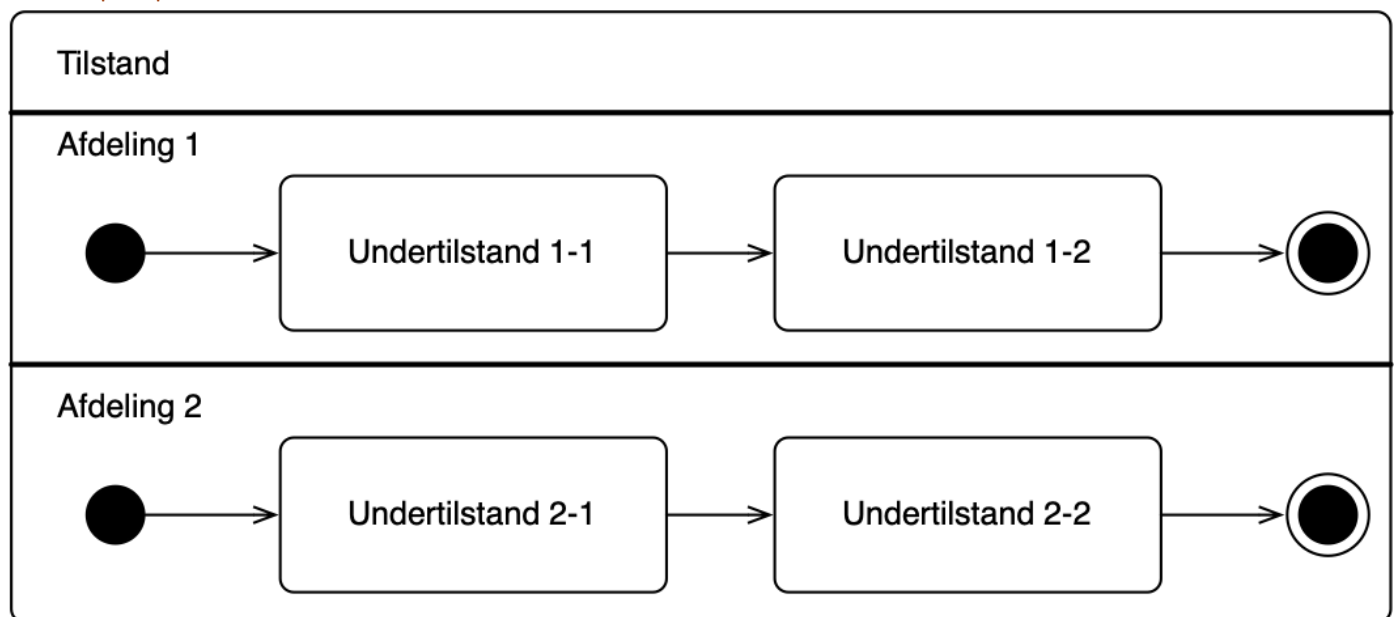
Dette er hvad der beskrives alt i alt i et tilstandsdiagram.

### Godt eksempel på tilstandsdiagram:

Nedenunder kan der ses en start på tilstandsdiagrammet, og der kan ses en loop, der altid tekst mellem stregerne i en tilstandsdiagram.



Eksempel på undertilstande:



Der er 5 forskellige tilstande her.

Alle kombinationer af 1-1 eller 1-2, og 2-1 og 2-2 kan fremstå.

## Review

Det er vigtigt at reviewe ens materiale, så man ved hvad man gør godt, og kan forbedre.

Når man laver en review, burde man udnytte disse emner:

Review: Analysis

- Domæne modeller

Er den betydningsfuld, effektiv, nogle problemer eller syntaxfejl?

Review: Design

Er den betydningsfuld, effektiv, nogle problemer eller syntaxfejl?

Review: Process

Man sender det videre til andre grupper, så der kan processeres et review.

## Software kvalitet

Software kvalitet er bare hvordan kvaliteten er opbygget programmeringsmæssigt.

Møder Krav

- (Hvis kravene er gode)
- Utility

Brugbart

- Usability

Velstruktureret

- Vedligeholdelsesvenligt

Lever op til standarder

- Programmering, Proces, Dokumentation

## Verificering & Validering

*Test, Inspektion og Review*

- Del af verificering og validering

Validering

- Kontrol af at produktet lever op til behovet

Verificering

- Kontrol af at produktet lever op til kravene

**!VIGTIGT! Validering og Verificering skal passe sammen**

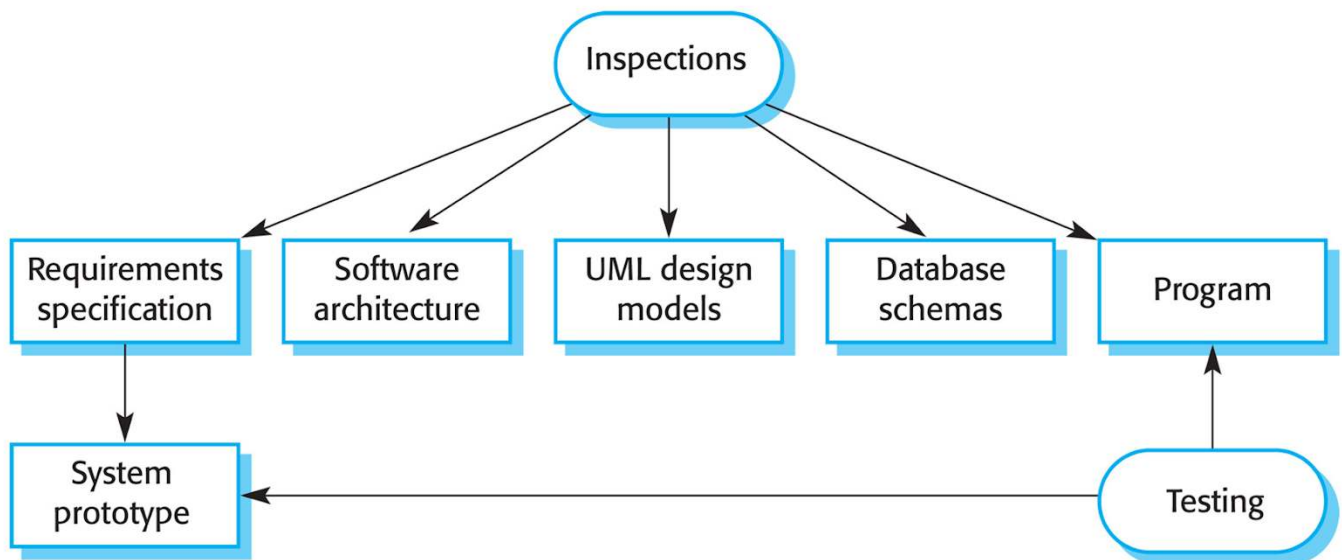
**Hvis krav og behov ikke matcher sammen, har man et problem!**

## Quality Assurance

Man har ofte en afdeling af quality assurance i et firma, det er hvor de tilgår at alt er som det er, de kan forkorte kode, teste kode, være testere for spil.

De kigger generelt set på inspektioner af de forskellige faser man indgår i et software projekt.

## Eksempel på inspektioner for quality assurance:



Copyright ©2016 Pearson Education, All Rights Reserved

## Projektplanlægning

Projektplanlægning indeholder forskellige principper, herunder disse:

### Project Scope Statement

Herunder betyde (PSS) Formål, Ønsket resultat, Kriterie, altså har en omfattende retning

### Project plan

Dette indeholder delopgaver, estimer, rækkefølge, deadlines, have en omfattende rute.

## CDIO

CDIO er et godt iterativt skabelon for projekt planlægning:

CDIO står jo for:

- Conceive
- Design
- Implement
- Operate

Man kan også udvide ens CDIO, så den er mere projekt planlægnings konstrueret:

### C(P)DIO(C)

- Conceive (intiate)  
Plan
- Design  
Plan
- Implement  
Execute
- Operate  
Monitor and control
- Close  
Learn from your mistakes

## Plan

Den bedste måde man kan udbygge en plan for, som projektleder er:

- Risk Management
- Project Schedule

- Budget
- Communication plan

Nedenstående står der dybdeangående forklarelse af alle punkter i en plan

## Risk management

Risici, hvilke risikoeer forefindes der.

### A. Identificer risici

Brainstorm

Hvad hvis folk ikke kommer?

Noget vigtigt bliver væk?

### B. Vurder risici

Lave en risk matrix, af mpact og probality

Hvilke risikoeer er acceptabelt, hvilket niveauer skal adresseres

### C. Imødegå risici

Uacceptable risici / kritiske risici

Transfer: Giv opgaven til en 3. part for hvilken risiko er mindre (Specialist)

Accept: Accepter risiko

Mitigate (Afbød): Reducér Impact og/eller Probability

Eliminate: Fjern risiko

## Project Schedule

Man kan bruge værktøjer til projektplaner, såsom Microsoft project, Asana, excel eller Jira, Asana er gratis og har en venlig brugerinterface.

### D. Work Breakdown Structure:

Nedbryd projektet, i overskuelige bidder, så projekterne er overskuelig.

- Et mindmap er en god idé
- nogle posts
- Scrum log, hvor man skriver to do liste, og hvad der er færdigt osv.

### E. Sæt opgaverne i rækkefølge

- Prioriterer hvad der er vigtigst.

### F. Identificerer Projektteamet

- Giv passende roller til de korrekte personer
- Hvor mange opgaver kan køre parallelt?

### G. Estimer varigheden af de forskellige opgaver

- Hvor lang tid forventer vi at det tager, udefra tidligere erfaring, teamets erfaring, ekspert erfaring.
- Man kan udnytte estimat-metoder, såsom

PERT formel:

$\begin{aligned} &> \text{Optimistisk} := 10 \\ &= \\ &> \text{Realistisk} := 30 \\ &= \\ &> \text{Pessimistisk} := 100 \\ &= \\ &> \frac{(\text{Optimistisk} + 4 \cdot \text{Realistisk} + \text{Pessimistisk})}{6} \end{aligned}$	$\begin{aligned} &\text{Optimistisk} := 10 \\ &\text{Realistisk} := 30 \\ &\text{Pessimistisk} := 100 \\ &38.33333334 \end{aligned}$
---	--

## H. Identifier kritisk vej

Den bedste vej, det tidsrum hvor man kan udnytte sin bedste udgang.

## I. Ansvarlig på Opgaver

Praktisk, hvem er ansvarlig?

### Opfølgning:

Man burde lave en opfølgning, så man kan se ens projekterede retning imod den tilbageværende tid.

Dette kan man gøre vha:

- Noter tidsforbrug hver dag
- Kontroller om tilbageværende tid er tilstrækkeligt, dette kan indeholde
  - Opjuster Estimat?
  - Sænk kvalitet?
  - Ryk Deadline?
  - Er kritisk vej i fare?

Og opdater ens plan i betragtning med ens opfølgning, så man opbygger sig i form af realiteten.

- L. Del projektet op i sprint, eller kvartiler, typisk opdeler man det i virksomheder i 3 ugers perioder.

### Budget

- Person timer
- Andre udgifter

### Communication Plan

Hvad	Hvem (ansvarlig)	Publikum	Hvordan	Hvornår
Team accountability session (Scrum møde)	Projektleder	Projekt-team		
Status report	Projektleder	Styregruppe	Møde	
Prototype update?				

### GANTT Diagram

Konstruktion af et gantt diagram, viser en tidsorden, hvori man kan være opmærksom på bindinger, og skabe en kritisk vej til den bedste projektilgang til ens projekt.

Man konstruerer et gantt diagram ved følgende træk:

Konstruér Gantt-diagram

- Brug et værktøj
- Parallelisér!
- Vær opmærksom på bindinger
- Find kritisk vej!

Kritisk vej kan ændre sig



- Opatér
- Monitorér

Sæt alle sejl på kritisk vej!

- Bedste ressourcer

Eksempel på gantt diagram:

