

Velkommen til E/SW/ST2ISE

Introduction to Systems Engineering

Dagens menu

- Kursus-introduktion
- Gruppedannelse, opgaver og kommunikation
- Hvad er *systems engineering*?
- Case study

Dagens menu

- Kursus-introduktion
- Gruppedannelse, opgaver og kommunikation
- Hvad er *systems engineering*?
- Case study

Introduktion til kurset

- Tema: Indledende Systems Engineering
- Formål:
 - *Metoder, processer og struktur* for udvikling af systemer bestående af hardware og software
 - *Teori og øvelser* som fundament for effektiv projekt-gennemførsel

Undervisere

- Kim Bjerge (kbe@ece.au.dk), 313E
 - Kursusansvarlig
- Frank Bodholdt Jakobsen(frabj@ece.au.dk)
 - Primært domæneanalyse og applikationsmodeller (UML)

Undervisning

Uge 5,6 og 8

	Mandag	Tirsdag	Onsdag	Torsdag	Fredag
0815-0900					
0915-1000					
1015-1100					
1115-1200					
1215-1300					ISE
1315-1400					ISE
1415-1500	ISE	ISE			
1515-1600	ISE	ISE			
1615-1700					

Standard

Materiale

- Kompendium (købes i bogladen)
- Online videoer, artikler m.m.
- Lektionsmateriale, herunder slides

Lessons and topics

- ***System Specification, Quality and Process (Kim)***
 - Specification, Use Cases
 - System Test
 - Quality Assurance
 - Development Processes
- ***System Architecture Design and SysML Diagrams (Kim)***
 - SysML structure diagrams
 - SysML behavior diagrams
 - Architecture and design
 - Interfaces
- ***Software Analysis and Design, Protocols and Implementation (Frank)***
 - Domain Analysis
 - Application Models
 - Design to Implementation
 - Protocols
- ***Project Management***
 - Project Management
 - Scrum
 - Project Documentation

Lektionsplan

Lektion	Uge	Navn	Dag	Dato	Underviser
L1	35	Introduktion	Mandag	30/8	KBE
L2	35	Kravspecifikation	Tirsdag	31/8	KBE
L3	35	Use Cases	Fredag(*)	3/9	KBE
L4	36	UC + Projekt	Mandag	6/9	KBE
L5	36	Systemtest	Tirsdag(**)	7/9	KBE
L6	36	SysML BDD	Fredag(*)	10/9	KBE
L7	37	SysML BDD+IBD	Mandag	13/9	KBE
L8	37	SysML IBD	Tirsdag	14/9	KBE
L9	37	SysML SD	Fredag(*)	17/9	KBE
L10	38	SysML STM	Mandag	20/9	KBE
L11	38	Arkitektur og design	Tirsdag	21/9	KBE
L12	39	Grænseflader	Mandag	27/9	KBE
L13	39	Kvalitetssikring	Tirsdag	28/9	KBE
L14	40	Projektledelse	Mandag	4/10	KBE
L15	40	Domæneanalyse	Tirsdag	5/10	FRABJ
L16	41	Domæneanalyse	Mandag	11/10	FRABJ
L17	41	Applikationsmodel	Tirsdag	12/10	FRABJ
	42	Efterårsferie			
L18	43	Applikationsmodel	Mandag	25/10	FRABJ

Brightspace – L1-L5 Specifikation og test

SW2ISE-01 Indledende System Engineering (E21...)

Course Home Content Course Tools Classlist Zoom Help

Standards + New Unit

L1-5 Specifikation og test

L1 Introduction

L2 Kravspecifikation

L3 Use Cases

L4 Use Cases

L4 Use Case Løsning

L5 Systemtest

Visible

Add Existing Create New

BeoSoundF_ConceptReport.pdf

Course Introduction F2021.pdf

Indhold:

- Introduktion til System Engineering
- Kursus indhold
- Gennemgang af retningslinier for gruppedannelse, aflevering af opgaver og kommunikation med undervisere i I2ISE
- Hardware/Software Udviklingsprojekter
- Eksempler
- Case arbejde

Læsestof:

http://en.wikipedia.org/wiki/Systems_engineering

BeoSoundF_ConceptReport.pdf skimmes

What is "Systems Engineering" ? | Elementary collection
Duration: (3:13)
User: 3dcata - Added: 30/09/11

Watch Video

Dagens menu

- Kursus-introduktion
- Gruppedannelse, opgaver, kommunikation og eksamen
- Hvad er *systems engineering*?
- Case study

Gruppedannelse

- Gruppedannelse – *jeres eget ansvar!*
 - 3-4 personer pr. gruppe
 - Se i øvrigt instruktioner på *Brightspace* -> *SW2ISE* -> "Afleveringsopgaver" og "Grupper"
- Deadline: Fredag d. 5. februar
 - Ved ikke → *send mail til Frank* (frabj@ece.au.dk)
- Ændringer i gruppessammensætning
 - *Kun* sammenlægninger ved bekræftet frafald

Obligatoriske opgaver

- 4 obligatoriske opgaver
 - A: Specifikation og validering
 - B: Systemstruktur og adfærd (SysML)
 - C: Domæneanalyse og applikationsmodel
 - D: Systemdesign (Valgfri opgave)
- Ud- og aflevering vha. Brightspace (BS)
- Aflevering -> Review (A+B) -> Godkendelse
- Tilbagemelding vha. BS

Slusesystem



Kaffeautomat



Obligatoriske opgaver

- Besvares i grupperne
- 1 besvarelse = 1 fil.
 - *Enten* 1 PDF-fil eller word
- Brug altid standard-forside
- Besvarelse uploades på BS

Review af opgave

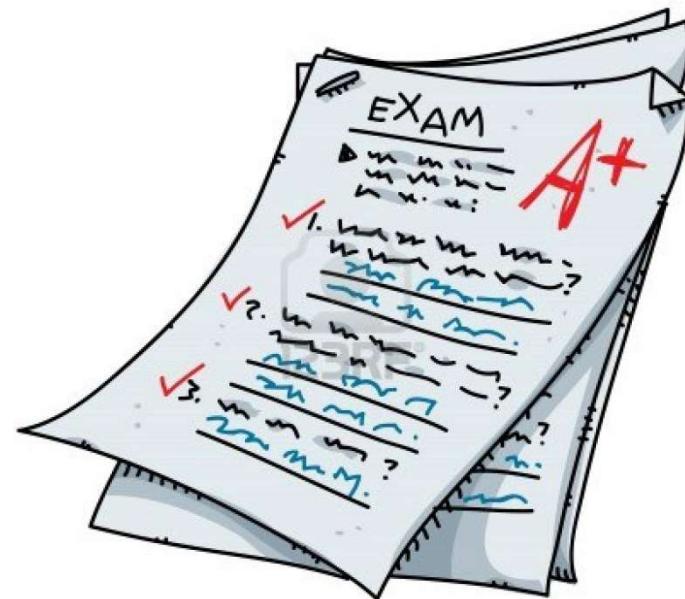
- Kopi af opgavebesvarelse sendes til review gruppen
- Opgaven reviews med bemærkninger
- Kommenteret review sendes til kontaktperson for besvarelsen som uploader en revideret opgavebesvarelse på BS

Obligatoriske opgaver - godkendelse

- Tre mulige bedømmelser:
 - Godkendt
 - Ikke godkendt – genafleveres (og hvad så?!?)
 - Ikke godkendt (og hvad så?!?)
- For sen aflevering == ikke godkendt
- Alle 4 opgaver skal godkendes for at kunne indstilles til eksamen – på baggrund af revideret besvarelse

Eksamens

- 3 timers skriftlig eksamen, intern censur, 7-skala
- Igen: Alle 4 obligatoriske opgaver *skal* være godkendt for at I kan gå til eksamen!



Dagens menu

- Kursus-introduktion
- Gruppedannelse, opgaver og kommunikation
- Hvad er *systems engineering*?
- Case study

Hvad er systems engineering?

- **Wikipedia: Systems engineering**
is an **interdisciplinary** field of engineering that focuses on **how complex engineering projects should be designed and managed** over the life cycle of the project [...].
- Systems engineering deals with **work-processes and tools** to handle such projects, and it **overlaps with both technical and human-centered disciplines** such as control engineering, industrial engineering, organizational studies, and project management.

Some systems engineering buzzwords

Interdisciplinary

Holistic

Technical AND
Human-centered

Managing
complexity

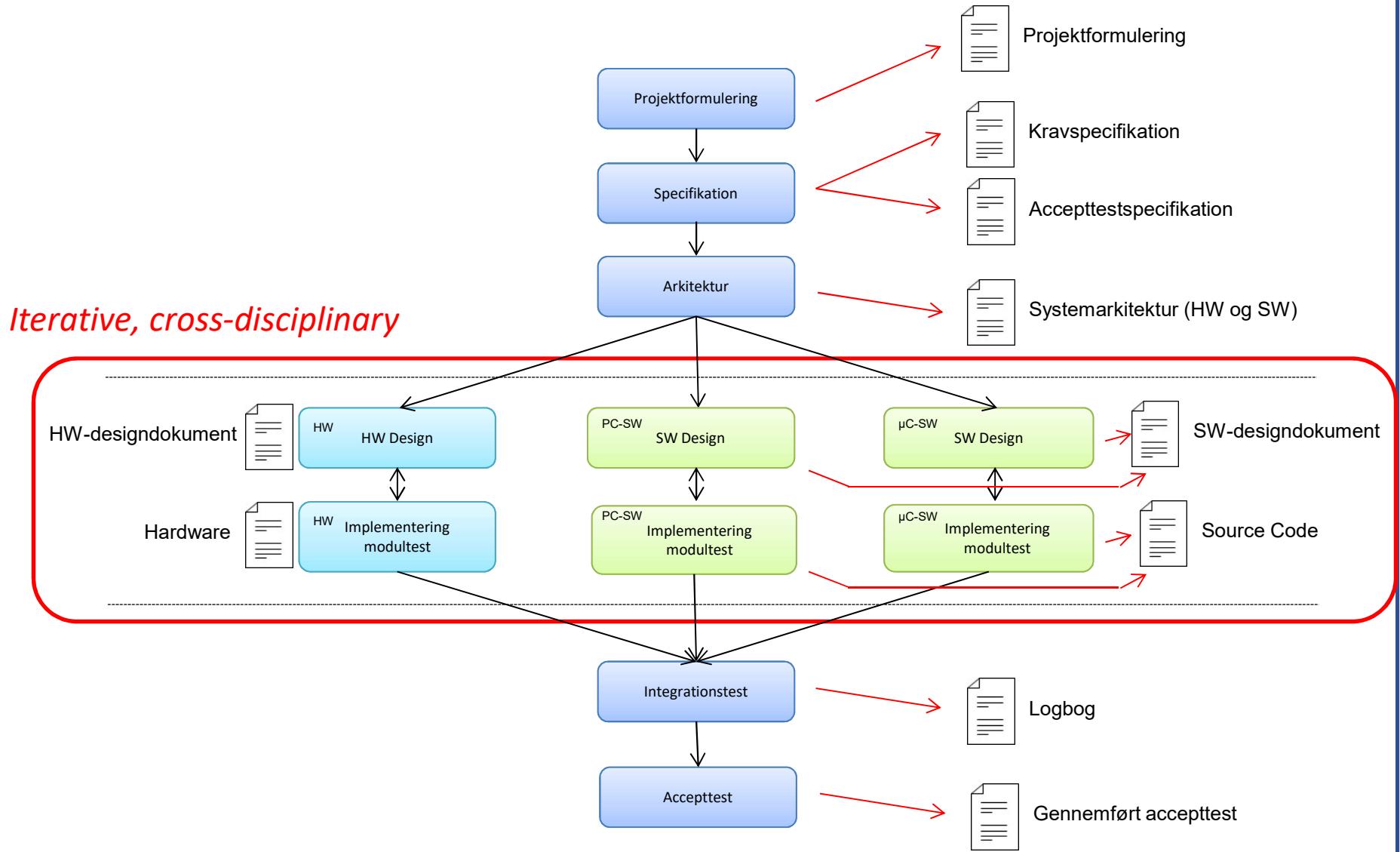
Tools and
methods

Art and science

Et par eksempler på systemer

- **Automation Systems:**
 - **Crisplant LS-4000E Tilt-Tray Sorter**
 - <https://www.youtube.com/watch?v=l9iSLdUT7zs>
- **Robotics:**
 - **Boston Dynamics All Prototypes**
 - <https://www.youtube.com/watch?v=bRHG7YObDuU>
- **Building Constructions - Tech trends 2019:**
 - https://www.youtube.com/watch?v=BkRsA_v5oY4

The ASE Process



Specifikation:

Hvad skal systemet gøre?

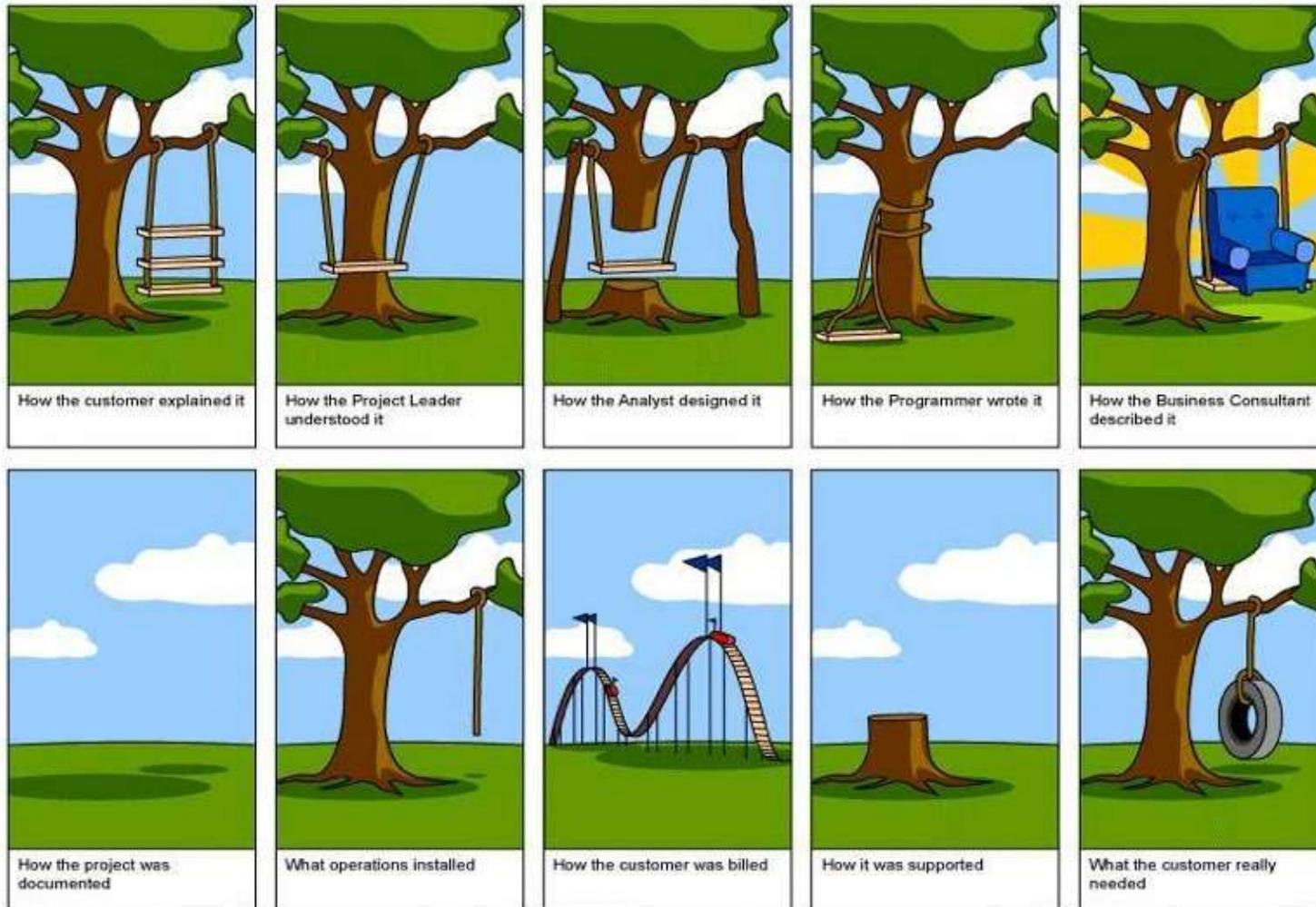
Analyse og Design:

Hvordan systemet gør det?

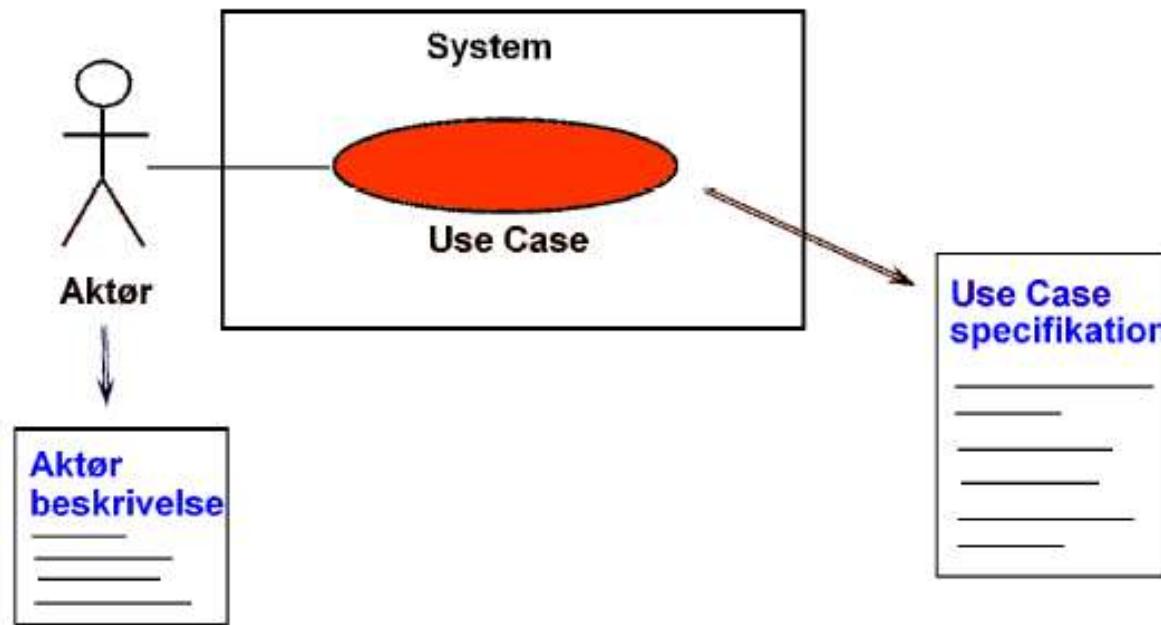
Verifikation og Test:

Virker systemet som forventet?

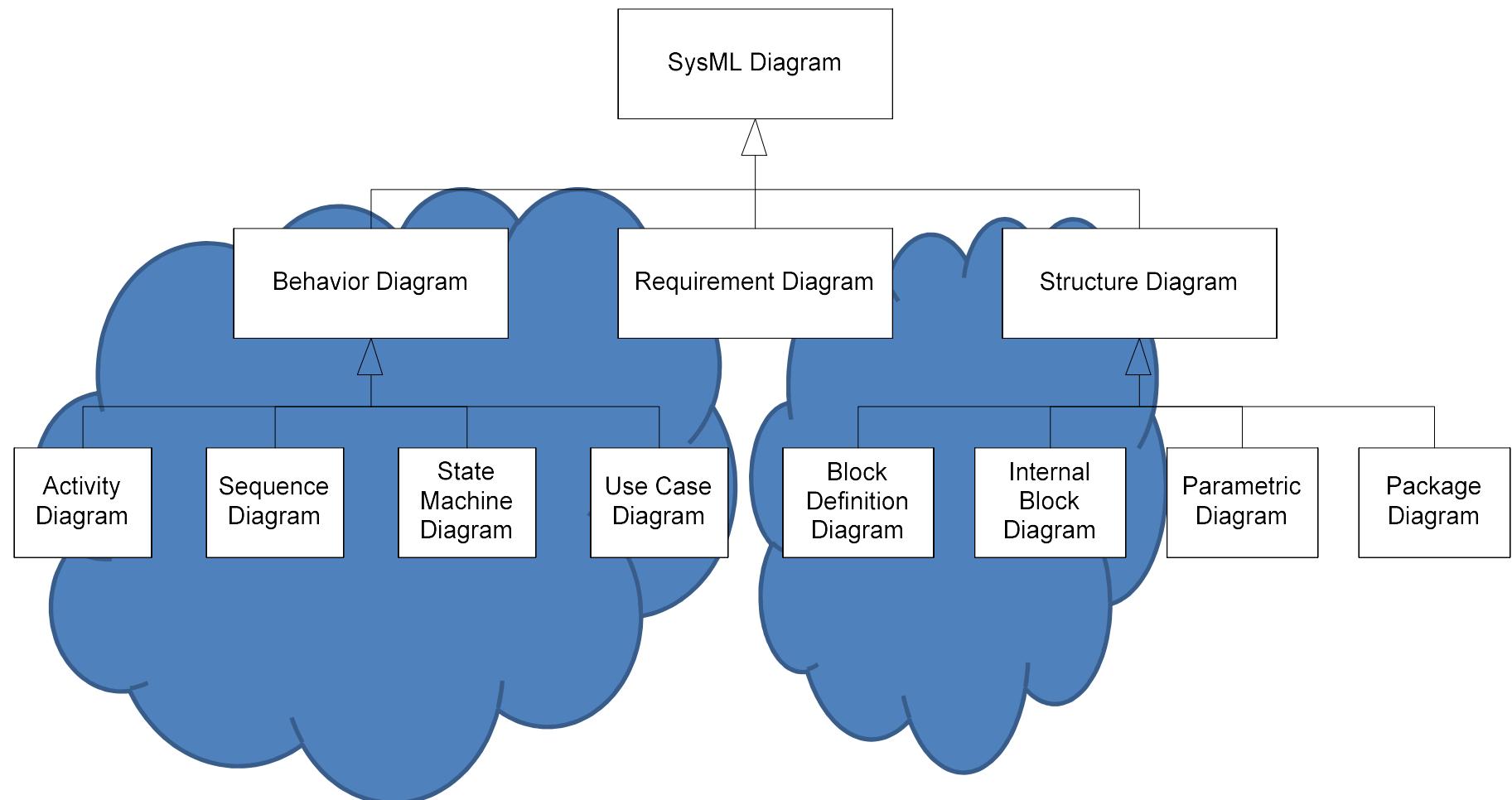
Udfordringen i at forstå systemet?



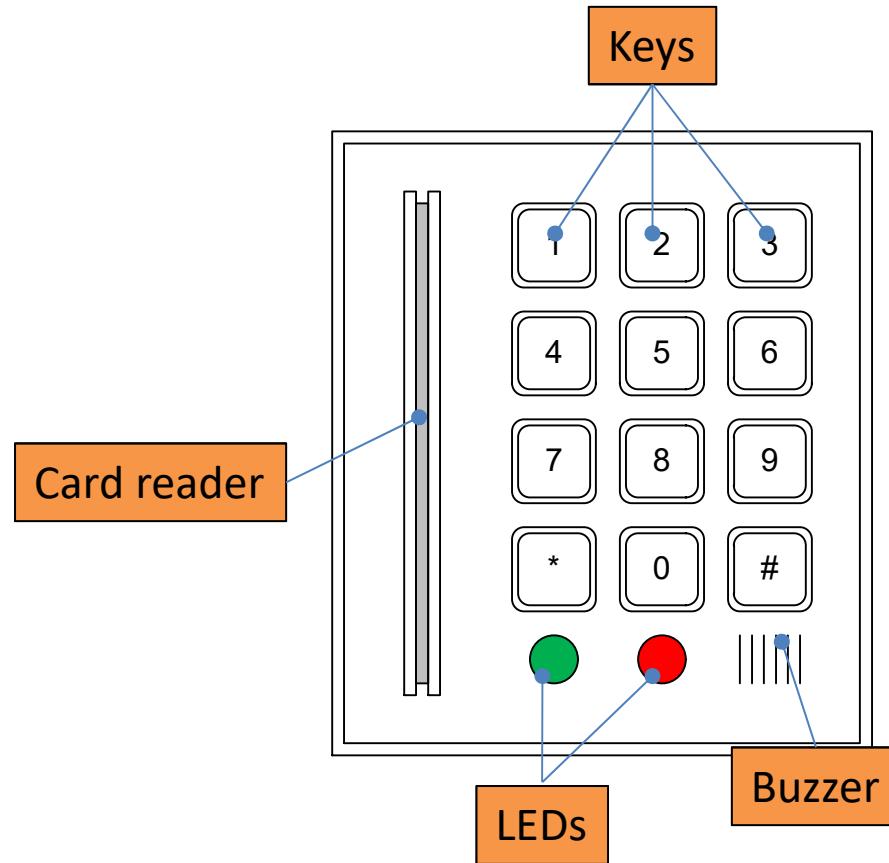
Specifikation med Use Cases



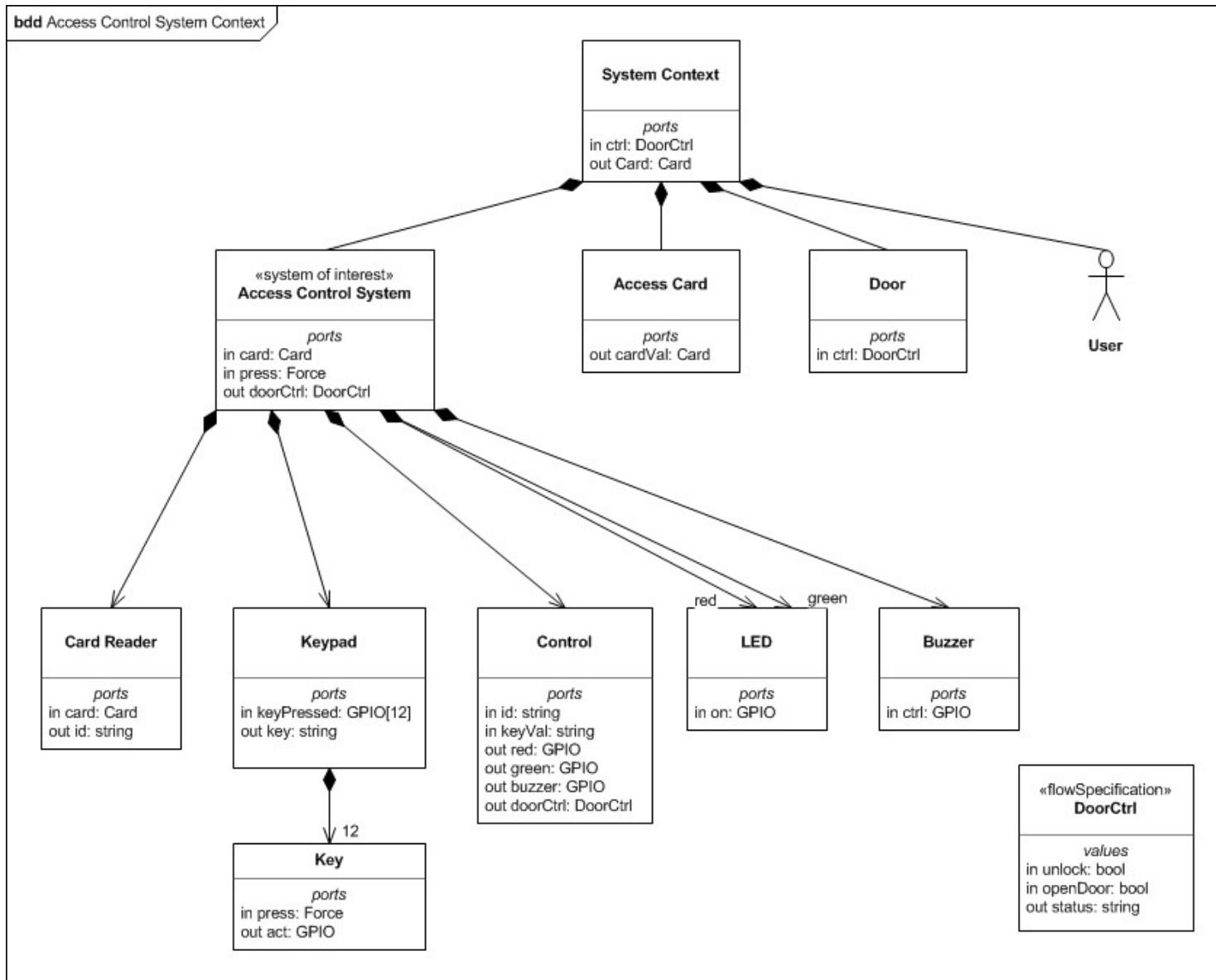
SysML – Systembeskrivelse med diagrammer



Example: Access Control System (1/3)

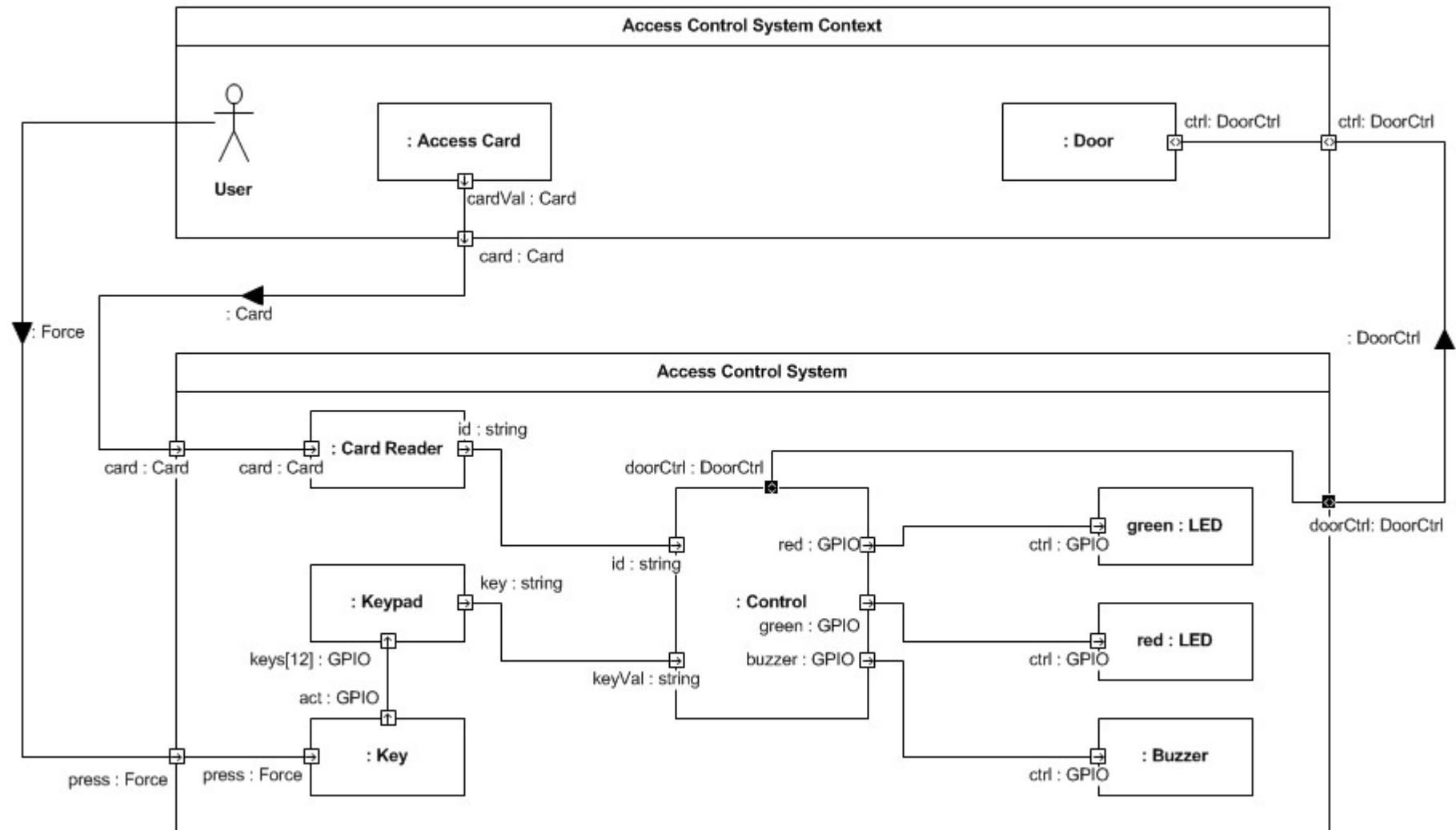


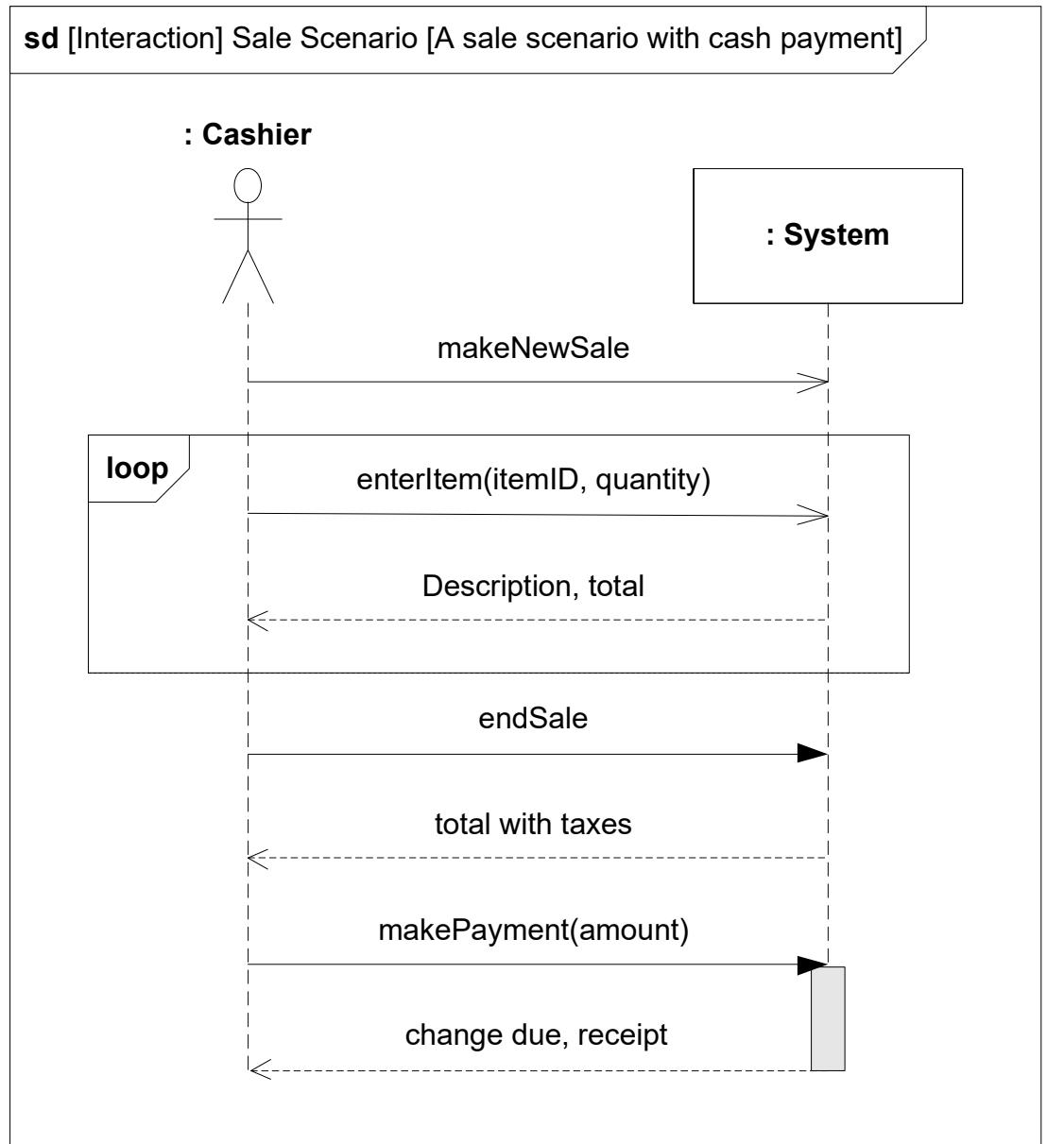
Example: Access Control System BDD (2/3)

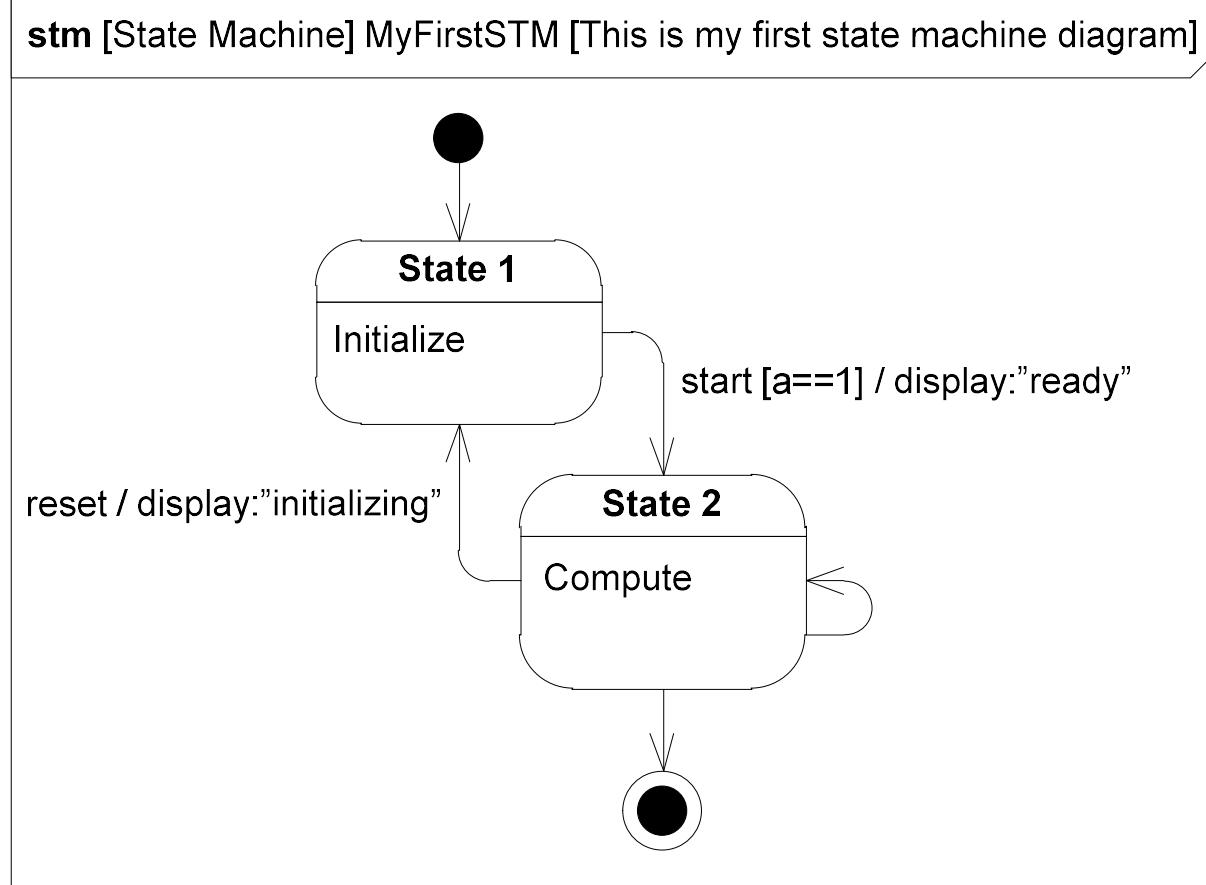


Example: Access Control System IBD (2/3)

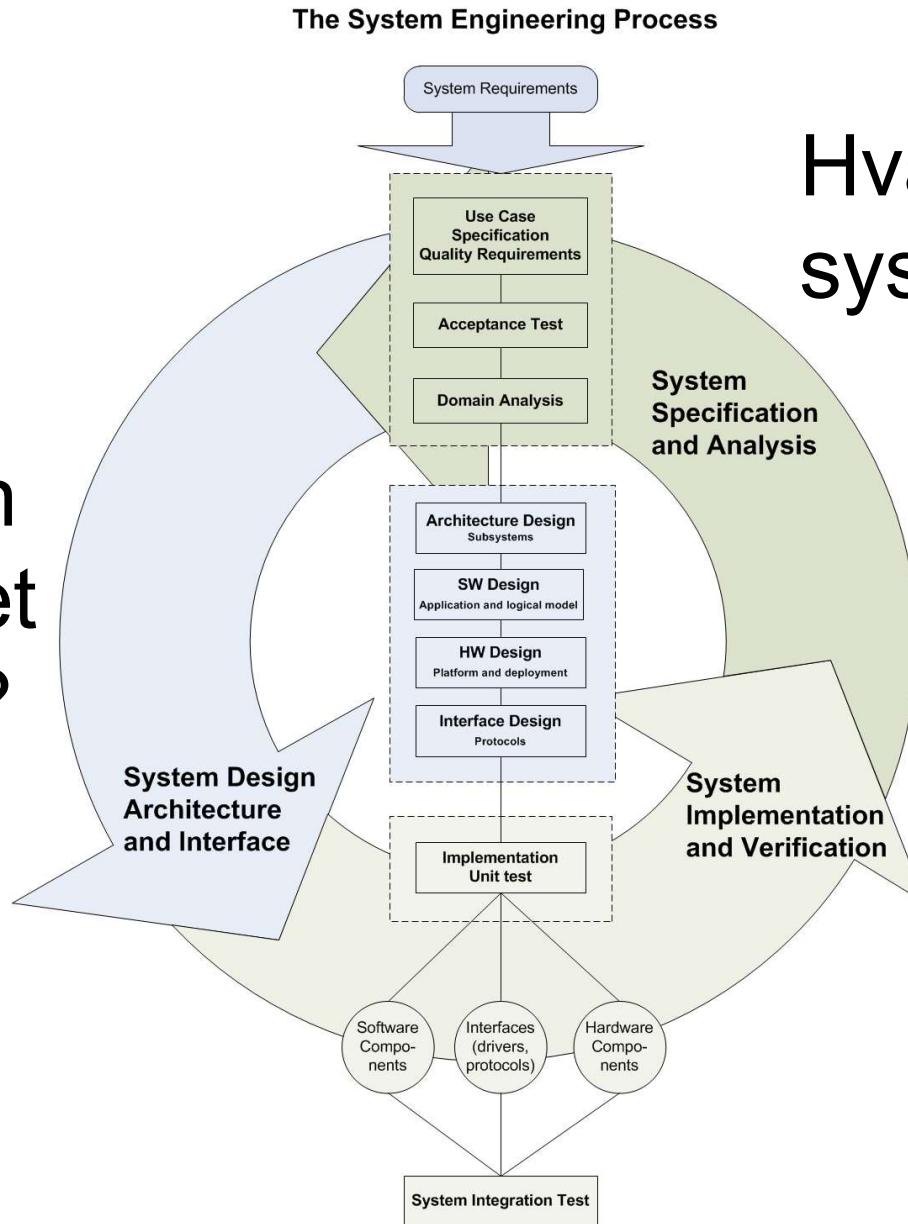
ibd Access Control System







Hvordan
systemet
gør det?



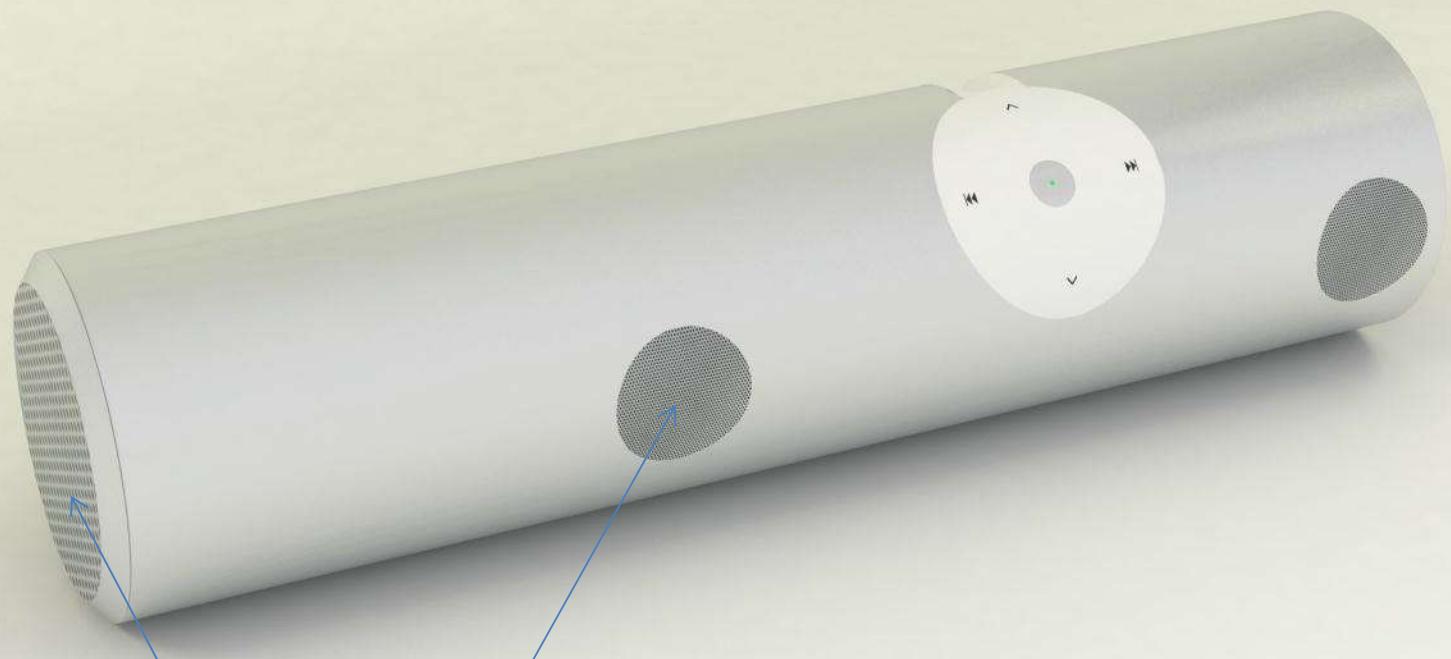
Hvad skal
systemet gøre?

Virker
systemet?

Dagens menu

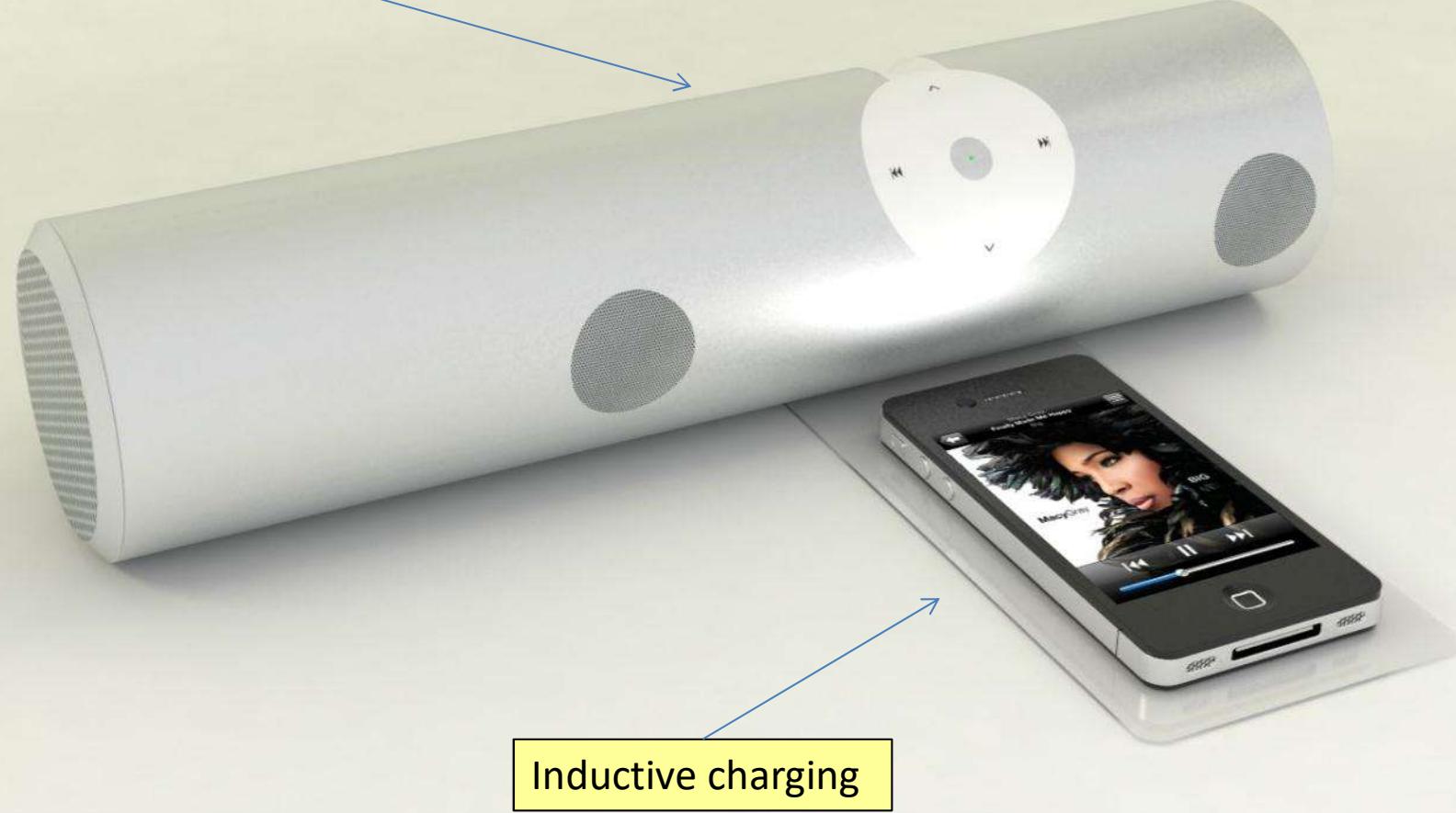
- Kursus-introduktion
- Gruppedannelse, opgaver og kommunikation
- Hvad er systems engineering?
- Case study

BeoSound F



High-quality Speakers

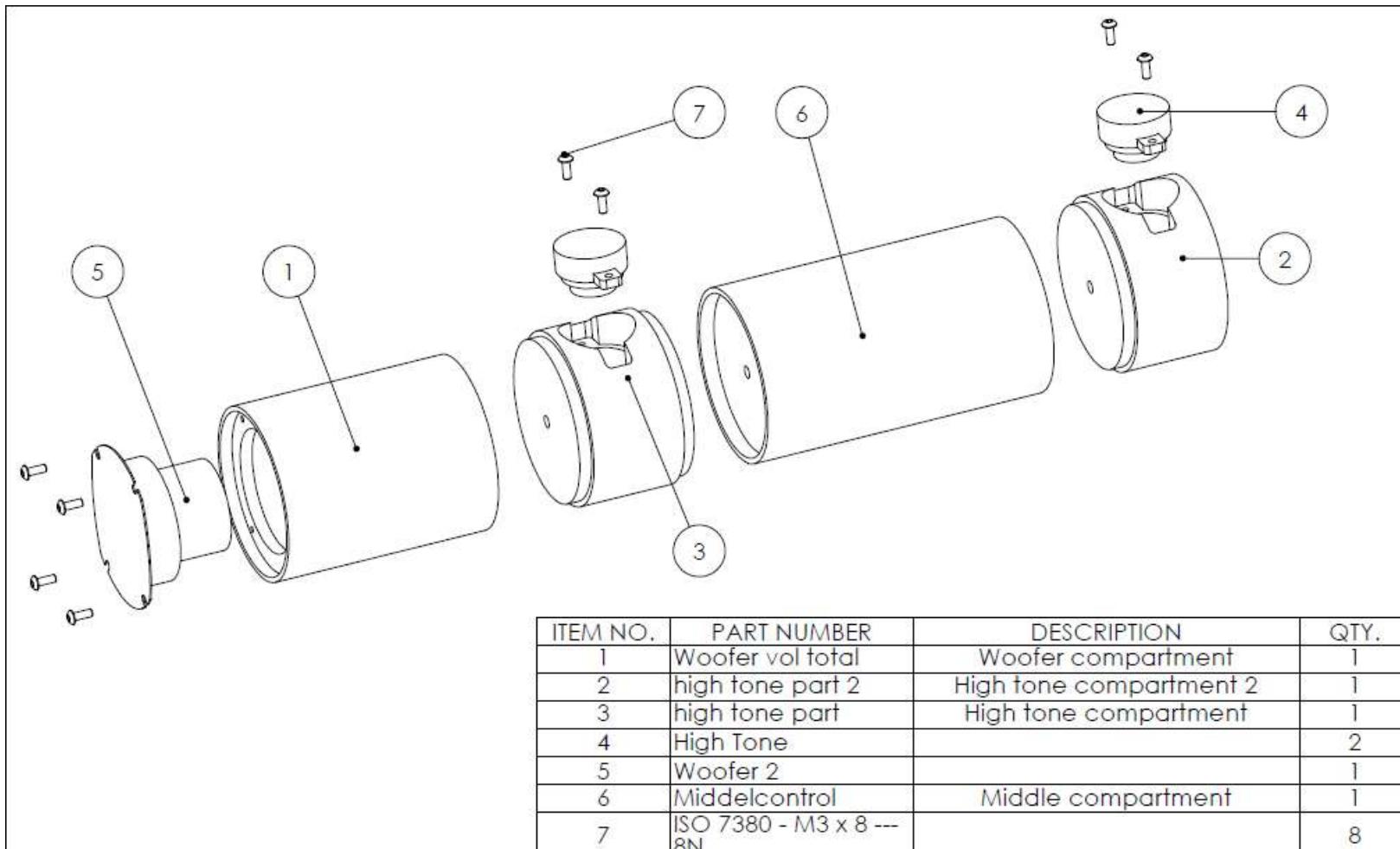
BeoSound F



BeoSound F

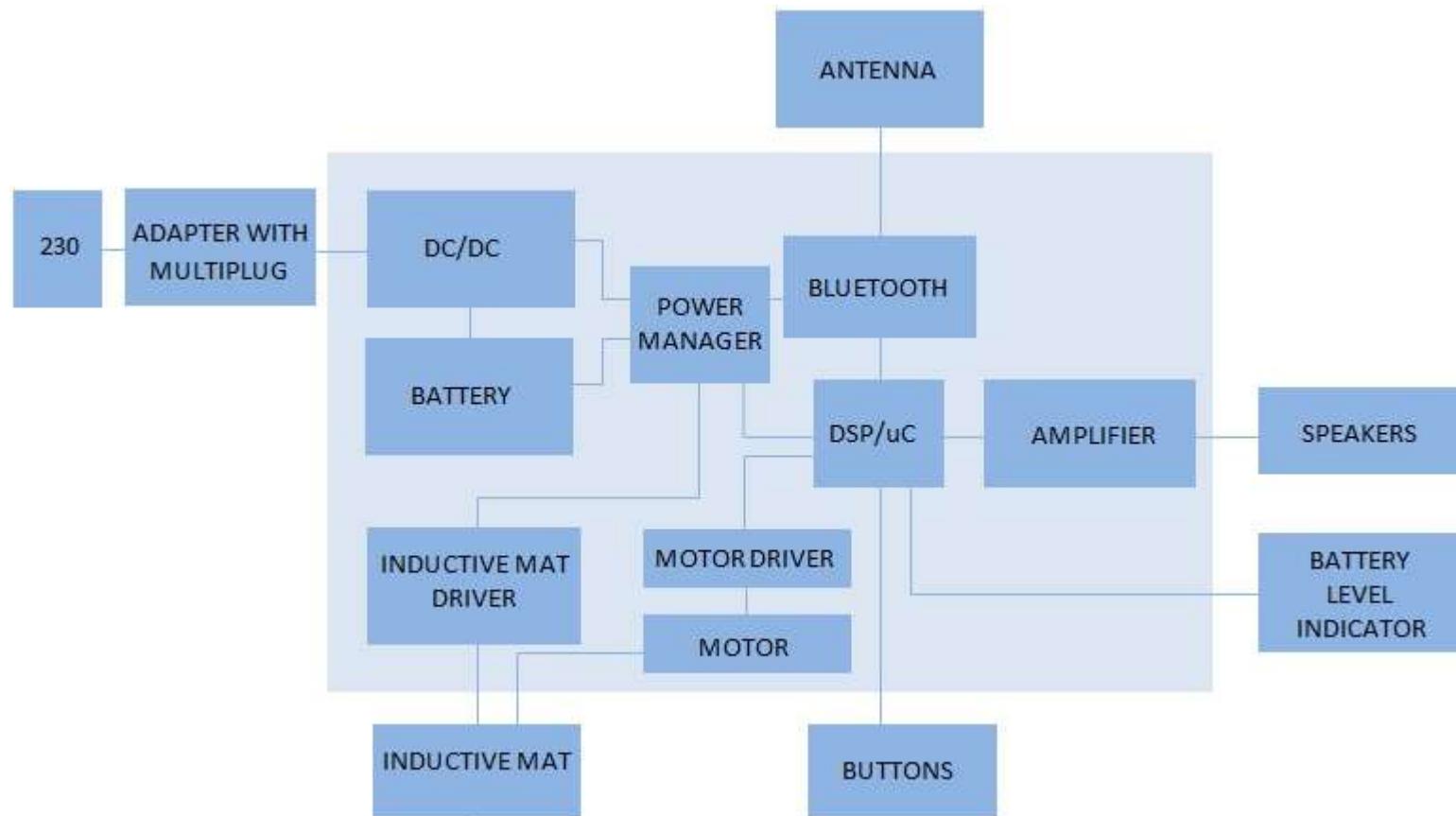


Exploded View

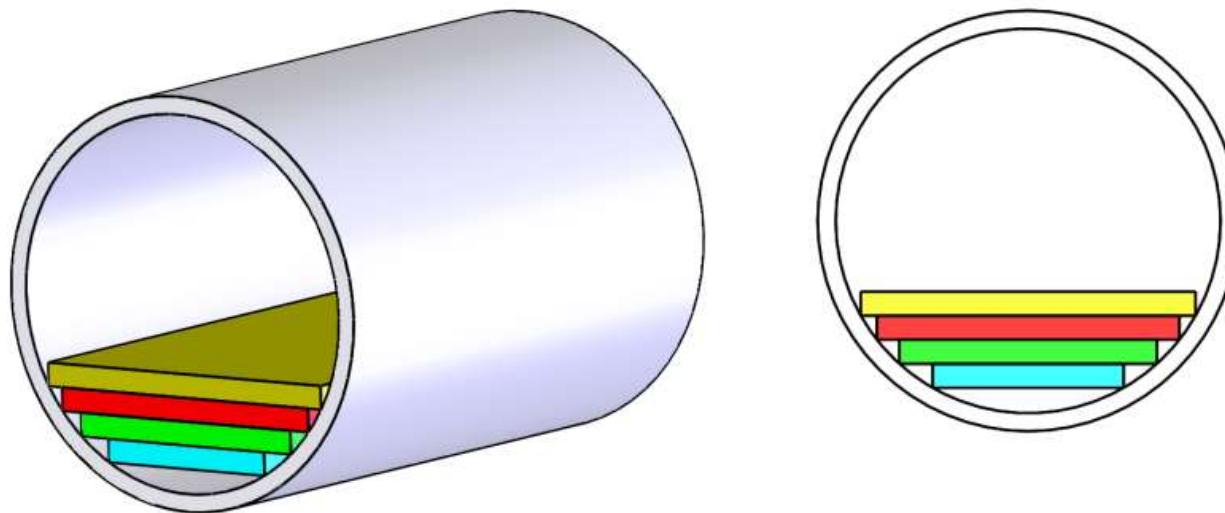


ITEM NO.	PART NUMBER	DESCRIPTION	QTY.
1	Woofer vol total	Woofer compartment	1
2	high tone part 2	High tone compartment 2	1
3	high tone part	High tone compartment	1
4	High Tone		2
5	Woofer 2		1
6	Middelcontrol	Middle compartment	1
7	ISO 7380 - M3 x 8 --- 8N		8

Block Diagram



Battery



6 HOURS

OF PLAYING MUSIC + 3 RECHARGES

Case Study

- Fokuser på elektroniskkonstruktion og softwaren
 - Check Blackboard-> **BeoSoundF_ConceptReport.pdf**
- Diskuter en fremgangsmåde for udvikling af systemet
 - Hvad mangler I at vide noget om? Hvad vil I gøre ved det?
 - Hvordan vil I planlægge? Organisere? Udvikle?
 - Hvornår er den færdig?
 - Hvilke udfordringer/risici ser I? Hvad vil I gøre ved det?
 - Hvad kræves der af hardware og software?
 - Hvordan sikrer I kvaliteten af produktet?

Questions?



Specification, Part 1

System Specification

Introduction to System Engineering

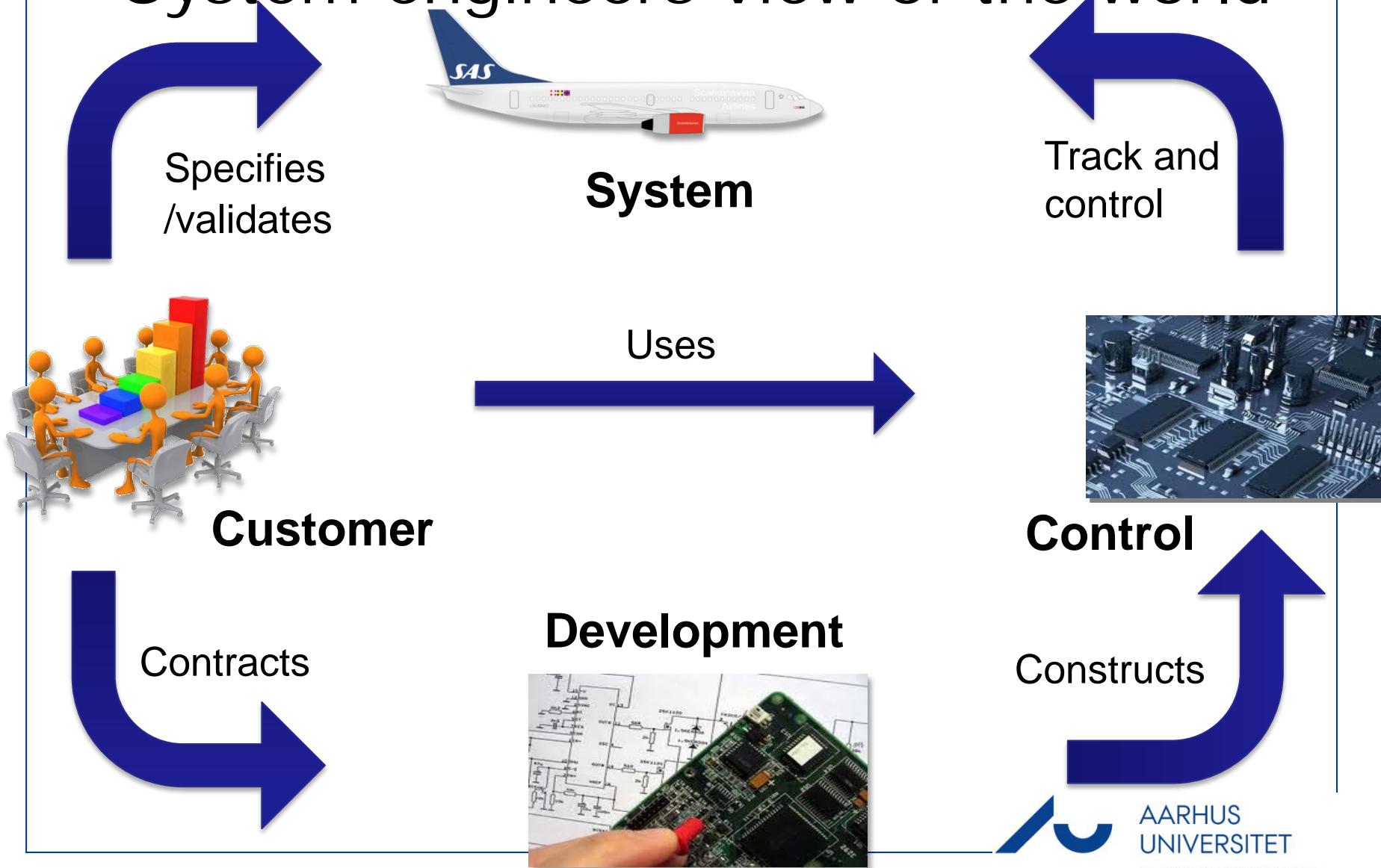


AARHUS
UNIVERSITET
INGENIØRHØJSKOLEN

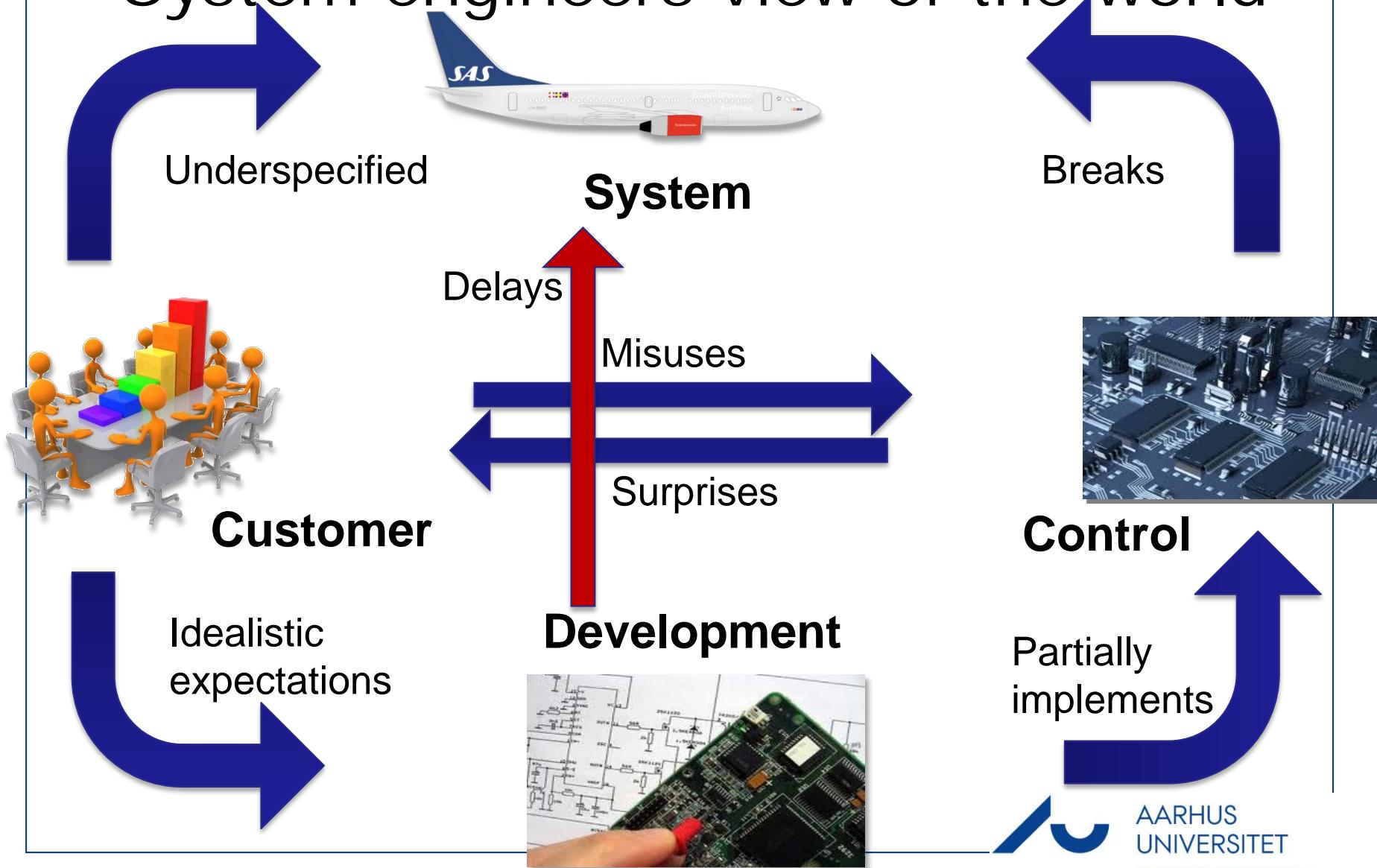
Agenda

- System Specification
- Types of requirements
- Specifying requirements
- Finding requirements (Elicitation)
- Good / Bad Requirements
- Traceability

System engineers view of the world



System engineers view of the world



What is a system specification?

- “a **statement that identifies a capability, characteristic, or quality factor** of a system in order for **it to have value** by a user or a customer to solve a problem or achieve an objective”

Ralph Young, *Requirements Engineering Handbook*, 2004

- Do the right thing
- The **what** and not the **how**



What is a system specification?

- The **Stakeholders** description of a desire functionality or behavior for a system to achieve an objective.
- Functional requirements state what the system is required to do.
- The primary input to the design process
- The baseline against which acceptance tests are carried out.



Why have focus on specification?

- Time, Effort, Price
- Correcting or changing functionality following specification
 - 3 x as much during the design phase
 - 5-10 x as much during implementation
 - 10-100 x as much after release



- Vasa, *10. aug. 1628 – †10. aug. 1628
- L: 69m H: 52m B: 11,7m D: 5m
- Requirements creep
- Inexperience
- Bad test conditions
- Schedule Pressure

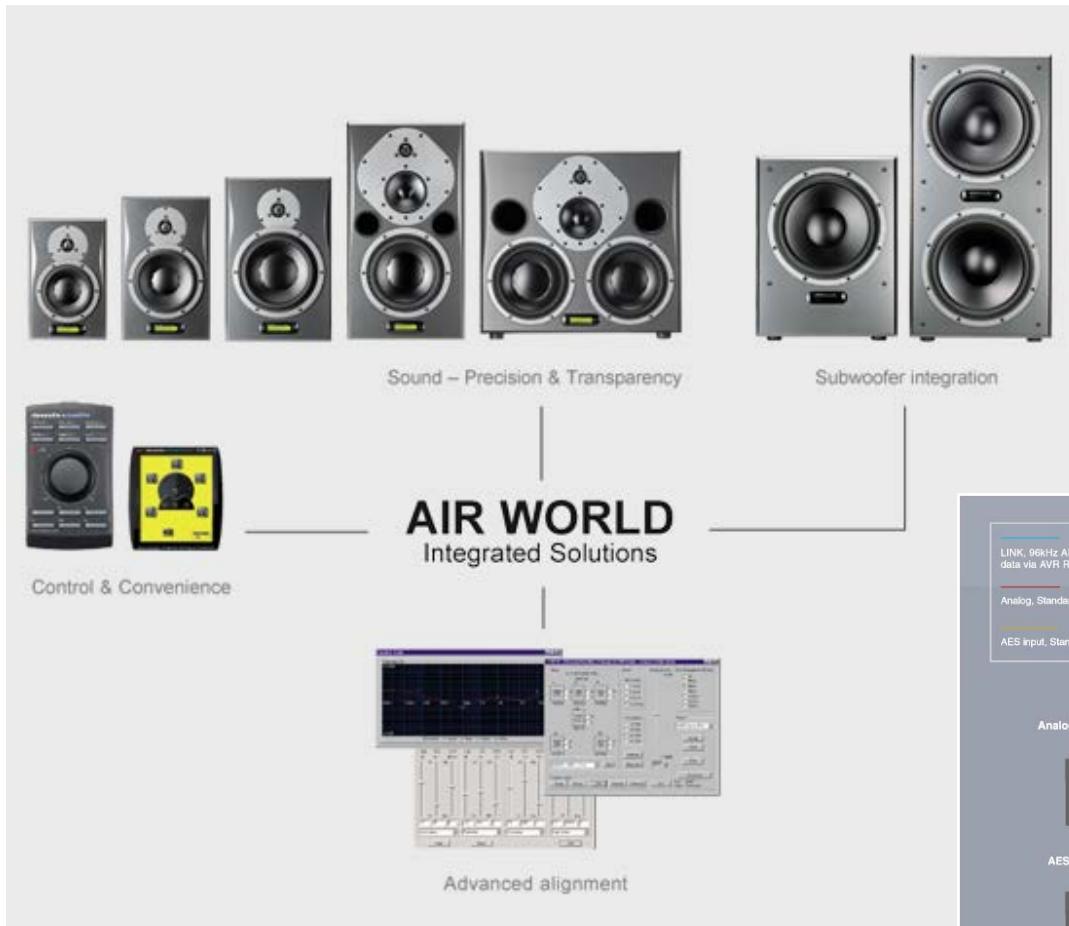


[Why the Vasa Sank: 10 Problems and Some Antidotes for Software Projects, *IEEE Software*, Fairley03]

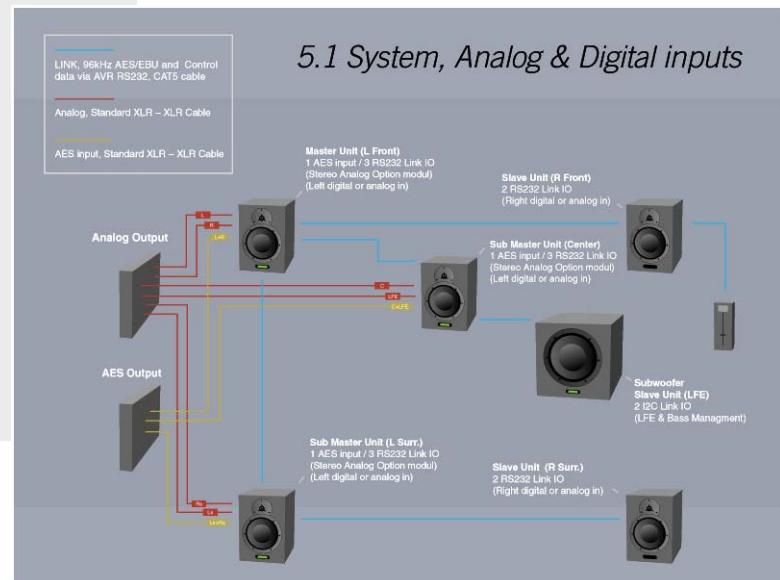


AARHUS
UNIVERSITET
INGENIØRHØJSKOLEN

Dynaudio AIR Series – Studio monitors



- Lack of specification
- Stereo and volume
- Multichannel 7.3



<http://dynaudioprofessional.com/air-series/>



AARHUS
UNIVERSITET
INGENIØRHØJSKOLEN

What to derive from these cases?

- A specification is more than functional requirements
- The specification should attempt to address:
 - Requirements creep / Requirements Change
 - Test conditions
 - Training
 - Realistic Schedule
 - External Constraints (physical, legal)
 - Ownership and Stakeholders
 - Existing systems/hardware

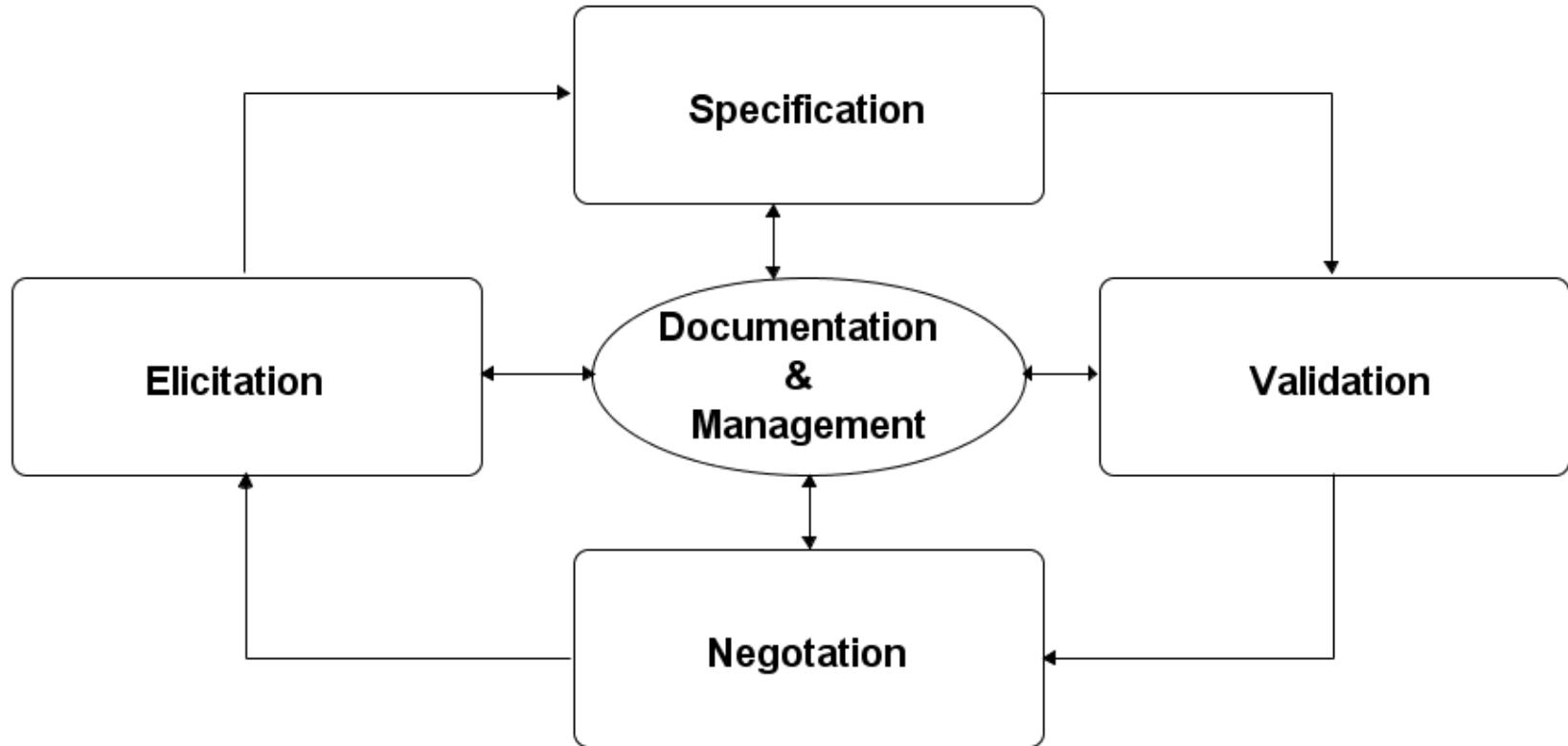


Who uses a specification

Stakeholder	Use of specification
Customer	Fullfillment of business goal Contract
Manager	Scheduling Progress measuring
System Engineer	Design Functionality Constraints
Test Engineer and QA personnel	Test planning Verification Validation



Specification Process



Requirements Specification

- Documentation of the specification
- Principally the outcome of elicitation
- Complete description of the behavior and constraints of the system to be developed
- Baseline for communication between stakeholders
- Often has high demands for versioning and traceability



Types of requirements

- ***Functional***
 - What the system should do (behaviours)
- ***Non-functional (Quality-demands)***
 - Qualities or criteria of the system, rather than specific behaviour
- Other categorizations exists:
 - Customer Requirements
 - Domain Requirements (Business Rules)



Prioritisation of Requirements

- MoSCoW Method (prioritisation technique)

M - MUST (skal) have this

S - SHOULD (bør) have this if possible

C - COULD (kan) have this if it does not affect anything

W - WON'T (vil ikke) have this time, but WOULD like in the future



- The system **must** be able to make a cup of coffee
- The system **should** be able to indicate need of service
- The system **could** be able to add milk
- The system **won't** be able to automatically refill water



Functional Requirements

- Functional
 - Describes system services
 - Requirement for each input and output
 - Behavioural requirements
 - Use Cases



Exercise 1



- Who would the stakeholders be?
- Write 5-10 functional requirements for the reverse vending machine
(Use MoSCoW)



Quality Demands/Non-functional Requirements

- Qualities or Constraints on the services or functions offered by the system

Qualities are properties or characteristics of the system that its stakeholders care about and hence will affect their degree of satisfaction with the system.

[Defining Non-Functional Requirements, Malan01]

- Quality demands/NFRs should satisfy two attributes
 - **Must be verifiable (measurable metrics)**
 - **Should be objective**



Example - Treasure Robot (3. Semester project)

- Driving robot
- Obstacle sensor
- Metal detector
- GPS

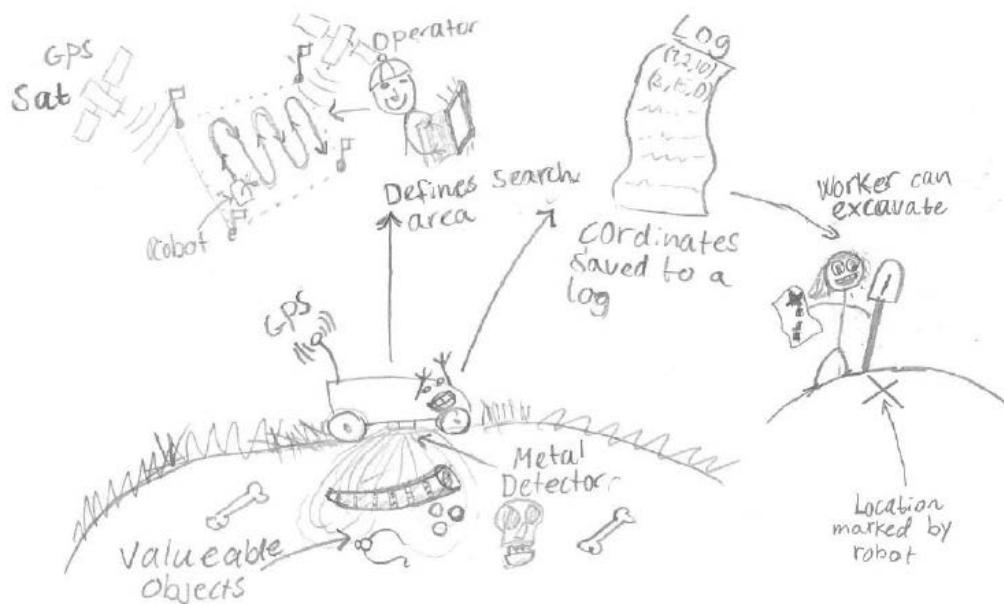
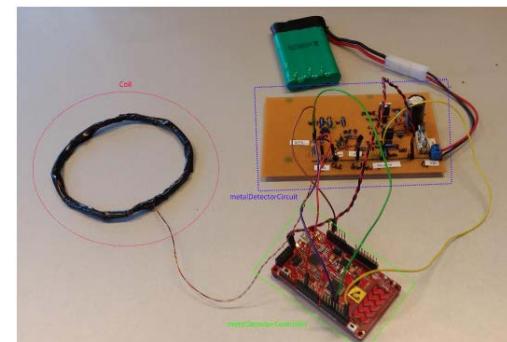
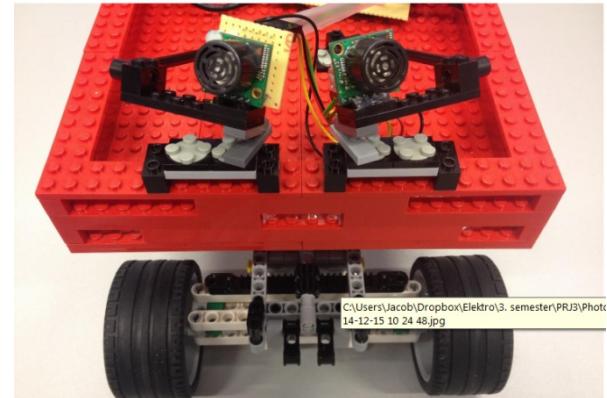


Figure 1 TreasureBot rich picture



Treasure Robot - Non-functional Requirements

3.1 Battery

- 3.1.1. Battery life **should** be minimum of 20 min when the car is in running mode
- 3.1.2. Battery life **should** be minimum of 1 hours when the car is idle

3.2 Robot

- 3.2.1. The robot **must** not exceed the dimension of 40 cm long, 25 cm wide and 15 cm tall
- 3.2.2. The robot **must** have enough storage capacity to be able to drive and record data for 20 min.
- 3.2.3. Should be able to save GPS-location every 5 sec. +/- ½ sec.

3.3 Metal detection

- 3.3.1. Must be able to detect metal to a depth of min. 5 cm from sensor, when driving on dirt or grass
- 3.3.2. Must be able to detect metal to a depth of min. 3 cm from sensor, when driving on gravel

3.4 Obstacle sensor

- 3.4.1. Must be able to detect a black box with dimensions 10x10x10 cm, from a distance of 1 m

3.5 GPS

- 3.3.1. Should have an accuracy of minimum 3 m radius on a clear day
- 3.3.2. Should have an accuracy of minimum 5 m radius on a cloudy day

Types of requirements

- FURPS+ (Robert Grady, Hewlett-Packard)

Functionality

Usability

Reliability

Performance

Supportability

+ (Design and Physical constraints, Interfaces, Legal, Test, Reuse, Economic constraints, Aesthetics, Comprehensibility, Technology tradeoffs)



Usability

- Characteristics
 - Operability
 - Accessibility
 - User Interface (If any)
 - Documentation
- Metrics
 - ~~The system should be easy to use~~
 - Max. number of errors made by users for a specific task over a time period.
 - Time to learn a certain functionality



Reliability (1/2)

- Characteristics
 - Reliability
 - is the probability of a system to perform a required function under stated conditions
 - Availability
 - is a function of how often failures occur, repair time and maintenance interval
 - Maintainability
 - is the ability of a system to restore to a specified condition



Reliability (2/2)

- Metrics
 - Reliability
 - Mean time between failure (MTBF)
 - Availability
 - Maintainability
 - Mean time to restore (MTTR)

$$\frac{MTBF}{MTBF+MTTR} == \frac{UPTIME}{UPTIME+DOWNTIME}$$



Performance

- Characteristics
 - Throughput
 - Response time
 - Start-up time
 - Capacity & Efficiency Constraints
- Metrics
 - Time
 - Specifics is out of scope for this course
 - *95% of the transactions shall be processed in less than 1 second at 80 % load*



Supportability

- Characteristics
 - Compatibility,
 - Installability,
 - Localizability
 - Maintainability,
- Metrics
 - Same as with Usability
 - Measure of success under specified scenarios



+

- Legal
 - Data Protection Act, Health and Safety act,
- Technology trade off
 - Balance between two incompatible features
- Test
 - Conditions, Environment, Access
- Reuse
 - Existing systems, parts, modules
- Environmental
 - RoHS, WEEE, EMC, etc.



Exercise 2

- Find 5-10 NFR / Quality requirements for this coffee machine using (F)URPS+

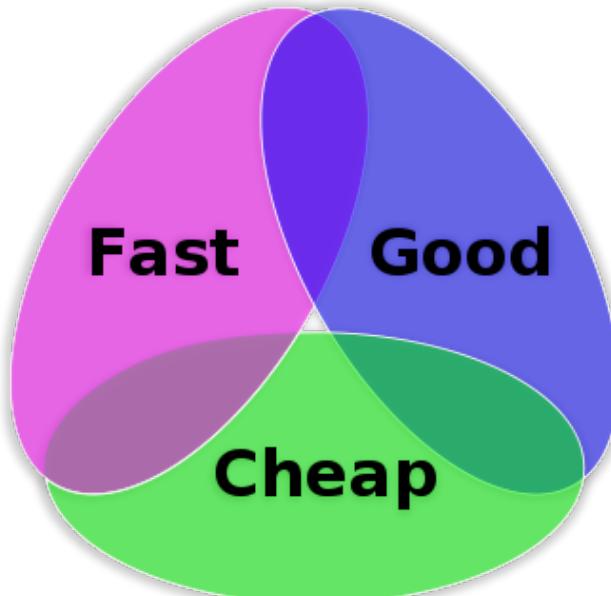


Usability
Reliability
Performance
Supportability



Challenges in Elicitation

- The larger the project, the more difficult it is to grasp
- Users' expectations are often unrealistic, and compromises must be made between economy and features



Requirements Elicitation Techniques

- Techniques
 - Interview
 - Stakeholder analysis
 - Brainstorm
 - Prototype
 - (Requirements Workshop)
 - (Task Demonstration)
 - (Role Playing)



Interview

- Simple direct technique
- Context-free questions can help achieve bias-free interviews
- Convergence on some common needs will initiate a “requirements repository” for use during the project.
- A questionnaire is no substitute for an interview



Stakeholder analysis

- Who are the stakeholders?
- What are their goals?
- Which risks and costs do they see?



Brainstorm

- Brainstorming involves both idea generation and idea reduction
- The most creative, innovative ideas often result from combining, seemingly unrelated ideas
- Various voting techniques may be used to prioritize the ideas created
- Open to all suggestions
- Suggestions often spawn new ideas



Prototype session

- Preliminary model built for demonstration purposes
- The customer may be more likely to view the prototype and react to it, than to read the specification and react to it
- The prototype provides quick feedback
- Prototype displays unanticipated aspects of the systems behavior



Good Requirements

- Correct – specifying something actually needed
- Unambiguous – only one interpretation
- Complete – includes all significant requirements
- Consistent – no requirements conflict
- Verifiable – all requirements can be proven by test
- Modifiable – changes can easily be made to the requirements
- Traceable – the origin of each requirement is clear



Requirements Traceability Matrix

- Determine the two-way mapping between Requirements and Features/Test
- Are all features mapped to a requirement? And are each requirements fulfilled by a feature?
- Requirements and Test/Verification, or Requirements and Features

Requirements	REQ 1	REQ 2	REQ 3	REQ 4	REQ 5
Test Cases					
1.1	X	X			
1.2			X		
1.3			X		
2.1			X		
2.2	X	X		X	X
2.3	X	X			
3.1	X	X	X	X	X

A specification that will not fit on
one page of 8.5x11 inch paper
cannot be understood.

Mark Ardis

Professor
Rochester Institute of Technology



AARHUS
UNIVERSITET
INGENIØRHØJSKOLEN

Exercise 3 - BeoSound F (pp.13-15)

- Write 10-20 good requirements
- (F)URPS+
 - Usability
 - Reliability: MTBF and availability
 - Performance
 - Supportability
- MoSCoW
 - Must (skal)
 - Should (bør)
 - Could (kunne)



Questions



Specification, Part 2

Use Cases

Introduction to System Engineering



AARHUS
UNIVERSITET
INGENIØRHØJSKOLEN

Agenda

- What is a Use Case
- Why use Use Cases
- Use Case diagrams
- Actors
- Scenarios : Format and styles
- Guidelines: Finding and describing actors and UC's
- Good and Bad Use Cases

What is a Use Case (1/2)

A use case is a specific way of using the system by performing some part of the functionality. Each use case constitutes a complete course of events initiated by an actor, and it specifies the interaction that takes place between an actor and the system

- A textual description of events between a user and a system
- In order to discover and describe requirements
- Are described from a users view or the environment (not from the systems view)



What is a Use Case (2/2)

“If you design a new house and you are reasoning about how you and your family will use it, this is use case-based analysis. You consider the various ways in which you’ll use the house, and these use cases drive the design.”

– Booch



What are they used for

- Used to specify the functional requirements
 - In FURPS+, they give emphasis to the F
- Outside-in approach, where the functionality is described from the users viewpoint
 - Not from the developers



Use Case Example

Buy Product

1. *Customer* browses through catalog and selects items to buy
2. *Customer* goes to check out
3. *Customer* fills in shipping information
4. *Customer* fills in credit card information
5. *System* authorizes purchase
- 5a [Authorization fails]
Customer may go to 4. or Cancel



Use Case terminology

Buy Product

Actor

1. *Customer browses through catalog and selects items to buy*

Reference

2. *Customer goes to check out*
3. *Customer fills in shipping information*
4. *Customer fills in credit card information*

5. *System authorizes purchase*

Extensions/
Alternate Flows

- 5a [Authorization fails] ←
Customer may go to 4. or Cancel

Use Case terminology

- A use case either reaches its goal or fails;
- If only a main success scenario is given, then success is assumed
- Use Case naming rule:
Described in terms of obtaining a goal for a given actor

Examples:

Withdraw Money
Check Balance



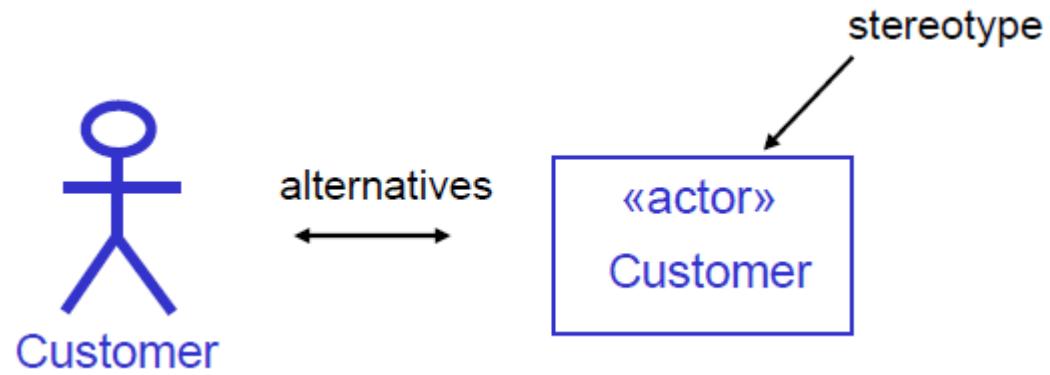
Why use Use Cases

- Capturing requirements of a system
- Validating systems (all use cases are realized)
- Can drive implementation and tests
- Use Case Modeling is
 - A simple concept
 - User-friendly (allow customers to contribute)
 - Effective
- Discovery and definition are the goals of Use Cases



Actors

- Role – more precise translation from Swedish
- An actor describes an object, external to the system, who interacts with the system
- Actors represents:
 - Persons
 - Other systems
 - HW devices
- UML notation:

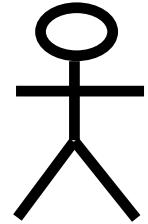


Actors

- Person actors:
 - Describes the role played by a person who interacts with the system
 - The same person can play different roles over time
- Other systems:
 - Describes the external systems who communicates with the system under development
- HW devices:
 - A concrete HW unit can also be subdivided in different logical roles played by the HW unit or device



Actor type



- Primary Actor
 - Has goals to be fulfilled by system
- Supporting Actor (alternatively Secondary Actor)
 - Provides service to the system
- Offstage Actor
 - Interested in the behavior, but no contribution



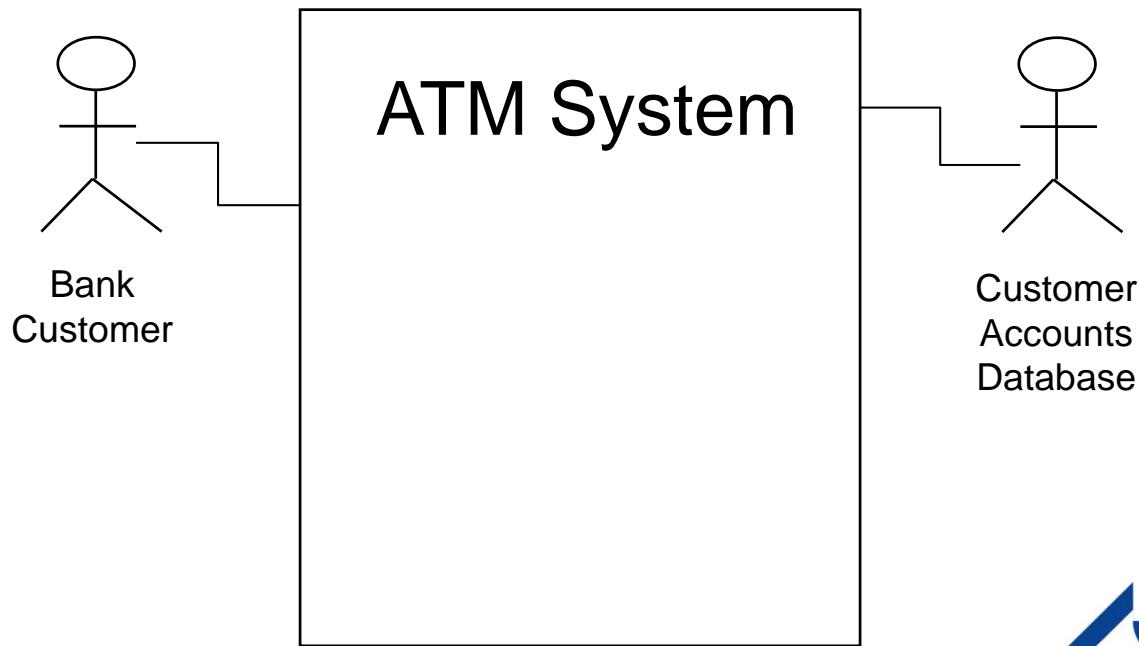
Actor description

Name of actor:	Shopper
Alternate references	Customer
Type:	Primary
Description:	<p>The Shopper is the sole end user of the system ...</p> <p>Wants to ...</p>



Actor-Context diagram

- Supplies overview of the actors in relation to the system boundary
- Essentially a diagram with actors, the system boundary and no use cases.



System Boundary

- Boundary between the inside and the outside of the system
- Determining the level of abstraction
- There may be systems within systems
 - One system may be made up of several subsystems, which interact with each other
- Identify interactions between the system and its environment (stimuli and responses)

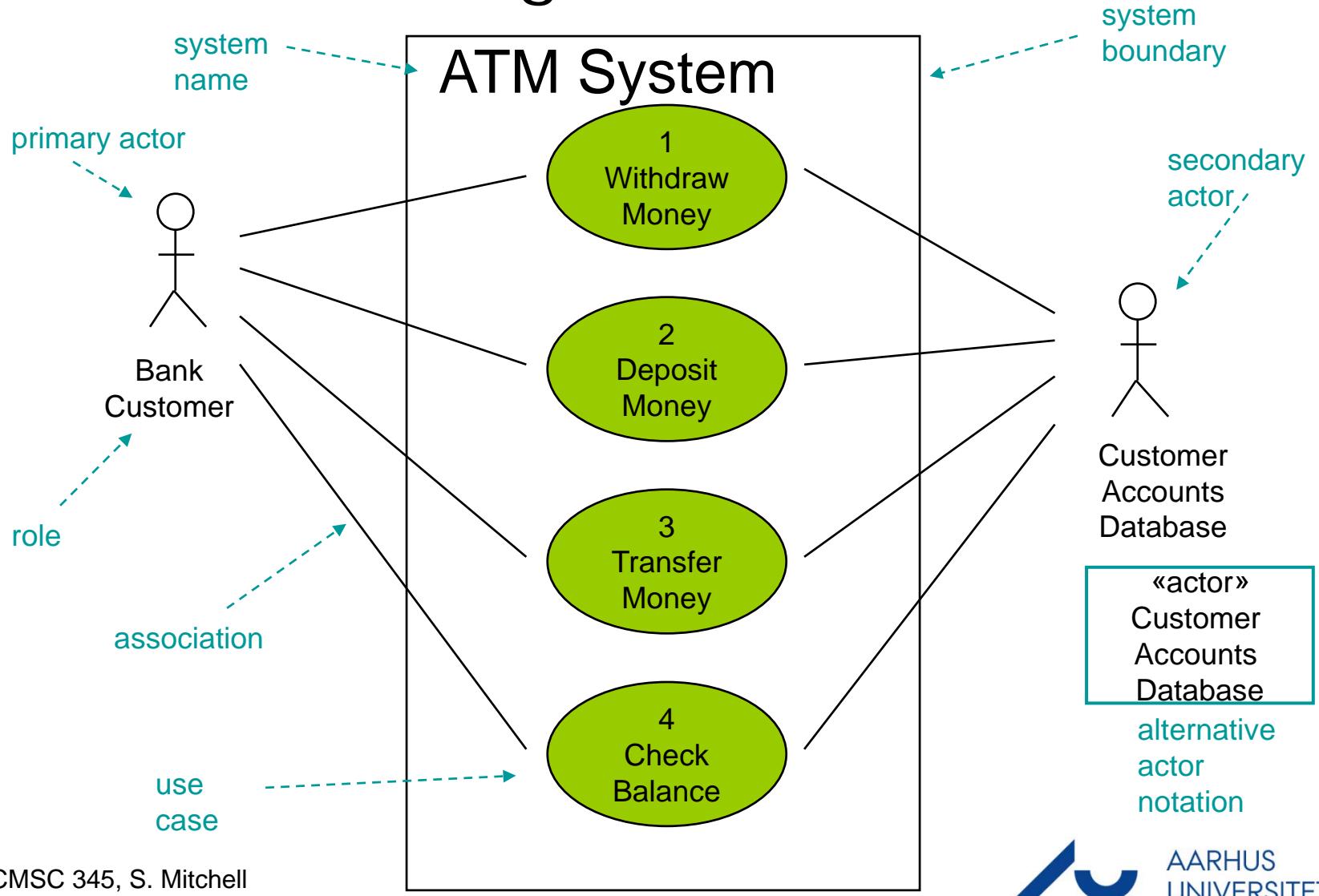


Why money is not an actor

- No interest in the scenario
- A means to fulfil the scenario



Use Case diagrams



Exercise 1

- Bomanlæg
 - Find Boundary
 - Actors
- Ismaskine
 - Find Boundary
 - Actors



Use Case diagrams

- A way of visualizing the relationships
 - between actors and use cases
 - among use cases
- A graphical table of contents for the use case set



Use Case diagrams (Parkerings-automat)

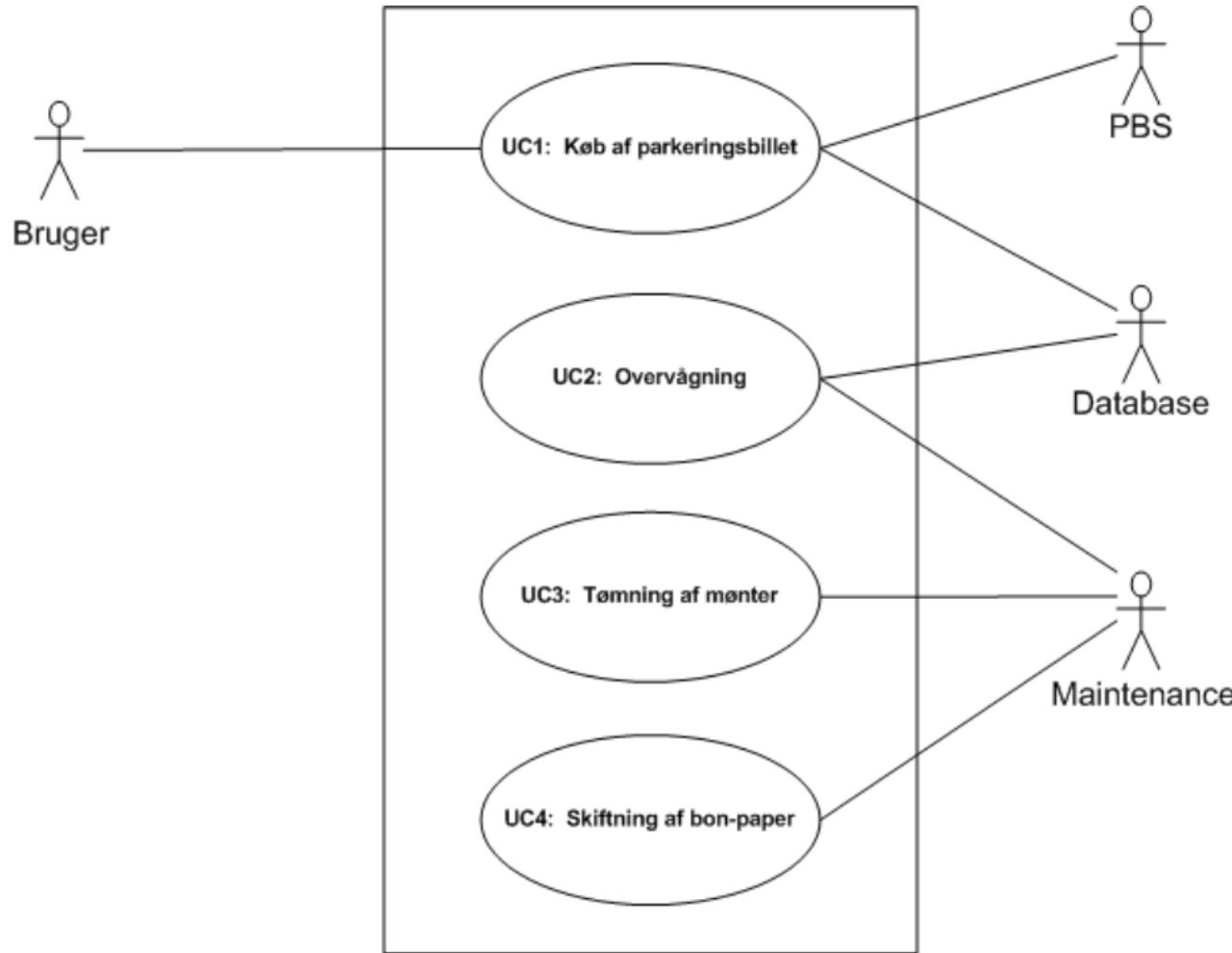


Diagram Purpose

- A Use Case diagram is an overview of Use case scenarios
- Diagrams shall only contain Use Cases that actually exists
- Do not join several Use Cases into one in a diagram



Exercise 2

- Medical Consultation System
- Draw a Use Case diagram of the following:

	Patient	Doctor	EPR/EPJ	National Board of Health
Make Appointment	Primary	Secondary	-	-
Cancel Appointment	Primary	Secondary	-	-
Access Patient Record	-	Primary	Secondary	Offstage

Scenarios

- The Main Success Scenario is the scenario where the actor obtains its goal
 - - is also called
 - Sunshine scenario
 - Happy scenario
- Extensions / Alternative Flows
 - The alternative flows can be more comprehensive than the main scenario



Styles

- Essential:
 - Focus is on intent
 - Free of technology and mechanisms
 - Avoid making user interface decisions
- Concrete:
 - UI decisions are embedded in the use case text
 - e.g. “Admin enters ID and password in the dialog box, (see picture X)”



Scenarios : Formats

- Different ways of structuring use cases:
 - Brief
 - (Casual)
 - Fully Dressed



Scenarios : Formats : Brief

- 1-6 sentence description of behavior
- Mention only most significant behavior and failures
- Short enough to put many on a page
- Used to
 - Estimate complexity
 - To get a sense of subject and scope



Scenarios : Formats : Brief : Example

Process Sale:

A customer arrives at a checkout with items to purchase.

The cashier uses the POS system to record each purchased item.

The system presents a running total and line-item details.

The customer enters payment information, which the system validates and records.

The system updates inventory.

The customer receives a receipt from the system and then leaves with the items.

Scenarios : Formats : Fully dressed

- Paragraphs written in a numbered form
- Includes all step and alternate flows in detail
- Normally follows a defined template of supporting sections



Fully-dressed Template

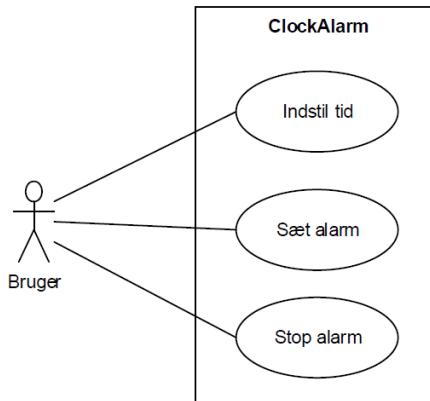
Name	Name of UC (Start with verb)
Goal	What is achieved by the UC
Initiation	Actor, System initiation
Actors and Stakeholders	List of actors Actor role (type)
References	Other use cases referenced
Number of concurrent occurrences	2, 10, none
Precondition	What must be true on start
Postcondition	What is true on completion
Main Scenario	Happy path scenario
Extension	Alternate flows
Data Variations List	e.g. Data Formats



Fully-dressed elements

- Actors and Stakeholders
 - list of stakeholders and their key interests in the use case
- Pre/Post conditions
 - assumptions before and success guarantees
- Data Variations List
 - technical variations in how data is defined





Fully-dressed example (Alarm Clock)

Navn:

Sæt alarm

Mål:

Bruger ønsker at sætte alarmtiden.

Initiering:

Bruger trykker på ALARM knappen

Aktører:

Bruger - primær

Samtidige forekomster:

1

Prækondition:

Uret er tændt og operationel

Postkondition:

Alarmen er sat til den ønskede tid

Hovedscenarie:

1. Bruger trykker på ALARM
2. Urets display viser tidligere alarm
[Extension 1a: Ingen tidligere alarm]
3. Bruger trykker på henholdsvis HOUR og MIN
4. Uret optæller time og minut visningen for alarm
5. Bruger trykker på ALARM for at afslutte indstillingen
6. Uret skifter tilbage til at vise klokken

Udvidelser/undtagelser:

[Extension 1a: Ingen tidligere alarm]

Alarm indstillingen starter ved 00:00.

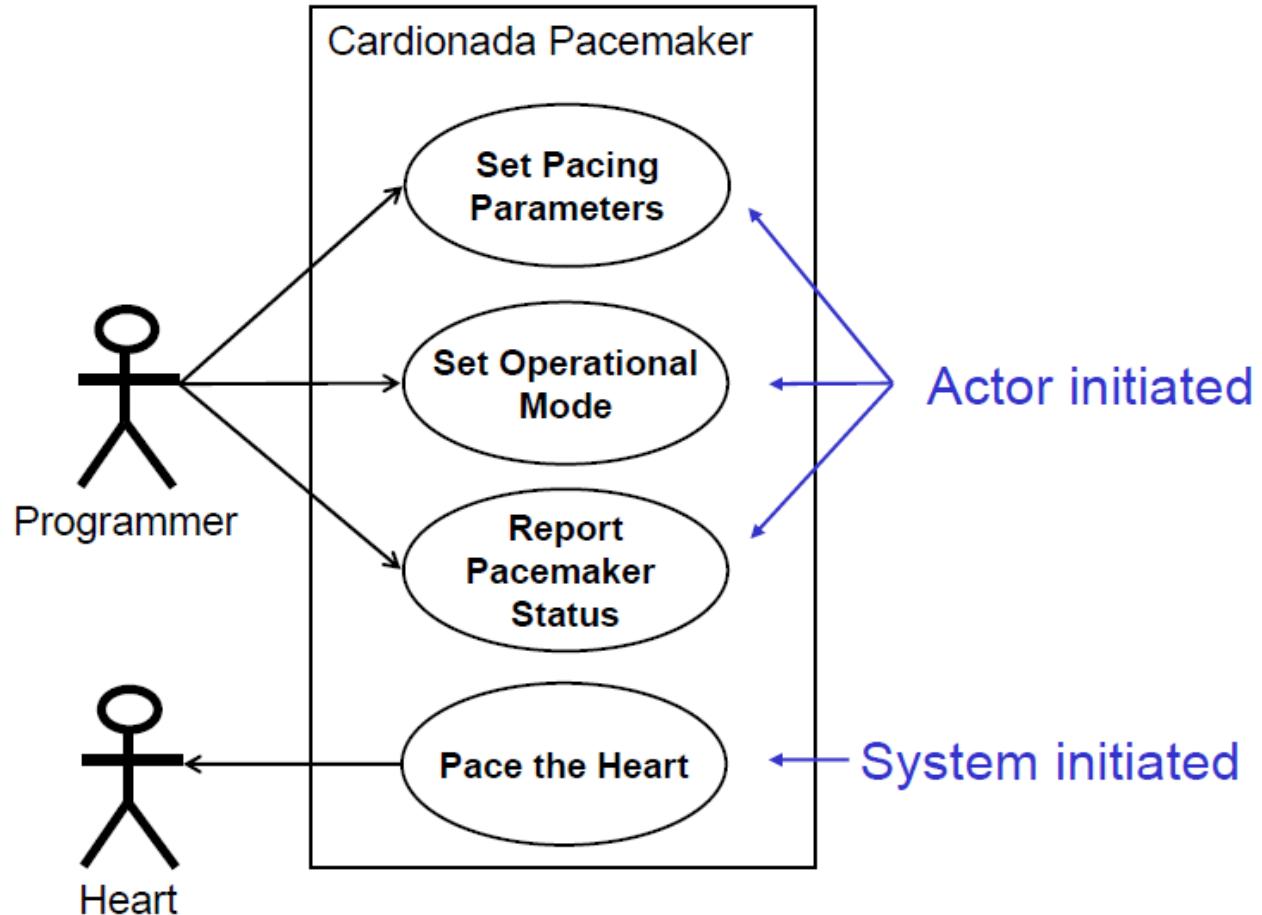


Use Case activation

- Actor initiated:
 - Actor takes initiative to activate a Use Case
- System initiated:
 - Use Cases activated by the system, is very common in technical systems
 - Periodic activated Use Cases
 - Aperiodic activated Use Cases, where a Use Case is started, when a given condition is true



Use Case activation



Guidelines: Building a System in UC's

1. Name the system scope
2. Brainstorm and list the primary actors
3. Brainstorm and exhaustively list user goals for the system.
4. Select one use case to expand
5. Write the main success scenario
6. Brainstorm and exhaustively list the extension conditions
7. Write the extension-handling steps

Finding and describing actors

- Identify
 - Ask the End-Users
 - Documentation
- Issues
 - Roles Vs. Job Titles
 - Time



Finding and describing use cases

- Scenario Driven
 - Find measurable value
 - Business events
 - Services actor needs / supplies
 - Information needed
- Actor/Responsibility
- Mission decomposition



Exercise 3

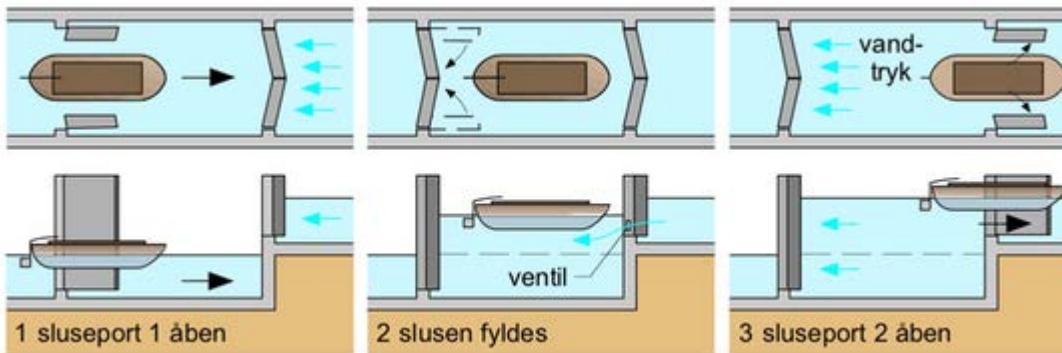
- Bilvaskehald.pdf



AARHUS
UNIVERSITET
INGENIØRHØJSKOLEN

Mandatory Exercise A

- Kaffeautomat
- Slusesystem



- Requirements, Use Cases and Accept Test



Extra topics for Use Cases



AARHUS
UNIVERSITET
INGENIØRHØJSKOLEN

Developing a Use Case

Start out by answering these questions:

- Who are the primary and secondary actors?
- What are each (primary) actor's goals?
- What preconditions must exist before the story begins?
- What main tasks or functions are performed by each actor?
- What exceptions will need to be considered as the story develops?
- What variations in the actors' interactions are possible?
- What system information will each actor acquire, produce, or change?
- What information does each actor need from the system?



Developing a Use Case

- Use case names should start with a verb.
 - **DO:** Rent Items
 - **DON'T:** Item Rental
- Actor names should be capitalized.
- Use cases should be written in the active voice, using actors.

DO: Customer arrives with videos to rent.

DON'T: Videos are brought to the cash register by a Customer.
- Be as terse as possible while still being clear.

DO: Clerk enters..., System outputs....

DON'T: The Clerk enters..., The System outputs...



Good Use Cases

- Keep it simple
- Use present tense
- Subject should be primary actor, system under design and secondary actors
- Must provide a meaningful result to primary actor
- Verb should be what actor does to successfully move the use case forward
- Avoid GUI: write in terms of goals, not details of the GUI

Applying UML and Patterns, C. Larman 2005

Writing Effective Use Cases, A. Cockburn, 2000



AARHUS
UNIVERSITET
INGENIØRHØJSKOLEN

Relationships between Use Cases

- You have three types of relationships:
 - Include
 - Extends
 - Generalization
- Use of these features will typically make a use case diagram more complex to read.
- So they should be used with caution.

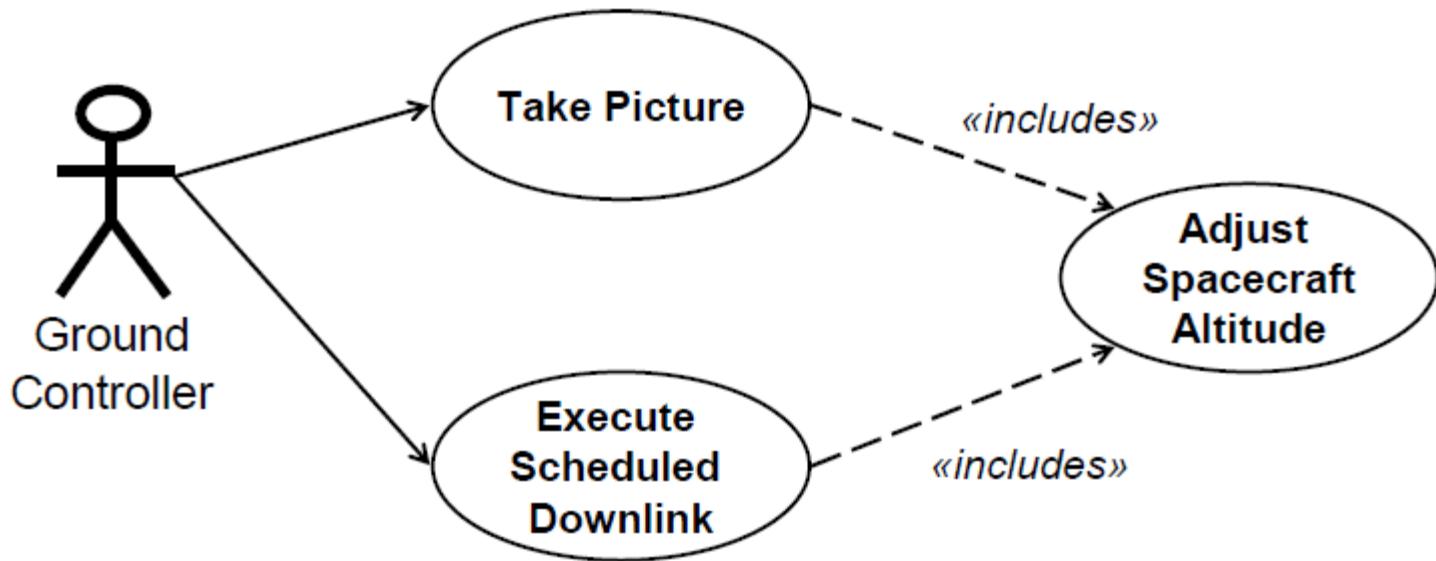


Includes

- An *include* relation is a structuring mechanism
- Used primarily to avoid redundancy in the specification
 - An include use case can be used and reused in many situations



Includes



Common functionality is here moved out and described by its own Use Case



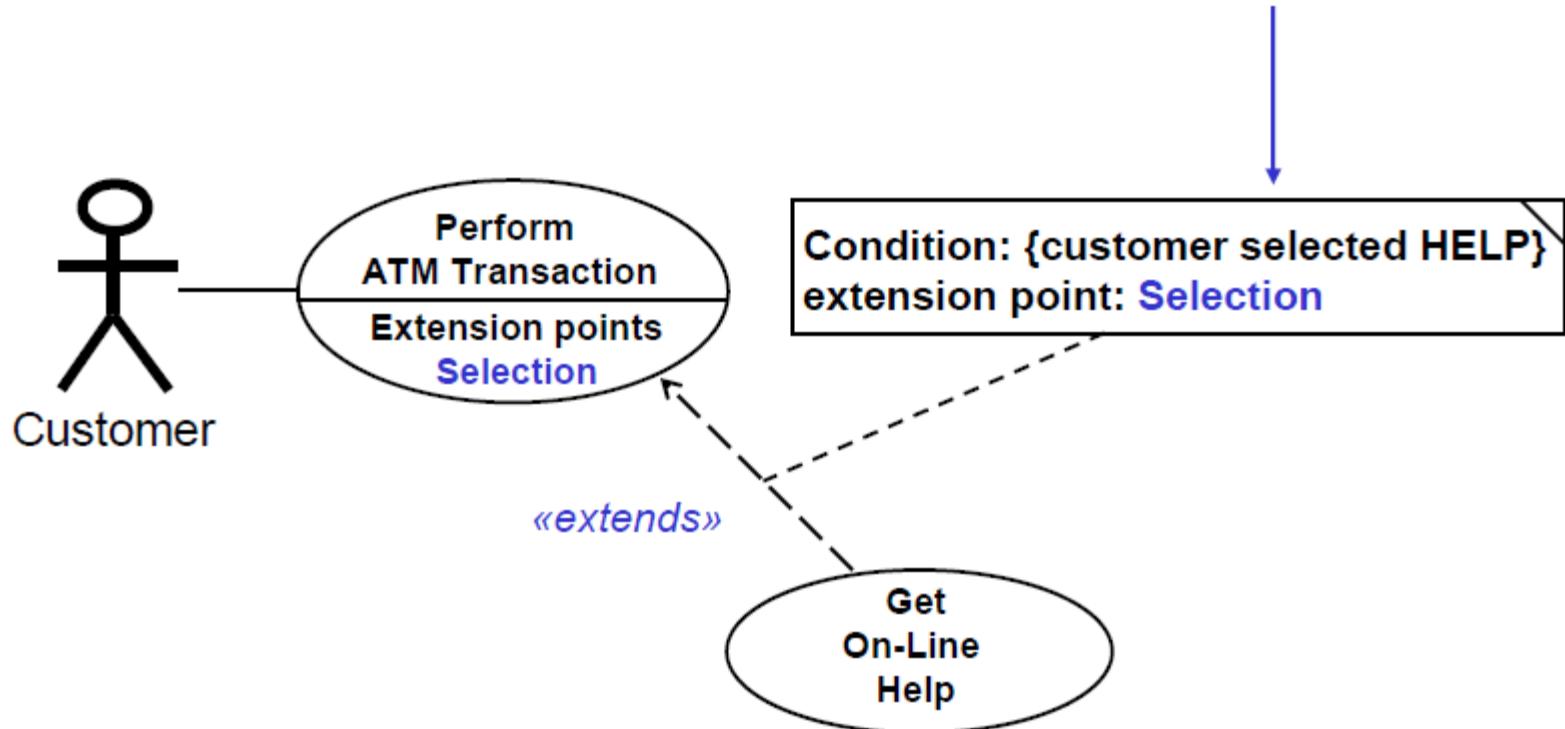
Extends

- An extends relation is a structuring mechanism:
 - Used to describe optional extensions
 - Used to describe special situations for example errors or other exceptional scenarios
- Can be used late in the development cycle – as a way to add functionality in a structured way, without disturbing the other use cases

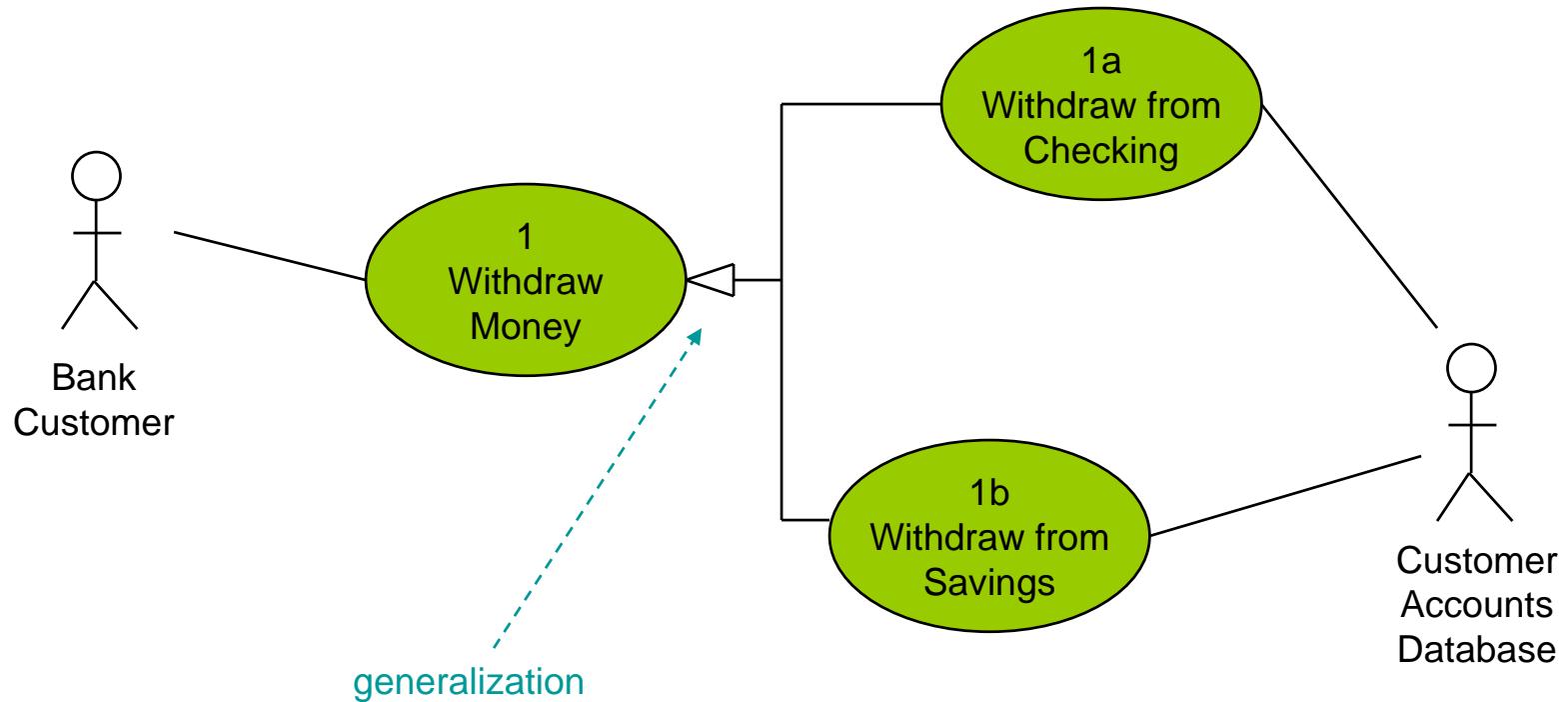


Extends

Example of an extension with a condition



Generalization

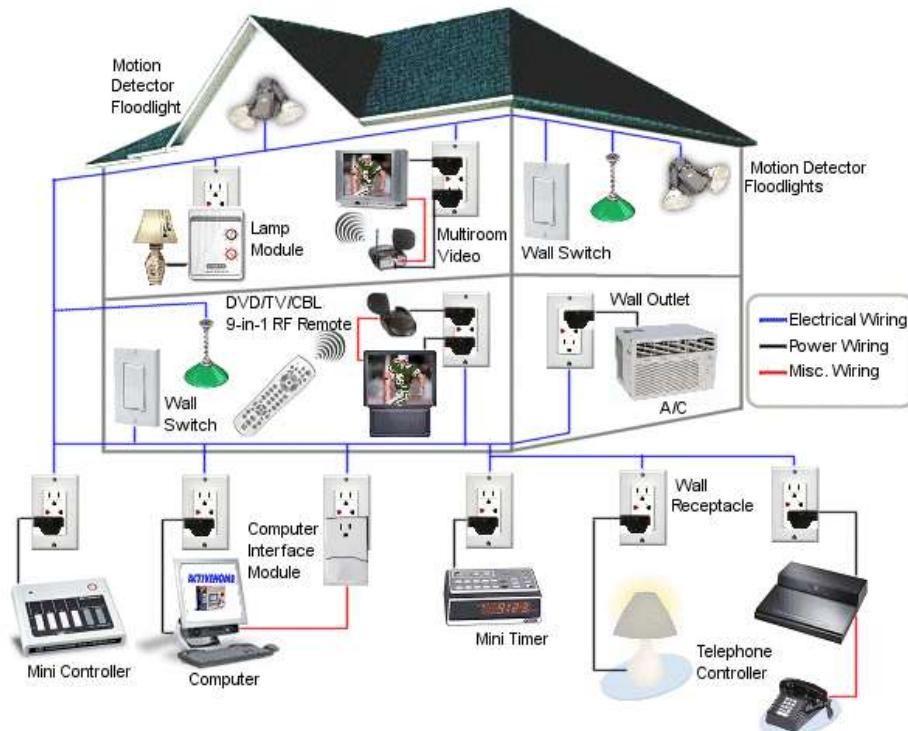


Questions?



2. Semesterprojekter

”Home Automation System”



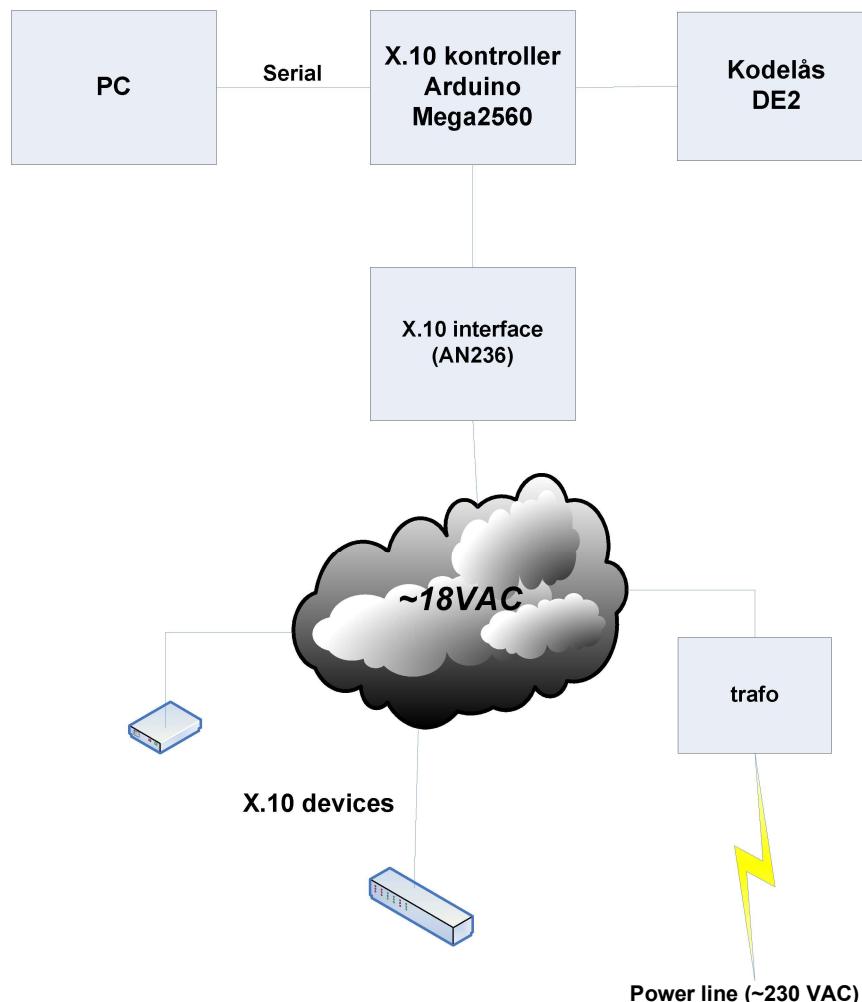
AARHUS
UNIVERSITET

Home Automation/Security – E/IKT

- Home Automation – det intelligente hjem
- Home Security – et konkret scenarie
 - Tyveriforebyggelse
 - Simulerer, at der er nogen hjemme
- Automatisk tænd/sluk af lys og apparater
- Konfigureres med forskellige scenarier
- Styres fra tilkoblet PC
- Central power line kommunikations kontroller
 - X.10 devices over "lysnættet"
- Applikationsnoten "AN236" forslag til HW konstruktion
- Sender (minimum) og modtager – lysdæmper
- Mere information om det intelligente hjem

http://en.wikipedia.org/wiki/Home_automation

Home Security System – E/IKT



Fagområder - Systemdesign

Software, Digitaldesign:

- Arduino board – C/C++ prog.
- Windows C++
- GUI
- VHDL – DE2

Kredsløbsdesign

Analog/Power:

- X.10 hardware
- Zerocrossing
- Analog filter
- Power Supply
- Lysdæmper (triac)



AARHUS
UNIVERSITET

Krav til projektet – E/IKT

- *Kombineret hardware og software-projekt*
 - Højniveau-software (C++)
 - Hardwarenær software (C/C++)
 - Hardware (Analog, digital, stærkstrøm)
- Grupper efter studieretning og Insights profil
 - Omrent 7-8 studerende i alt
 - Vi danner grupperne
- Home Security Systemet
 - Se [projektoplæg og dokumenter på BlackBoard](#)

Formål med projektet

- At bringe kendt og ny viden i **anvendelse**
- At lære at **søge og anvende** ny viden
- At gennemføre et **tværfagligt og tværdisciplinelt** projekt
- At lære at **strukturere et vellykket samarbejde**

Læringsmål (E2PRJ2)

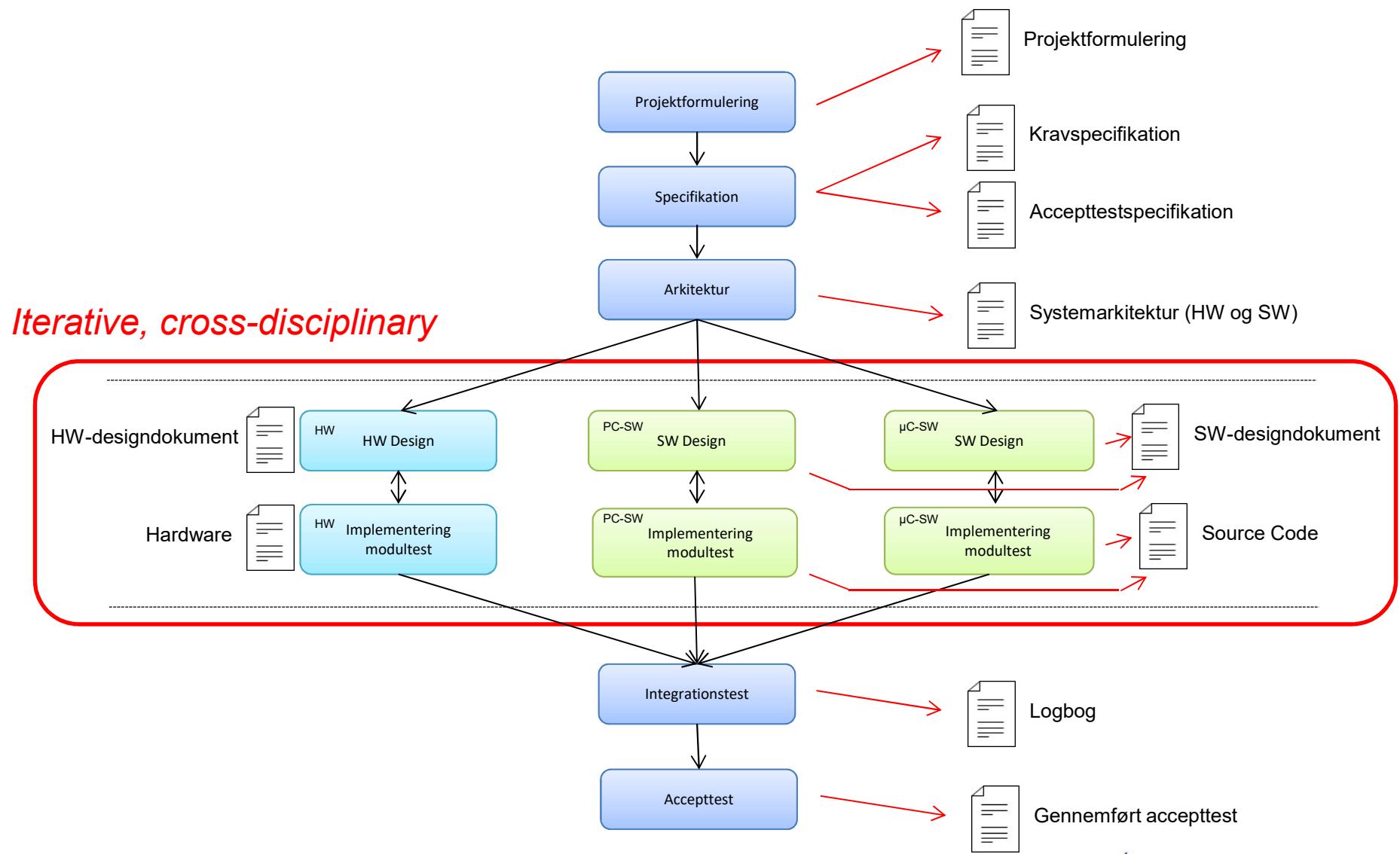
Når kurset er afsluttet, forventes den studerende at kunne:

Udarbejde en teknisk rapport ud fra et projekt oplæg

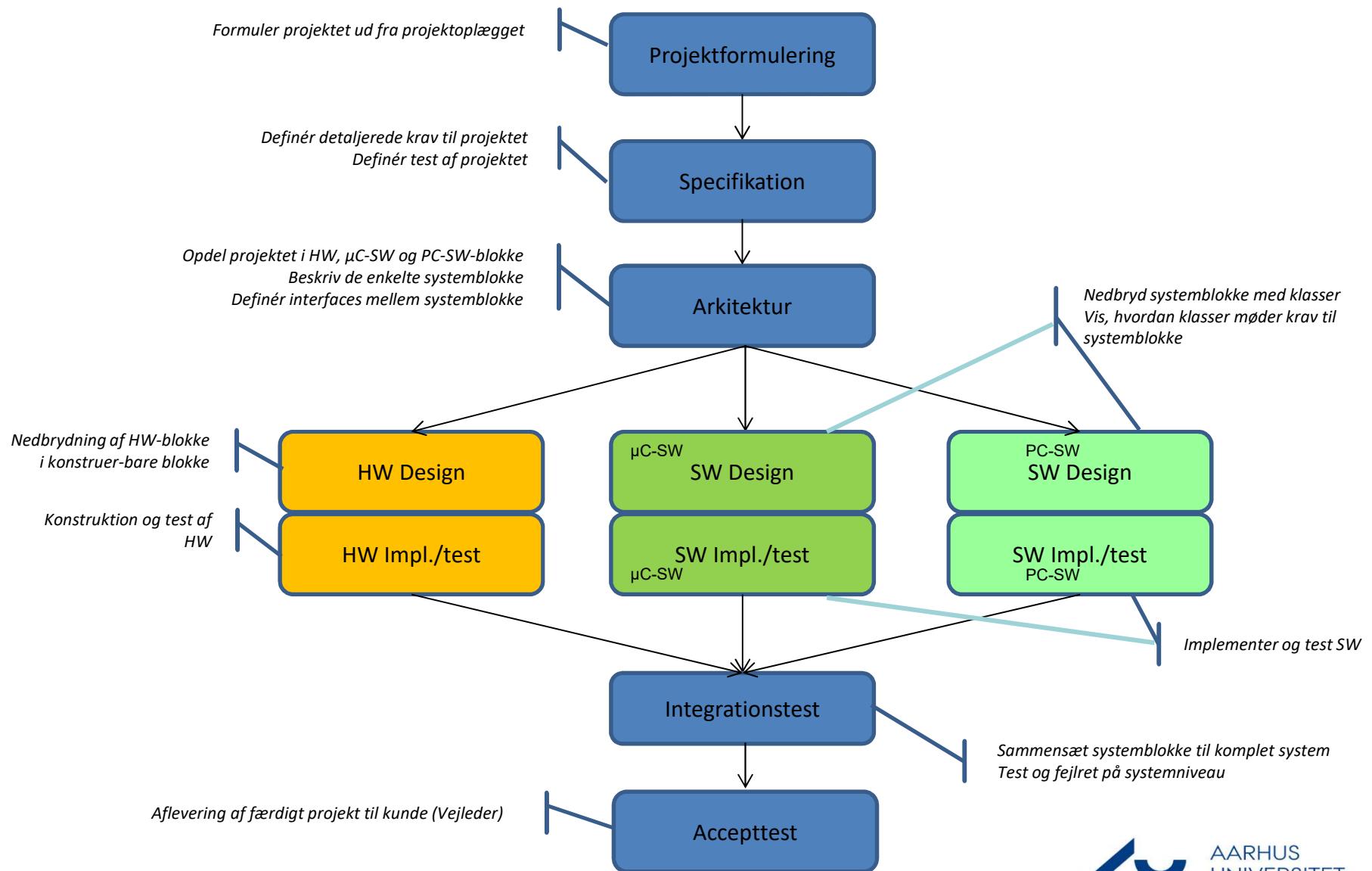
- Anvende en beskrevet **udviklingsproces** til gennemførelsen af produktudvikling
- Foretage og modtage **review** af en anden projektgruppens arbejde
- Foretage fælles **planlægning** og uddelegering af opgaver
- Anvende **mødeledelse** med dagsorden og referat i en projektgruppe
- Formulere egne ingeniørfaglige **styrker og svagheder** i projektarbejdet
- Formulere og anvende en use case-baseret **kravspecifikation**
- Anvende kravspecifikation til udformning af **accepttest**
- Beskrive **systemarkitektur** og design ved hjælp af SysML og UML
- Anvende korrekt **fagterminologi**
- Designe og implementere en **prototype**, der indeholder egen udviklet HW og SW
- **Kombinere viden** fra flere af semestrets kurser og anvende denne i projektet
- **Vurdere og evaluere** projektets udviklingsproces, produkt og resultater
- Udvælge og anvende supplerende viden i projektarbejdet med angivelse af **referencer**
- Præsentere projektets resultater ved et **mundtligt forsvar**

I øvrigt lægger faget sig tæt op af I2ISE

The ASE Process



Projektforløbet - faser



Projektstyringsform

- Overordnet tidsplan med milestones
- Samarbejde efter Scrum konceptet (Iterative fase)
 - Benyttes i design og implementerings faserne (Sprints)
- ***Projektformulering 17./9 som godkendes af vejleder***
- ***1. Review – deadline 9./10.***
Omfatter dokumenterne: **Projektformulering, tidsplan, kravspecifikation og accepttestspezifikation**
- ***2. Review – deadline 17./11.***
Omfatter dokumenterne: **Systemarkitektur-dokument og evt. første version af HW/SW-designdokument, hvis det foreligger.** Udgangspunktet er den nu reviderede Kravspecifikation, der også sendes til review gruppen.
- ***Demonstration og accepttest – deadline 17./12.***
Projektet funktionalitet fremvises for vejleder, iflg. accepttestspezifikationen.
- ***Aflevering af rapport og dokumentation – deadline 8./1.***

Reviews

- Kravspecifikation, accepttestspecifikation og systemarkitektur-dokumenter skal reviewes
- Review foretages ”på kryds” med reviewgruppe
 - Tidspunkt aftales med reviewgruppe
 - Vejleder(e) deltager hvis muligt.
- Review-feedback *skal* anvendes
 - Revideret dok. + ”list of changes” afleveres til vejleder
 - List of changes: $\frac{1}{2}$ -1 A4-side med beskrivelse af indførte ændringer i dokumentationen



Eksamens

- I skal *alle* have kendskab til hele projektet, dvs...
 - Projektrapport
 - Kravspecifikation
 - Accepttestspezifikation
 - Arkitektur-dokumentation
- I skal *hver især* have indgående kendskab til
 - Jeres individuelle dele af design og implementering
- Bemærk: Det er *jeres* ansvar at godtgøre, at I har opfyldt læringsmålene!
 - gennem projektrapport og eksamen

Gode råd

- Lav en *samarbejdsaftale* og få afstemt forventningerne i jeres gruppe både for *omfanget* af jeres projekt, *ambitionsniveau* og *arbejdsmoral*
- Det giver *ikke ekstra point* af få lavet *print* eller implementerer et *avanceret GUI* eller brug af *teknologi*, som *ikke er krævet* af jer
- Det er *OK at specificere mere end i realisere i jeres prototyper*, men det *skal fremgå af dokumentationen hvor i begrænser* jer
- Vælg et *minimum af funktionalitet der endeligt skal realiseres og demonstreres*, som tænd/sluk eller justering af lysstyrken
- I skal have *fokus på læringsmålene*, det er det i bliver vurderet efter
- Vigtig med en *god og velskrevet rapport* med tilhørende dokumentation af jeres produkt

Jeres første opgave

- Gruppedannelse
 - Angiv din studieretning i dokument i link på Blackboard
 - Grupperne er dannet efter retning og tildeling af vejleder kommer i løbet af ugen
- Projektformulering (2-3 sider)
 - Med udgangspunkt i projektoplægget, hvad kunne I tænke jer at lave (vision)?
 - Home Security er kun et forslag – kravet er power line kommunikation og X.10
 - Design og implementering af hardware til X.10 transmitter og receiver
 - hjertearytmimonitorering defineres brugsscenerier og funktioner til GUI på PC
 - Brug udstyr, transducer og interface til PC



AARHUS
UNIVERSITET

Projektvejledning (Semesterprojekt)

Vejlederens rolle

- Processen – samarbejdet - fremdriften
- Milestones – review
- Faciliteter og vejleder:
 - Diskussioner
 - Forslag
 - Konsulent
- Faglig vejleder på sit eget fagområde
 - Tekniske spørgsmål henvises til jeres respektive faglærer

4 typer af vejledning

- Produkt orienteret
- Proces orienteret
- Laissez-faire vejledning
- Kontrolleret vejledning

Husk:

Vejlederen er ikke en lærer.

Afklar forventninger med jeres gruppe og vejleder. Vejledning tilpasses gruppens modenhed og aktuelle situation.

Forventninger til gruppen

- Aftal **møder med vejlederen** ca. 1 gang pr. uge af $\frac{1}{2}$ time, husk vejlederen har kun 30 timer totalt!
- Send **dagsorden** inden mødet, med punkter i vil have diskuteret
- **Beslutningsreferat** fra alle jeres **møder**
 - Dato, deltagerfortegnelse, dagsorden, beslutningsreferat
 - Mødeleder og referent
- **Logbog** – projektdagbog, hvori væsentlige aktiviteter registreres. Godkendte revisioner af dokumenter, programmer og konstruktioner.

Mere information

- Inspiration – BlackBoard
 - Vejledninger for gennemførelse og dokumentation
 - Projektoplæg
 - Arbejde i projektgrupper
 - ApplicationNote (AN236)

System Test

Introduction to Systems Engineering
I2ISE

Here's a fact about test

Testing can only show the *presence* of errors, never their *absence*

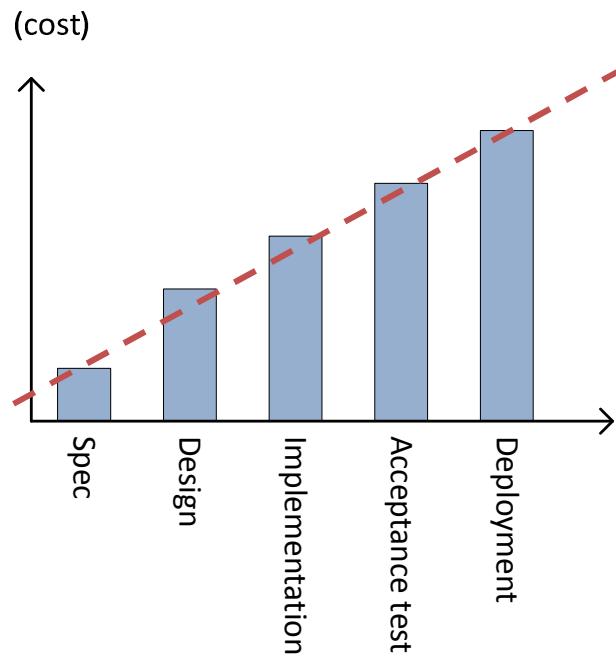
- What does this mean? What are the consequences?

A short discussion

- What is the *value* of testing?
 - For the system
 - For the developer
 - For the company
 - For the customer
 - For the users
- What is the *cost* of testing?

The cost of errors

- Finding errors early is in the best interest of you and your company



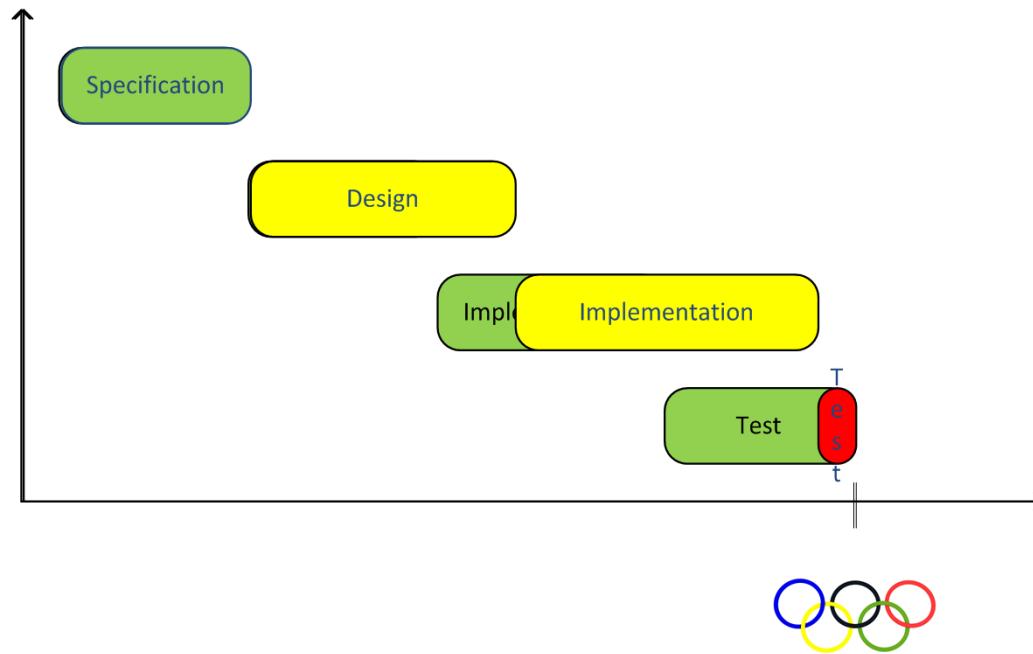
To this, add damage done to

- humans
- property
- company image
- loss of productivity
- follow-on sales

The test mantra:
*Test early, test often,
test enough*

When to test?

The nightmare, all-too-often-seen scenario

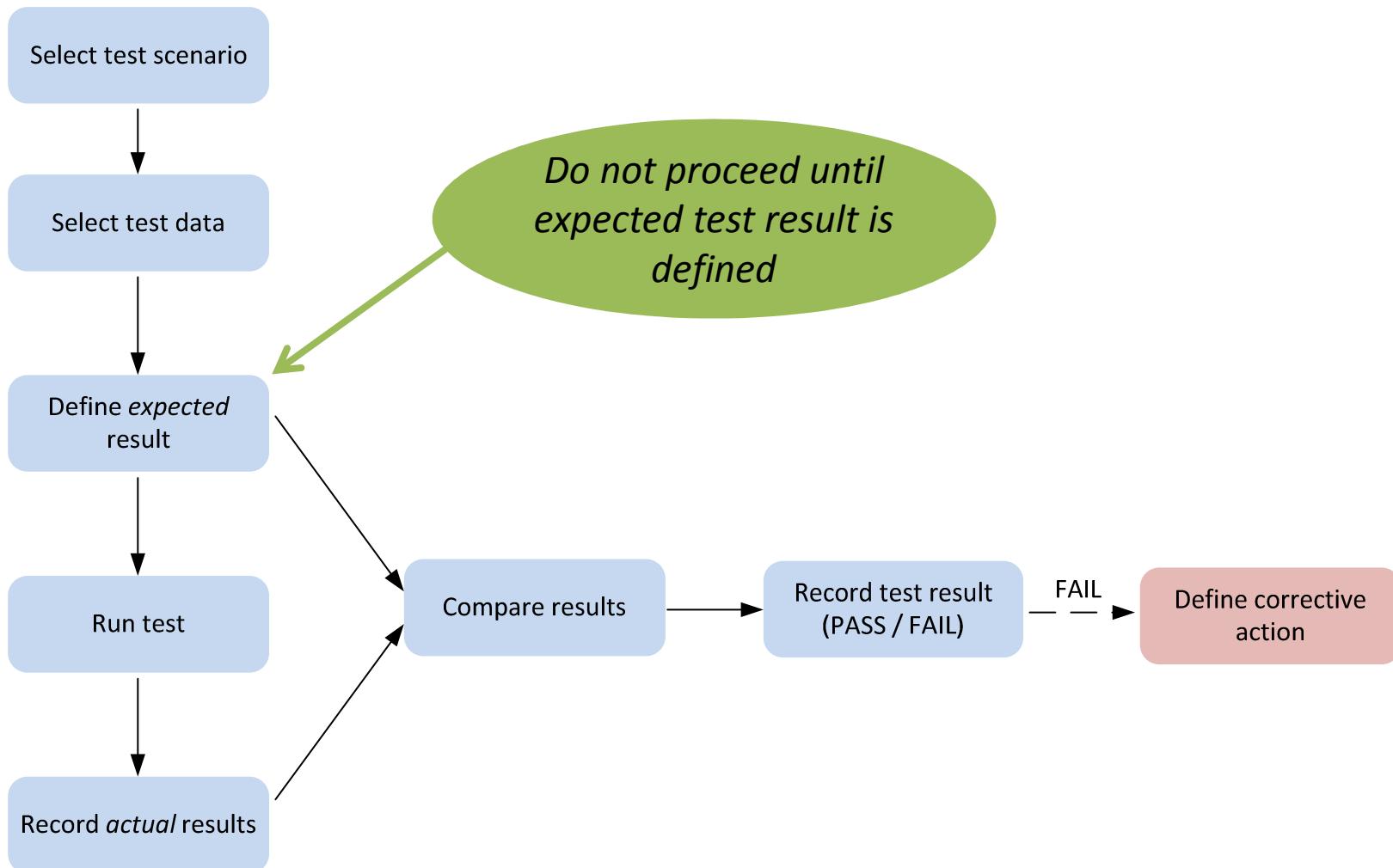


- What happens to the test effort in this case?

Properties of a good test

- What are the properties of a good, valuable test?
- The test should be
 - independent
 - simple
 - repeatable
 - fine-grained
 - quick to run

Defining a test



Selecting test data and Equivalence Classes

- Definition:
 - An Equivalence Class is a collection of input that should be processed and react equally.
- Characteristic:
 - All elements in an equivalence class will either fall or pass.
 - Is used to reduce the number of tests
- Limitation:
 - May require knowledge of: type of processor, programming languages or algorithms

Simple Example

bool Big(int x)

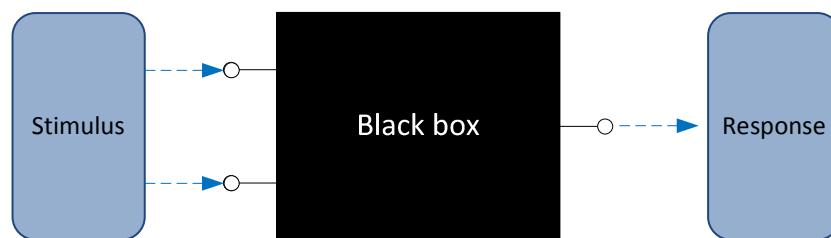
- x can assumed a value in the 1 - 100 range
- if $x < 10$
 - x is "small" => false
- else
 - x is "big" => true

At least 4 Equivalence Classes:

- $x < 1$: invalid, but possible input.
- $1 \leq x < 10$: valid data, 1, 5 and 9 chosen as test data.
- $10 \leq x < 100$: valid data, chosen values 10, 49 and 99.
 - Where 10 and 99 is boundary values
- $100 < x$: invalid, but possible input.

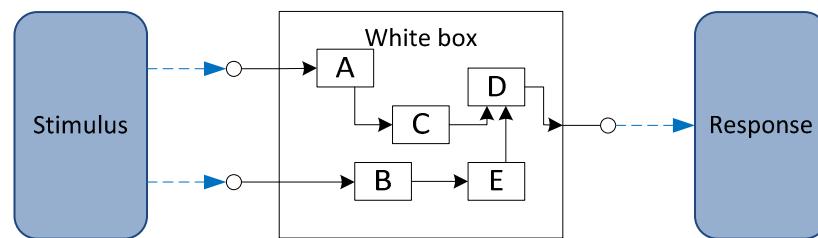
Test types: Black vs. white box testing

- Black box testing, AKA *functional* testing
 - Test only through system interfaces
 - No knowledge of internal workings
- Complete test → complete set of input tested (valid and invalid)



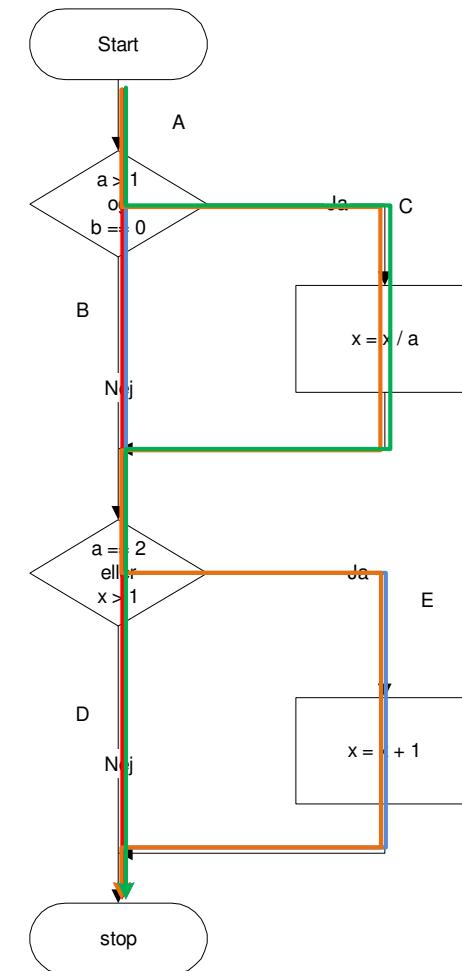
Test types: Black vs. white box testing

- White box testing
 - Test through system interfaces, but *with* knowledge of internal workings
- Complete test → complete *route coverage*

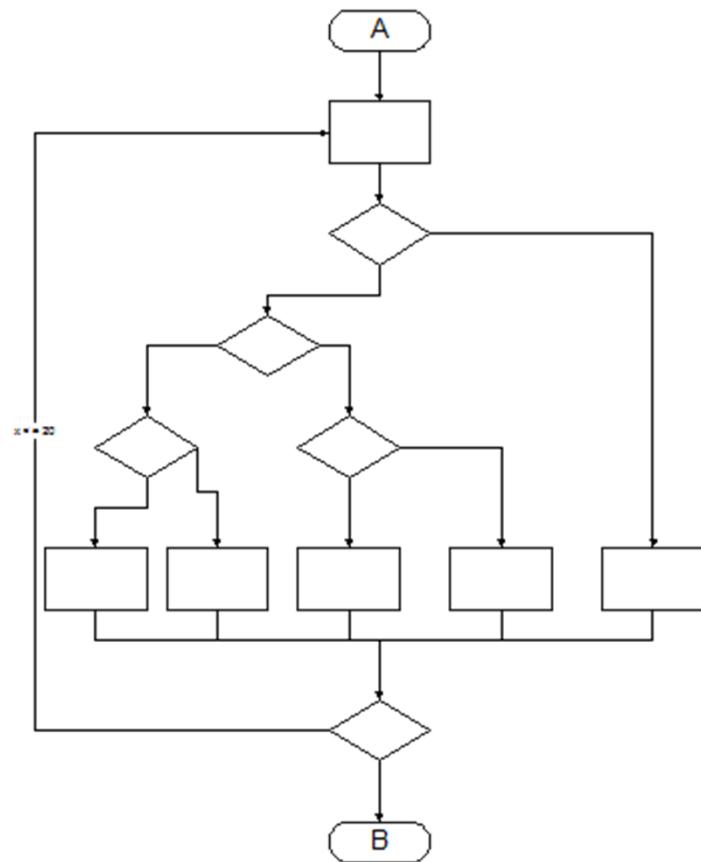


Route coverage - example

```
void f(a, b, x)
{
    if ((a > 1) && (b == 0))
        x = x / a;
    if ((a == 2) || (x > 1))
        x = x + 1 ;
}
```

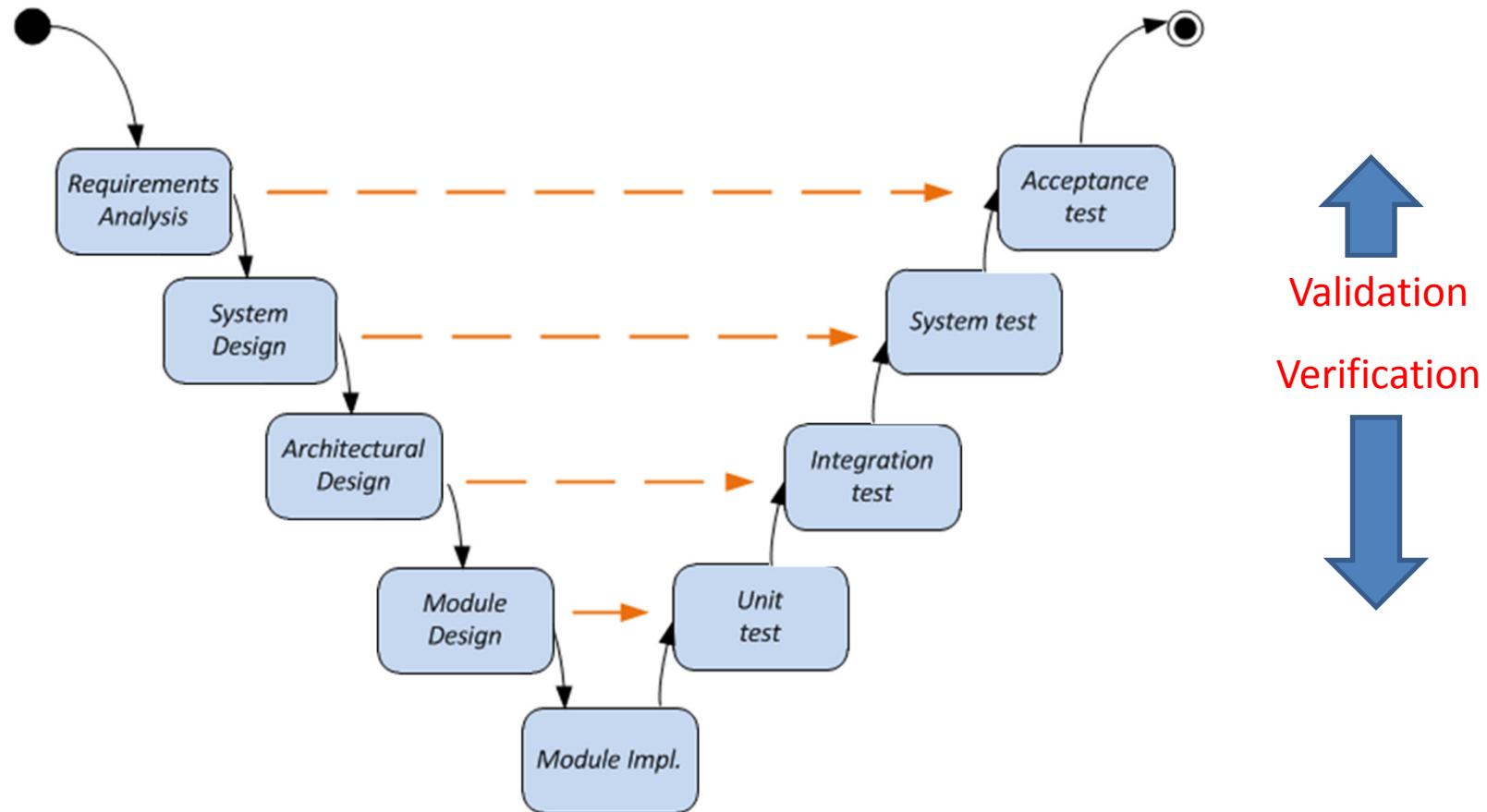


Route coverage - example



- 5 routes, up to 20 loops
- Independent decisions
→ 10^{14} routes
- 1 us/test → 3.17 years

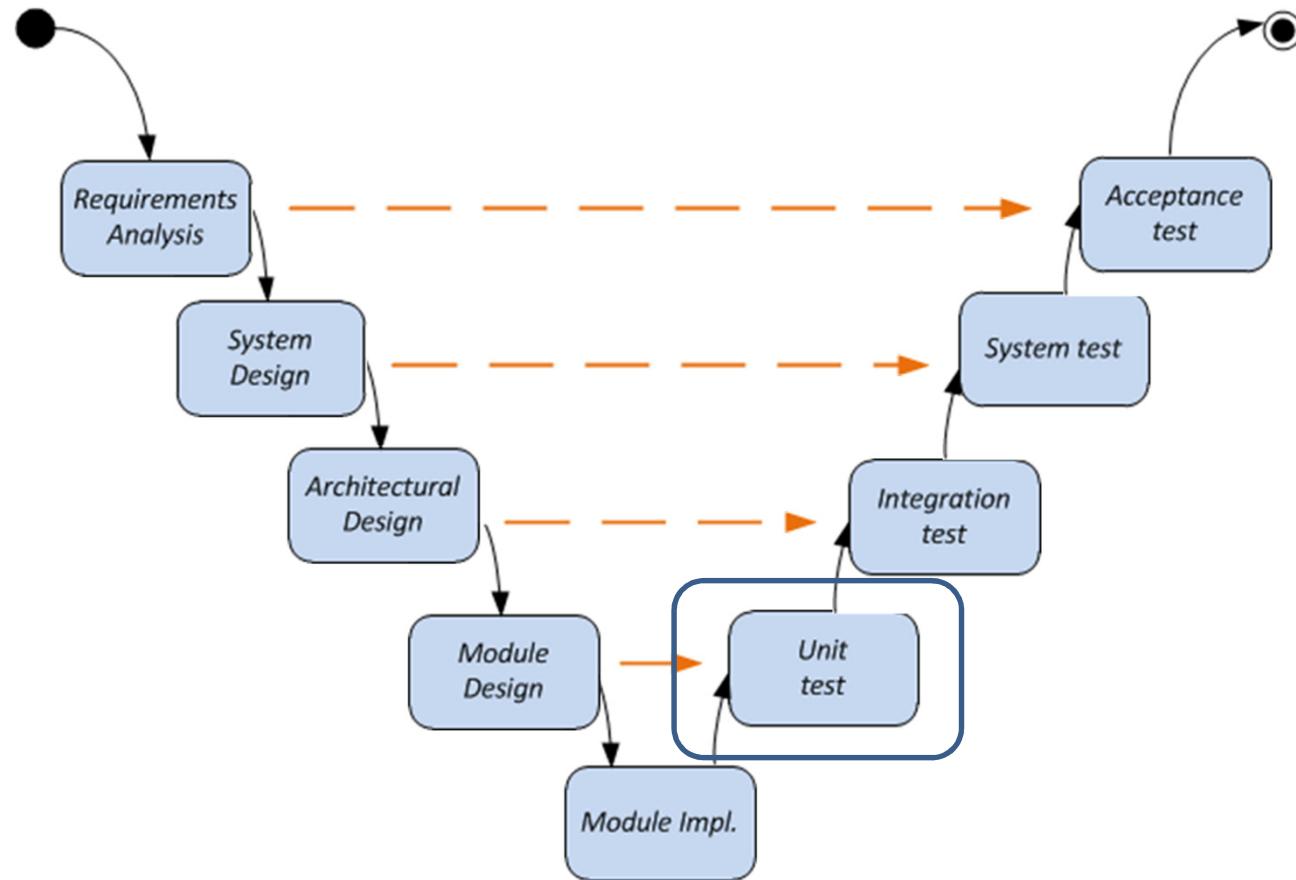
V-Model and Test levels



Validation & Verification

- Validation
 - Build the right thing
 - Checking and testing the product against the requirements and user needs
 - At the end of the development process
 - External process (**System + Acceptance Test**)
- Verification
 - Build the thing right.
 - Complies with a sub requirements regulation, design principles
 - At any given development phase
 - Internal Process (**Unit + Integration Test**)

Test levels



Test levels: Unit test

- Unit testing is *by far* the most efficient bug-squasher
- Find a bug in unit testing ?
 - correct the bug, re-run the test
- Find *same* bug in acceptance testing ?
 - Explain to customer, schedule new test, damage control, correct bug, regression-test system, ...

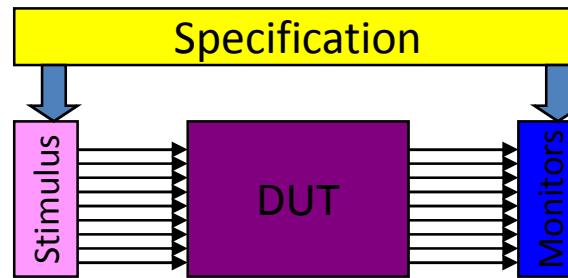


Unit testing in software

- Write software, then write test. Run test, correct bugs, move on...
- Or better yet: Write test, then write software
 - Test Driven Development (TDD)
 - The test becomes a *specification*
 - Red-green-refactor cycle

Unit testing in hardware

- Create component/subsystem, strap to test bench
- Deduct and apply stimulus, observe results
 - Stimuli signals and monitor expected behavior

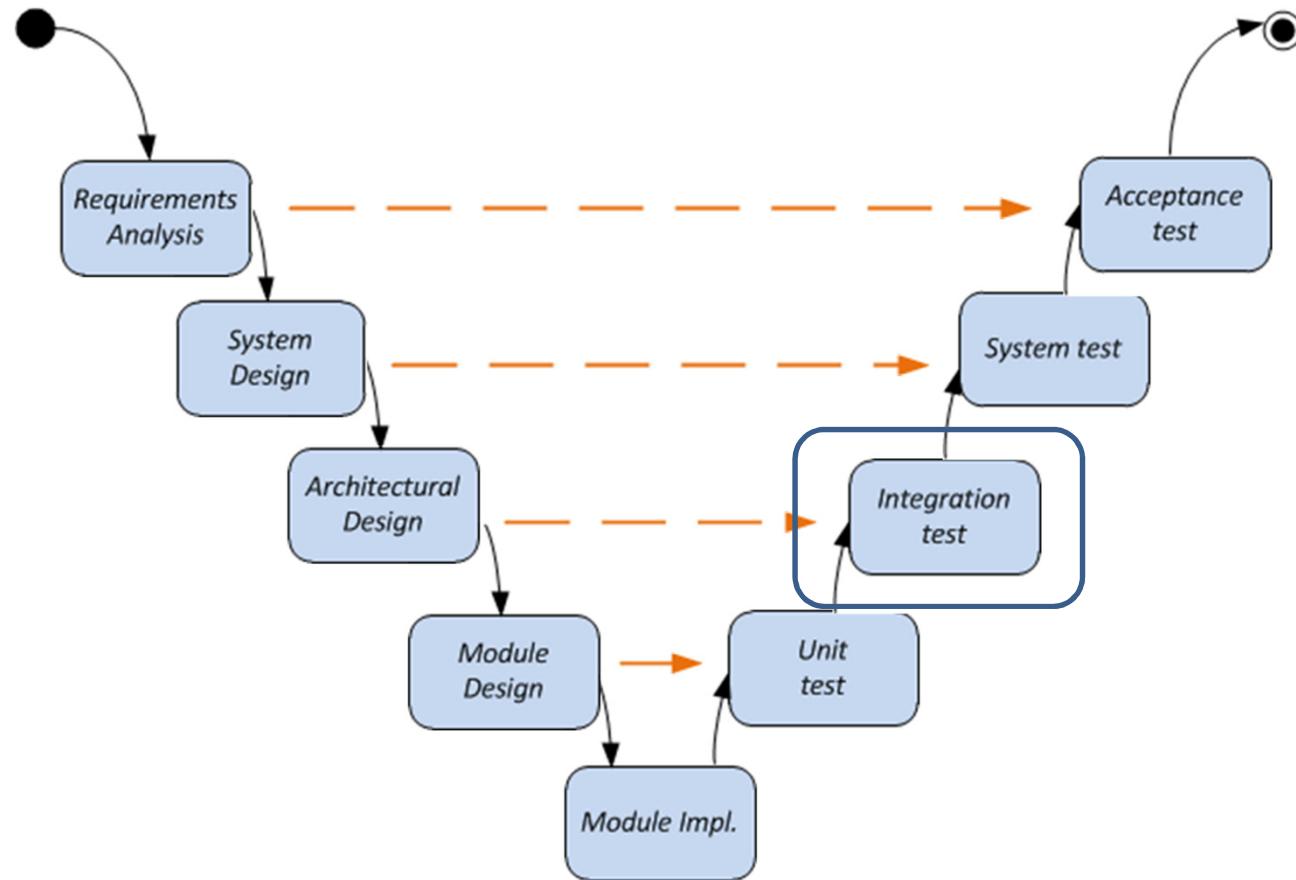


Unit testing

- Unit testing is closely related to design and implementation
- Most often done by implementor – *a problem?*
- Automate tests whenever possible
 - Machines have no feelings



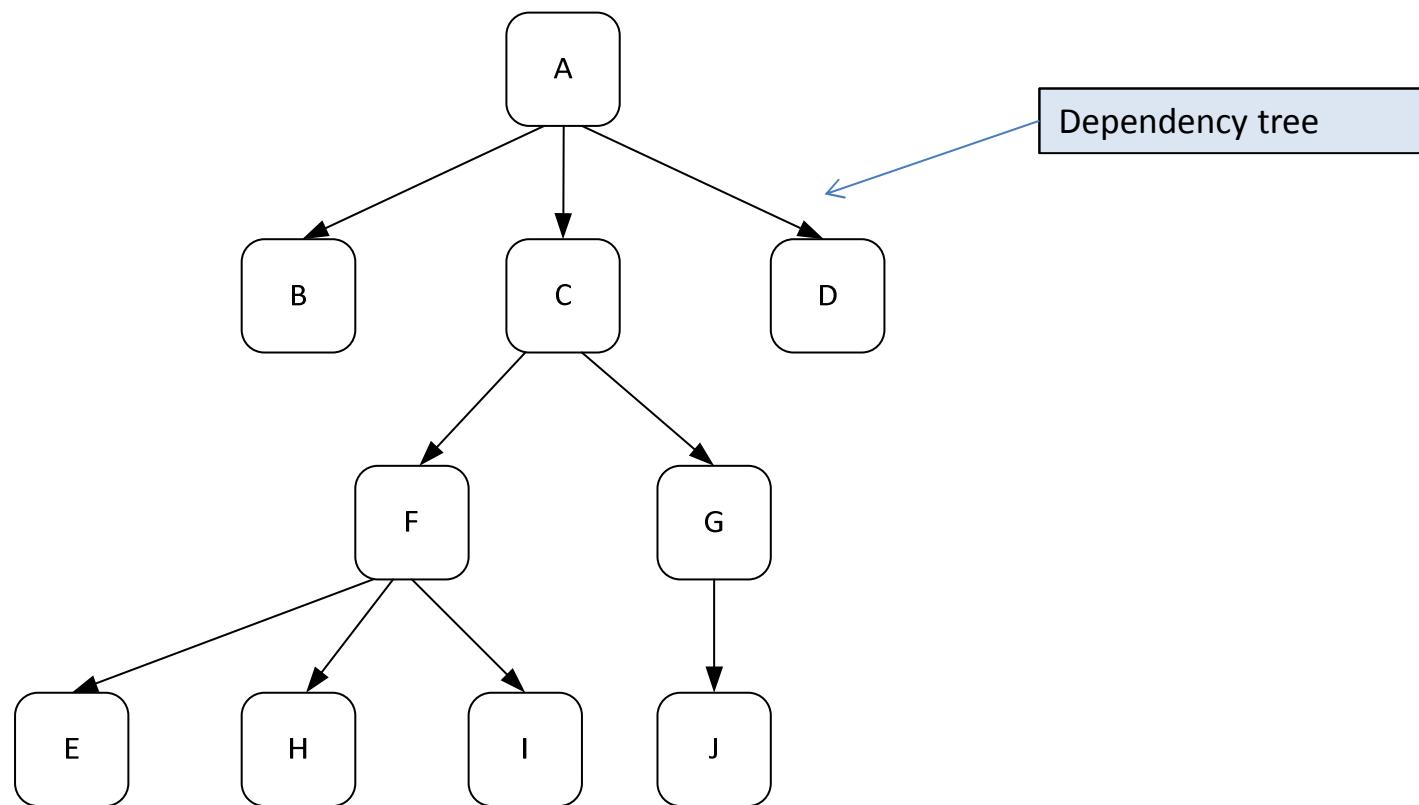
Test levels



Test levels: Integration test

- Integration test: Integrating dependent (unit-tested) components
- Various strategies:
 - Big-bang 
 - Bottom-up
 - Top-down
 - Sandwich (other hybrids)

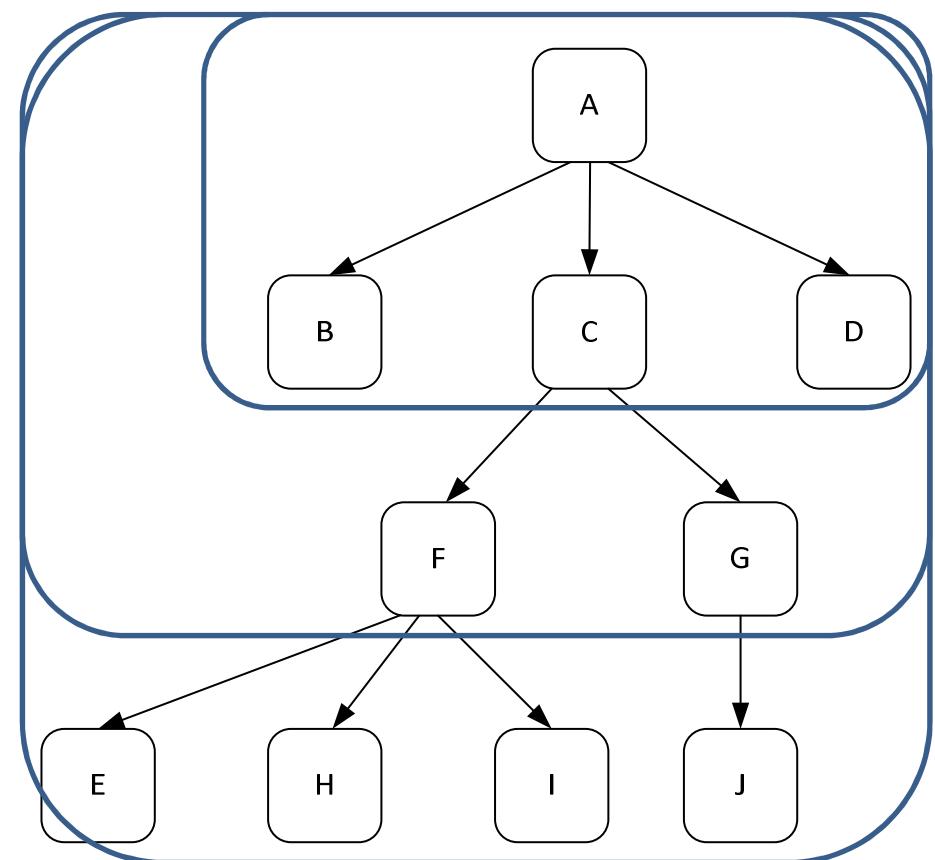
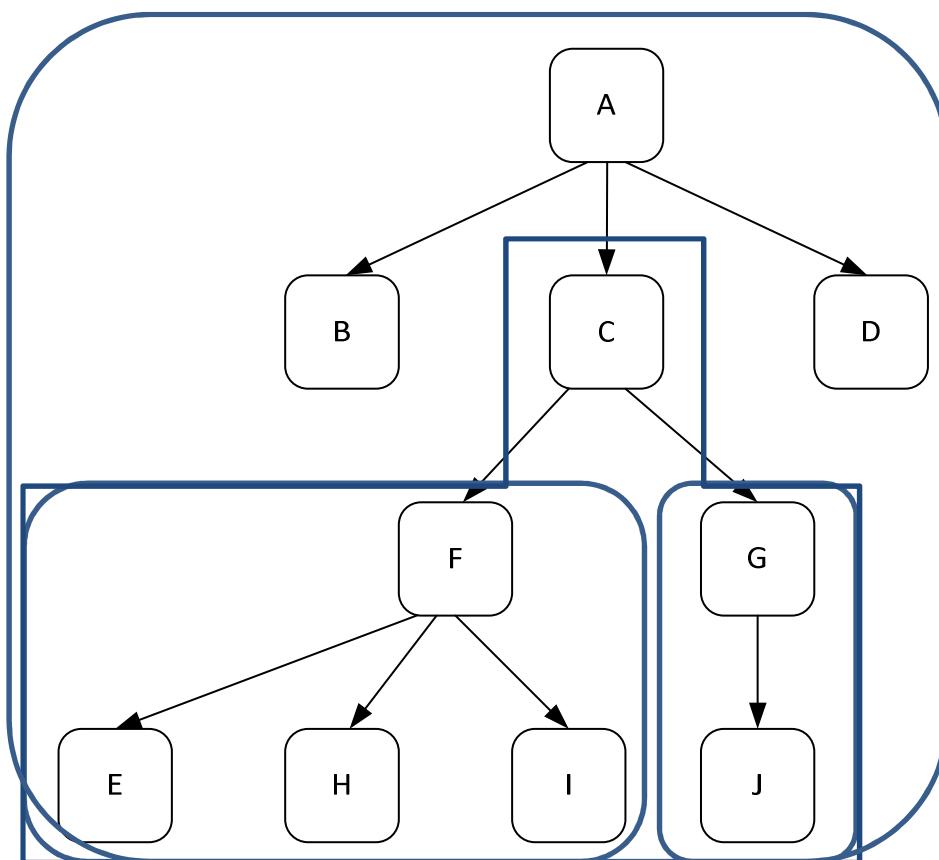
Integration test: Mapping dependencies



Integration test:

Bottom-up –
requires *drivers*

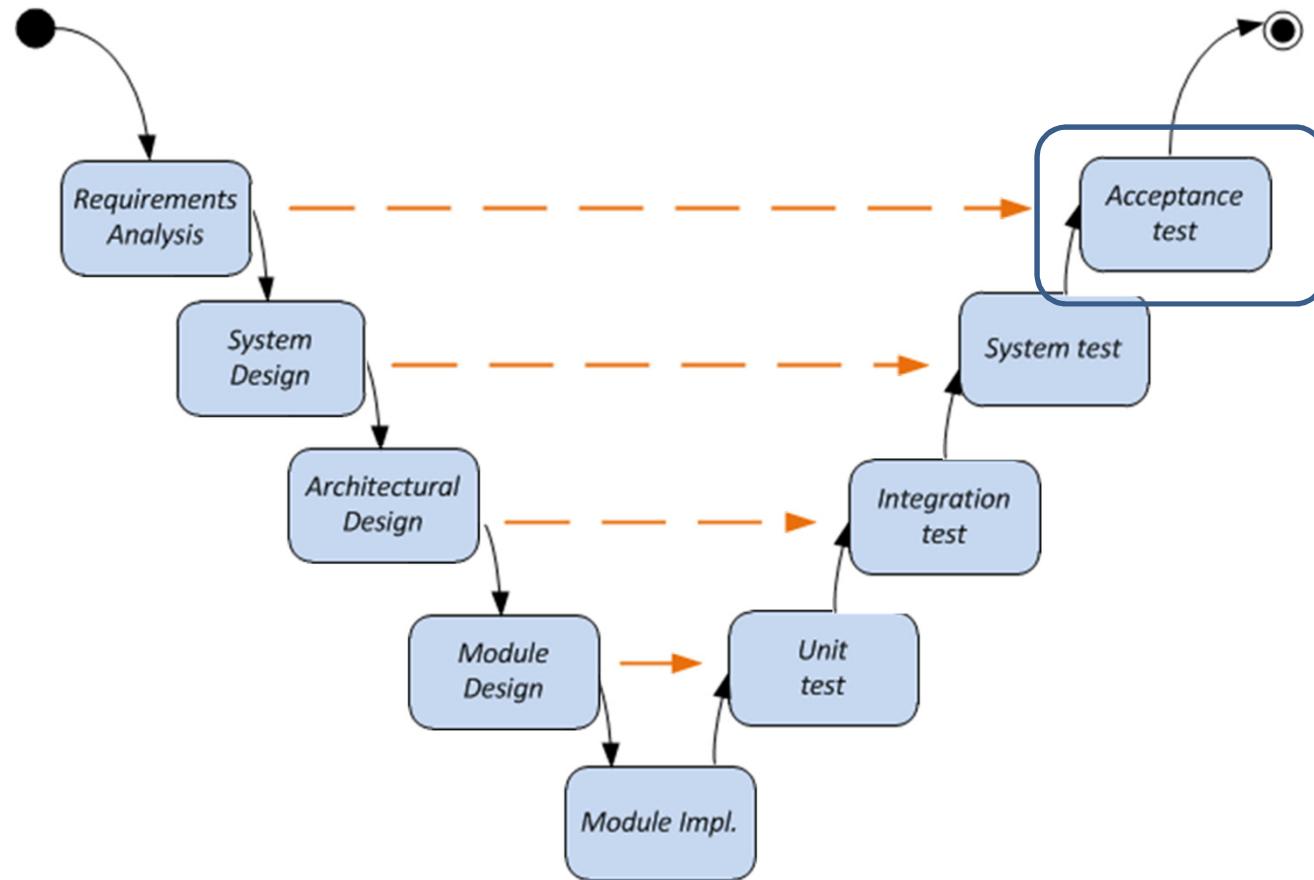
Top-down
- requires *stubs*



Integration test: Discuss

- What are the benefits of top-down integration testing?
- What are the benefits of bottom-up integration testing?
- What is applicable when?
- Do we have to make a one-or-the-other choice?

Test levels



Acceptance test: UCs versus test

- Conducted with customer – signs off.
- The Use cases (UCs) for the system must map to the acceptance test – why?
- How do we make this happen?

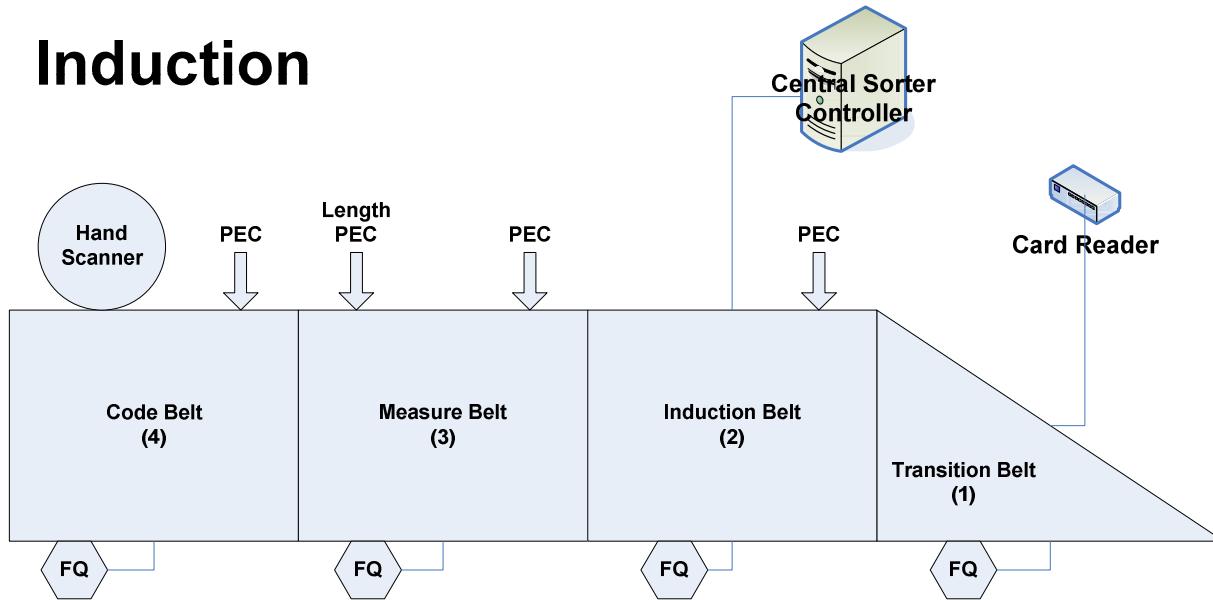
Mapping Use Cases to Acceptance test

- Essentially, performs the use case
 - Scenario maps to steps in the Test
 - Repeatable, because of pre-defined:
 - Input, Output, flow
- Remember:
 - Pre conditions; are they valid?
 - Post conditions; do they hold?
 - A use case may describe multiple paths
 - *For each path you need a test scenario*
- Used to validate the use case

Exercise: **AcceptTestOvelse.pdf**

Exercise – System Test (Black box)

Induction



- Find test scenarios
- Find possible test objects
- Think about error scenarios
- Equivalence classes

<https://www.youtube.com/watch?v=neSLiifHEBM>

<https://www.youtube.com/watch?v=tprsTfIRUII>



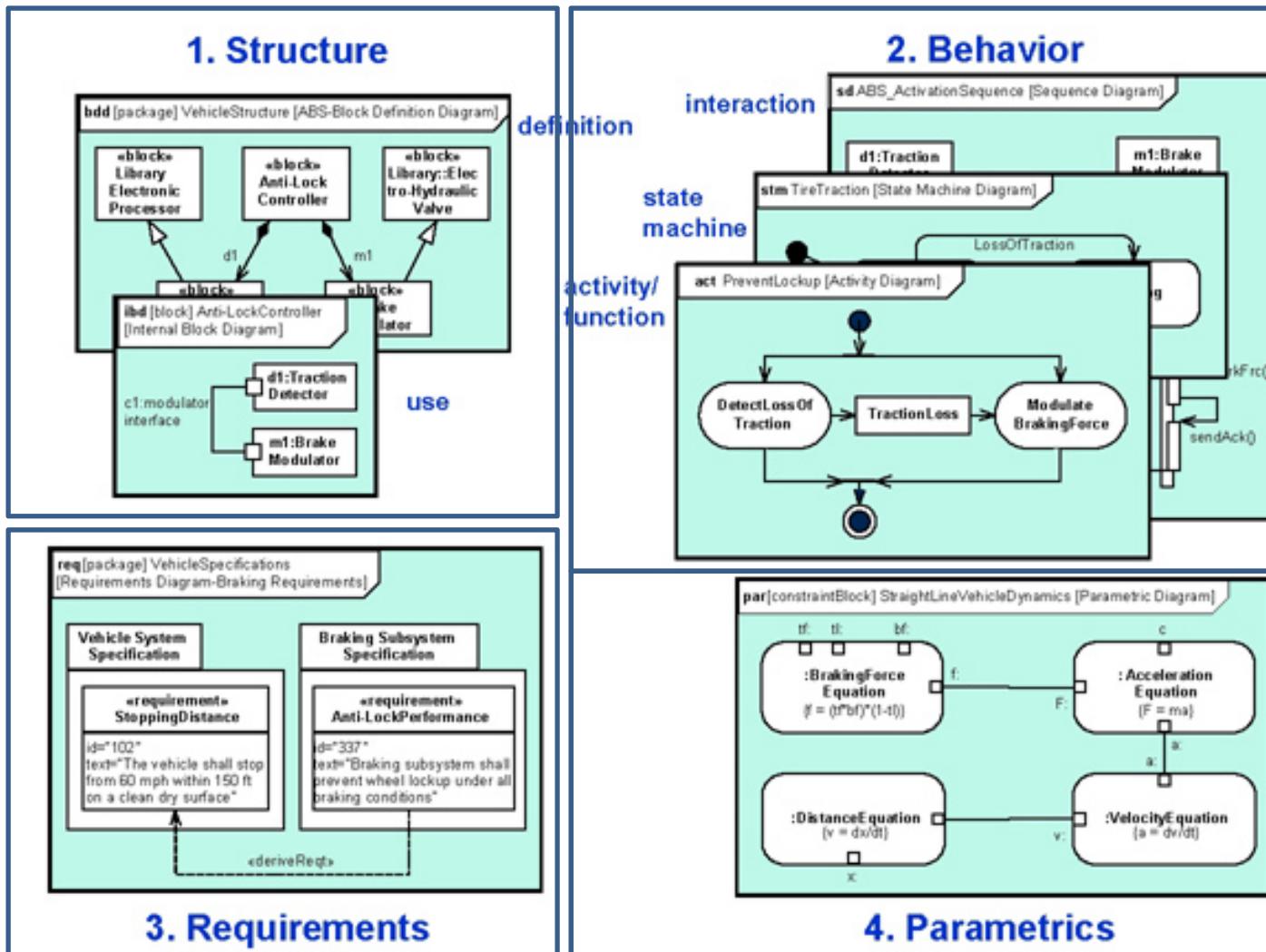
SysML Introduction

Introduction to Systems Engineering
I2ISE

Introduction to SysML

- SysML = *System Modeling Language*
 - Supports analysis, specification, design, and verification/validation of *systems* (hardware, software, mechanics, personnel)
- Allows the formation and communication of a system *model* using *diagrams*
- Elements in different (types of) diagrams are reused to convey different aspects of the elements' use
- SysML is an enabler of *Model-Based Systems Engineering*
 - The *model*, not documentation, is in focus

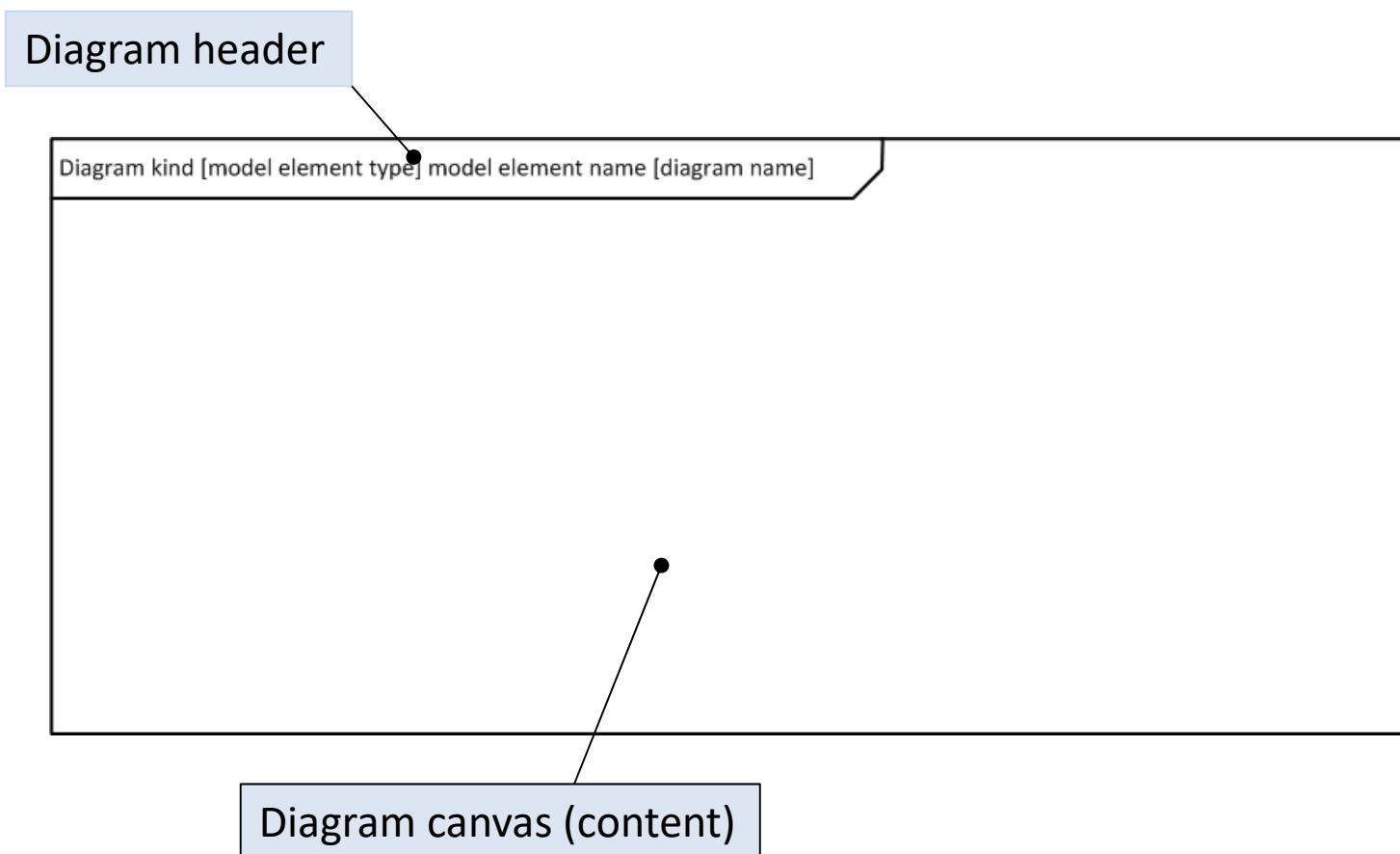
SysML “pillars” (diagrams)



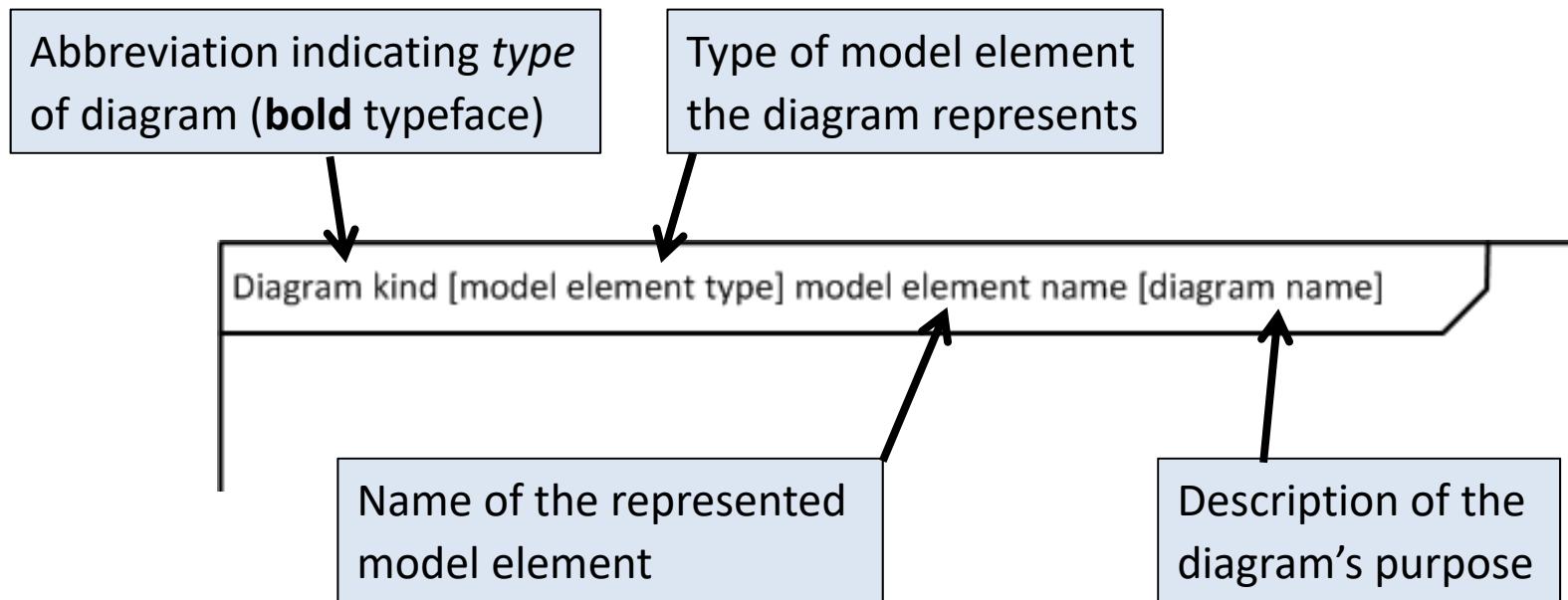
Note that the Package and Use Case diagrams are not shown in this example, but are respectively part of the structure and behavior pillars

SysML: Diagram frame

- The diagram frame consists of header and canvas



SysML: Diagram header



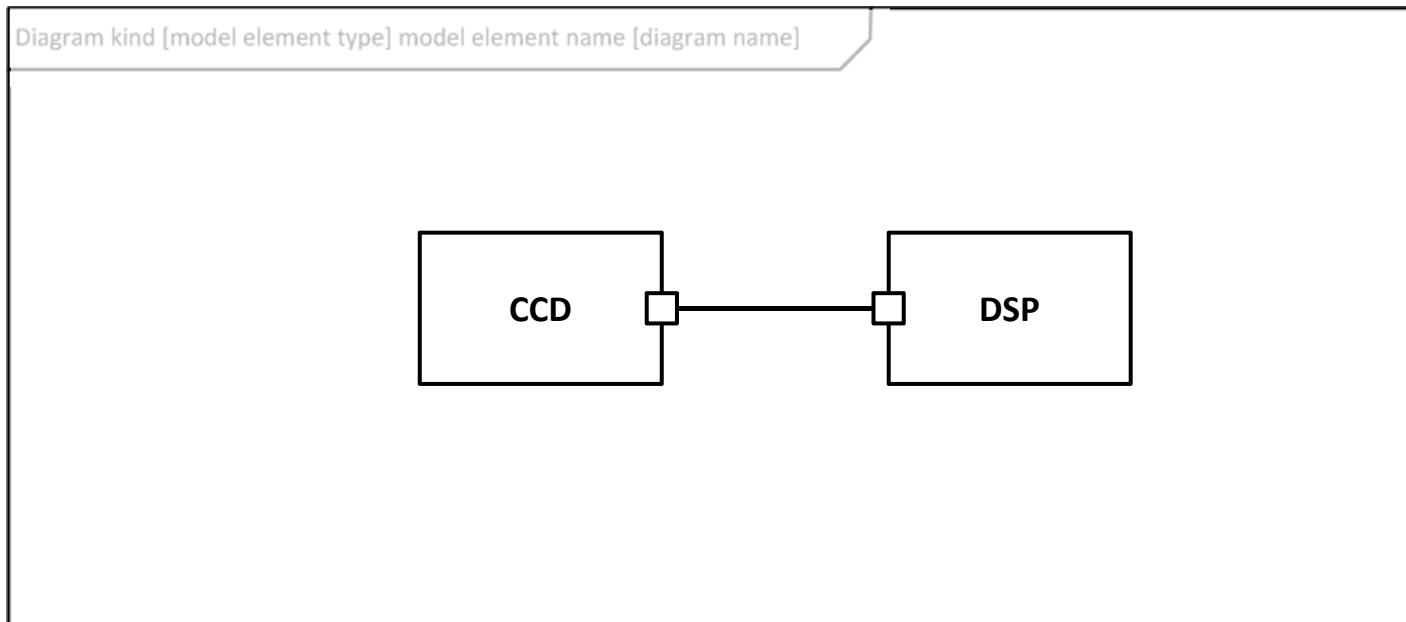
SysML: Diagram header - example

bdd [block] Camera [Hierarchical system structure]

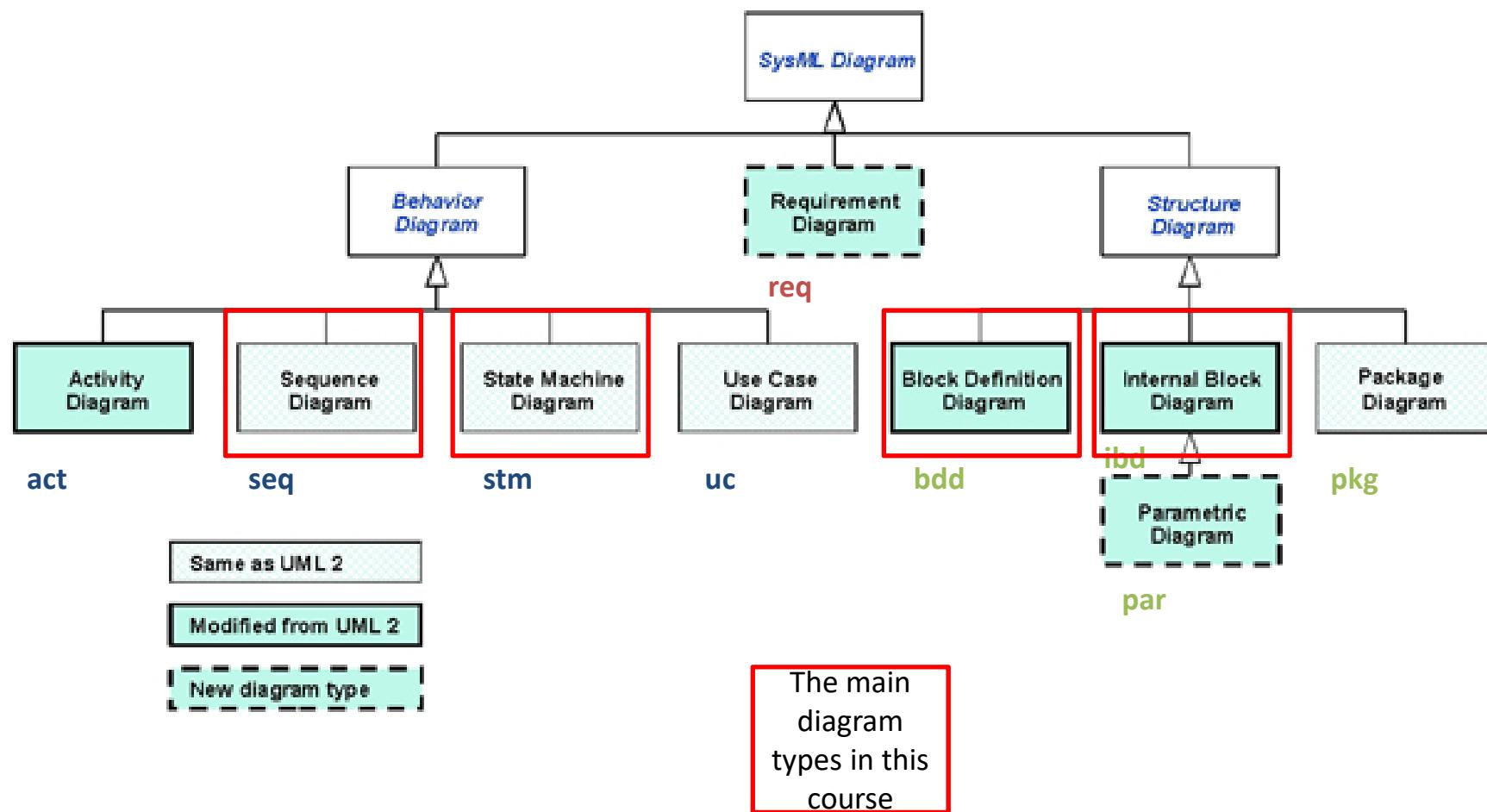
- This is a *block definition diagram (bdd)*,
 - for the [*block*]
 - *Camera*
 - describing its [*Hierarchical system structure*]
-
- Items in brackets are optional
 - *model element type* [*block*] is frequently omitted,
 - *diagram name* [*Hierarchical ...*] frequently included

SysML: Diagram canvas

- The diagram canvas holds the modeling elements



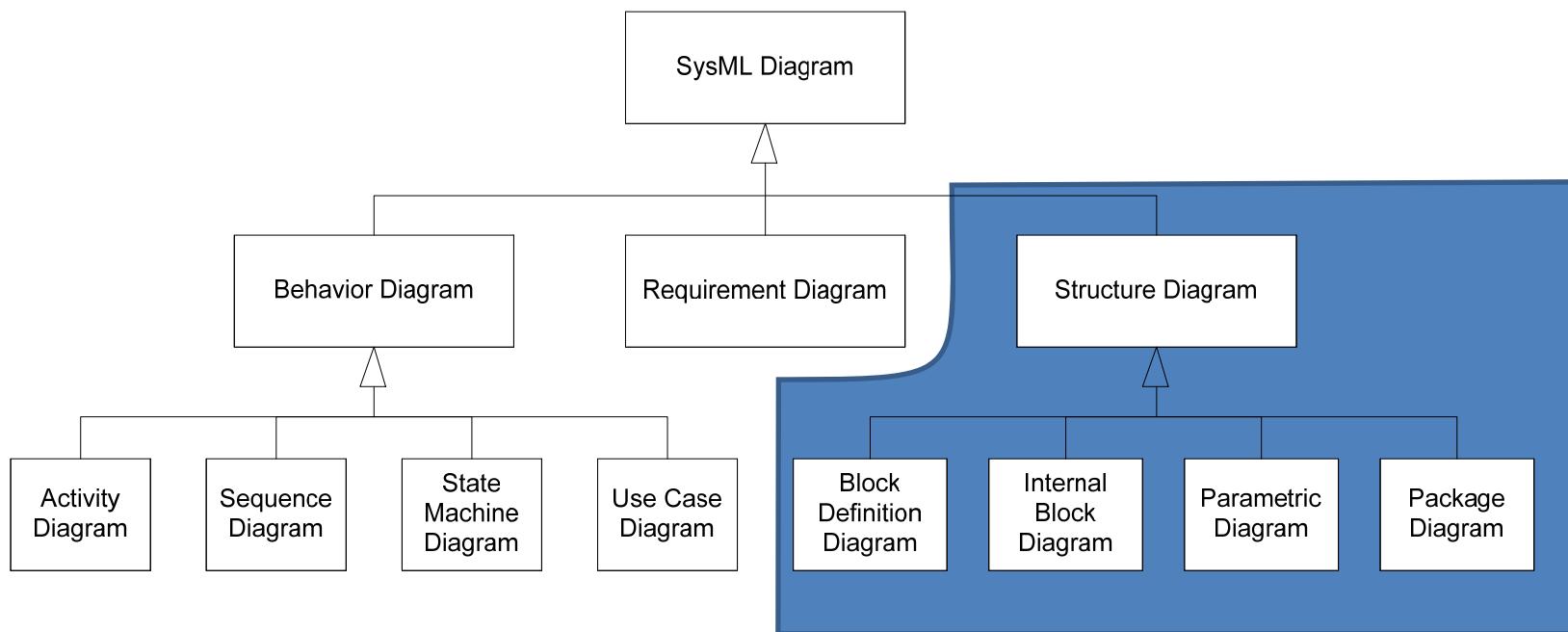
SysML: Diagram types compared to UML



SysML Structural Diagrams 1

Introduction to Systems Engineering
I2ISE

SysML: Diagram types



Introduction

- There are 4 different types of structural diagrams:

- Block Definition Diagram (bdd) – Structural system elements called *blocks* and their composition
- Internal Block Diagram (ibd) – Interconnection and interfaces between the *parts* of a block
- Parametric diagram (par) – Constraints on property values
- Package diagram (pkg) – The organization of a model into *packages* that contain model elements

Blocks

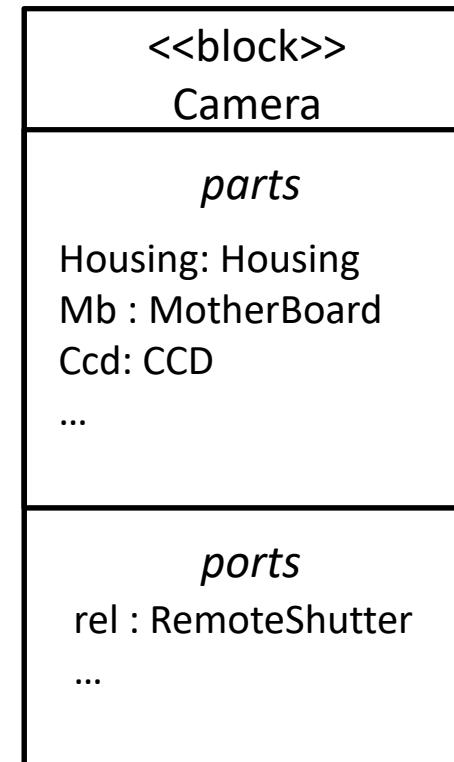


SysML structural diagrams – the *blocks*

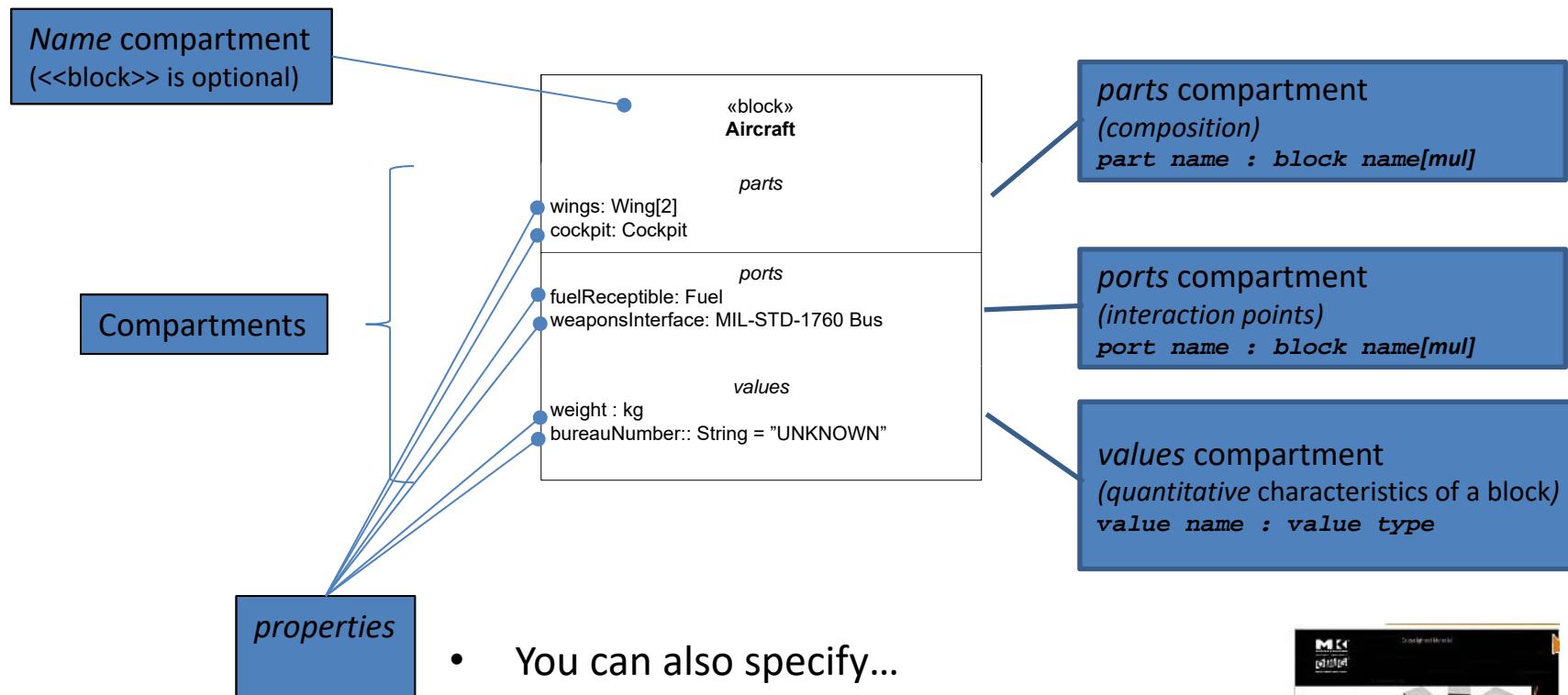
- The *block* is the fundamental model element for describing system structure
 - Hardware, software, person, facility, water, atmosphere, files,...
- The block is a *type*
 - A common description of similar *instances*, just like a C++ class

Blocks

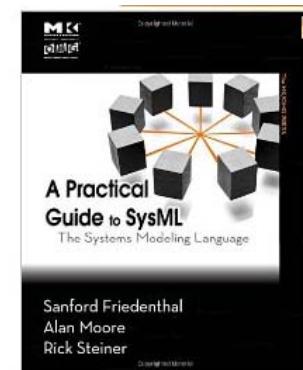
- The block is drawn as a *rectangle* on a diagram canvas
- The block may be divided into *compartments*
- The top compartment always contains the block's *name*
 - *Name* is mandatory
 - `<<block>>` is optional
- Other compartments may be used to represent other block features
 - Parts, operations, ports, ...
- Each compartment contains *properties*



Blocks – the works

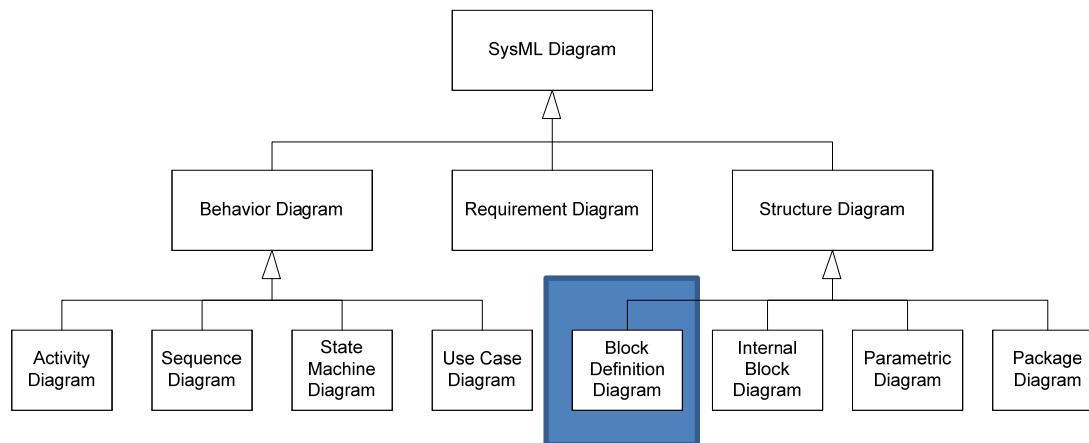


- You can also specify...
 - *references* (weaker connections)
 - *value types* and their *units* and *dimensions*
 - *read-only properties*
 - *initial property values*, their *distribution*
 - ...



SysML

Block Definition Diagrams



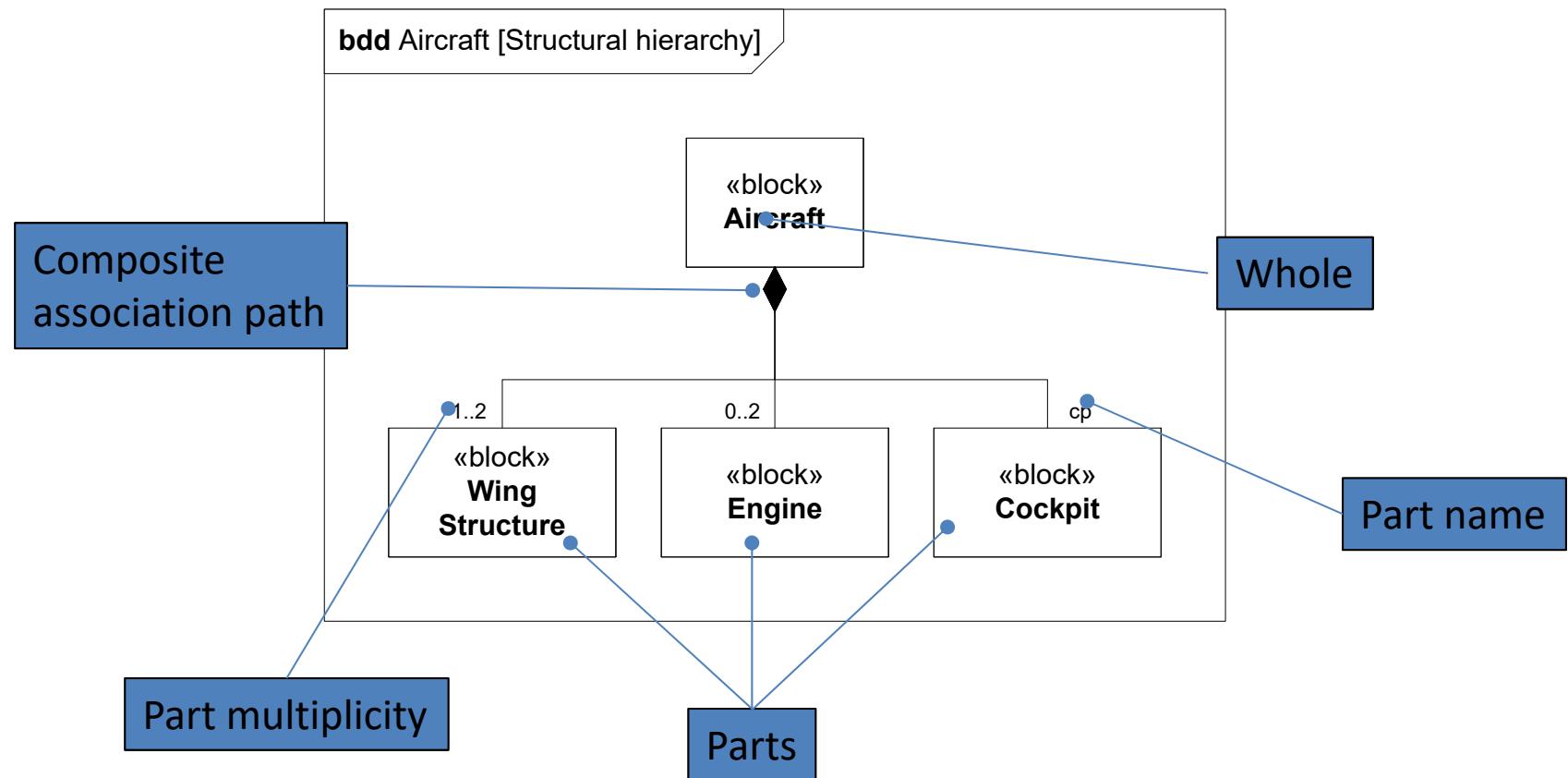
SysML: Block definition diagram

- A *Block Definition Diagram (BDD)* is used to define *blocks* and their relationship other blocks (their *composition*)
- A BDD may be used to define any kind of structure
 - Logical, physical, electrical, software, etc.
- BDDs are also used to define other relationships between blocks, e.g. allocation of functions to physical entities

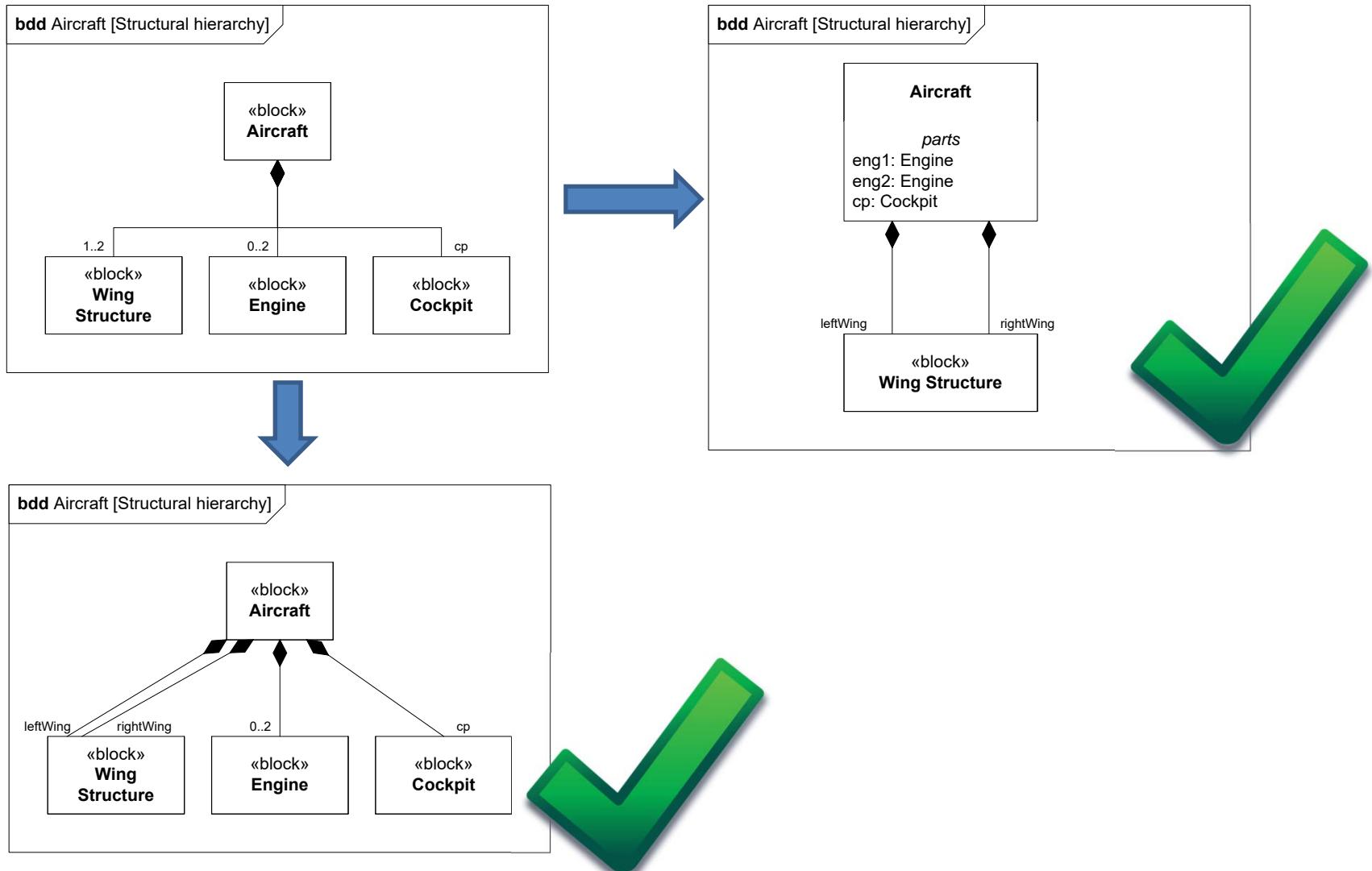
bdd: Composition relationships

The most common kind of relationship is *composition*:

- "Consists-of" or "whole-part" relationship, e.g. "an Aircraft consists-of 1-2 wings, 0-2 engines and 1 cockpit"

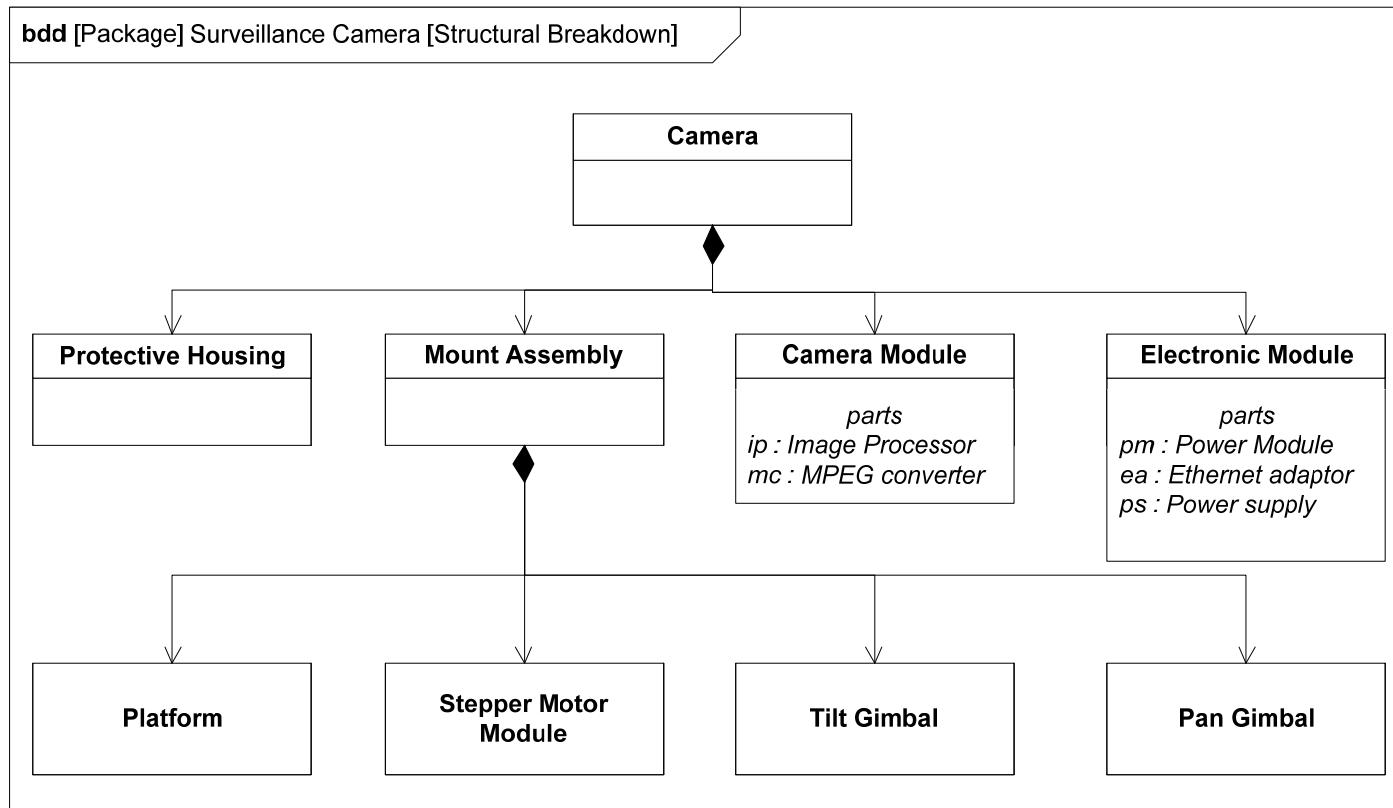


bdd: Variants

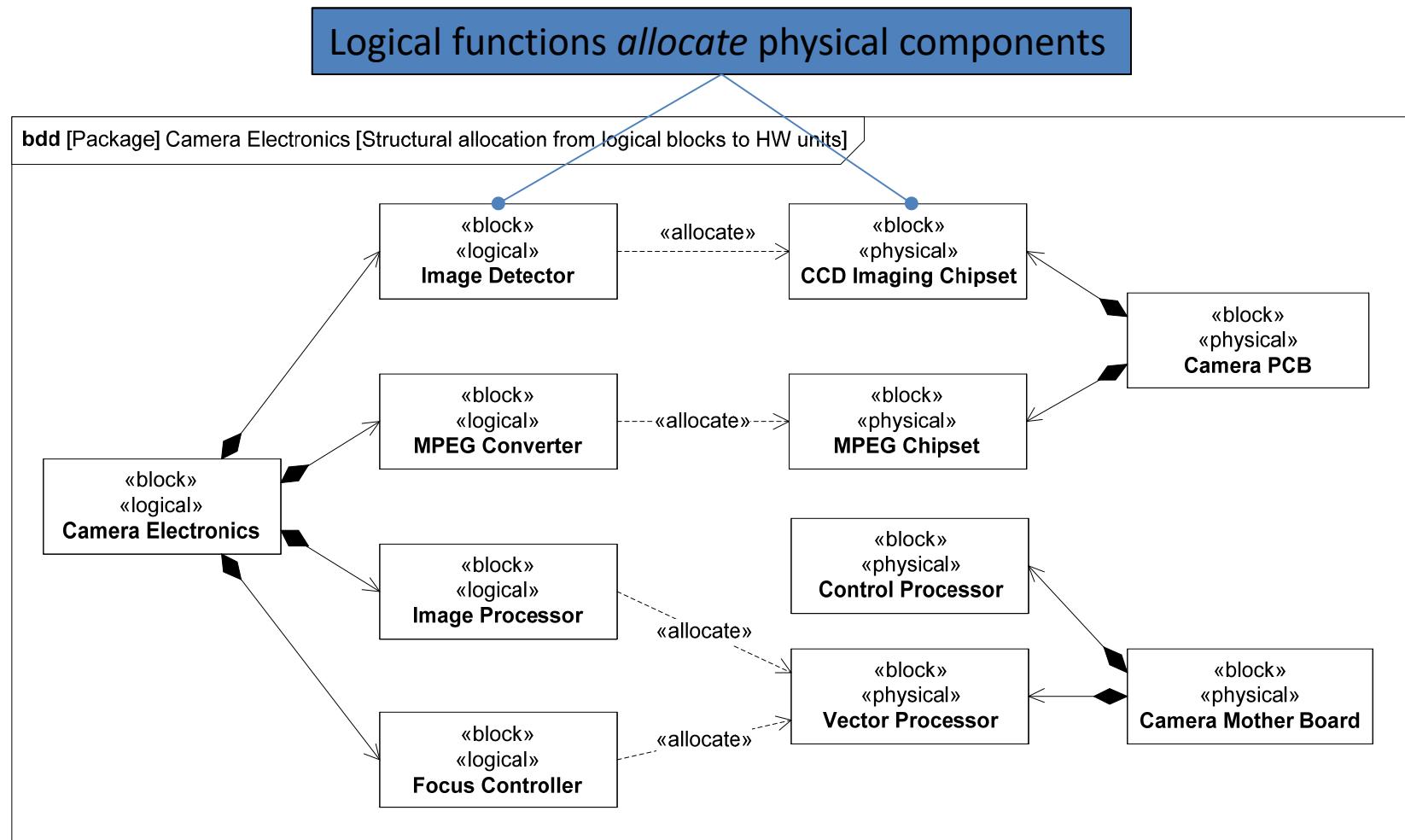


bdd: Deeper hierarchy

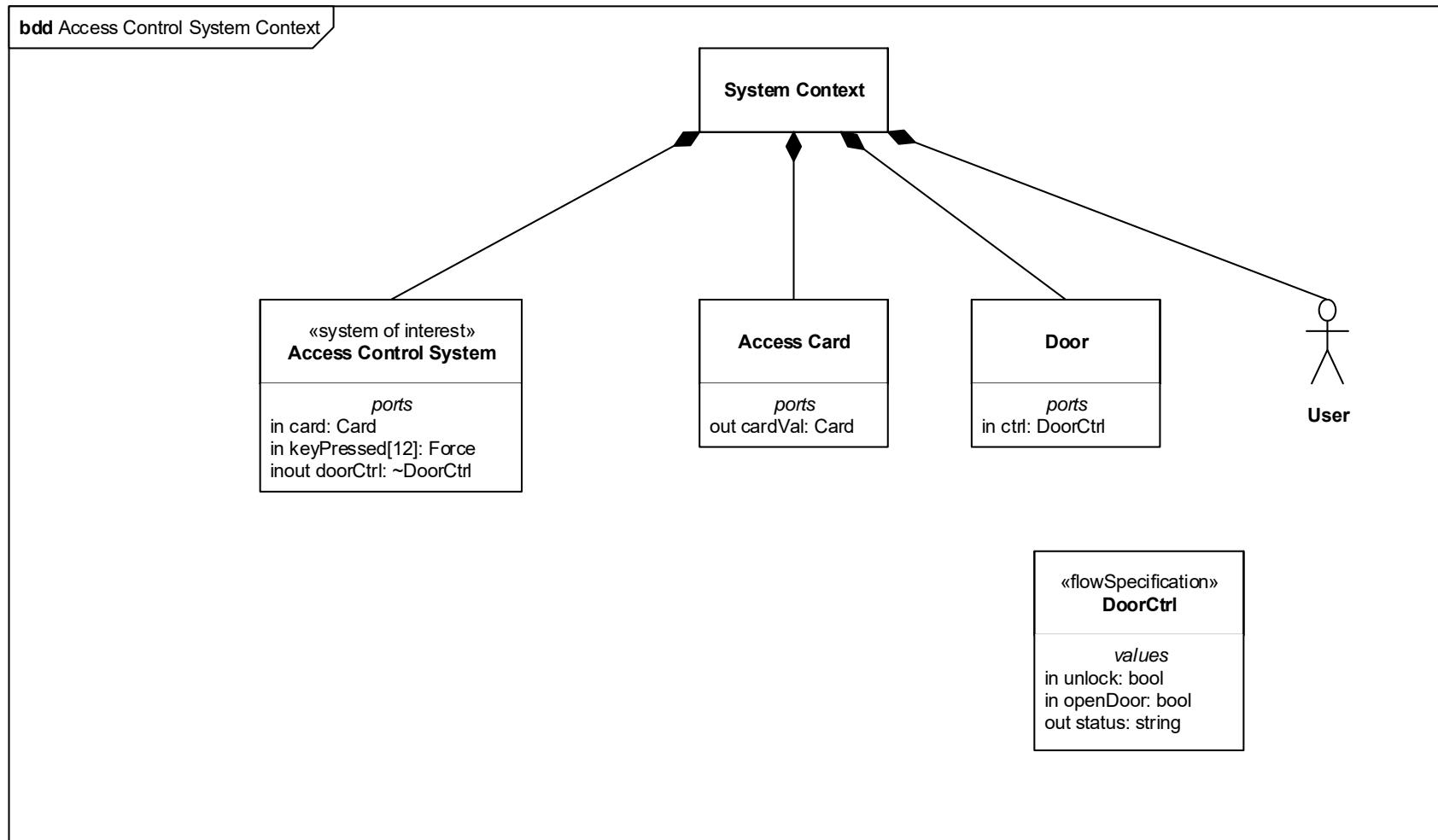
How would you read this diagram? "A camera consists of..."



bdd: Another use

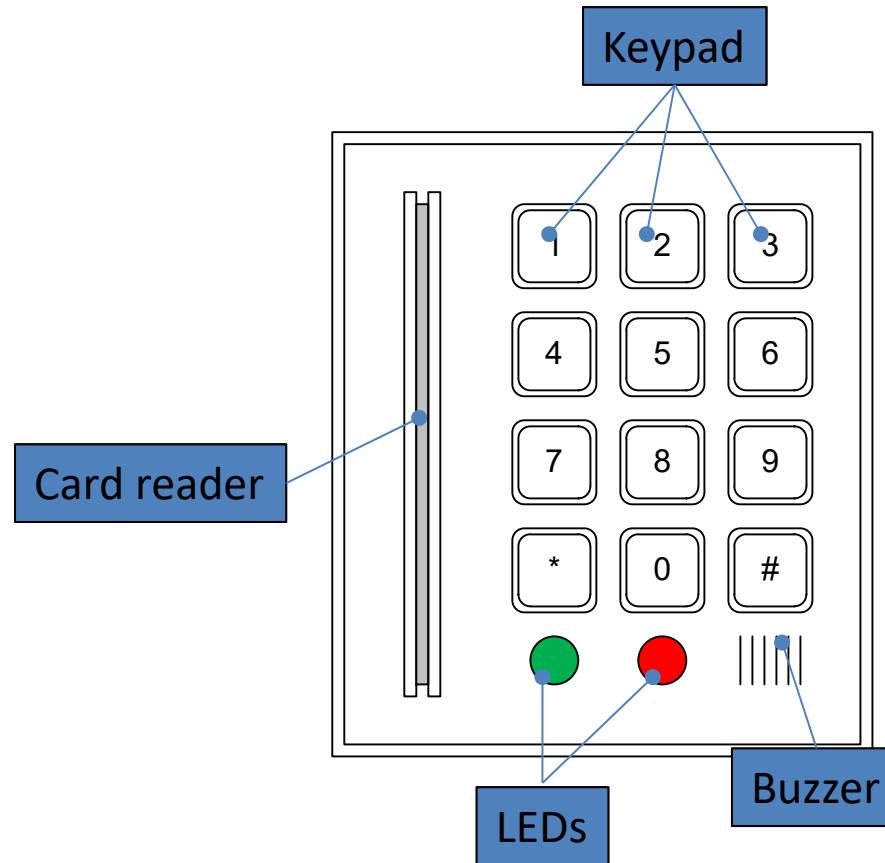


bdd: Defining the system's context



Your turn!

- Create a bdd for an access control system



BDD for BeoSoundF

Create a bdd for BeosoundF

Blocks:

- Speaker
3 speakers with names: T1, T2, W
- Amplifier
- CPU Board
- Bluetooth
- Power Supply
- Motor
- Inductive Mat
- User IF

ISE

Quick Guide for SysML Diagrams and Symbols

SysML Structure diagrams

A Practical Guide to SysML. Sanford Friedenthal, Alan Moore and Rick Steiner. Elsevier 2008.

SysML Reference Guide: Table A3, A4, A5, A8 and A9.

Table A.3 Block Definition Diagram Nodes for Representing Block Structure and Values

Diagram Element	Notation	Description	Section
Block Node	<pre> <block> <Name> parts <Part>:<Block>[<Multiplicity>] references <Reference>:<Block>[<Multiplicity>] values <ValueProperty>:<ValueType>=<ValueExpression> operations <Operation>(<Parameter>,...):<Type> receptions <>signal<>Signal(<Parameter>,...) </pre>	<p>The block is the fundamental modular unit for describing system structure in SysML.</p> <p>Compartments are used to show structural features (parts, references, values) and behavioral features (operations, receptions) of the block. See the following tables in this section for more block compartments.</p> <p>Additional properties on blocks are {encapsulated, abstract}. Abstract may also be indicated by italicizing the <Name>.</p> <p>Additional properties on structural features include: {ordered, unordered, unique, nonunique, subsets <Property>, redefines <Property>}.</p> <p>A forward slash (/) before a property name indicates that it is derived.</p>	6.2, 6.3, 6.5.2
Dimension and Unit Nodes	<p>«unit» <Name></p> <p>dimension ==<Dimension></p> <p>«dimension» <Name></p>	A dimension identifies a physical quantity such as length, whose value may be stated in terms of defined units, such as meters or feet. A unit must always be related to a dimension.	6.3.3
Value Type Node	<pre> <valueType> <Name> values <ValueProperty>:<ValueType>=<ValueExpression> operations <Operation>(<Parameter>,...):<Type> dimension=<Dimension> unit=<Unit> </pre>	A value type is used to provide a uniform definition of a quantity with units that can be shared by many value properties.	6.3.3
Enumeration Node	<p>«enumeration» <Name></p> <p><EnumerationLiteral></p>	An enumeration defines a set of named values called literals.	6.3.3
Actor Node	<p>«actor» <Name></p> <p><Name></p>	An actor is used represent the role of a human, an organization, or any external system that participates in the use of some system being investigated.	11.3

Table A.4 Block Definition Diagram Nodes for Representing Interfaces

Diagram Element	Notation	Description	Section
Flow Specification Node	<pre>«flowSpecification» <Name></pre> <pre>flowProperties <Direction> <FlowProperty>:<Item></pre>	A flow specification defines the set of input and/or output flows for a noncomposite flow port. <Direction> may be one of: in, out, or inout.	6.4.3
Interface Node	<pre>«interface» <Name></pre> <pre>operations <Operation>(<Parameters>,...);<Type></pre> <pre>receptions «signal»<Signal>(<Parameter>,...)</pre>	An interface is used to specify the set of behavioral features either required or provided by a standard (service-based) port.	6.5.3
Port Compartments for Block Node	<pre>«block» <Name></pre> <pre>standardPorts <Port>:<Interface></pre> <pre>flowPorts <Direction> <Port>:<Type></pre>	Ports can be shown in separate compartments labeled flow ports and standard ports. <Direction> may be one of: in, out, or inout. Non-atomic flow ports do not have a direction but may have the keyword {conjugated}.	6.4.3, 6.5.2
Nonatomic Flow Port Node	<pre><Name>:<FlowSpecification>[<Multiplicity>] <></pre> <pre><Name>:<FlowSpecification>[<Multiplicity>] <></pre>	A nonatomic flow port describes an interaction point where multiple different items may flow into or out of a block. A shaded symbol implies a conjugate port.	6.4.3
Atomic Flow Port Node	<pre><Name>:<Item>[<Multiplicity>] →</pre> <pre><Name>:<Item>[<Multiplicity>] ⇠</pre> <pre><Name>:<Item>[<Multiplicity>] ←</pre>	An atomic flow port describes an interaction point where an item can flow into or out of a block, or both, as indicated by the direction of the arrow in the Atomic Flow Port Node.	6.4.3
Standard Port Node	<pre><Interface> ○ —> <Name>[<Multiplicity>]</pre> <pre><Interface> ⊙ —> <Name>[<Multiplicity>]</pre>	A standard port defines the service-based interaction points on the interface to a block. The shape of the <Interface> symbol indicates whether services are required (socket) by or provided by (ball) the block.	6.5.3
Interface Realization Path	<pre>-----→</pre>	A realization dependency asserts that a block will declare a behavioral feature for each behavioral feature in an interface.	6.5.3
Usage Dependency Path	<pre>-----→></pre>	A uses dependency asserts that a block requires a set of behavioral features defined by an interface.	6.5.3

Table A.5 Block Definition Diagram Paths

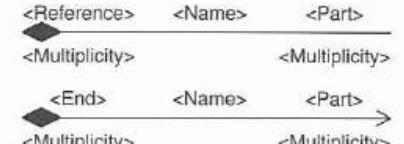
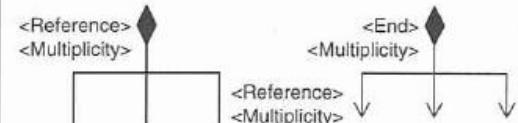
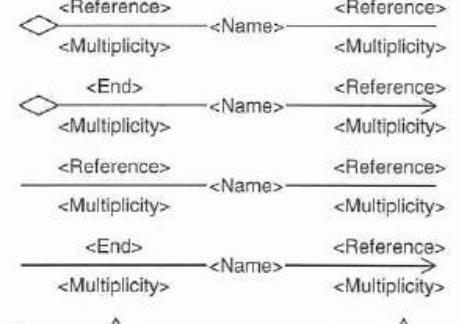
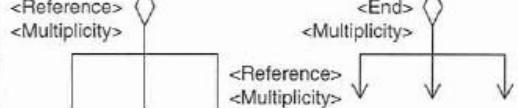
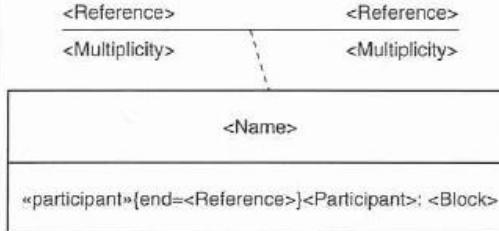
Diagram Element	Notation	Description	Section
Composite Association Path	 	<p>A composite association relates a whole to its parts showing the relative multiplicity at both whole and part ends. A composite association always defines a part property in the whole (indicated by <Part>).</p> <p>Where there is no arrow on the nondiamond end of the association it also specifies a reference property to the whole in the part (indicated by <Reference>).</p> <p>Otherwise when there is an arrow, the name at the whole end simply gives a name to the association end (indicated by <End>).</p>	6.3.1
Reference Association Path	 	<p>A reference association can be used to specify a relationship between two blocks. A reference association can specify a reference property on the blocks at one or both ends.</p> <p>The white diamond is the same as no diamond, but profiles can be used to differentiate them by specifying additional constraints.</p>	6.3.2
Association Block Path and Node		<p>An association block, as the name implies, is a combination of an association and a block, so it can relate two blocks together but can also have internal structure and other features of its own.</p> <p>Participants are placeholders that represent the blocks at each end of the association block, and are used when it is desired to decompose a connector.</p>	6.3.2
Generalization Path		<p>A generalization describes the relationship between the general classifier and specialized classifier. A set of generalizations may either be {disjoint} or {overlapping}. They may also be {complete} or {incomplete}.</p>	6.6

Table A.8 Internal Block Diagram Nodes

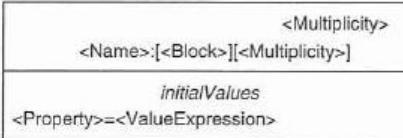
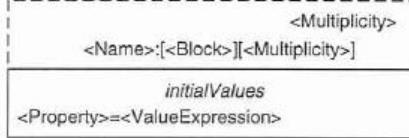
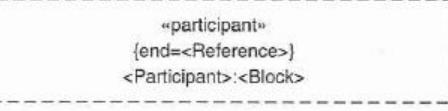
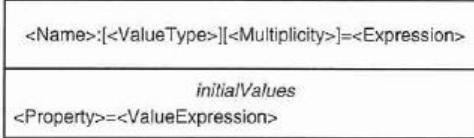
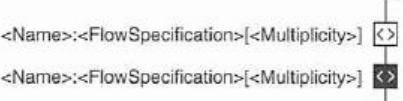
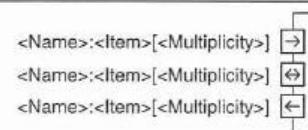
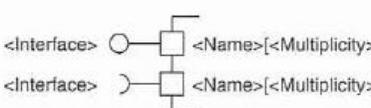
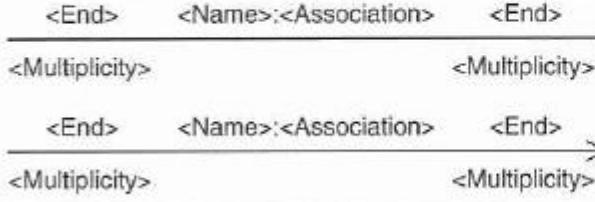
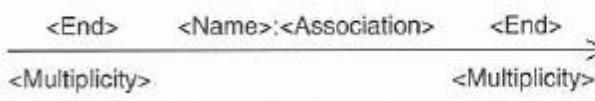
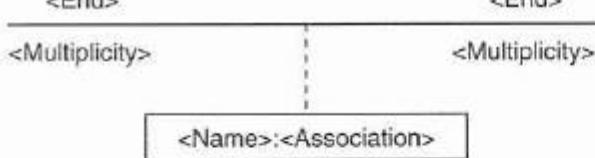
Diagram Element	Notation	Description	Section
Part Node		A part is a property of an owning block that is defined (typed) by another block. The part represents a usage of the defined block in the context of the owning block. Note that a Part Node may have the same compartments as a Block Node. [<Block>] represents a property-specific type.	6.3.1, 6.6.5
Actor Part Node	 $\langle\text{Name}\rangle:\langle\text{Actor}\rangle[\langle\text{Multiplicity}\rangle]$	An actor part is a property of a owning block that is defined (typed) by an actor.	11.5
Reference Node		A reference property of a block is a reference to another block. Note that a Reference Property Node may have the same compartments as a Block Node. [<Block>] represents a property-specific type.	6.3.2
Participant Property Node		A participant property represents one end of an association block. Using a participant property, a modeler can show the relationship between the internal structure of the association block and the internal structure of its related ends.	6.3.2
Value Property Node		A value property describes the quantitative characteristics of a block. Note that a Value Property Node may have the same compartments as a Value Type Node. [<ValueType>] represents a property-specific type.	6.3.3
Nonatomic Flow Port Node		A nonatomic flow port describes an interaction point that allows multiple different items to flow into or out of a block. A nonatomic flow port is typed by a flow specification. A shaded symbol implies a conjugate port that reverses the items' allowable in and out flow direction.	6.4.3
Atomic Flow Port Node		An atomic flow port describes an interaction point where an item can flow into or out of a block, or both, as indicated by the direction of the arrow in the Atomic Flow Port Node.	6.4.3
Standard Port Node		A standard port describes a service-based interaction point on a block. A standard port is defined by its interface. The shape of the <Interface> symbol indicates whether services are required by or provided by the block.	6.5.3

Table A.9 Internal Block Diagram Paths

Diagram Element	Notation	Description	Section
Connector Path	 $\langle\text{End}\rangle \quad \langle\text{Name}\rangle;\langle\text{Association}\rangle \quad \langle\text{End}\rangle$ <hr/> $\langle\text{Multiplicity}\rangle \qquad \qquad \qquad \langle\text{Multiplicity}\rangle$  $\langle\text{End}\rangle \quad \langle\text{Name}\rangle;\langle\text{Association}\rangle \quad \langle\text{End}\rangle$ <hr/> $\langle\text{Multiplicity}\rangle \qquad \qquad \qquad \langle\text{Multiplicity}\rangle$	A connector is used to bind two parts (or ports) and provides the opportunity for those parts to interact, although the connector says nothing about the nature of the interaction.	6.3.1
Connector Property Path and Node	 $\langle\text{End}\rangle \qquad \qquad \qquad \langle\text{End}\rangle$ <hr/> $\langle\text{Multiplicity}\rangle \qquad \qquad \qquad \langle\text{Multiplicity}\rangle$ <div style="border: 1px solid black; padding: 5px; display: inline-block;"> $\langle\text{Name}\rangle;\langle\text{Association}\rangle$ </div>	More detail can be specified for connectors by typing them with association blocks. An association block, as the name implies, is a combination of an association and a block, so it can relate two blocks together but can also have internal structure and other features of its own.	6.3.2
Item Flow Node	 $\langle\text{Name}\rangle;\langle\text{Item}\rangle, \dots$ <hr/> <div style="text-align: center;">  </div> $\langle\text{Name}\rangle;\langle\text{Item}\rangle, \dots$	An item flow is used to specify the items that flow across a connector in a particular context. An item flow specifies the type of the item that is flowing and the direction of flow. It may also be associated to a property, called an item property, of the enclosing block to identify a specific usage of an item in the context of the enclosing block.	6.4.2

SysML Behavior diagrams

A Practical Guide to SysML. Sanford Friedenthal, Alan Moore and Rick Steiner. Elsevier 2008.

SysML Reference Guide: Table A11, A12, A13, A14, A15, A16, A18, A19 and A20.

Table A.15 Sequence Diagram Structural Nodes

Diagram Element	Notation	Description	Section
Lifeline Node		A lifeline represents the relevant lifetime of an instance that is part of the interaction's owning block, which will either be represented by a part property or a reference property.	9.4
Single-compartment Fragment Node		A combined fragment can be used to model complex sequences of messages. A number of combined fragments have operators with only a single compartment for all operands, shown as <UnaryOp>. These are: seq, opt, break, strict, loop, neg, assert, critical.	9.7.1, 9.7.2
Multi-compartment Fragment Node		<p>Two combined fragments have operators with a compartment per operand, shown as <N-aryOp>. These are par and alt.</p> <p>The lifelines that participate in the fragment overlay on top of the fragment (i.e., are visible) and lifelines that don't participate are obscured behind the fragment. (Note: This is also true of Single-Compartment Fragment Nodes.)</p>	9.7.1
Filtering Fragment Node		There are two combined fragments with filter operators: consider and ignore, shown as <FilterOp>. Inside such a construct, messages that have been explicitly ignored (or not considered) may be interleaved with valid traces.	9.7.2
State Invariant Symbol		A state invariant on a lifeline is used to add a constraint on the required state of a lifeline at a given point in a sequence of event occurrences. The invariant constraint can include the values of properties or parameters, or the state of a state machine.	9.7.3
Interaction Use Node		An interaction use allows one interaction to reference another as part of its definition. The lifelines that participate in the interaction are obscured behind the fragment, and lifelines that don't participate overlay on top of the fragment (i.e., are visible).	9.8

Table A.16 Sequence Diagram Paths and Activation Nodes

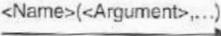
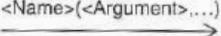
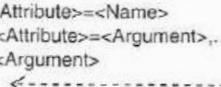
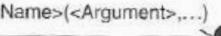
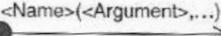
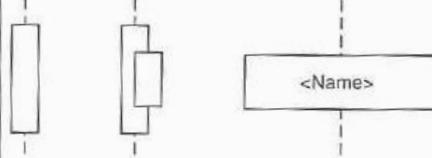
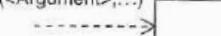
Diagram Element	Notation	Description	Section
Synchronous Message		A synchronous message corresponds to the synchronous invocation of an operation, and is generally accompanied by a reply message.	9.5.1
Asynchronous Message		Asynchronous messages correspond to either the sending of a signal or to an asynchronous invocation (or call) of an operation, and do not require a reply message.	9.5.1
Reply Message		A reply message shows a reply to a synchronous operation call, together with any return arguments.	9.5.1
Found Message Path		A lost message describes the case where there is sending event for the message but no receiving event.	9.5.2
Lost Message Path		A found message describes the case where there is receiving event for the message but no sending event.	9.5.2
Focus of Control (Activation) Node		Focus of control bars or activations are overlaid on lifelines and correspond to executions; they begin at the execution's start event, and end at the execution's end event. When executions are nested, the focus of control bars are stacked from left to right. An alternate notation for activations is a box symbol overlaid on the lifeline with the name of the behavior or action inside.	9.5.4
Create Message Path		The creation of an instance is indicated by the receipt of a create message.	9.5.5
Destroy Event Node		An instance's destruction is indicated by the occurrence of a destroy event.	9.5.5
Coregion Symbol		Within a coregion, there is no implied order between any messages sent or received by the lifeline.	9.7.1

Table A.18 State Machine Diagram State Nodes

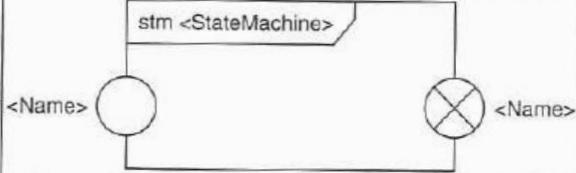
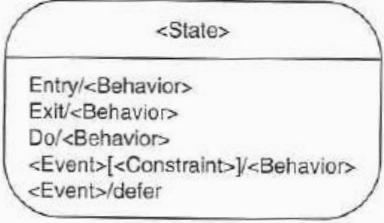
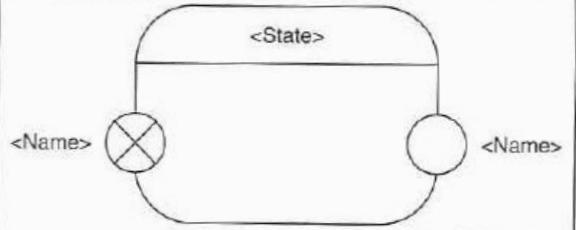
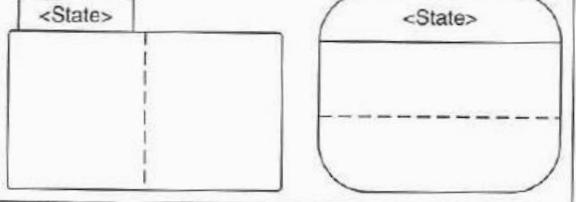
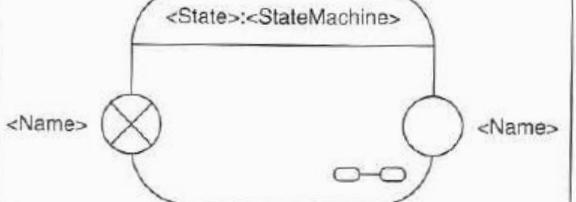
Diagram Element	Notation	Description	Section
State Machine with Entry- and Exit-Point Pseudostate Nodes		A state machine may have entry- and exit-point pseudostates, which are similar to junctions. On state machines, entry-point pseudostates can only have outgoing transitions and exit-point pseudostates can only have incoming transitions.	10.6.5
Atomic State Node		A state represents some significant condition in the life of a block, typically because it represents some change in how the block responds to events. Each state may have entry and exit behaviors that are performed whenever the state is entered or exited, respectively. In addition, the state may perform a do activity that executes once the entry behavior has completed and continues to execute until it completes or the state is exited.	10.3
Composite State with Entry- and Exit-Point Pseudostate Nodes		A composite state is a state with nested regions; the most common case is a single region. A composite state may have entry- and exit-point pseudostates that act like junction pseudostates. Entry points have incoming transitions from outside the state and exit points have the opposite.	10.6.1
Composite State Node with Multiple Regions		A composite state may have many regions, which may each contain substates. These regions are orthogonal to each other and so a composite state with more than one region is sometimes called an orthogonal composite state.	10.6.2
Sub-State Machine Node with Connection Points		A state machine may be reused using a kind of state called a submachine state. A transition ending on a submachine state will start its referenced state machine. Transitions may also be connected to connection points on the boundary of the state.	10.6.5

Table A.19 State Machine Diagram Pseudostate and Transition Nodes

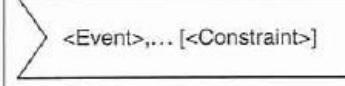
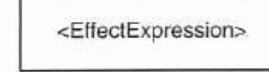
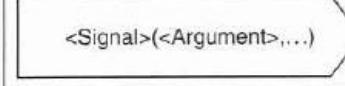
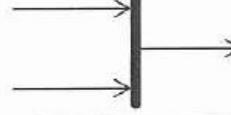
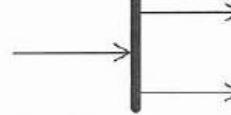
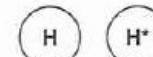
Diagram Element	Notation	Description	Section
Terminate Pseudostate Node		If a terminate pseudostate is reached, then the behavior of the state machine terminates.	10.3
Initial Pseudostate Node		An initial pseudostate specifies the initial state of a region.	10.3
Final State Node		The final state indicates that a region has completed execution.	10.3
Choice Pseudostate Node		The outgoing transitions of a choice pseudostate are evaluated once it has been reached.	10.4.2
Junction Pseudostate Node		A junction pseudostate is used to construct a compound transition path between states.	10.4.2
Trigger Node	 <code><Event>, ... [<Constraint>]</code>	This node represents all the transition's triggers, with the descriptions of the triggering events and the transition guard inside the symbol.	10.4.3
Action Node	 <code><EffectExpression></code>	<code><EffectExpression></code> describes the effect of the transition, either the name of a behavior or the body of an opaque behavior.	10.4.3
Send Signal Node	 <code><Signal>(<Argument>, ...)</code>	This node represents a send signal action. The signal's name, together with any arguments that are being sent, are shown within the symbol.	10.4.3
Join Pseudostate Node		A join pseudostate has a single outgoing transition and many incoming transitions. When all of the incoming transitions can be taken, and the join's outgoing transition is valid, then all the transitions happen.	10.6.2
Fork Pseudostate Node		A fork pseudostate has a single incoming transition and many outgoing transitions. When an incoming transition is taken to the fork pseudostate, all of the outgoing transitions are taken.	10.6.2
History Pseudostate Node		A history pseudostate represents the last state of its owning region, and a transition ending on a history pseudostate has the effect of returning the region to the state it was last in.	10.6.4



Table A.20 State Machine Diagram Paths

Diagram Element	Notation	Description	Section
Time Event Transition Path	$\xrightarrow{\text{after } <\text{TimeExpression}>[<\text{Constraint}>]/<\text{Behavior}>}$ $\xrightarrow{\text{at } <\text{TimeExpression}>[<\text{Constraint}>]/<\text{Behavior}>}$	Time events indicate either that a given time interval has passed since the current state was entered (after), or that a given instant of time has been reached (at). The transition can also include a guard and effect.	10.4.1
Signal Event Transition Path	$\xrightarrow{<\text{Signal}>(<\text{Attribute}>, \dots)[<\text{Constraint}>]/<\text{Behavior}>}$	Signal events indicate that a new asynchronous message has arrived. A signal event may be accompanied by a number of arguments, which may be assigned to attributes. The transition can also include a guard and effect.	10.4.1
Call Event Transition Path	$\xrightarrow{<\text{Operation}>(<\text{Attribute}>, \dots)[<\text{Constraint}>]/<\text{Behavior}>}$	Call events indicate that an operation on the state machine's owning block has been requested. A call event may also be accompanied by a number of arguments, which may be assigned to attributes. The transition can also include a guard and effect.	10.5
Change Event Transition Path	$\xrightarrow{\text{when } <\text{Expression}>[<\text{Constraint}>]/<\text{Behavior}>}$	Change events indicate that some condition has been satisfied (normally that some specific set of attribute values hold). The transition can also include a guard and behavior/effect.	10.7

Table A.11 Activity Diagram Structural Nodes

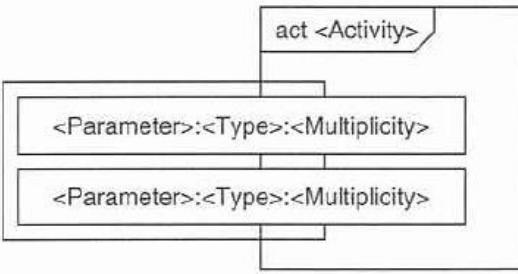
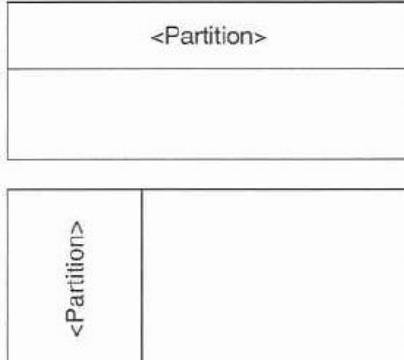
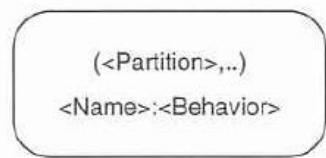
Diagram Element	Notation	Description	Section
Activity Parameter Node		<p>Activity parameter node symbols are rectangles that straddle the boundary of the activity frame.</p> <p>Other annotations include: «noBuffer», «optional», «overwrite», «continuous», «discrete», {rate=<Expression>}.</p> <p>Parameters can be organized into parameter sets, indicated by a bounding box around the parameters in the set. Parameter sets may overlap, and may have an annotation: {probability=<Expression>}.</p>	8.4.1
Interruptible Region Node		An interruptible region groups a subset of the actions within an activity and includes a mechanism for stopping their execution. Stopping the execution of these actions does not effect other actions in the activity.	8.8.1
Activity Partition Node		A set of activity nodes can be grouped into an activity partition (also known as a swimlane) that is used to indicate responsibility for execution of those nodes. <Partition> may be the name of a block or name and type of a part/reference. Partitions may overlap in a grid pattern.	8.9.1
Activity Partition in Action Node		An alternative representation for an activity partition for call actions is to include the name of the partition or partitions in parentheses inside the node above the action name. This can make the activity easier to layout than when using the swimlane notation.	8.9.1

Table A.12 Activity Diagram Control Nodes

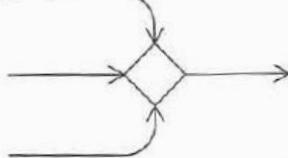
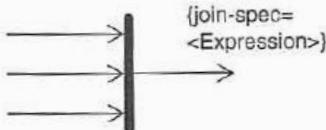
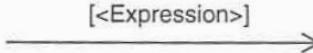
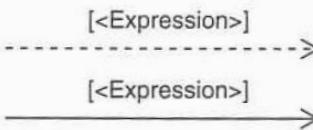
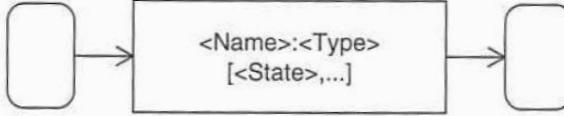
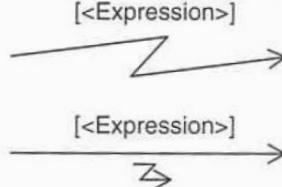
Diagram Element	Notation	Description	Section
Merge Node		A merge node has one output flow and multiple input flows—it routes each input token received on any input flow to its output flow. Unlike a join node, a merge node does not require tokens on all its input flows before offering them on its output flow. Rather it offers tokens on its output flow as soon as it receives them.	8.5.1, 8.6.1
Decision Node	 	A decision node has one input flow and multiple output flows—an input token can only traverse one output flow. The output flow is typically established by placing mutually exclusive guards on all outgoing flows and offering the token to the flow whose guard expression is satisfied. A decision node can have an accompanying decision input behavior, which is used to evaluate each incoming object token and whose result can be used in guard expressions.	8.5.1, 8.6.1
Join Node		A join node has one output flow and multiple input flows—it has the important characteristic of synchronizing the flow of tokens from many sources. Its default behavior can be overridden by providing a join specification, which can specify additional control logic.	8.5.1, 8.6.1
Fork Node		A fork node has one input flow and multiple output flows—it replicates every input token it receives onto each of its output flows. The tokens on each output flow may be handled independently and concurrently.	8.5.1, 8.6.1
Initial Node		When an activity starts executing a control token is placed on each initial node in the activity. The token can then trigger the execution of an action via an outgoing control flow.	8.6.1
Activity Final Node		When a control or object token reaches an activity final node during the execution of an activity, the execution terminates.	8.6.1
Flow Final Node		Control or object tokens received at a flow final node are consumed but have no effect on the execution of the enclosing activity. Typically they are used to terminate a particular sequence of actions without terminating an activity.	8.6.1

Table A.13 Activity Diagram Object and Action Nodes

Diagram Element	Notation	Description	Section
Call Action Node	<p><Name>;<Type>[<State>,...] → <Name>;<Behavior> → ↓ target → <Name>;<Operation> → localPrecondition Constraint localPostcondition Constraint</p>	<p>Call actions can invoke other behaviors either directly or through an operation, and are referred to as call behavior actions and call operation actions, respectively. A call action must own a set of pins that match in number and type of the parameters of the invoked behavior/operation. A called operation requires a target.</p> <p>Streaming pins may be marked as {stream} or filled (as shown).</p> <p>Where the parameters of the called entity are grouped into sets, the corresponding pins are as well. Pre- and postconditions can be specified that constrain the action such that it cannot begin to execute unless the precondition is satisfied, and must satisfy the postcondition to successfully complete execution.</p>	8.1, 8.3, 8.4.2
Central Buffer Node	<p>«centralBufferNode» <Name>;<Type> [<State>,...]</p>	A central buffer node provides a store for object tokens outside of pins and parameter nodes. Tokens flow into a central buffer node and are stored there until they flow out again.	8.5.3
Datastore Node	<p>«dataStore» <Name>;<Type> [<State>,...]</p>	A datastore node provides a copy of a stored token rather than the original. When an input token represents an object that is already in the store, it overwrites the previous token.	8.5.3
Control Operator Action Node	<p>→ <controlOperator> <Name>;<ControlOperator> → {control}</p>	A control operator produces control values on an output parameter, and is able to accept a control value on an input parameter (treated as an object token). It is used to specify logic for enabling and disabling other actions.	8.6.2
Accept Event Action Node	<p><Event>,...</p>	An activity can accept events using an accept event action. The action has (sometimes hidden) output pins for received data.	8.7
Accept Time Event Node	<p><TimeExpression></p>	A time event corresponds to an expiration of an (implicit) timer. In this case the action has a single (typically hidden) output pin that outputs a token containing the time of the accepted event occurrence.	8.7
Send Signal Action	<p>signal → <Signals> target →</p>	An activity can send signals using a send signal action. It typically has pins corresponding to the signal data to be sent and the target for the signal.	8.7
Primitive Action Node	<p>“<ActionType>” <Expression></p>	Primitive actions include: object access/update/manipulation actions, which involve properties and variables, and value actions, which allow the specification of values. The <Expression> will depend on the nature of the action.	8.12.1

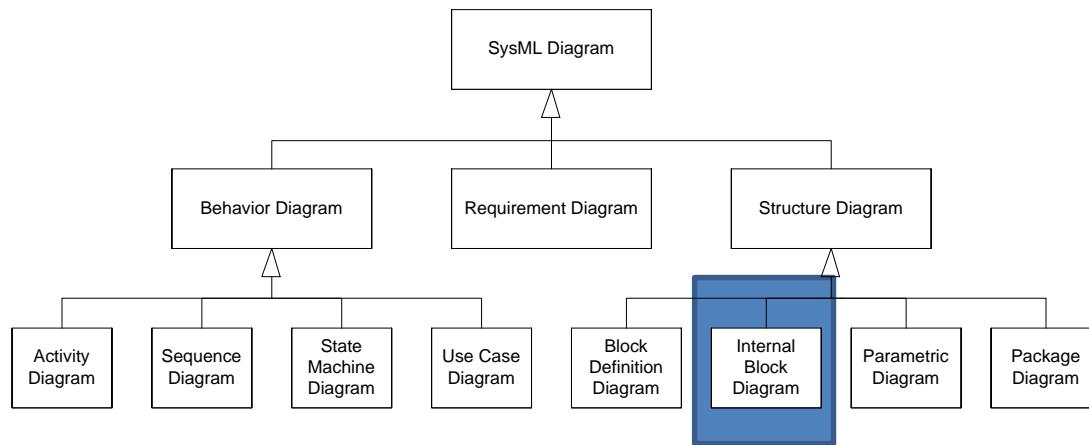
Table A.14 Activity Diagram Paths

Diagram Element	Notation	Description	Section
Object Flow Path		Object flows connect inputs and outputs. Additional annotations include «continuous», «discrete», {rate=<Expression>}, {probability=<Expression>}.	8.1, 8.5
Control Flow Path		Control flows provide constraints on when, and in what order, the actions within an activity will execute. A control flow can be represented using a solid line, or using a dashed line to more clearly distinguish it from object flow.	8.1, 8.6
Object Flow Node		When an object flow is between two pins that have the same characteristics, an alternative notation can be used where the pin symbols are elided and replaced by a single rectangular symbol called an object node symbol.	8.5
Interrupting Edge Path		An interrupting edge interrupts the execution of the actions in an interruptible region. Its source is a node inside the region and its destination is a node outside it.	8.8.1

SysML Structural Diagrams 2

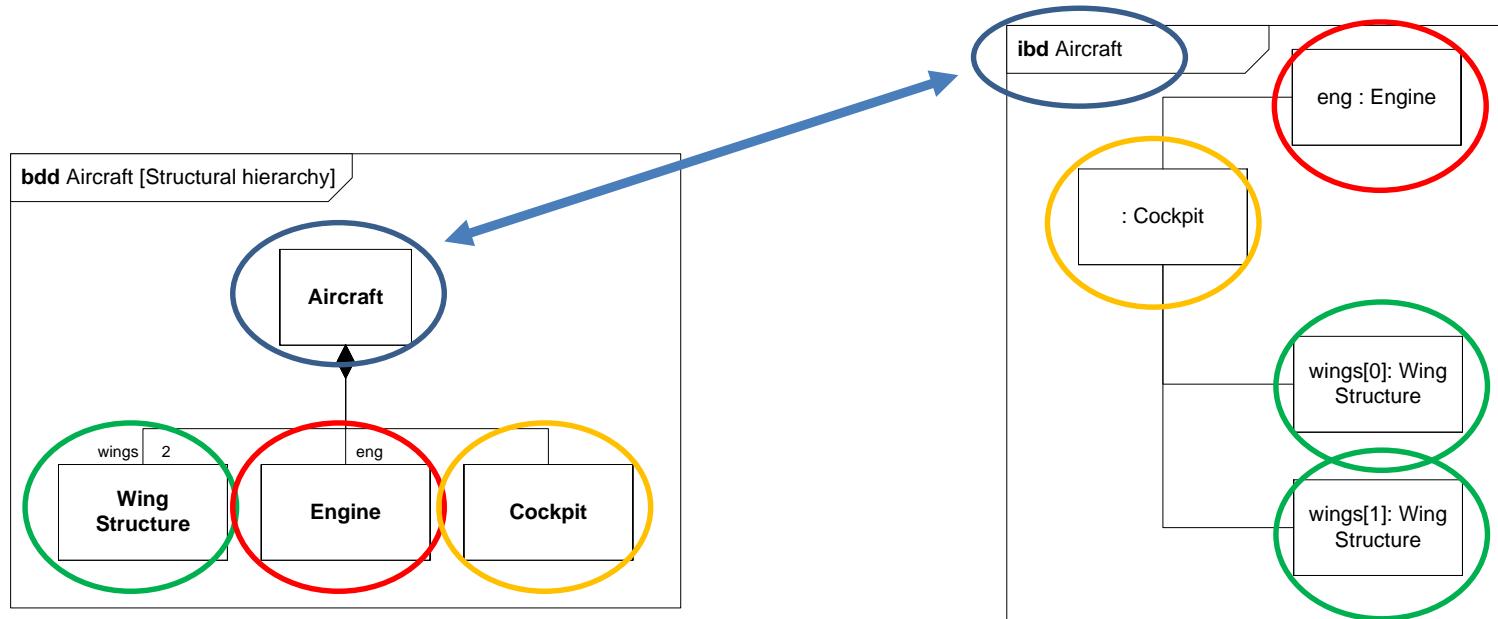
Introduction to Systems Engineering
I2ISE

SysML Internal Block Diagrams



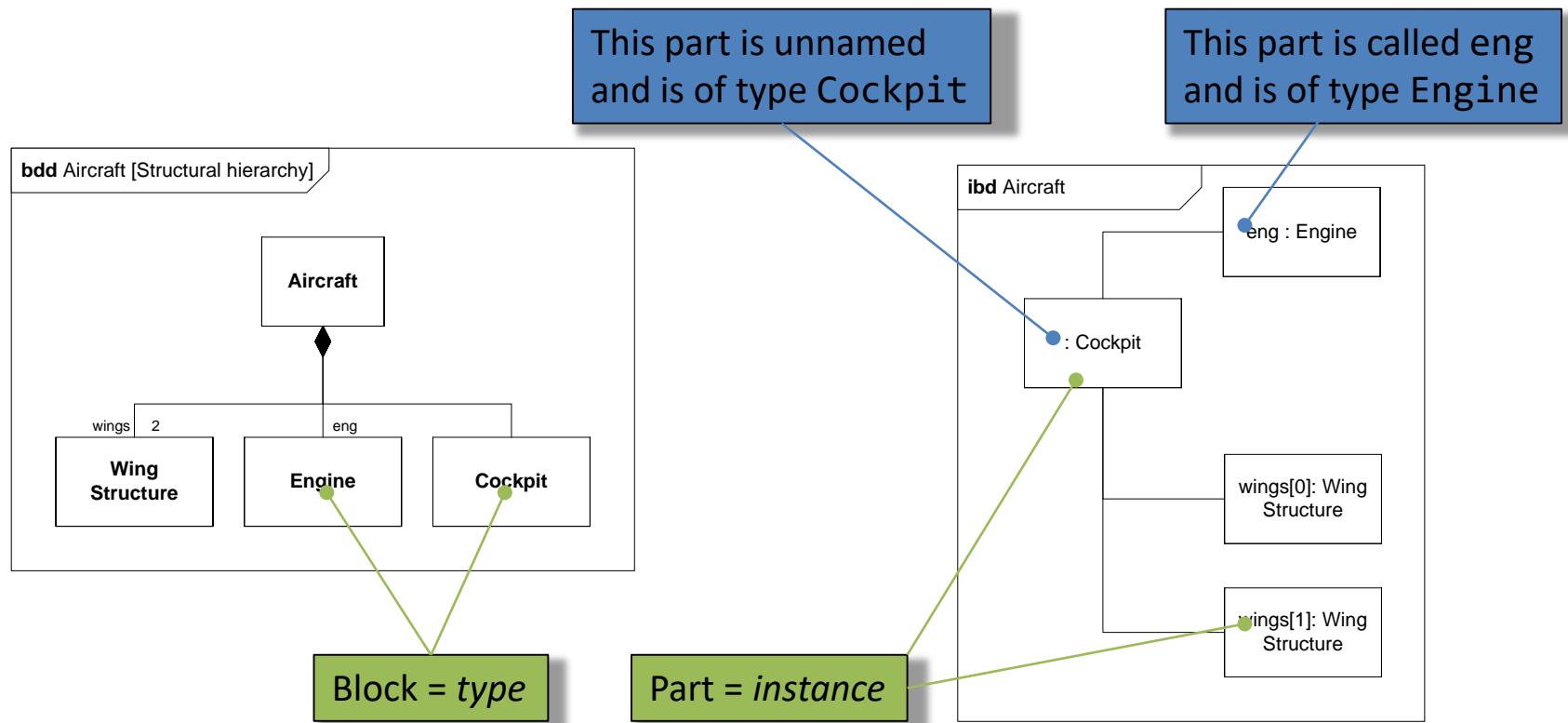
SysML: Internal Block Diagram

- An *Internal Block Diagram (ibd)* is used to define
 - the *interconnection* and *interfaces* of the parts of a block, and
 - the *information flow* between parts
- An ibd **always** relates to a block on a bdd. It shows the internal connections of the block's constituents



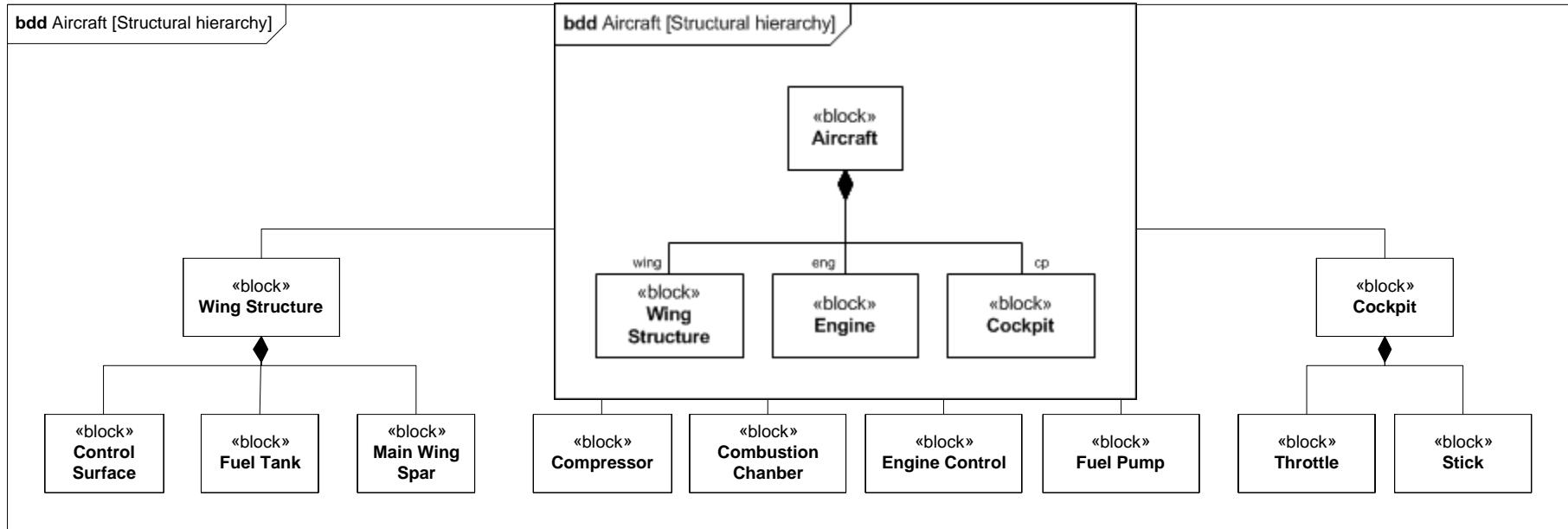
SysML: Blocks and parts

- A block is a *type definition* – there can be only one block with a given name
- A part is an *instance* of a block – there can be many instances of the same block

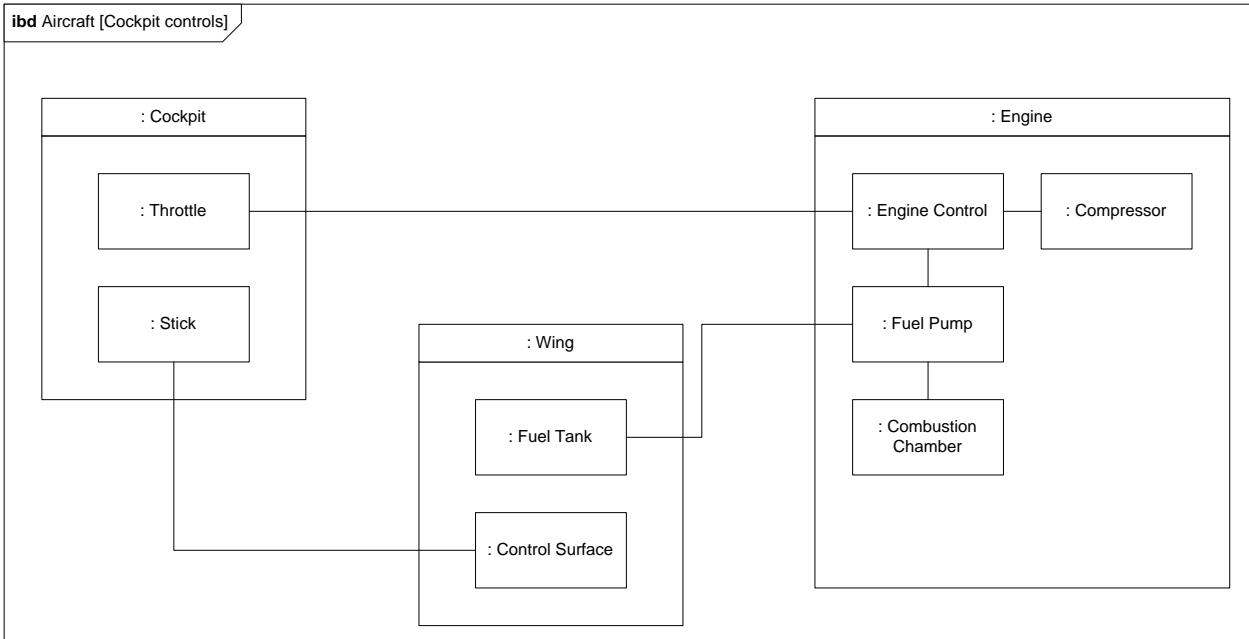
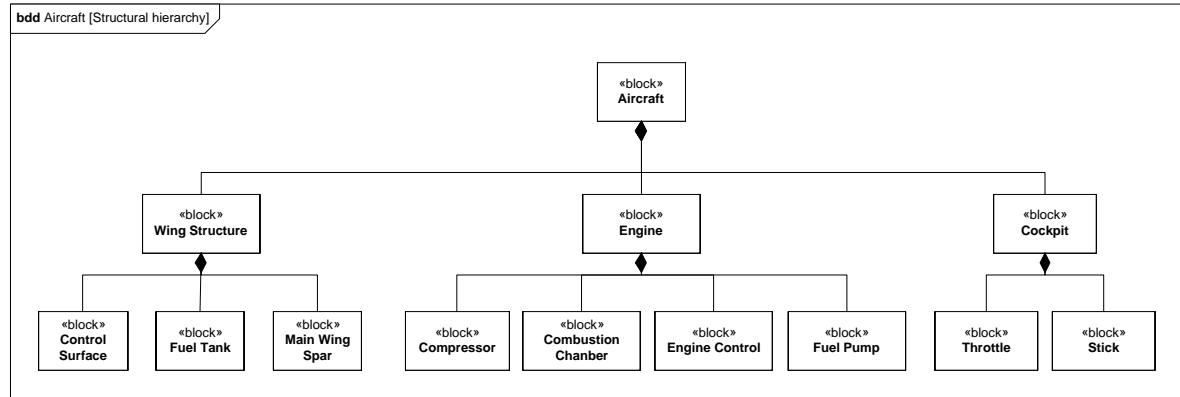
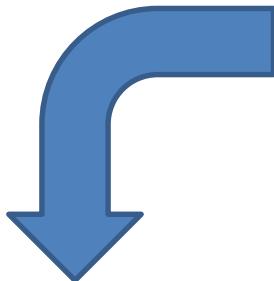


ibd: Aircraft - deep structure

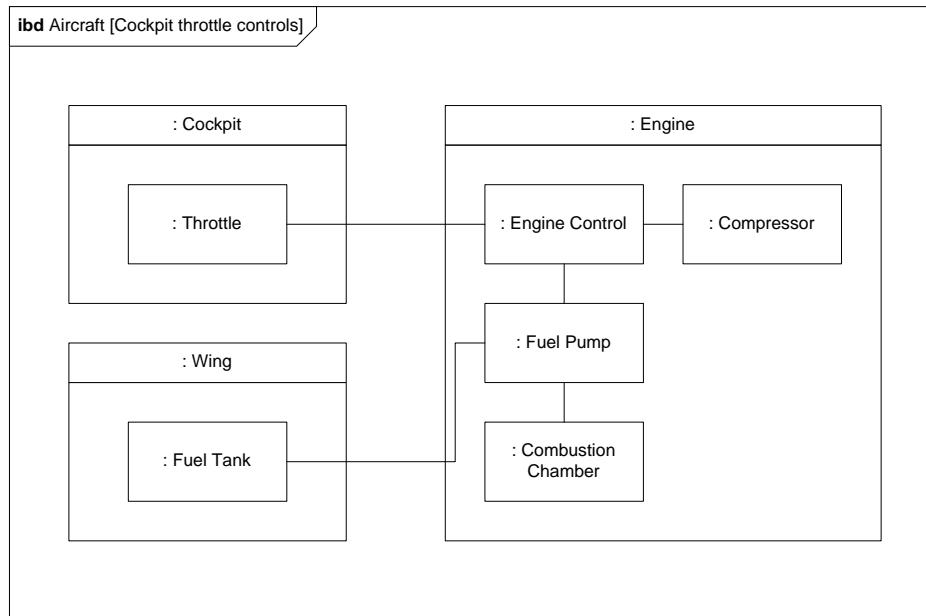
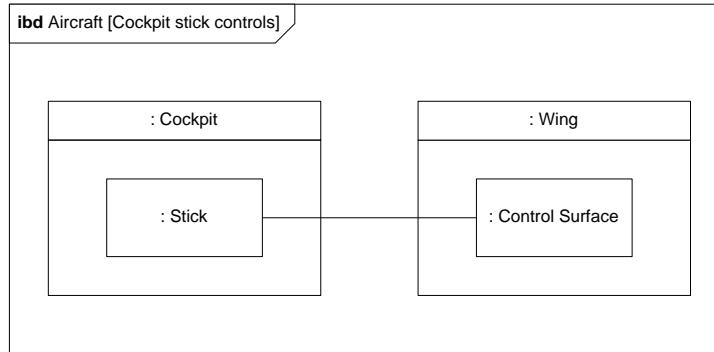
- Deep structure on a bdd can be shown in an ibd:



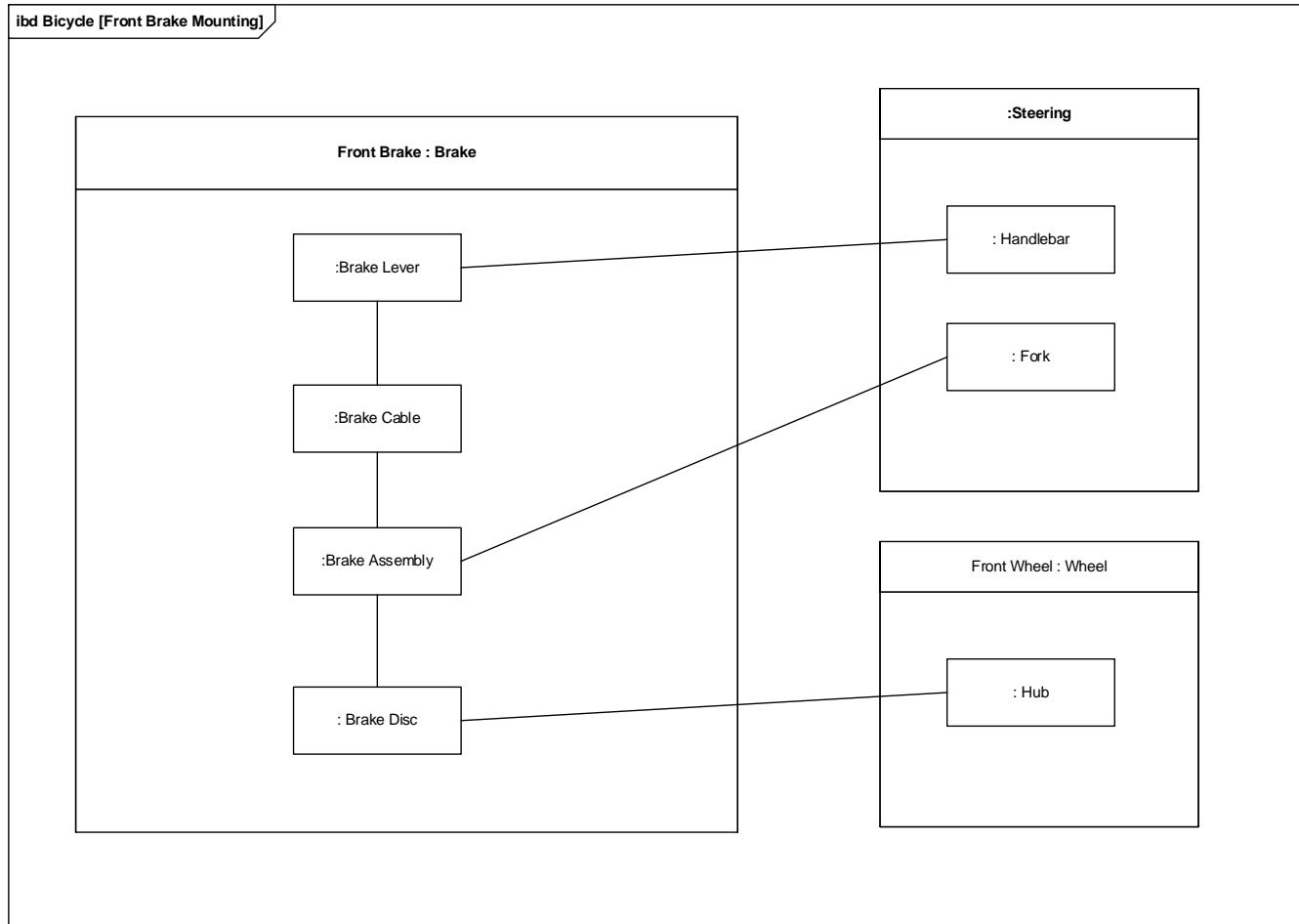
ibd: Aircraft - deep structure



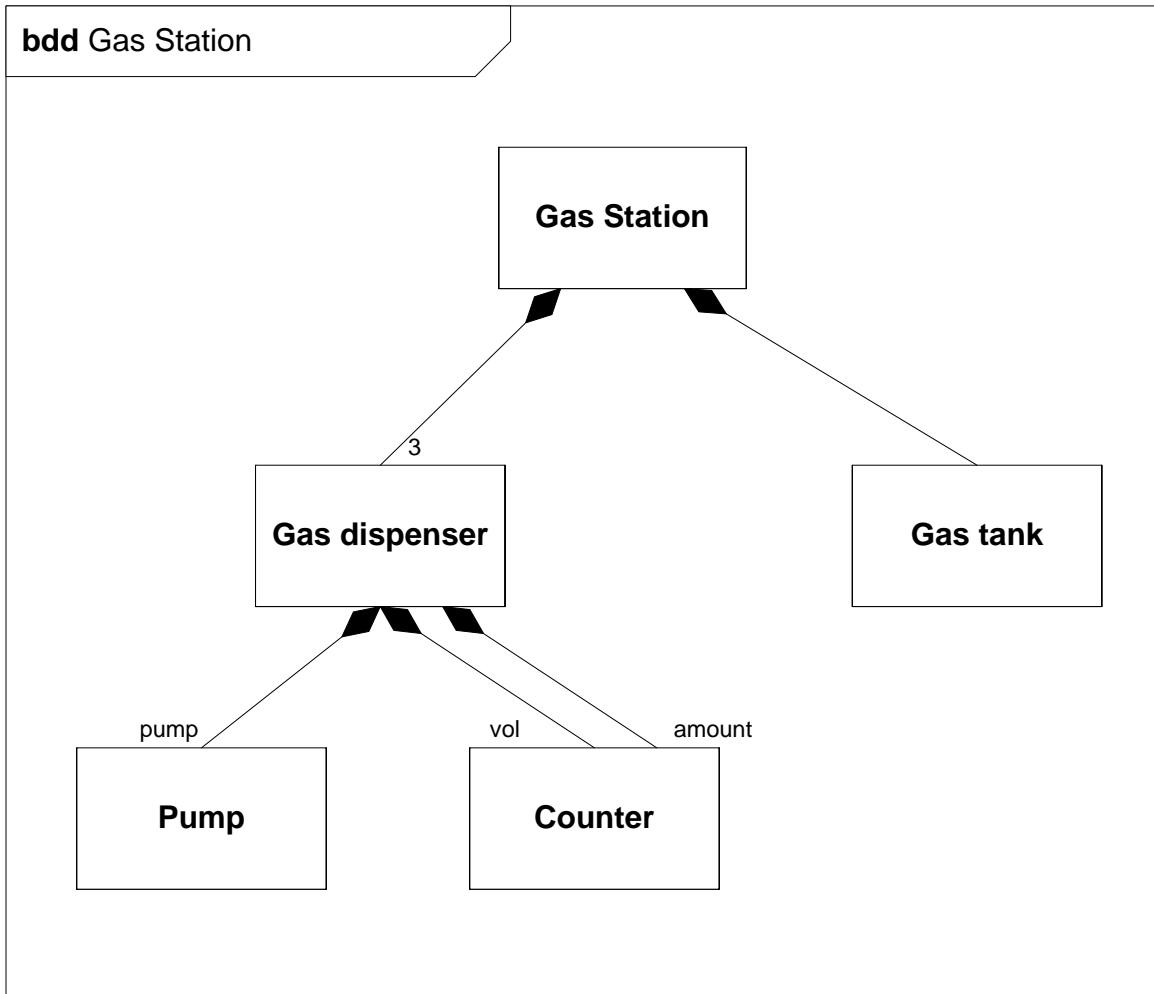
ibd: Aircraft – better deep structure



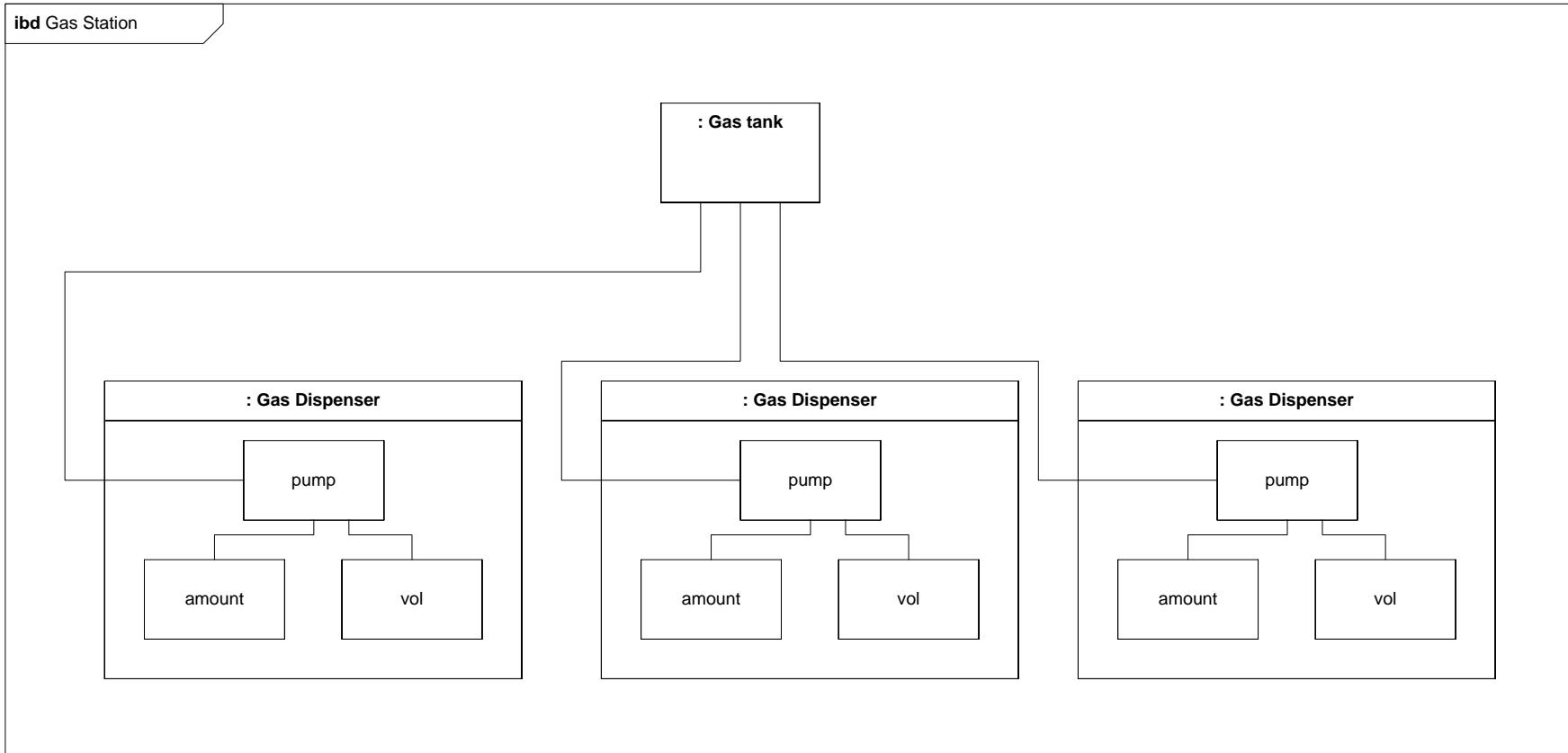
ibd : Bicycle – Front Brake Mounting



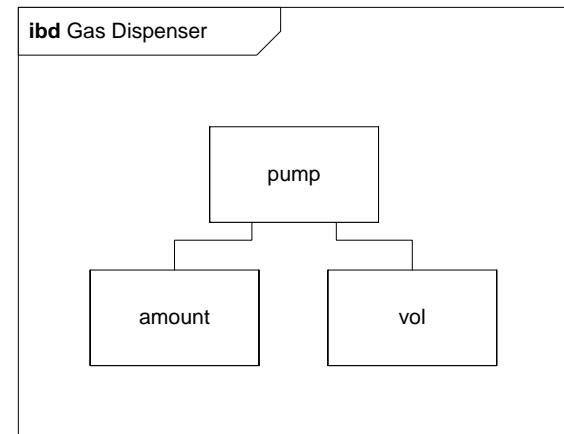
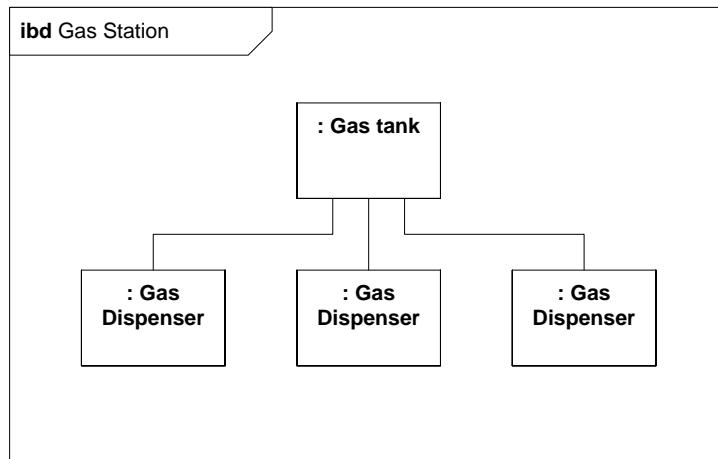
Gas station example - BDD



ibd: Gas station example

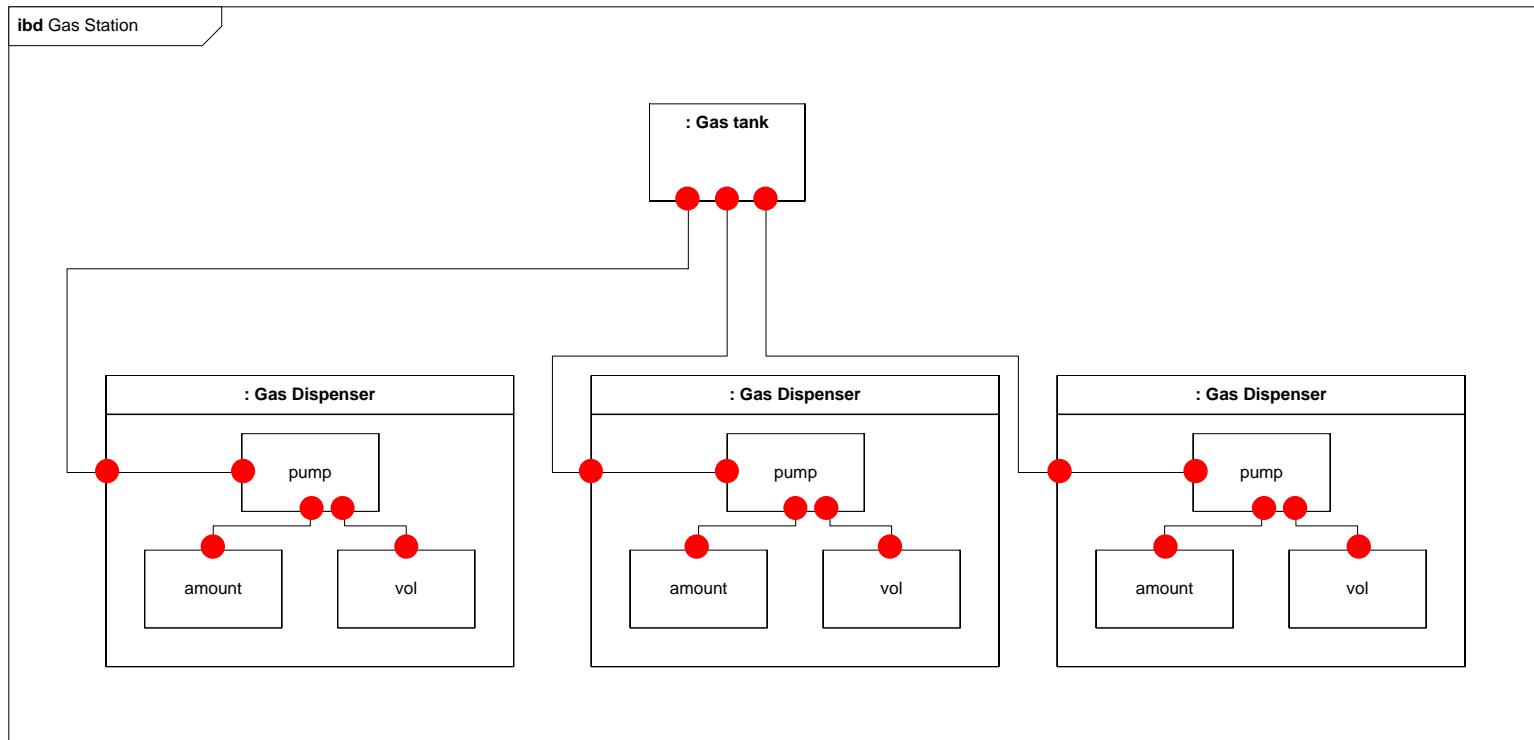


ibd: Better Gas station example



So far, so good...

- We can say a lot about the structure of a system in terms of *blocks* and *parts*...but what about their *interfaces*?



SysML

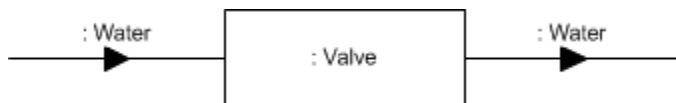
Modeling interfaces
using *items*, *item flows* and *ports*

Modeling interfaces

- We would like to express more about the *connection* between parts on the ibd
 - This would help us to define the *interface* of the parts
- To do this, we must define *items*, *item flows* and *ports*!

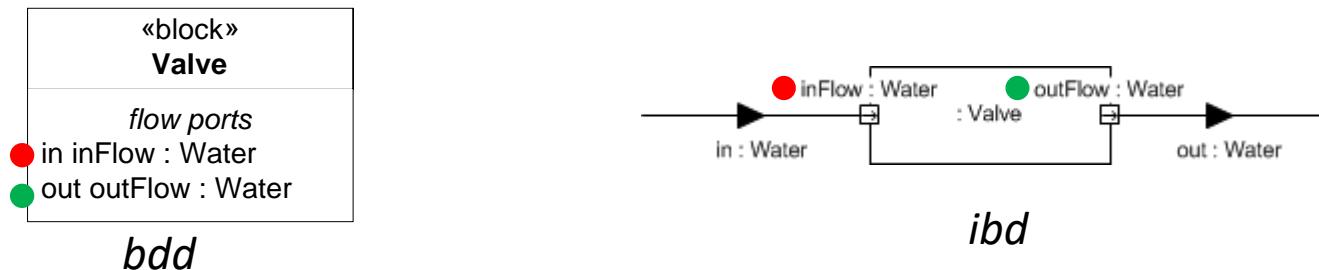
Items and item flows

- An *item* describes an entity that flows through a system (blocks, value types or signals)
 - Physical flow, information flow, energy, ...
 - Simple or complex
- An *item flow* is used to describe a flow of items (!) on a connector between two blocks on an ibd
 - Item flow = item *type* + flow *direction*



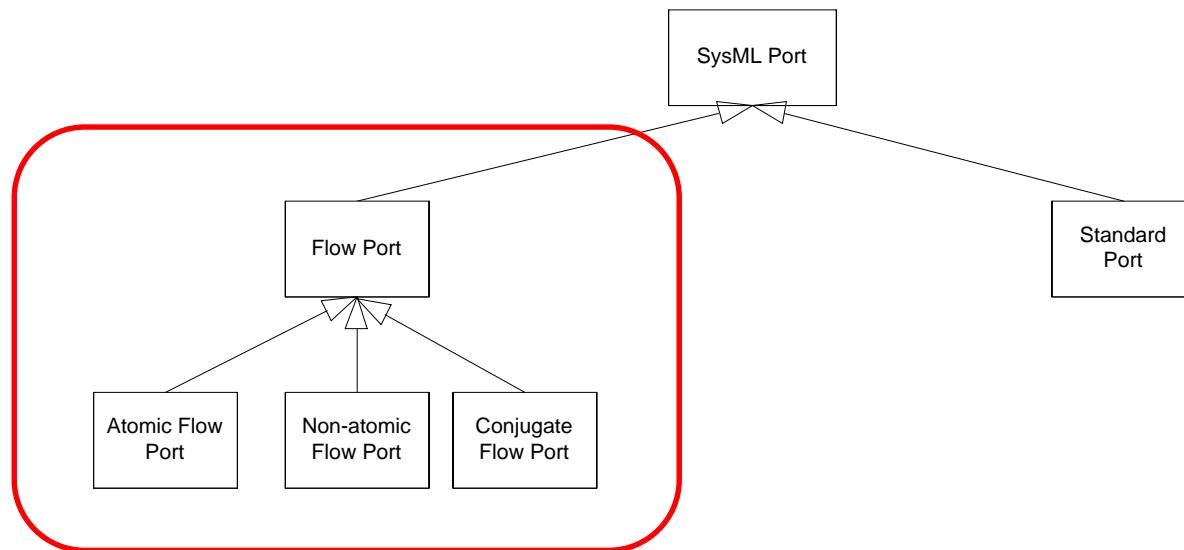
Ports

- A *port* is an interaction point on the boundary of a block
 - Ports are where the items flow into / out of
 - One block can have many ports
- Ports are *defined* on the blocks on a bdd and used to connect *parts* on ibds



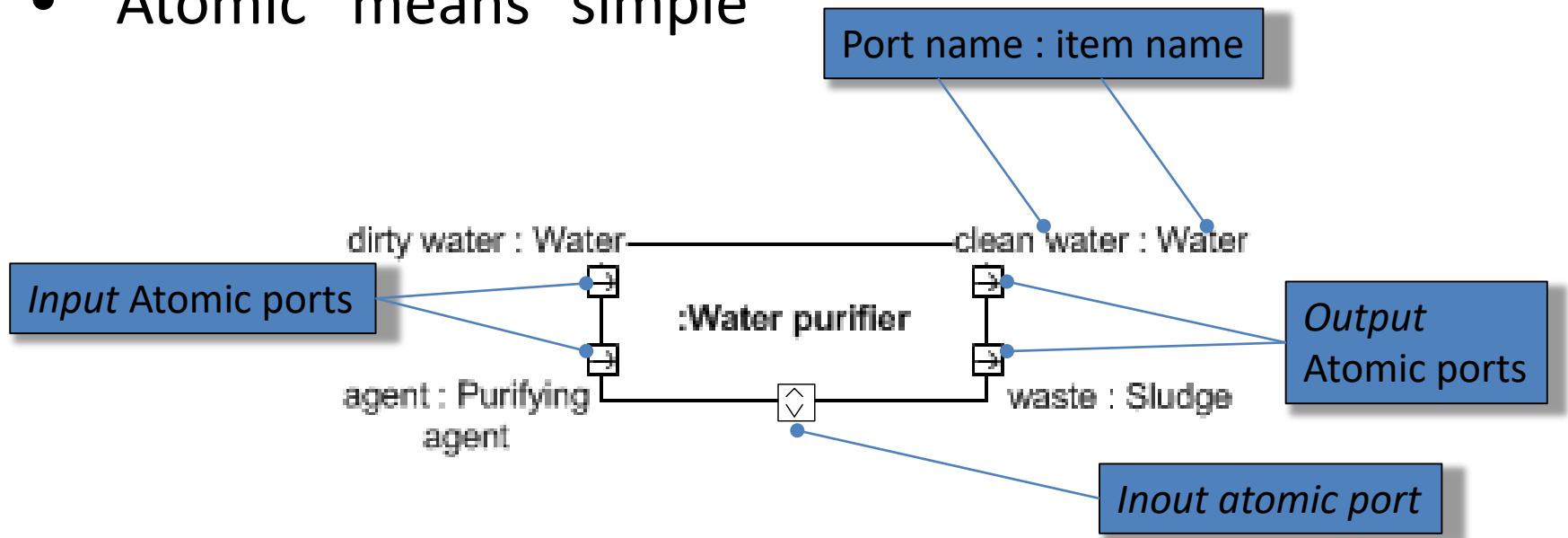
Ports

- Ports come in different flavours, each with different meaning and use
- We will concentrate on *flow* ports



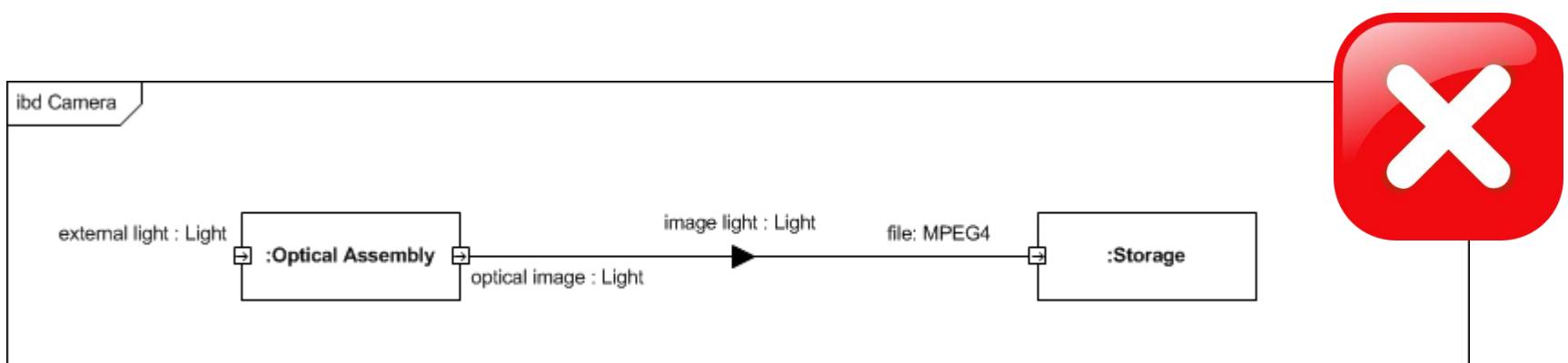
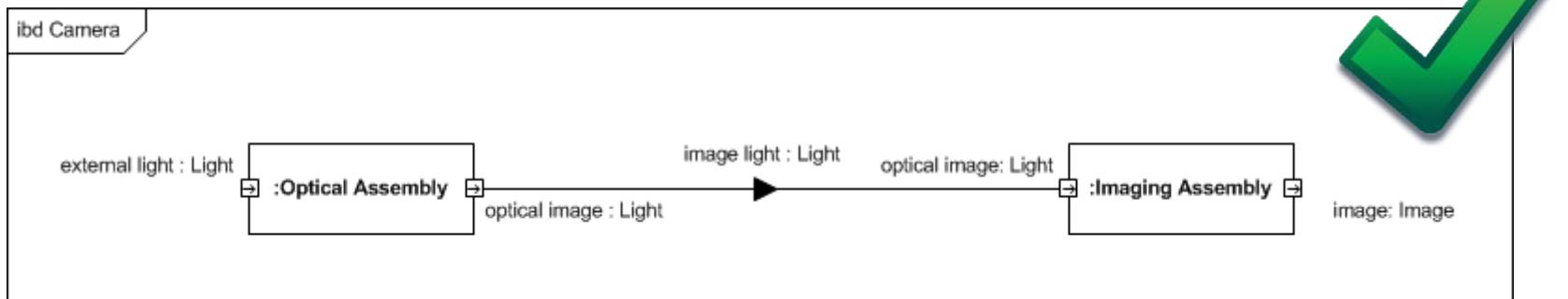
Atomic flow ports

- *Atomic flow ports* are used to describe flows of a single, simple type of item flow to/from a block
 - Directions: In, out or inout
- "Atomic" means "simple"



Atomic flow ports

- Atomic flow ports can be connected only if directions and item flow are compatible:

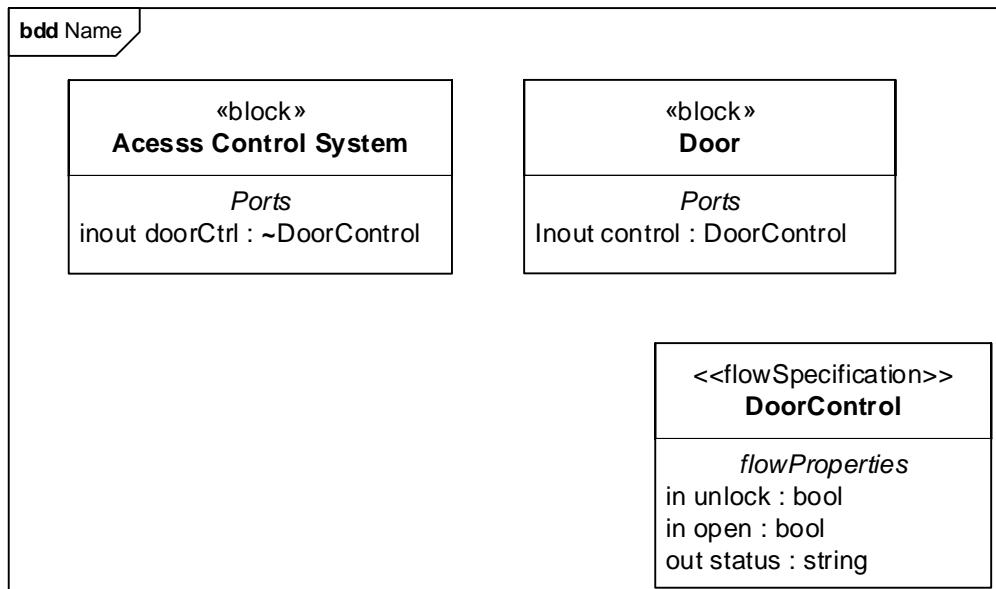


Nonatomic flow ports

- Nonatomic flow ports are used for composite interfaces
 - “Nonatomic” means “composed of several things”
- A nonatomic flow port must be matched by a *flow specification* on a bdd
 - Each component given as a flow property (type and direction)
- You may also use a *conjugate flow port* (see next slide)

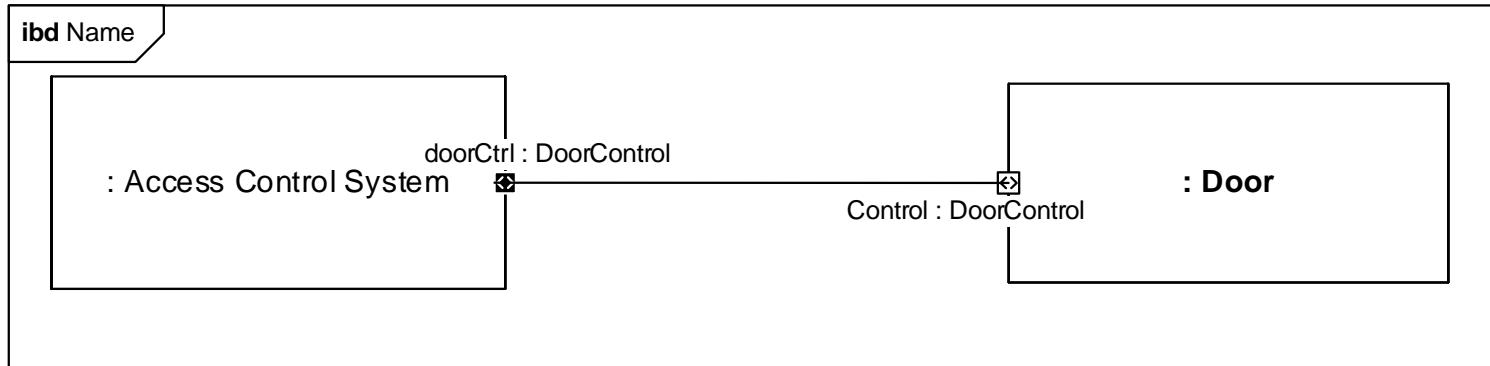
Nonatomic flows on BDDs

- Nonatomic flows are always *inout*
- A conjugated nonatomic flow port is indicated with ~ (tilde)
- For a conjugated flow – in and out are exchanged!



Nonatomic flows on IBDs

- Nonatomic flows are always *inout*, indicated by the double arrow
- A conjugated nonatomic flow port is indicated with the negative double arrow
- The \sim (tilde) is not used, if you use the negative symbol!
- *inout* is never written when using the arrow symbols!

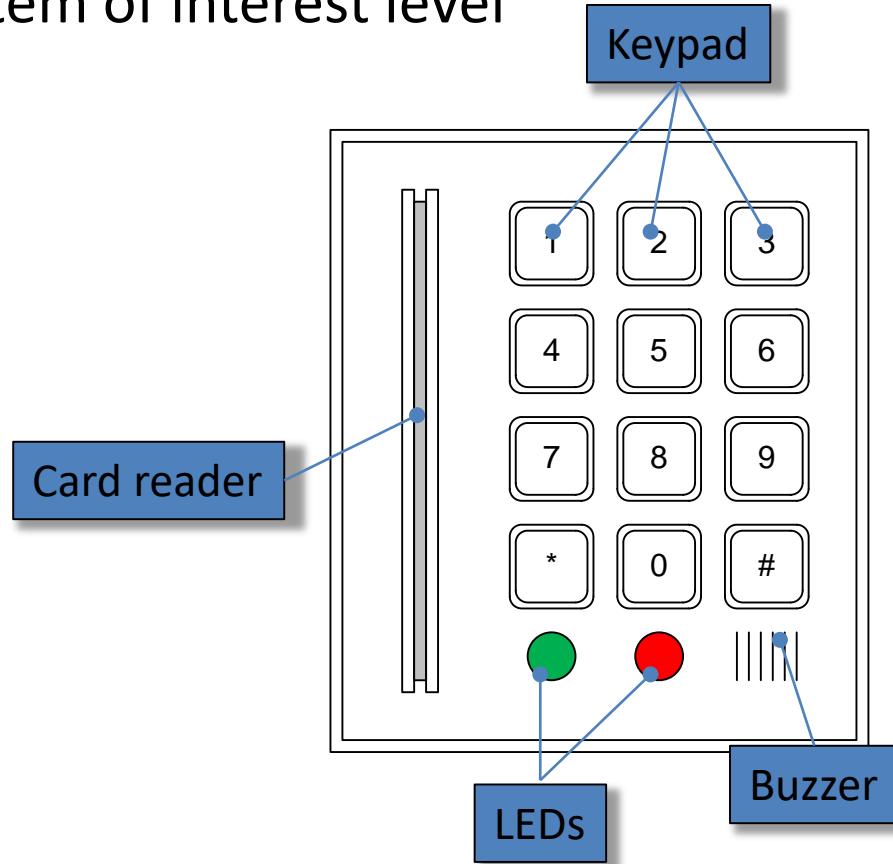


Flow port rules

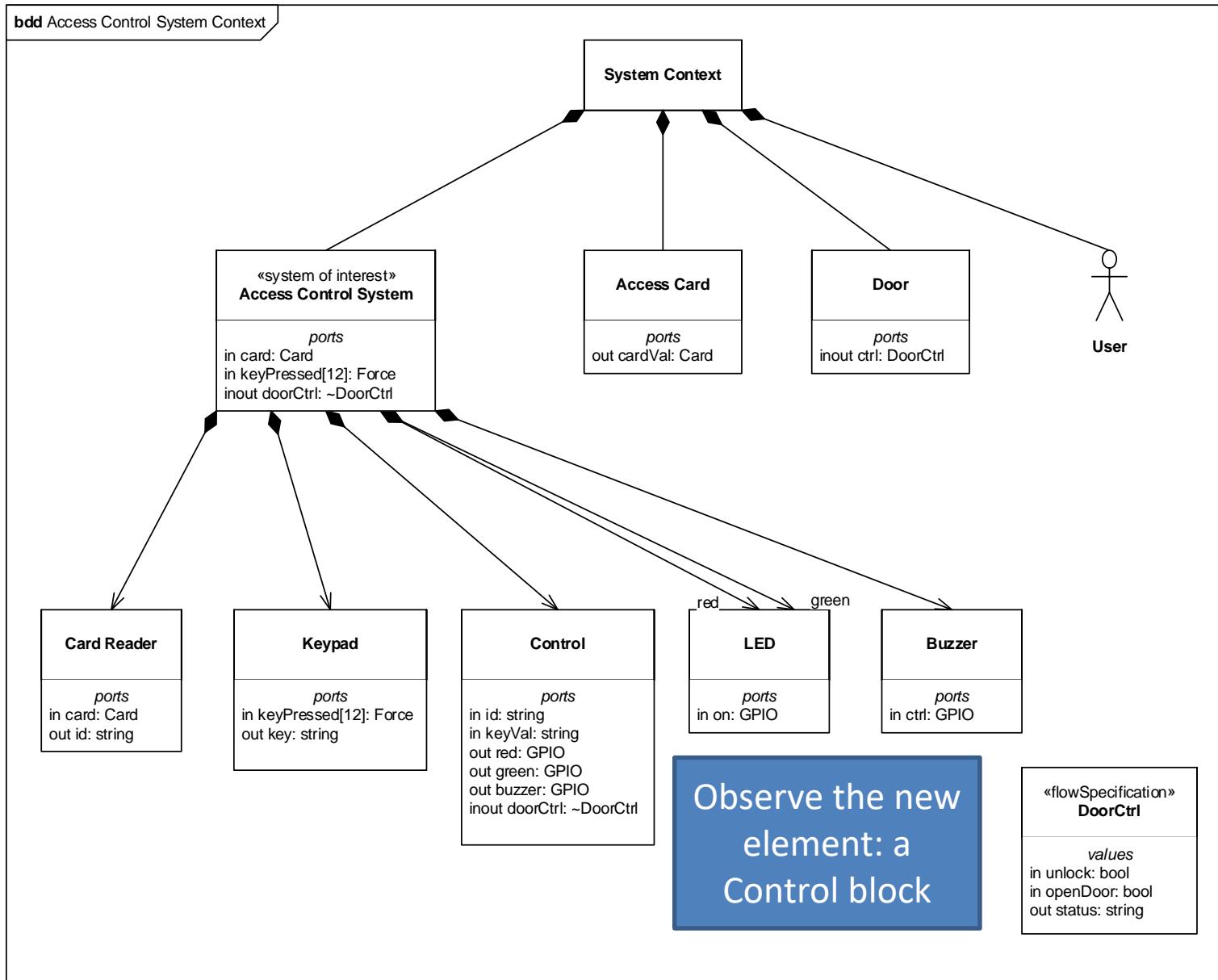
- *in, out, inout* is **NEVER** used, when you are using the arrow port symbols!
- Flow items and flow ports can be used on the same connection, but both are not necessary! The directions must match!
- Be consistent, if you use flow ports: there must be a flow port in each end, and the directions must make sense!
- Flow ports (but not flows) *can* be used on BDDs according to the SysML standard, but we never do it here at ASE!

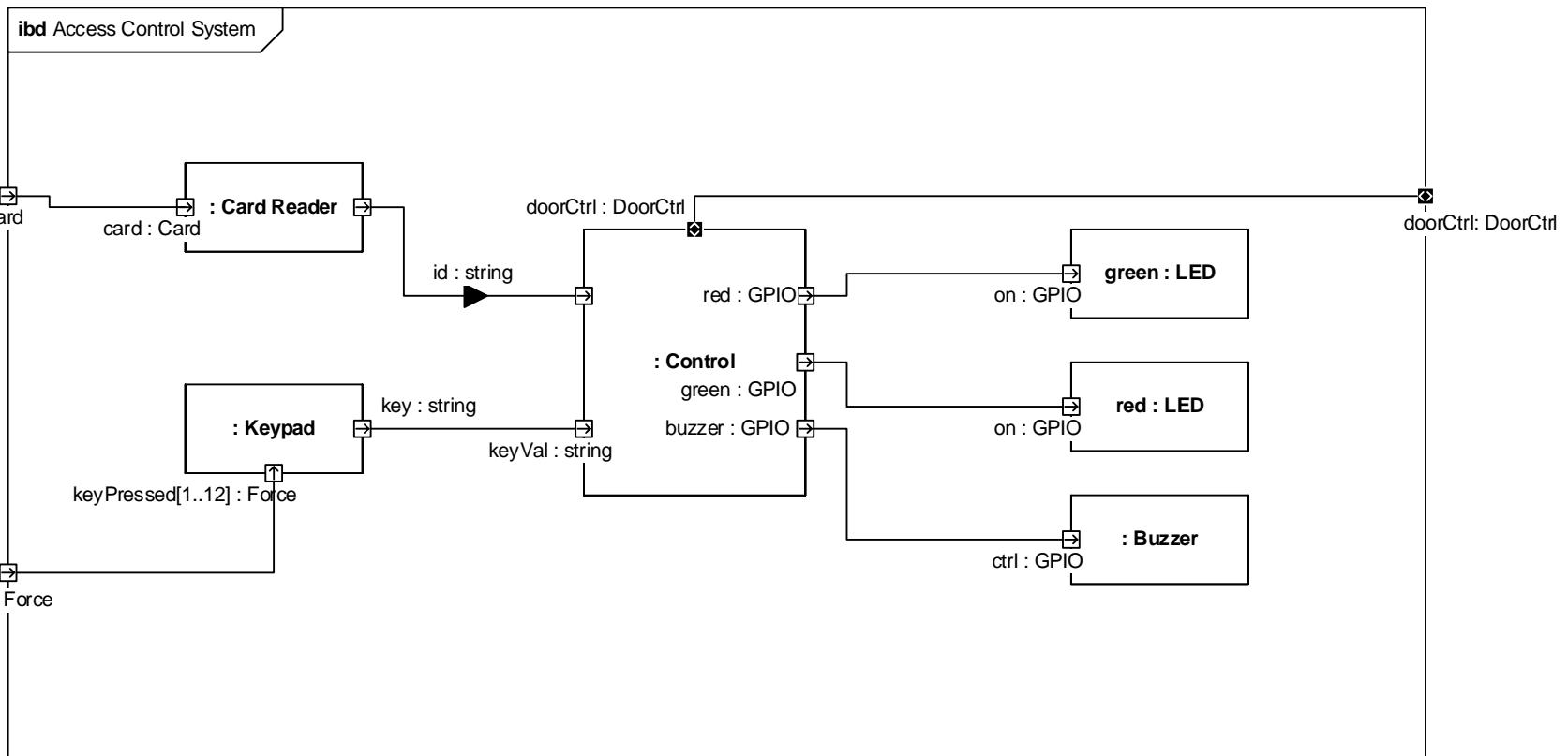
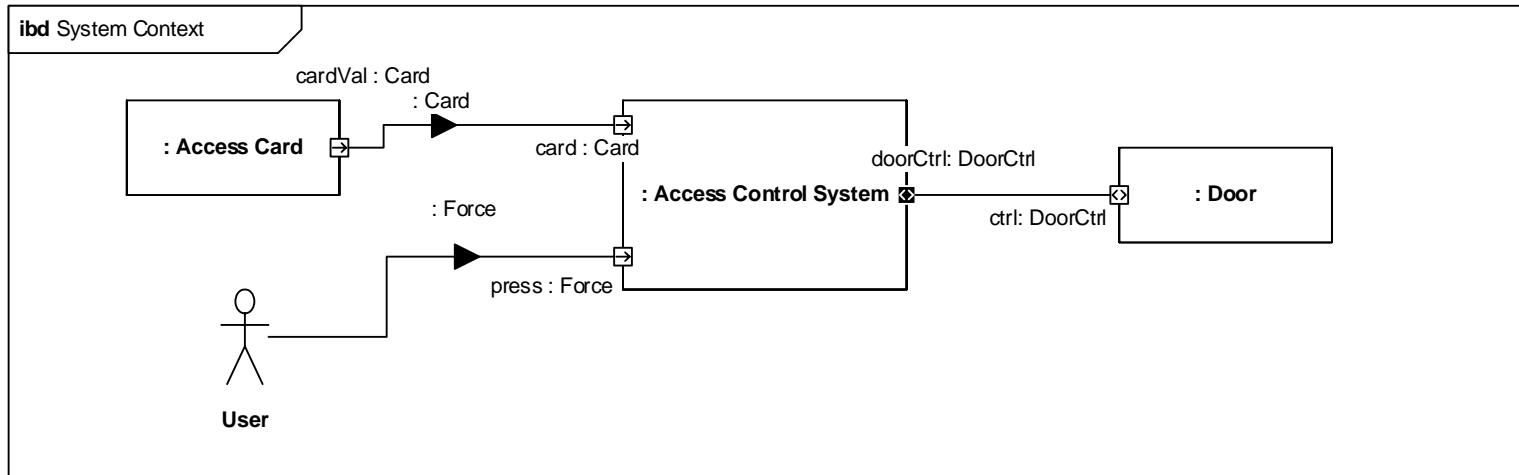
Your turn!

- Given a bdd for an access control system, create 2 ibds incl. ports and item flows
 - At system context level
 - At system of interest level



Your turn!



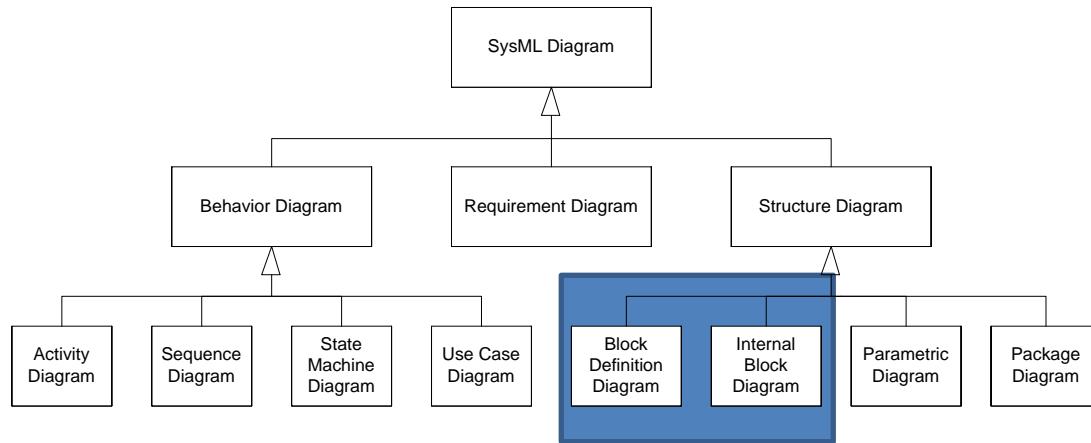


SysML Structural Diagrams 3

Introduction to Systems Engineering
I2ISE

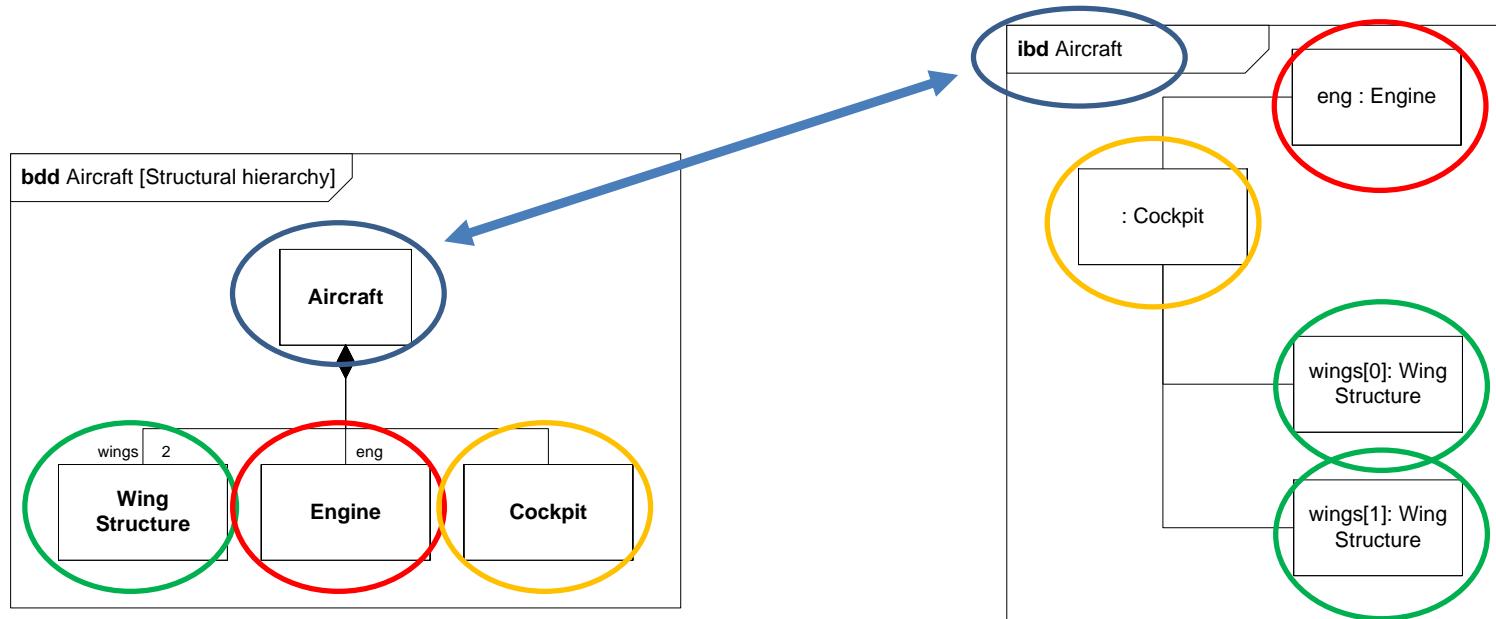
SysML

Block Definition Diagrams & Internal Block Diagrams



SysML: Internal Block Diagram

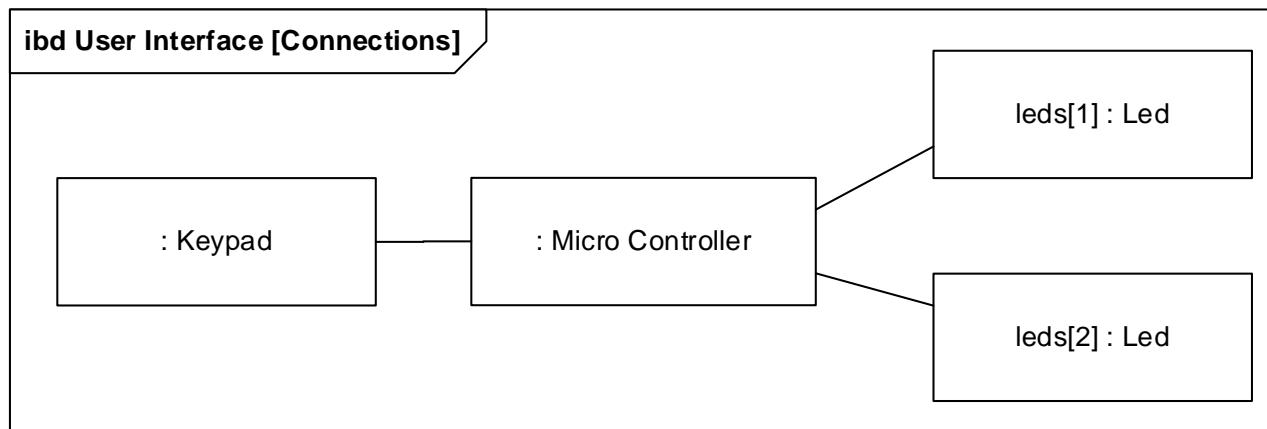
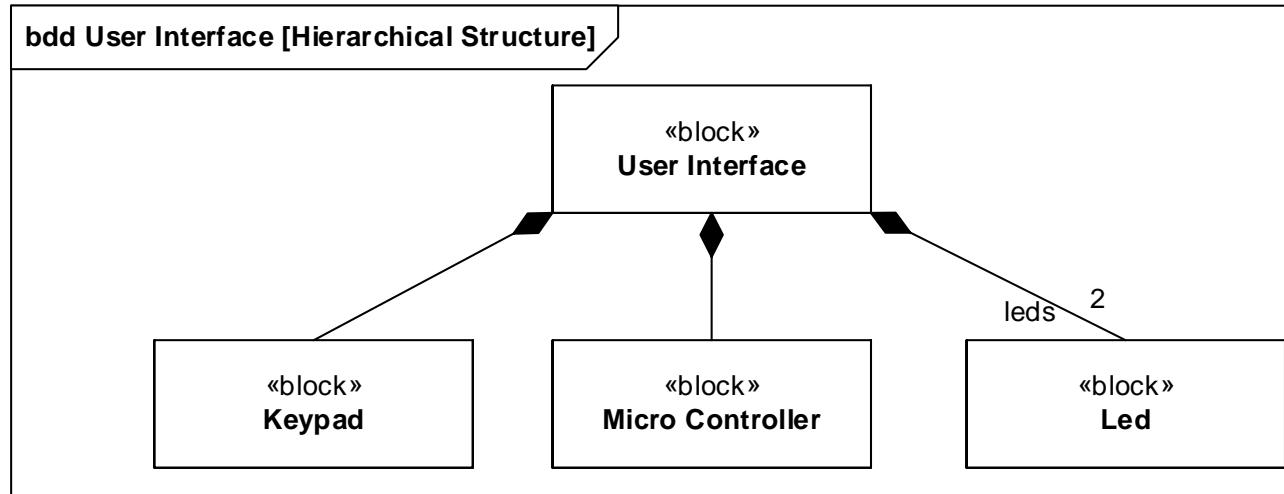
- An *Internal Block Diagram (ibd)* is used to define
 - the *interconnection* and *interfaces* of the parts of a block, and
 - the *information flow* between parts
- An ibd **always** relates to a block on a bdd. It shows the internal connections of the block's constituents



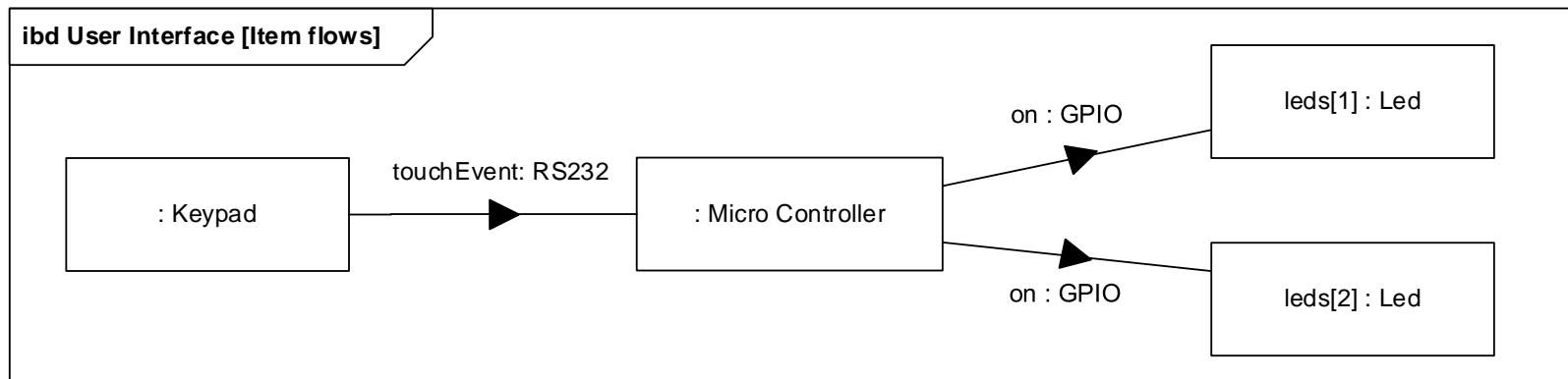
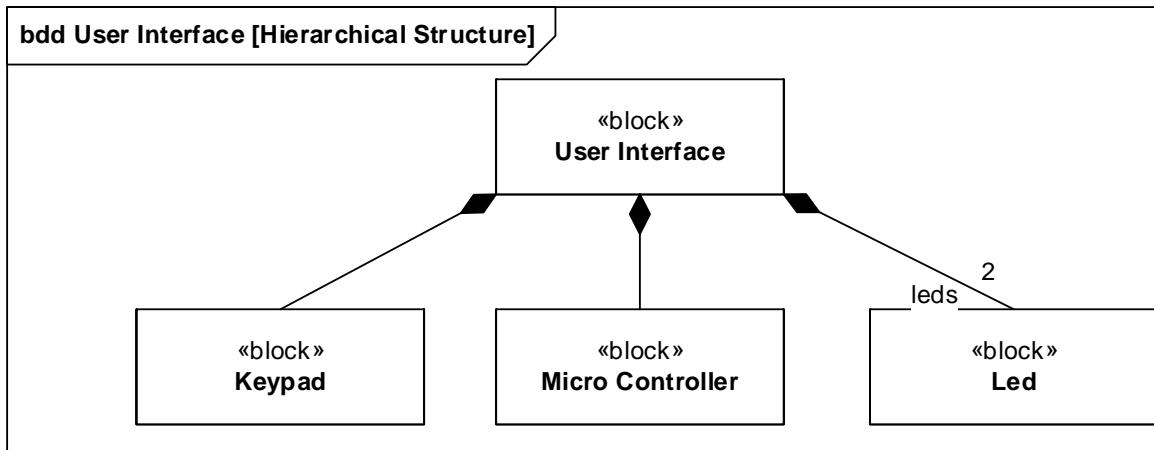
Modeling connections

- We would like to express more about the *connection* between parts on the ibd
 - This would help us to define the *interface* of the parts
- Connections
- Item flows
- Flow Ports
 - Atomic Flow Ports
 - Nonatomic Flow Ports

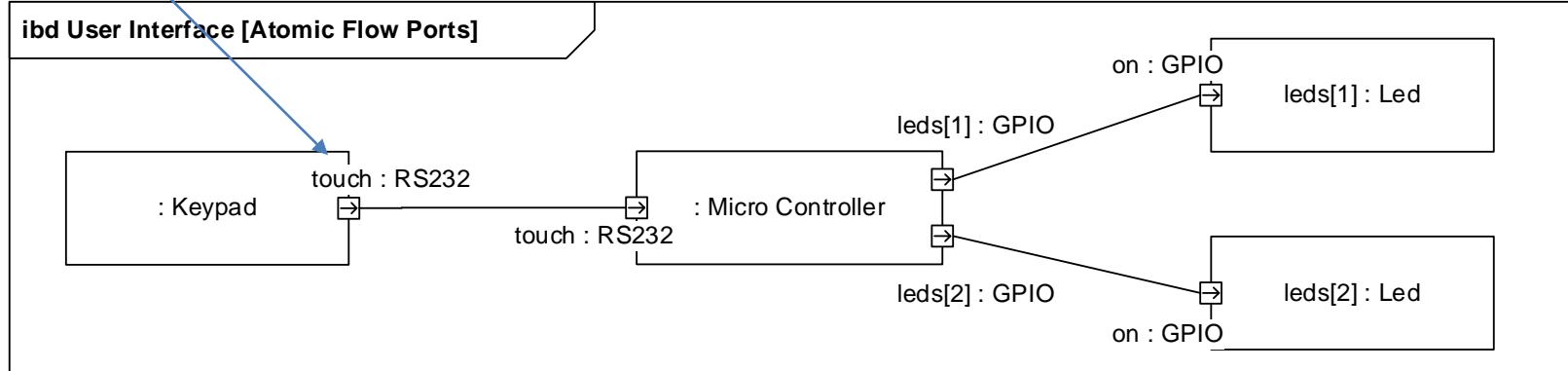
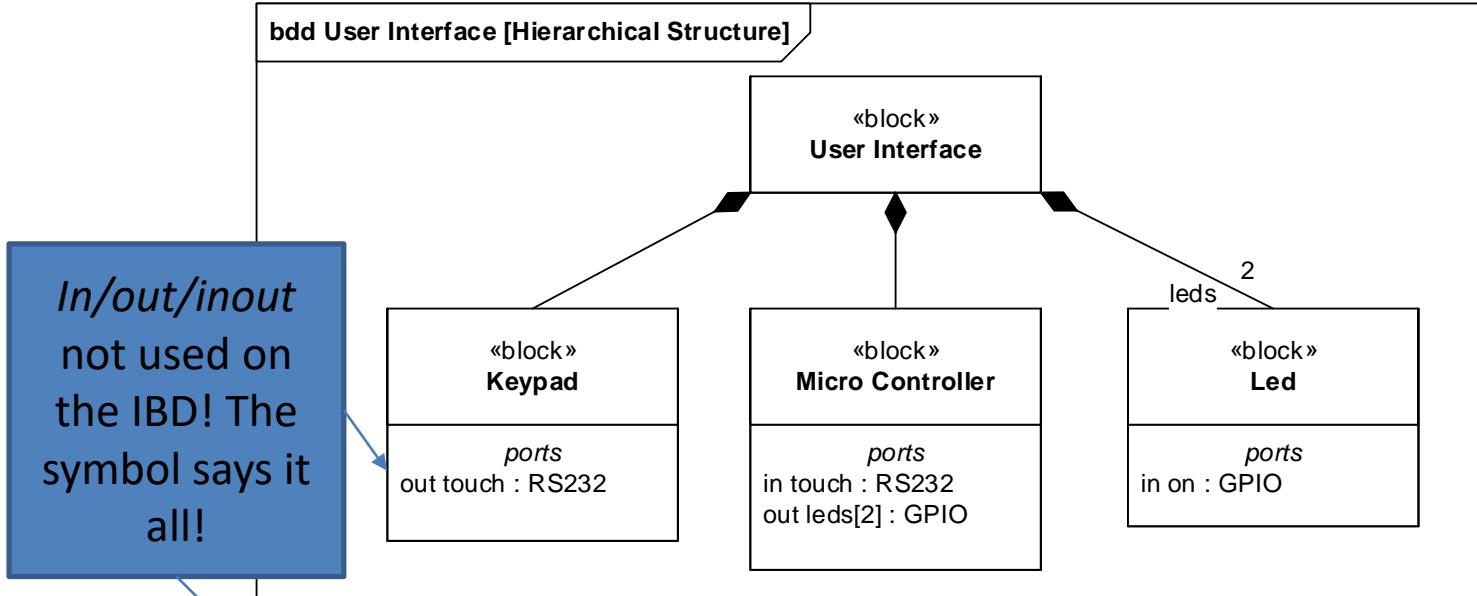
Simple connections



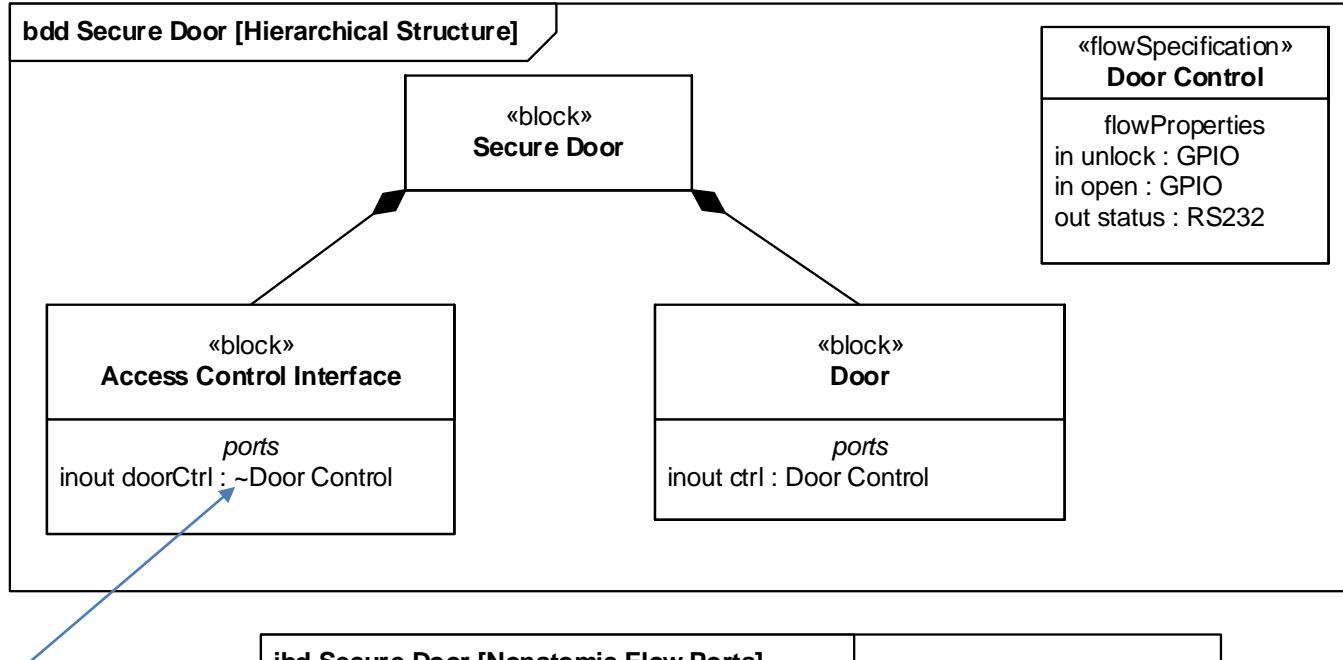
Flow Items



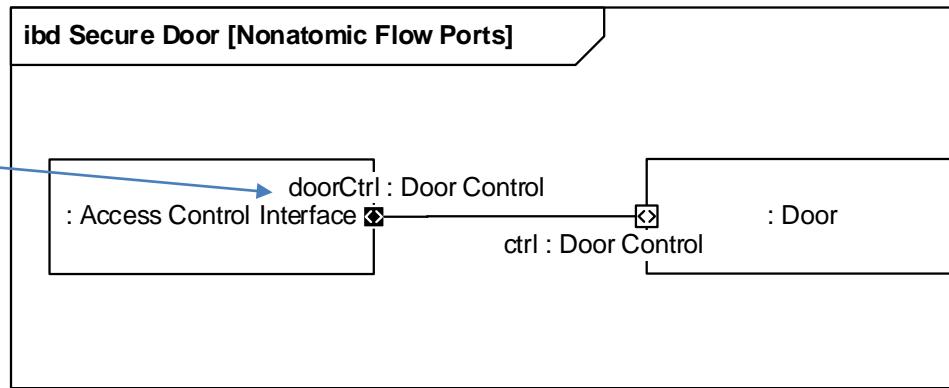
Atomic Flow Ports



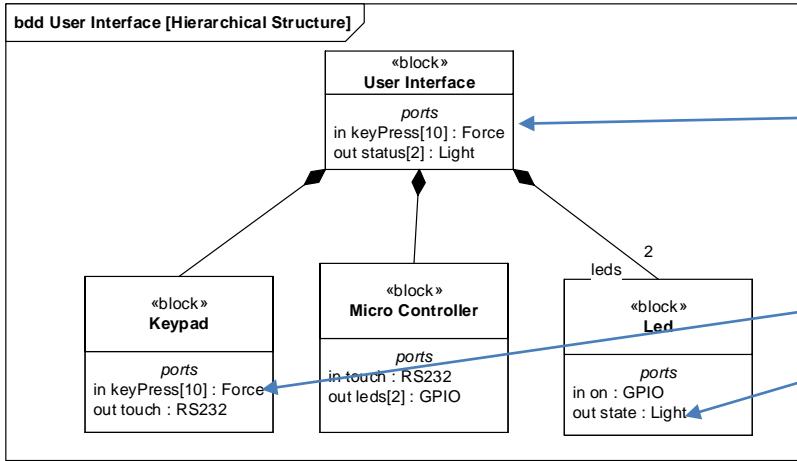
Nonatomic Flow Ports



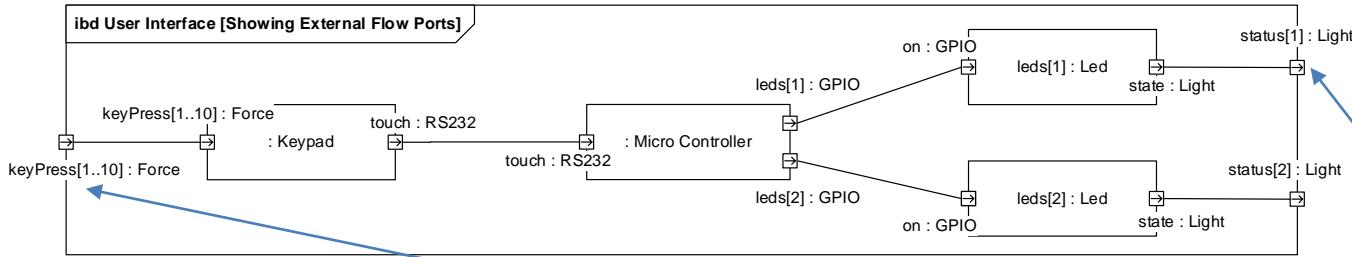
The ~ (tilde) is
not used on
the IBD! The
symbol says it
all!



Ports to the outside

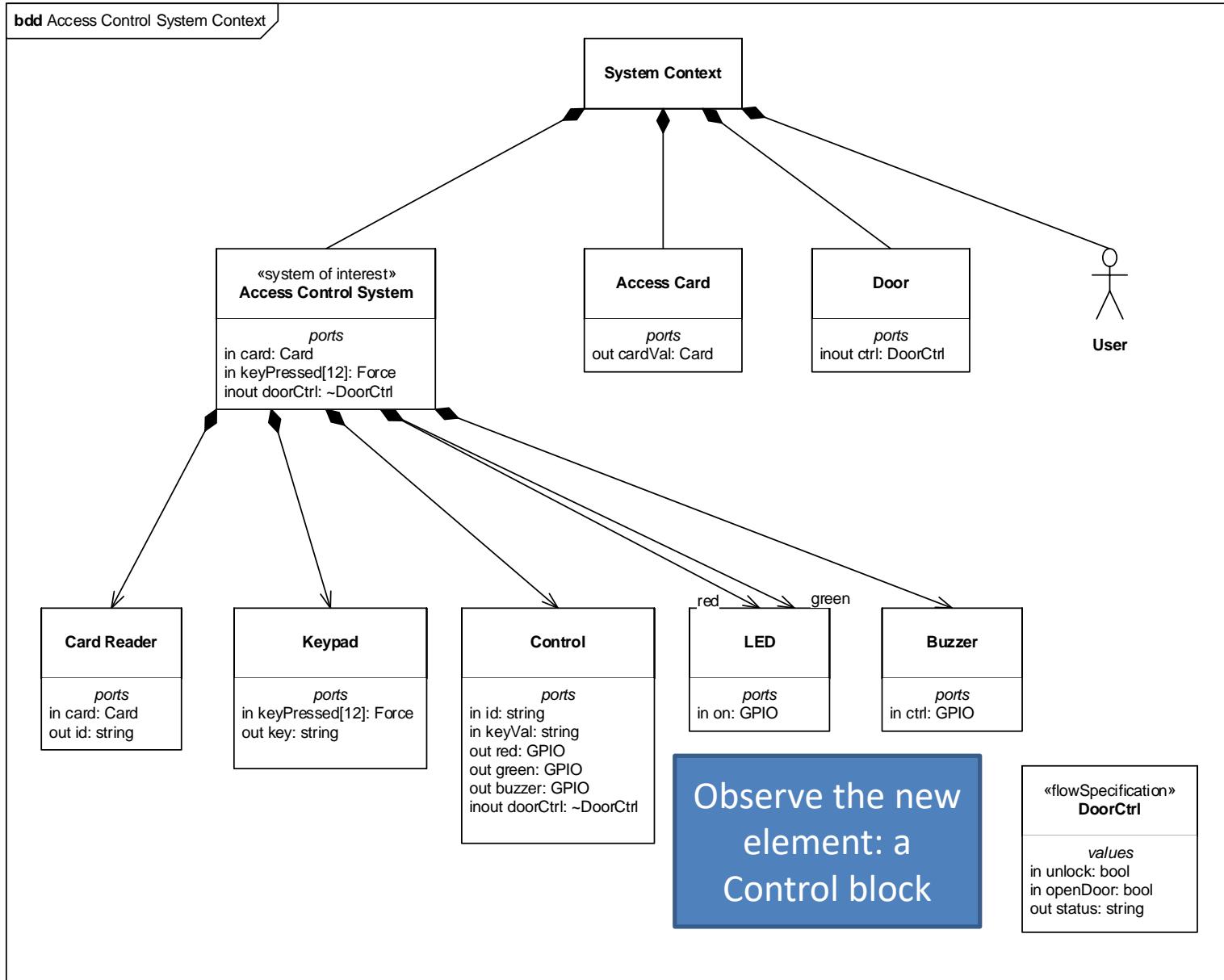


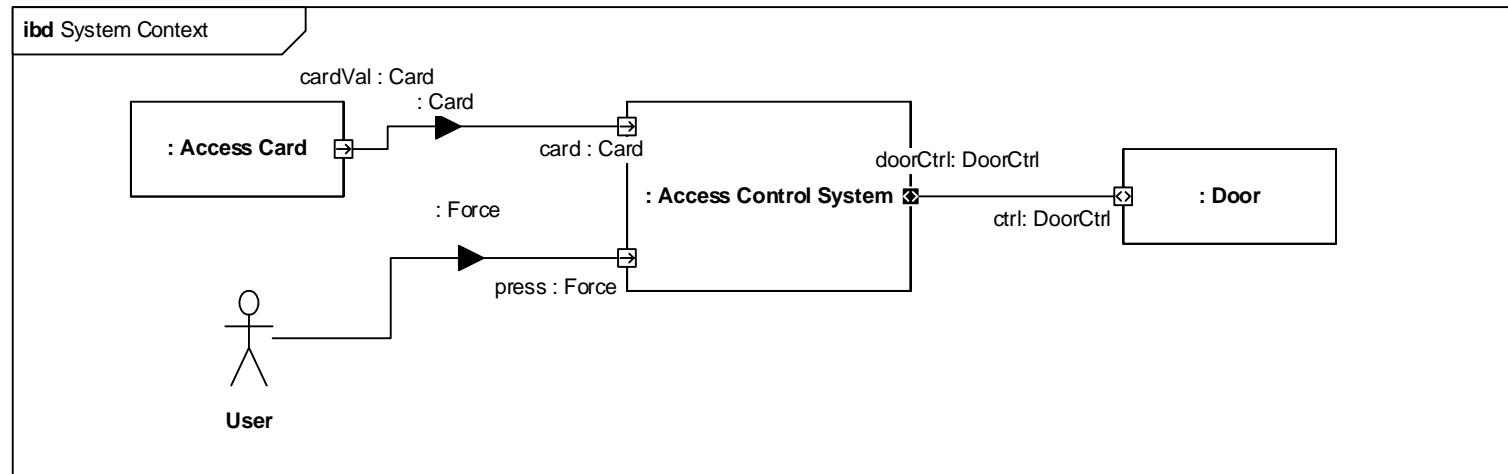
These ports must be implemented by one of the parts on the BDD!



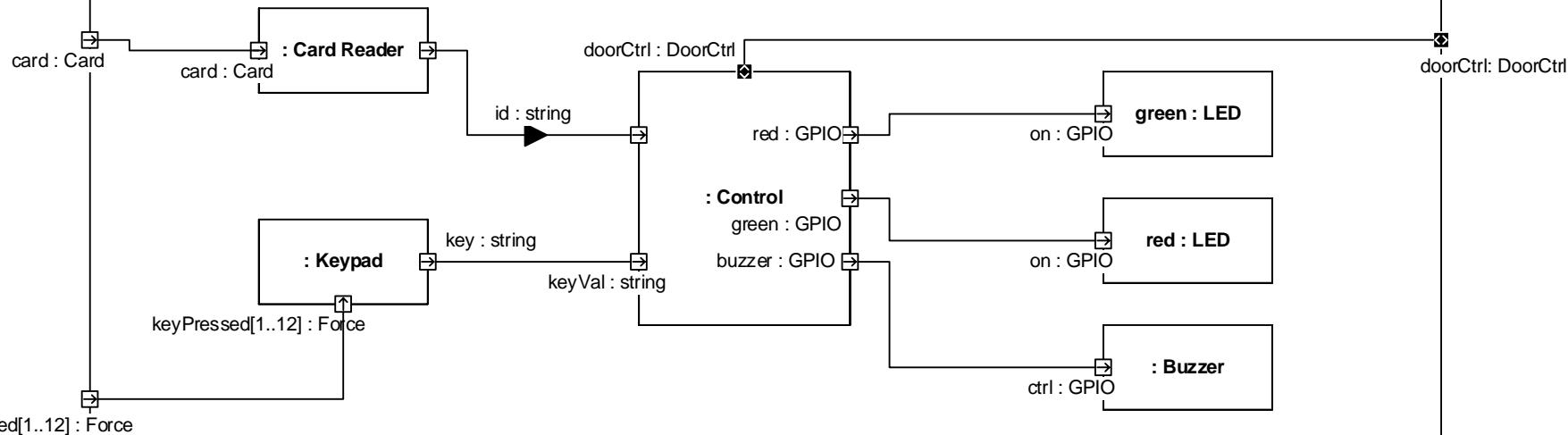
They are shown on the IBD as connected to the outside

Exercise from last – solutions on next slide



ibd System Context**ibd Access Control System**

Observe how the conjugated flow port is repeated, when going to the outside!



Today's exercises

- The solutions to last lecture's exercises
 - BDD for Parkeringsautomat
 - BDD for Smart Fridge
- Can be found on Blackboard
- Use them as input to create IBD's for
 - The User Interface block for Parkeringsautomat
 - The complete Smart Fridge

SysML Behavioural Diagrams

Sequence Diagrams

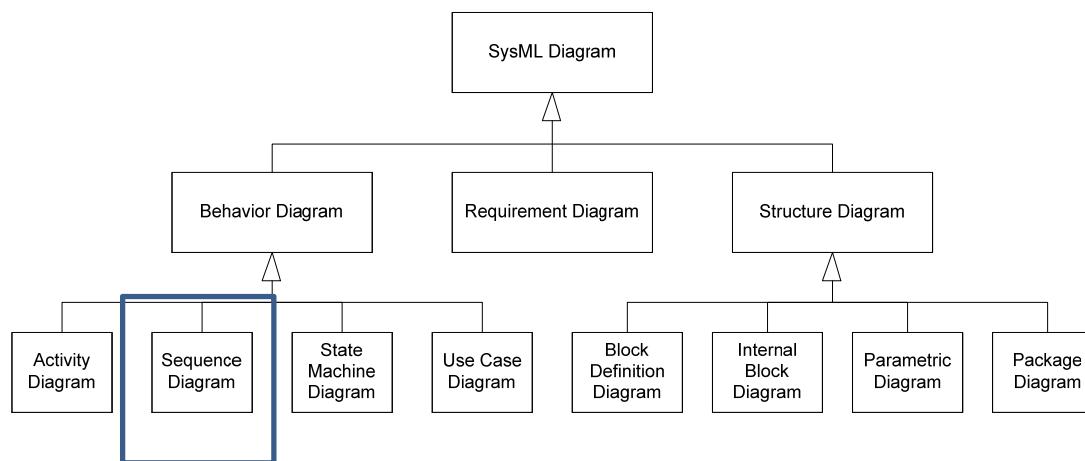
Introduction to Systems Engineering
I2ISE

SysML Structure vs. behaviour

- We have learned a lot about how to model *structure* in SysML
 - Block Definition Diagrams
 - Internal Block Diagrams
- Now, we will look at how we can model *behaviour* in SysML
 - Sequence diagrams
 - State Machines

Sequence diagrams

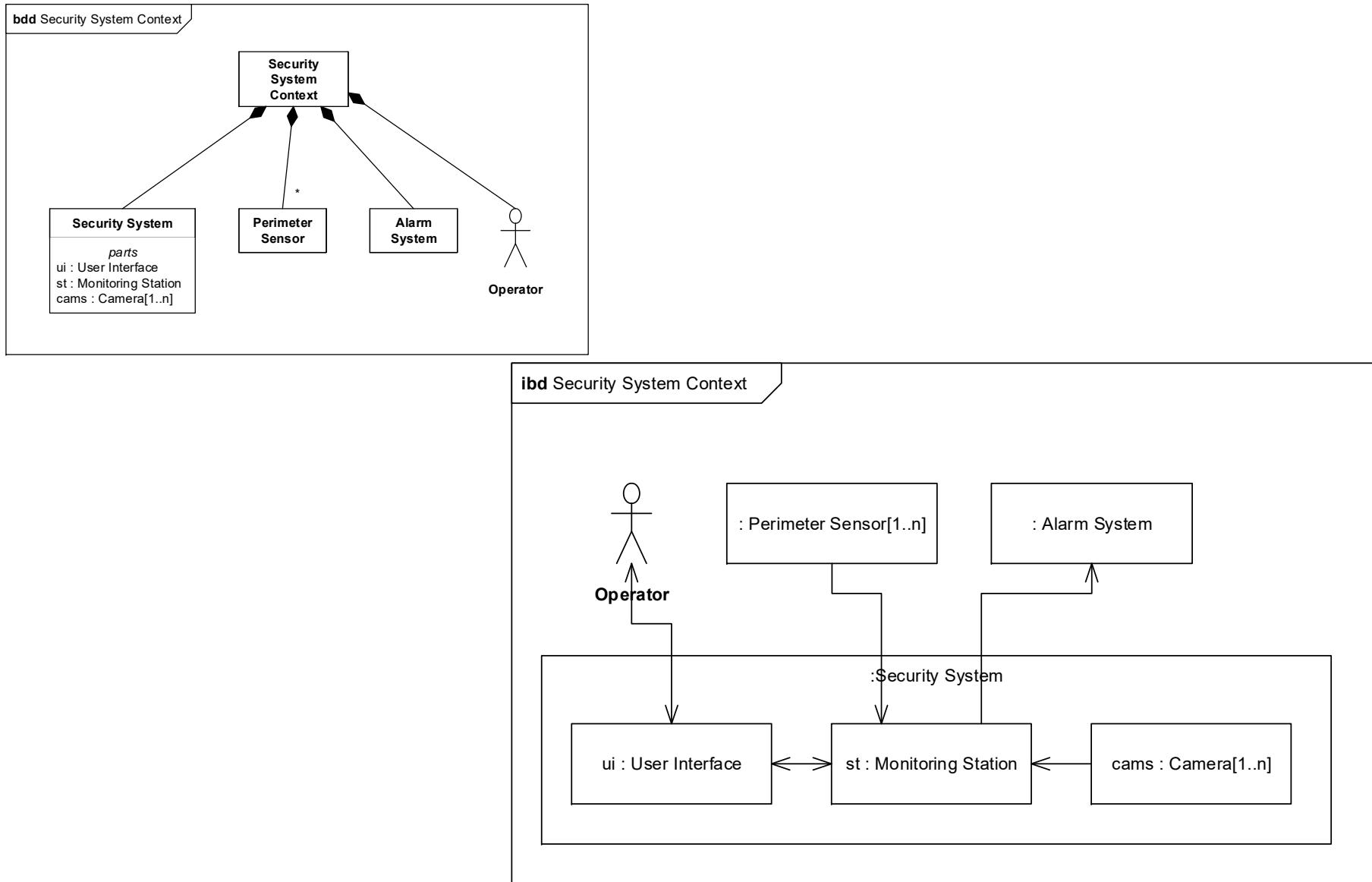
- Sequence diagrams (diagram type *sd*) model interactions between *parts* of a block



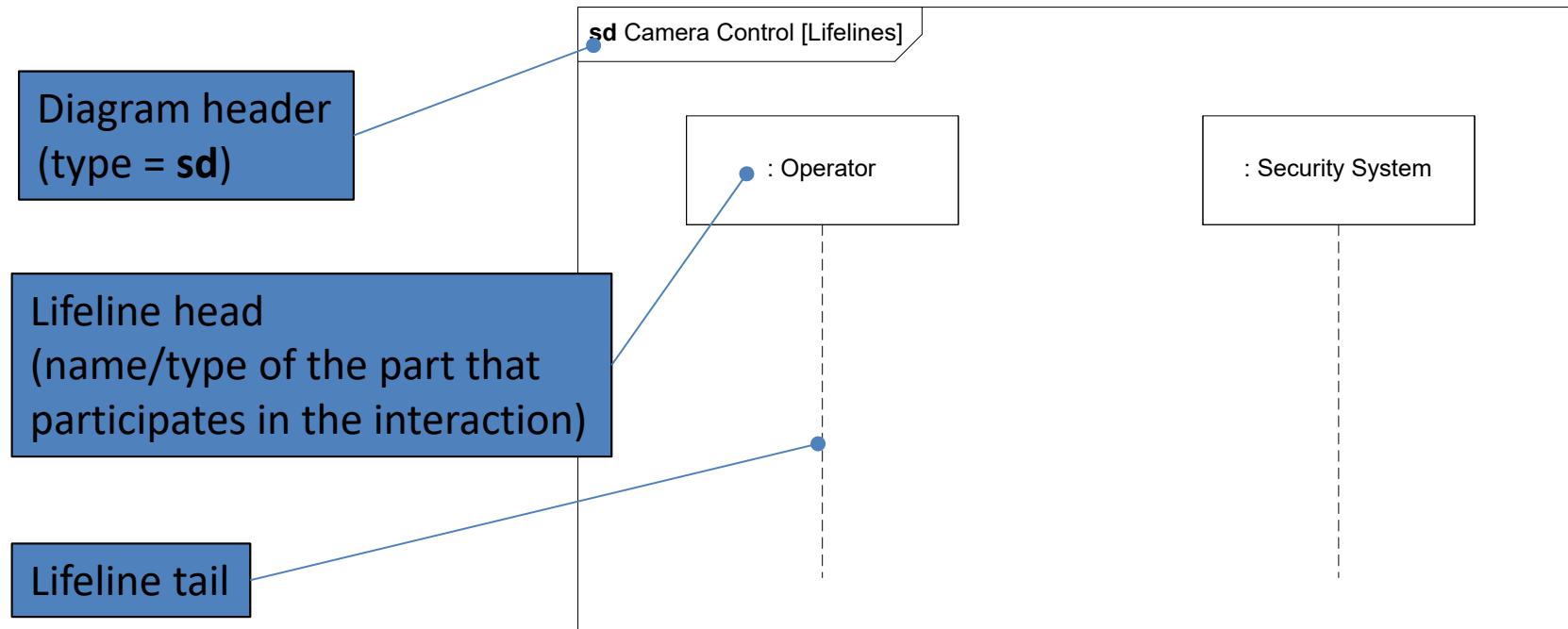
Sequence diagrams

- Sequence diagrams are used to model *message-based* behaviour
- The interactions take place within a block between its elements of internal structure (parts)
- The basic diagram consists of *lifelines* with *messages* between them.

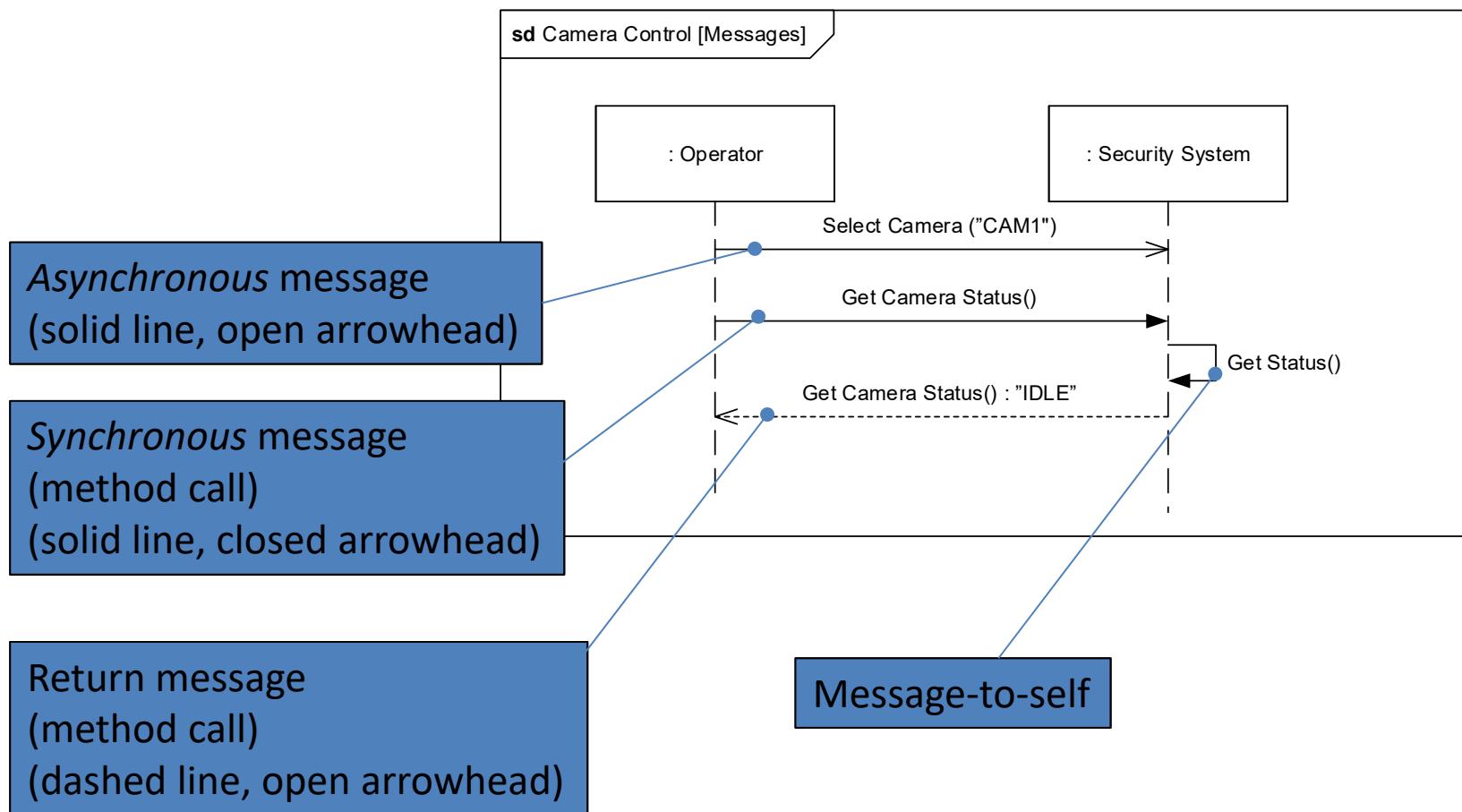
SD's – example system (structure)



SD's – lifelines



SD's – messages



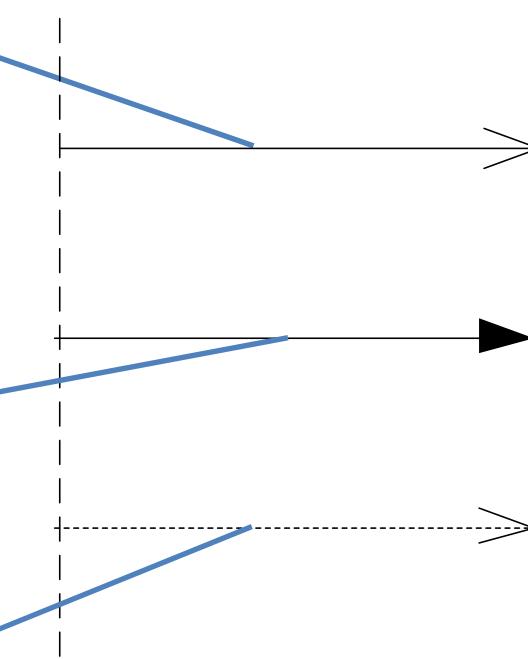
SD's – async/sync/reply

There are two basic types of messages: **asynchronous** and **synchronous**. A sender of an asynchronous message continues to execute immediately after sending the message, whereas a sender of a synchronous message waits until it receives a reply from the receiver.

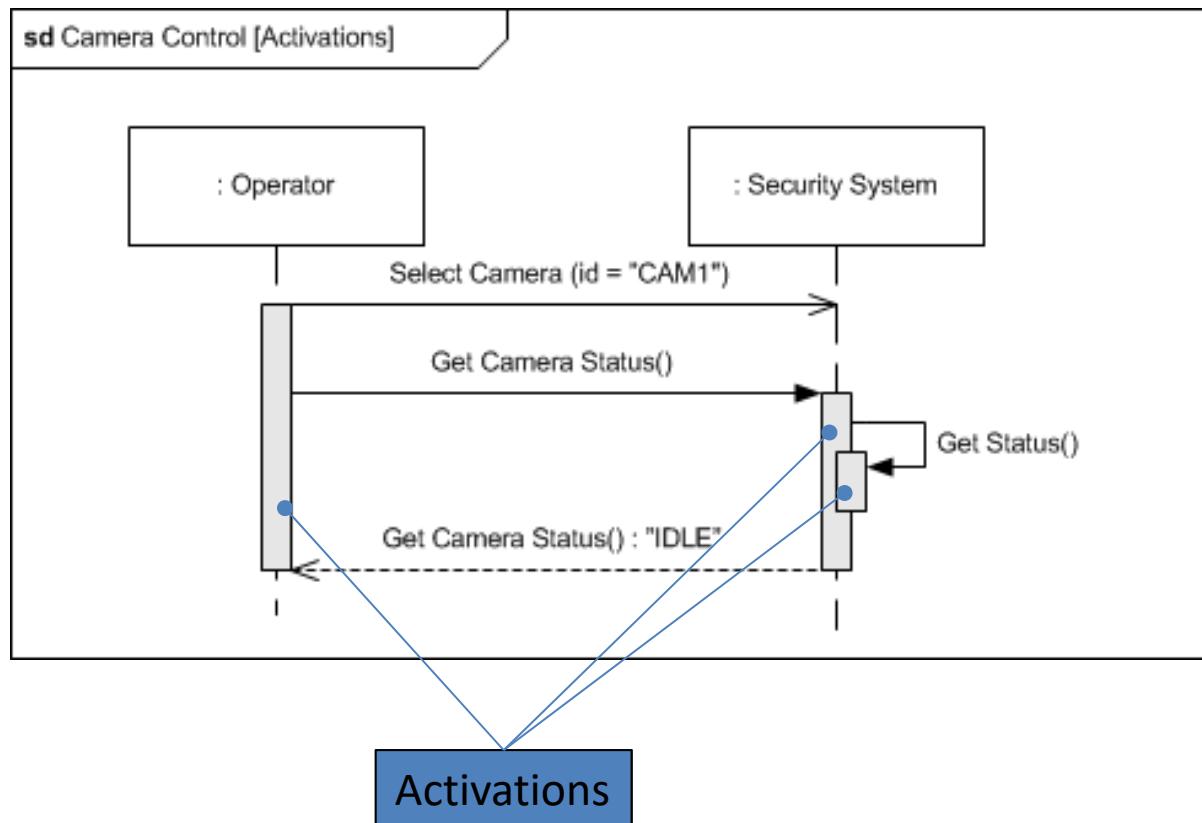
An open arrowhead means an **asynchronous message**. Input arguments associated with the message are shown in parentheses as a comma-separated list after the message name.

A closed arrowhead means a **synchronous message**. The notation is the same as for asynchronous messages.

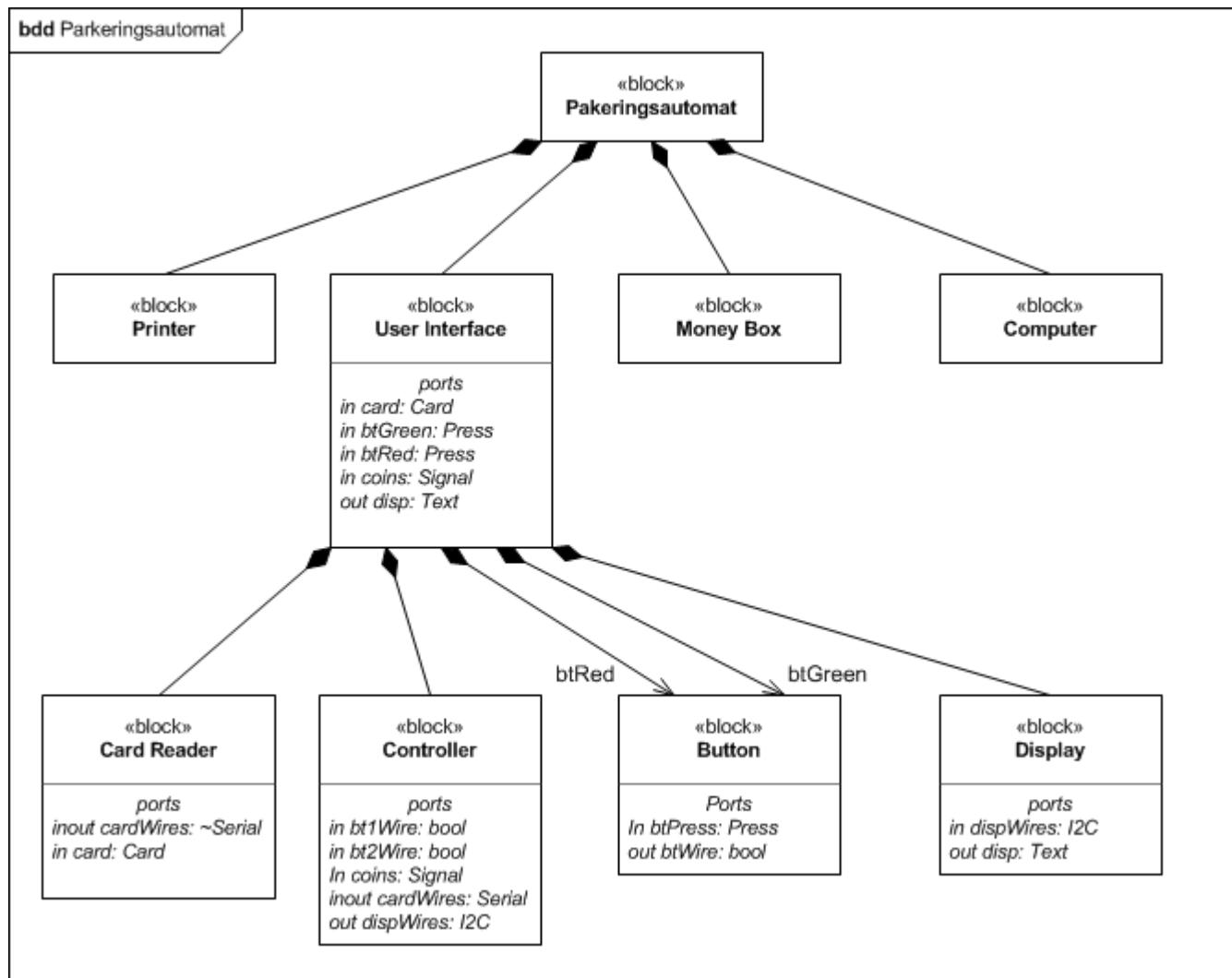
An open arrowhead on a dashed line shows a **reply message**. Output arguments associated with the message are shown in parentheses after the message name, and the return value, if any, is shown after the argument list.



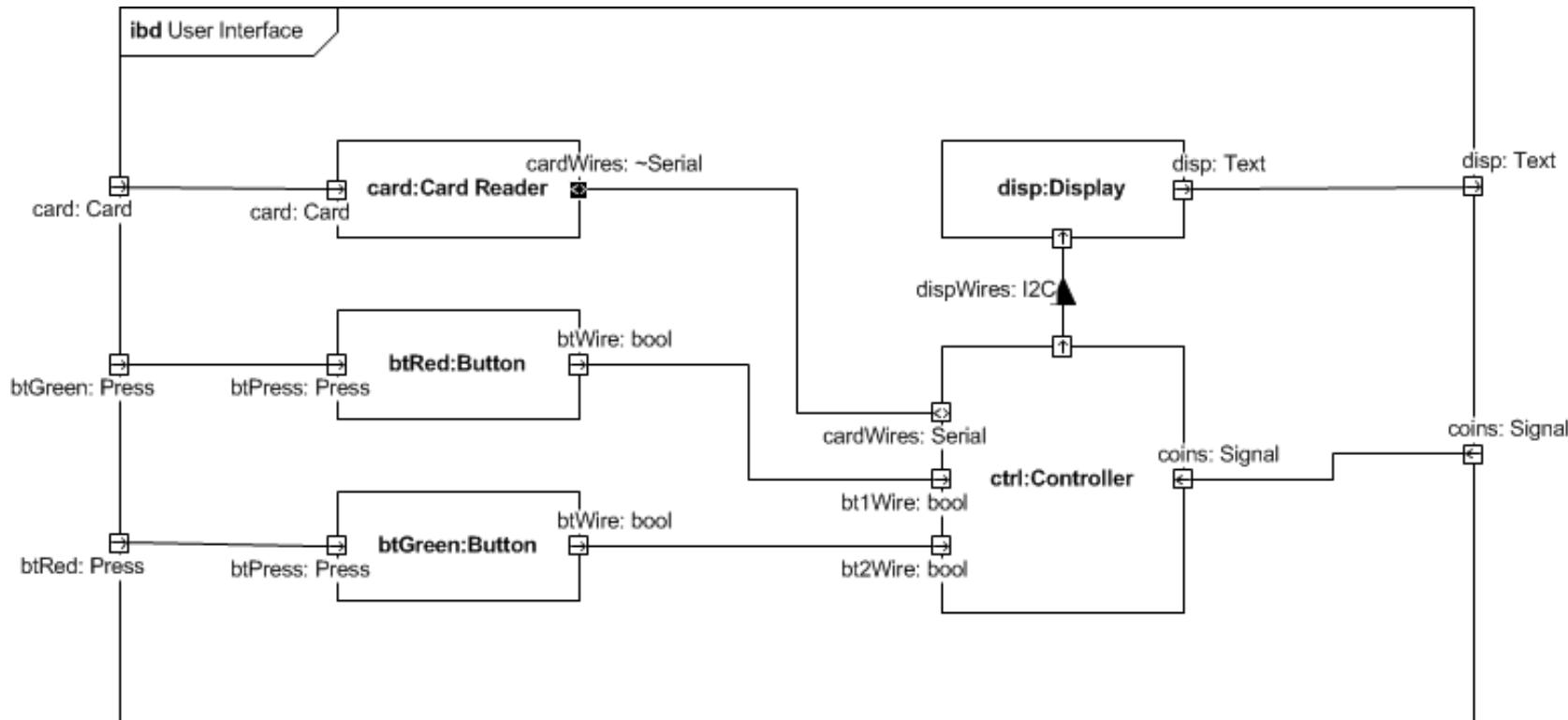
SD's – activations



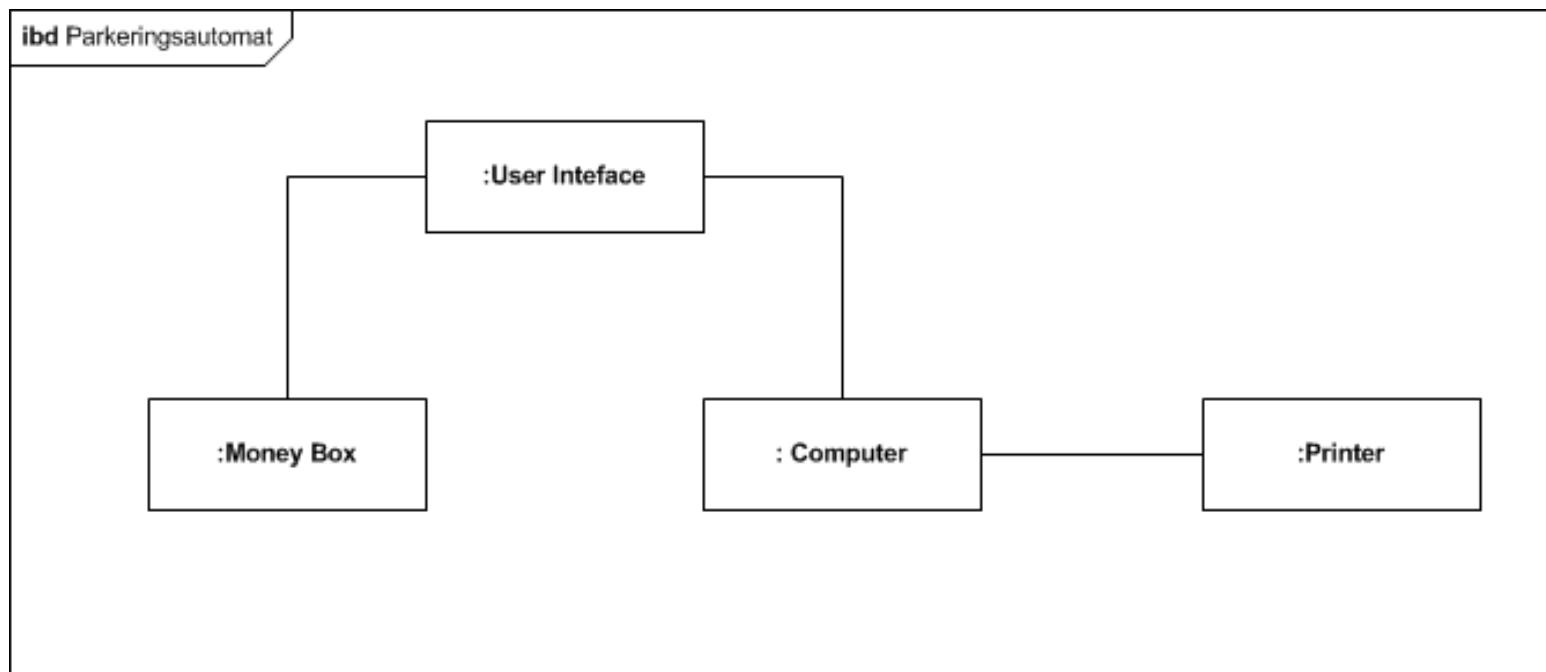
bdd Parkeringsautomat



ibd User Interface



ibd Parkeringsautomat



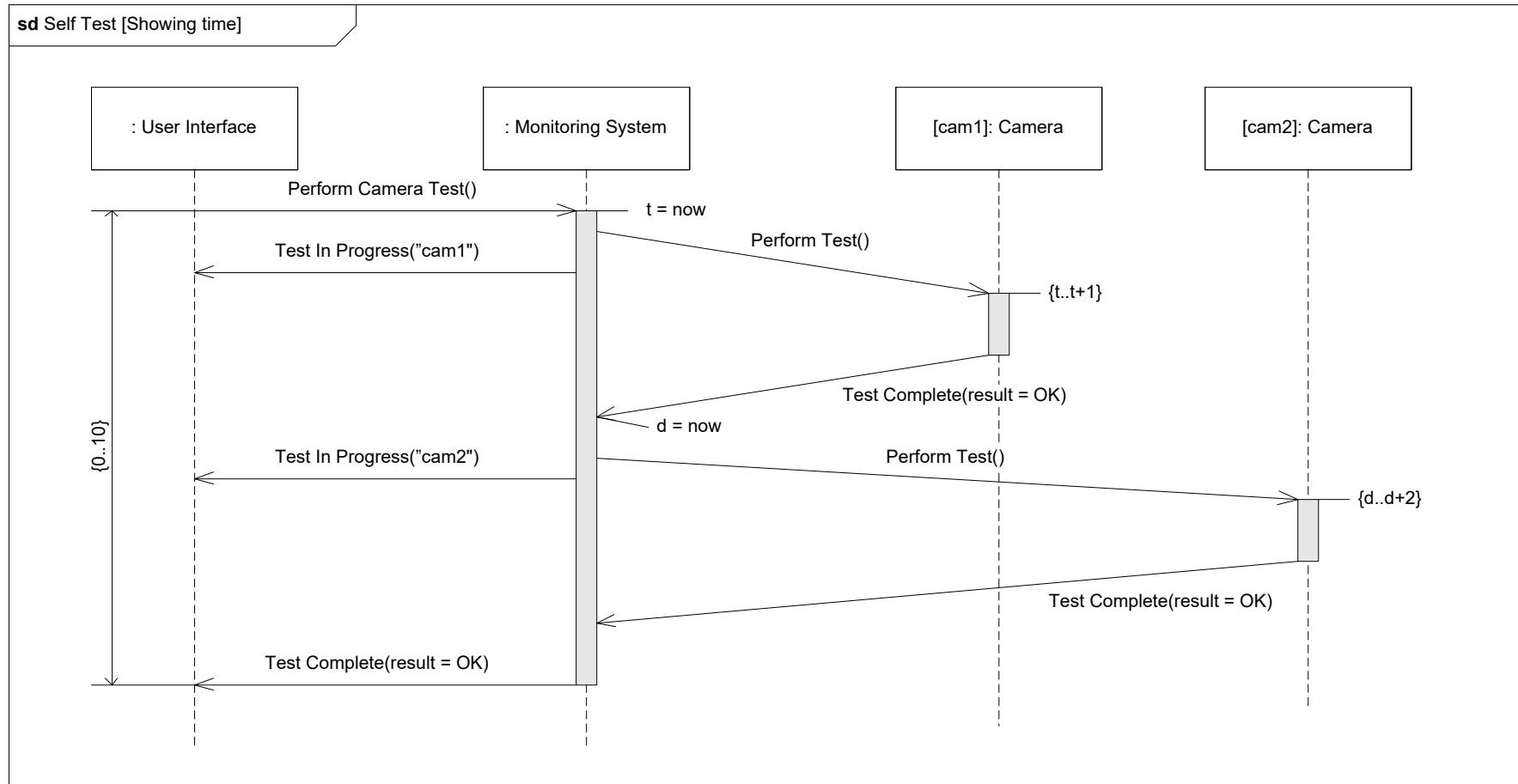
SD's – your turn – Parking Machine

- Draw a sequence diagram for buy ticket
- Use actor and blocks:
User Interface, Computer, Money Box and Printer

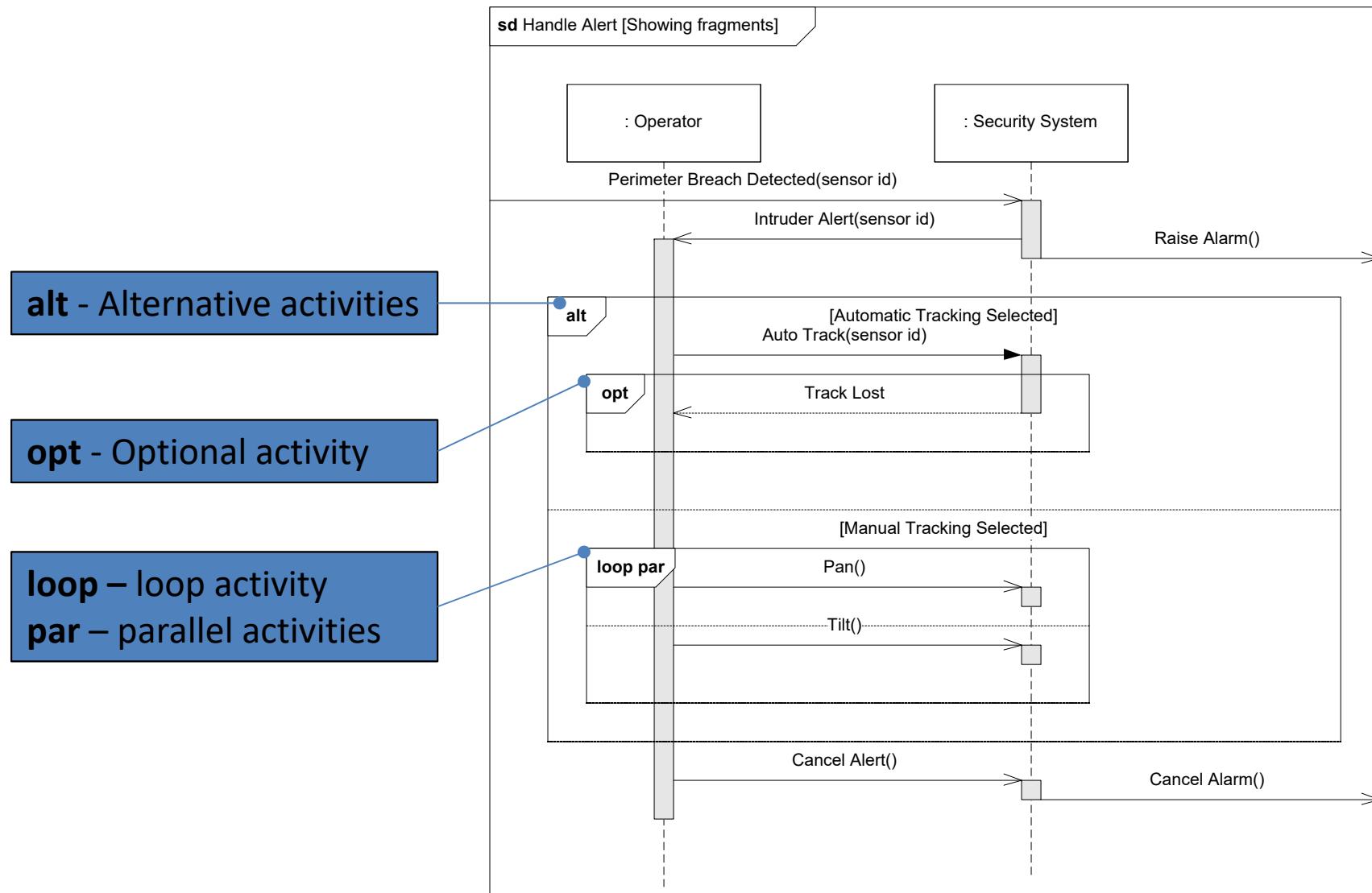
UC: Buy Ticket - Main scenario

1. User inserts coins in the Parking Machine
2. Parking Machine displays total amount and time
3. User press the pay button (green)
4. Parking Machine prints a ticket

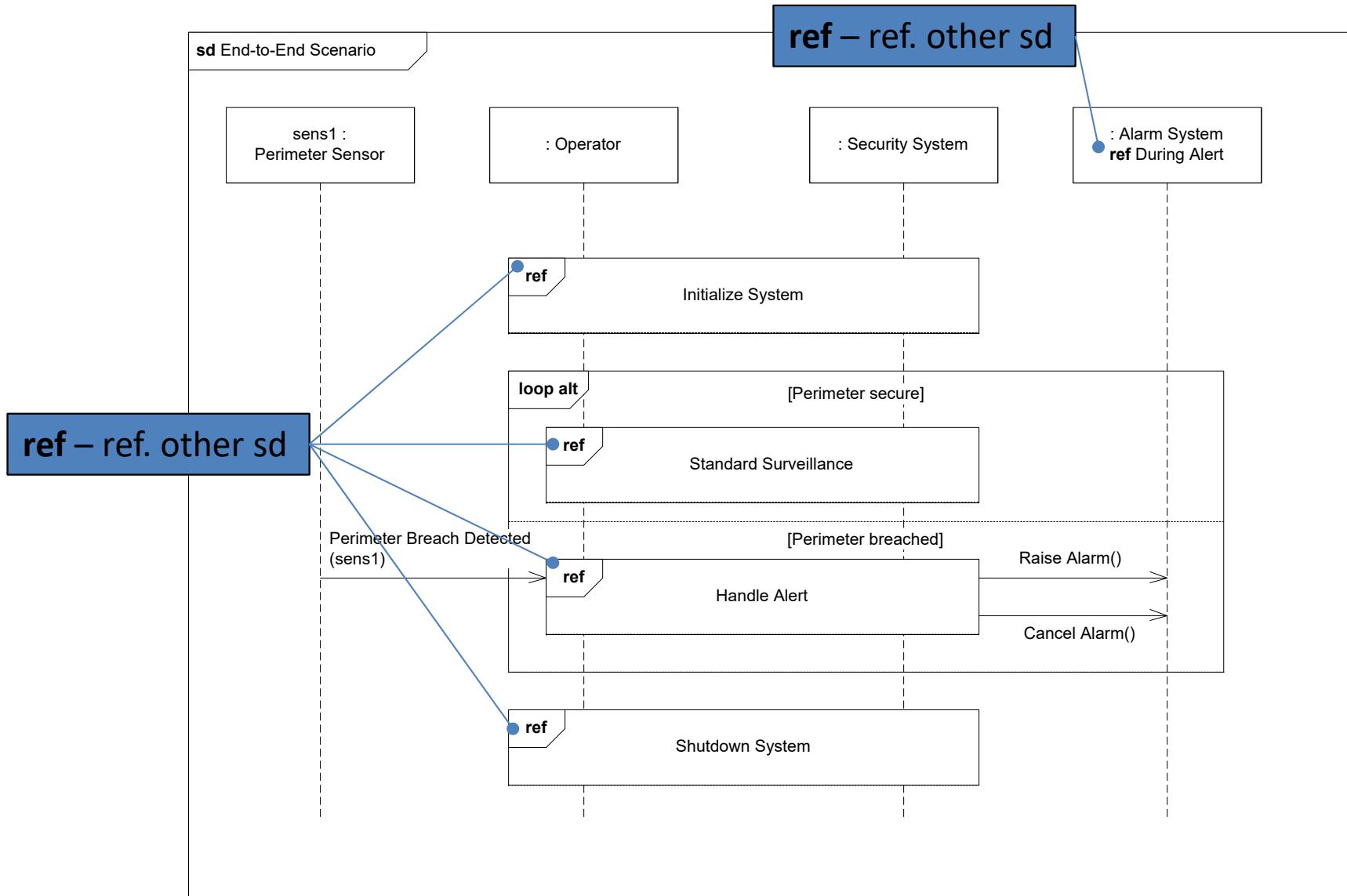
SD's – representing time



SD's – fragments



SD's – reference blocks



SD's – your turn!

- Create a sequence diagram for the RVM scenario *Recycle Containers* below
 - Participants: *User* and *RVM*
- Add operations to the RVM on a BDD

Main Scenario for Use Case *Recycle containers*

1. User arrives at RVM and is informed to insert containers.
2. User places container in the in-feed.
3. RVM scans container and *either*
 - a) accepts the container, collects the container from the in-feed, adds the return deposit to the collected amount, and displays the type and value of the accepted container and the total collected amount; *or*
 - b) does not accept the container, rejects the container to User, and displays that the container is not accepted and the total collected amount.

Step 2 through 3 is repeated until User is done feeding containers.

1. User request the return deposit receipt.
2. RVM prints out the return deposit receipt, and resets the collected amount.

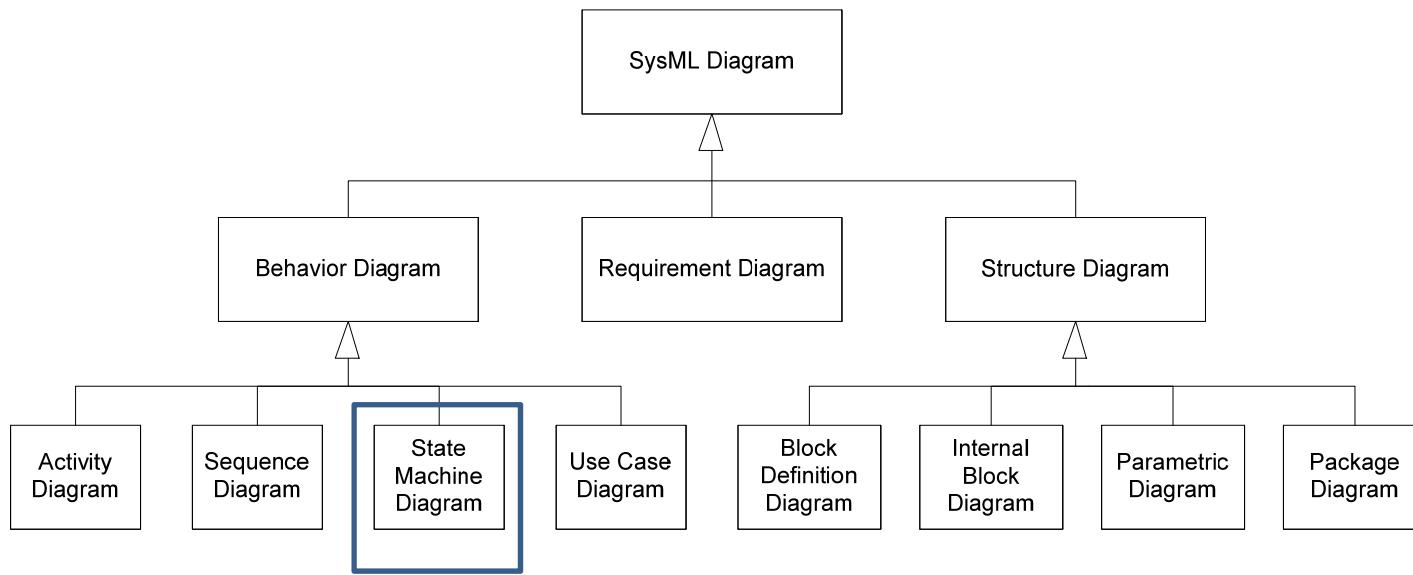
SysML Behavioural Diagrams

State Machine Diagrams

Introduction to Systems Engineering

I2ISE

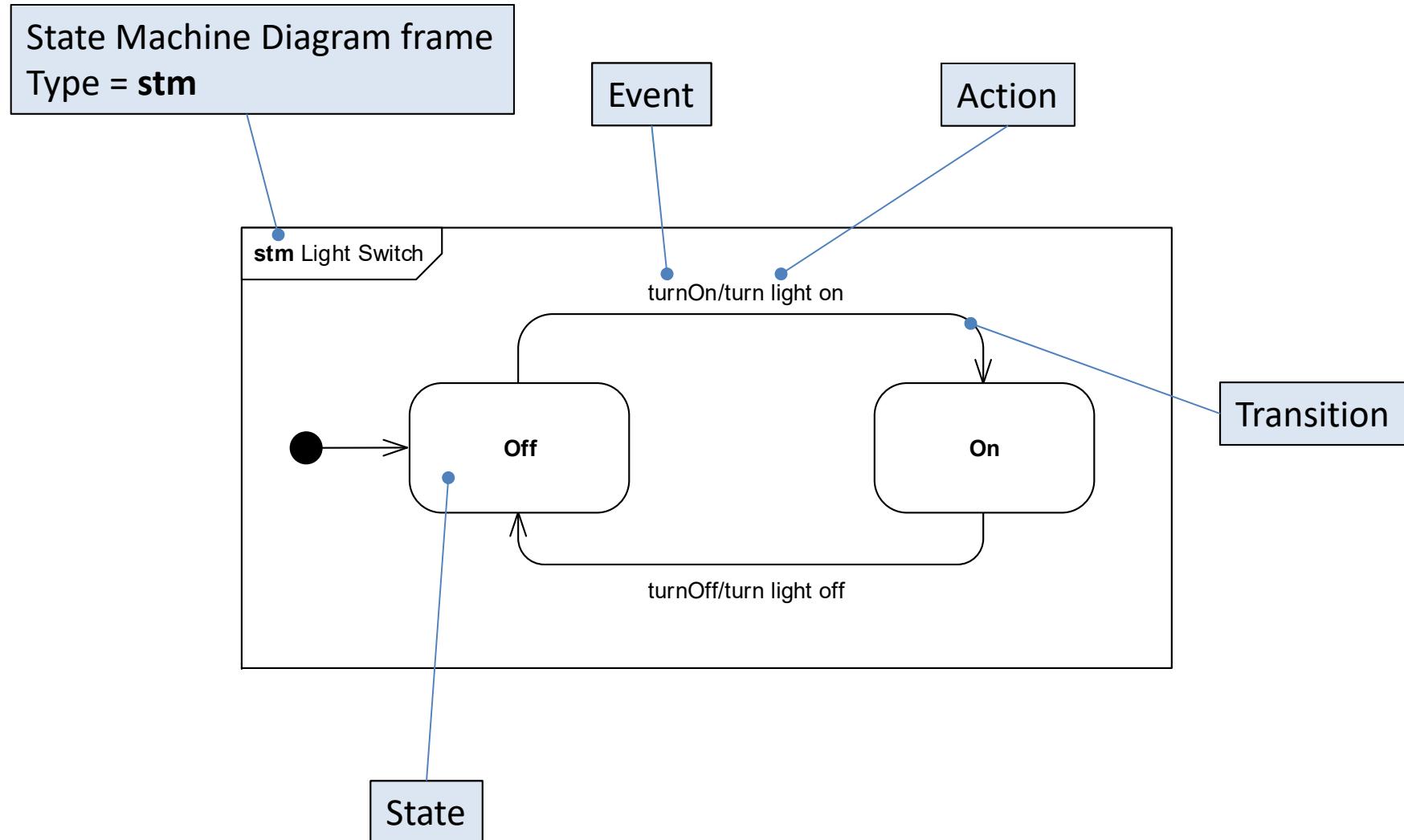
Introduction



State Machines

- State Machines Diagrams (**stm**), aka *state charts*, are used to model *state-dependent* behaviour of a block throughout its lifecycle
- A *state* is some significant condition in the life of a block
 - Typically, different states respond differently to same events
- A *state machine* is always in a certain *state* and will remain there until some *event* causes it to *transition* to another state.
- Any examples?

States and transitions - basics



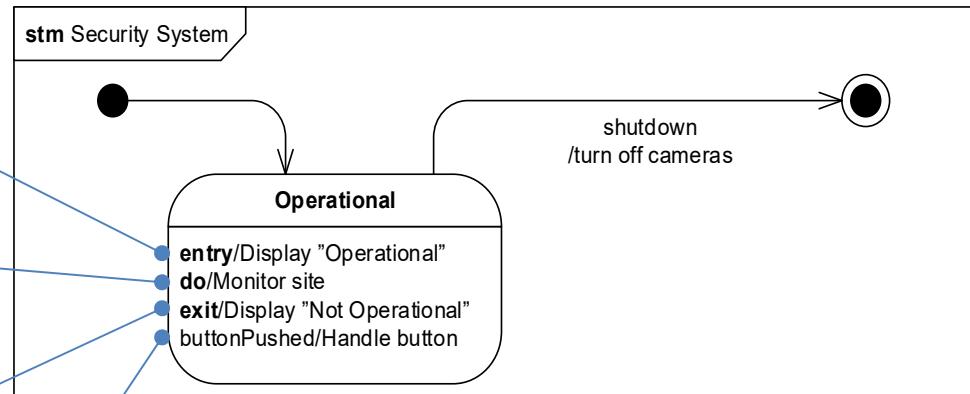
States in detail

entry behaviour is executed on entry into the state

do behaviour is continuously executed after entry until exit

exit behaviour is executed just prior to exit of the state

When events (*buttonPushed*) occurs, *do* behaviour is interrupted and action (Handle button) is executed. Then, *do* is resumed



Events:

1. Start ->

..

4. buttonPushed ->

..

7. Shutdown ->

..

Actions:

1. Display "Operational"

2. Monitor site

3. Monitor site

4. Handle button

5. Monitor site

6. Monitor site

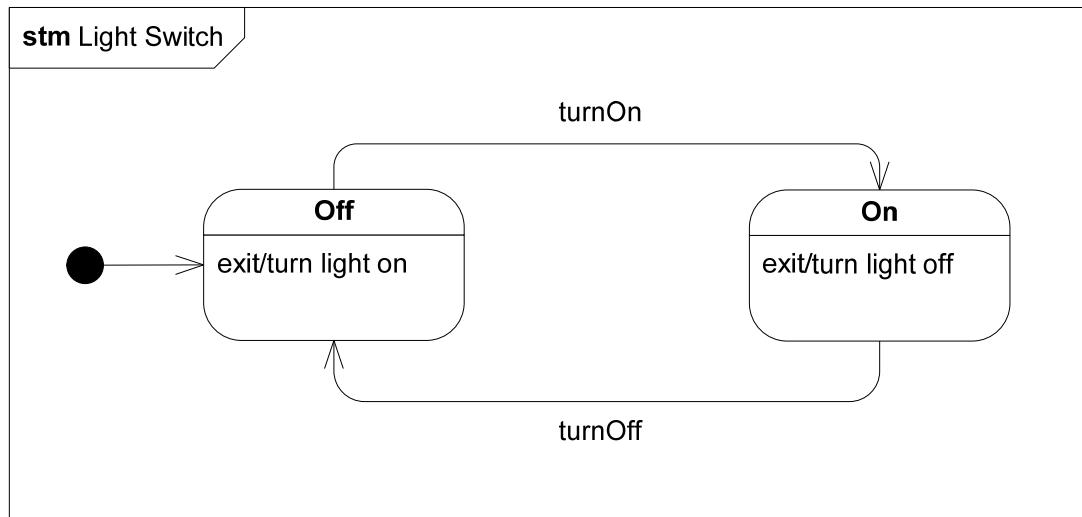
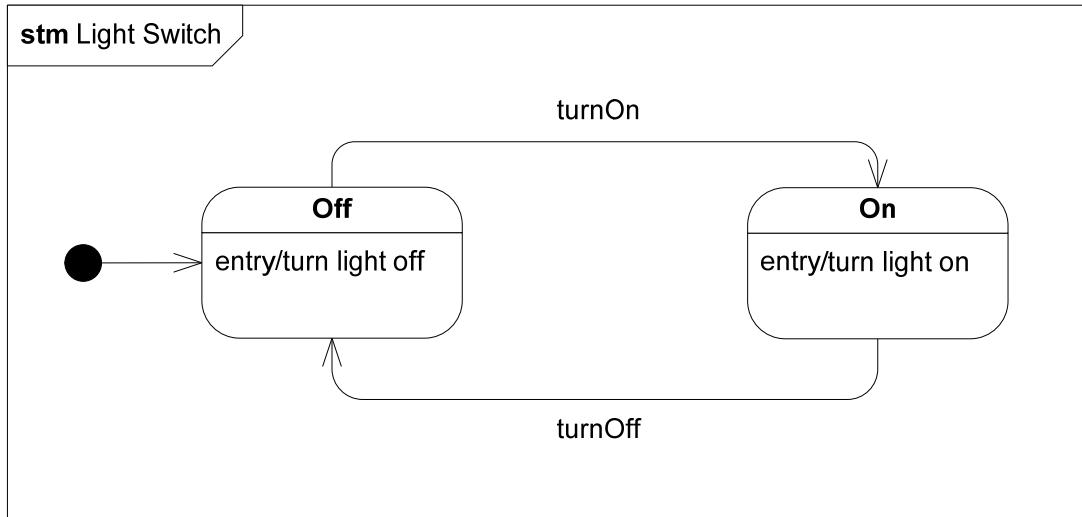
7. Display "Not Operational"

8. turn off cameras

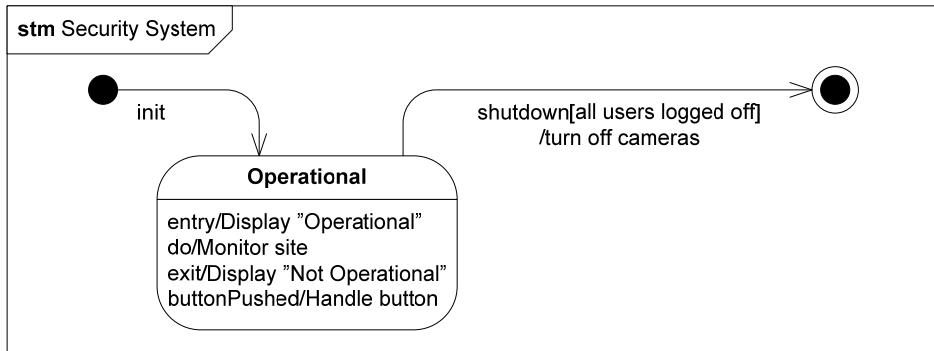
Example: Light switch



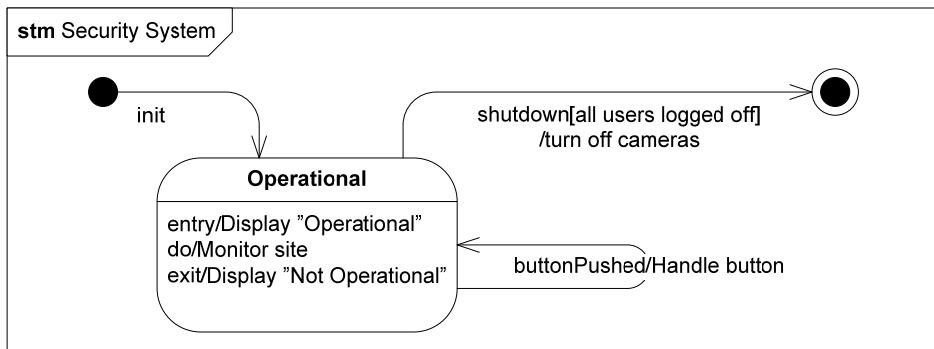
Best



States in detail – what's the difference?



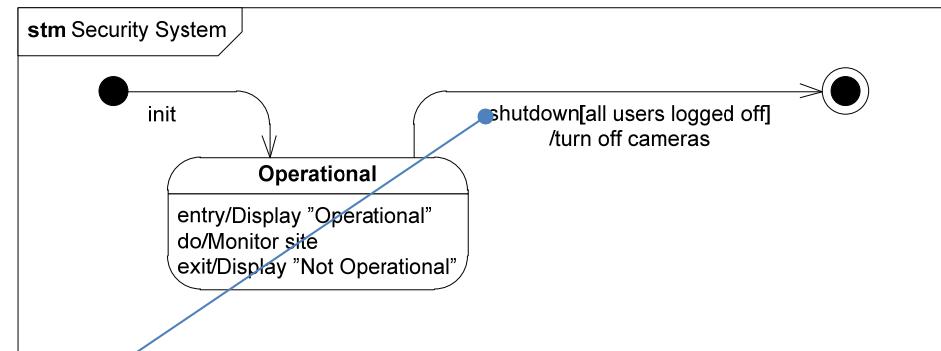
buttonPushed →
1. *Handle button*



buttonPushed →
1. *Display "Not operational"*
2. *Handle button*
3. *Display "Operational"*

Transitions in detail

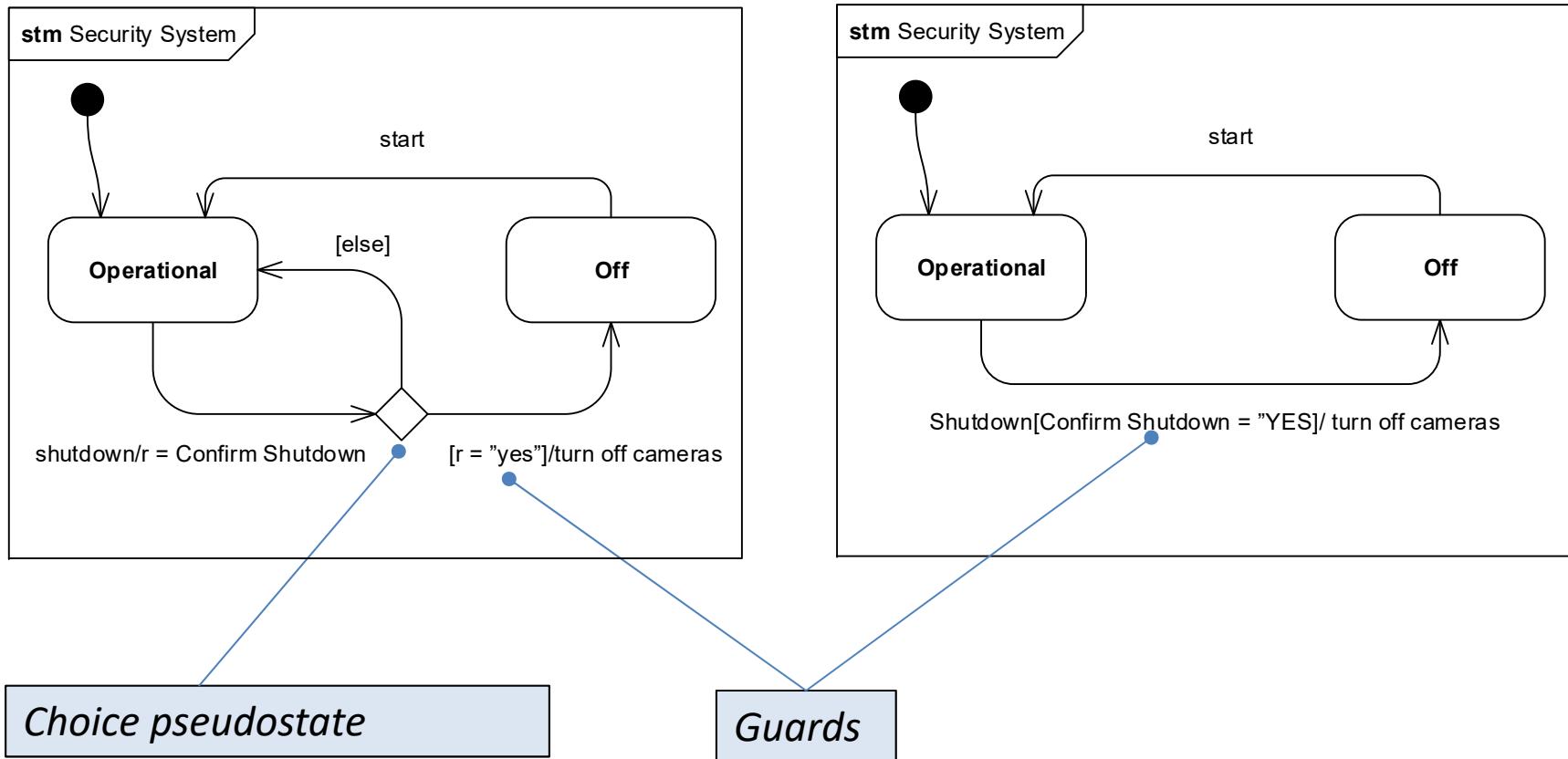
- Transitions consist of *trigger (event)*, *guard* and *effect (action)*:
trigger[guard]/effect
- When trigger occurs, guard is evaluated.
 - If guard is true, effect occurs.
 - If not, trigger is consumed without effect



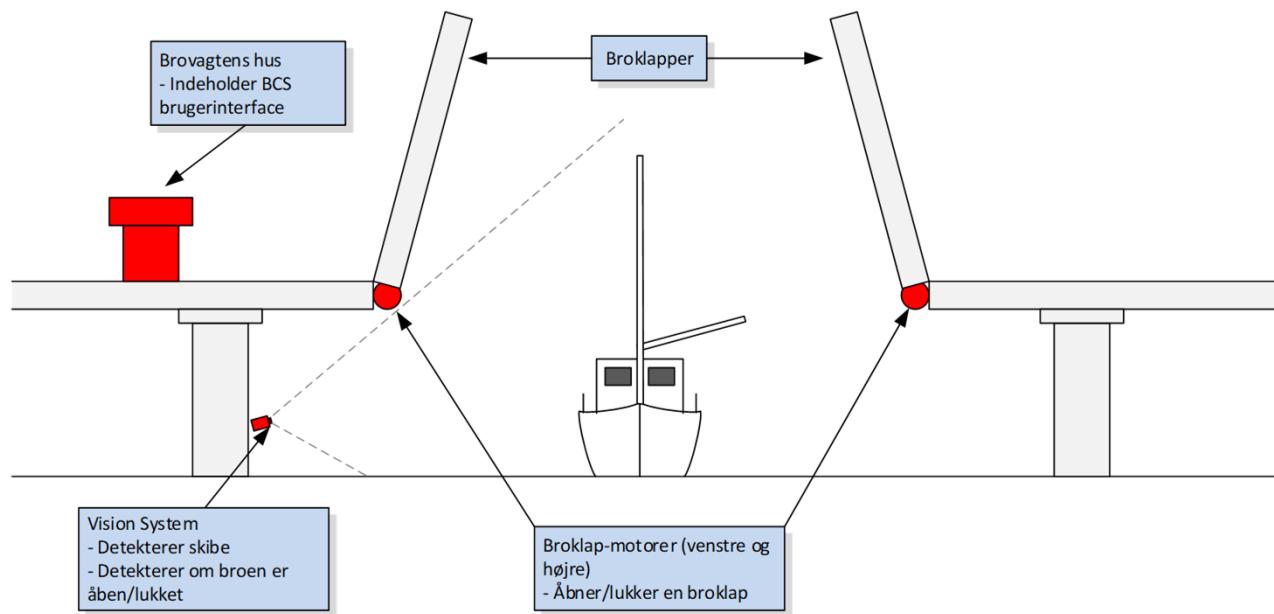
Trigger = shutdown
Guard = all users logged off
Effect = turn off cameras

What happens if some user is still logged on?

Choice pseudostate

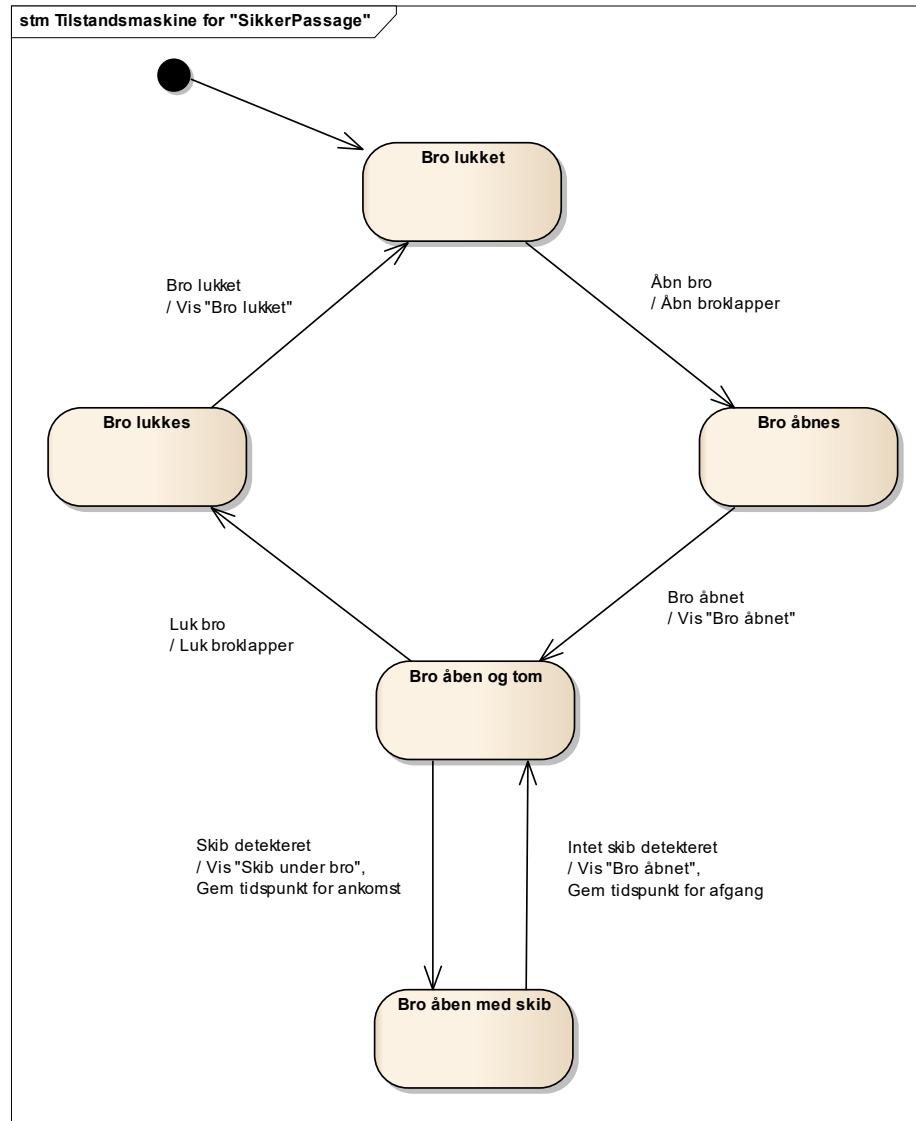


Bridge Control System (BCS)

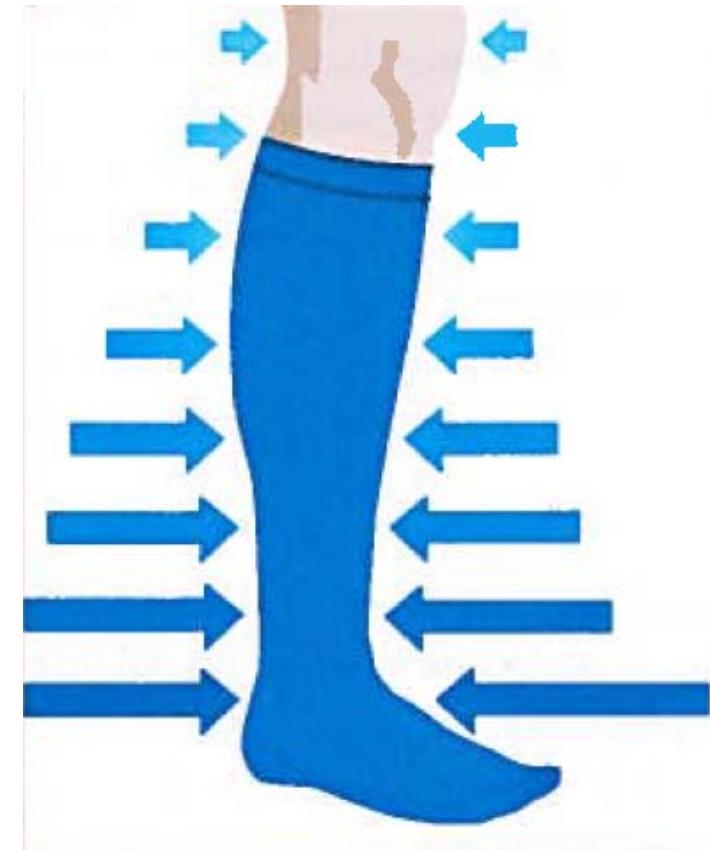


Skitse af en bro med "Bridge Control System". Med rødt: Brovagterns hus, vision systemet og broklap-motorer

Bridge Control System (STM)



Exercise: Compression stocking



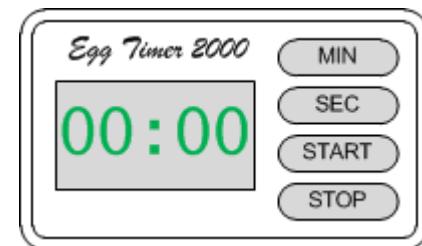
Exercise: Compression stocking

- Create a state machine diagram for a compression stocking
 - When the RED button is pushed, the stocking compresses.
 - When the GREEN button is pushed, the stocking decompresses.
 - Each second the battery level is checked. If it falls below 2.8V, the stocking decompresses and enters a FAIL SAFE state



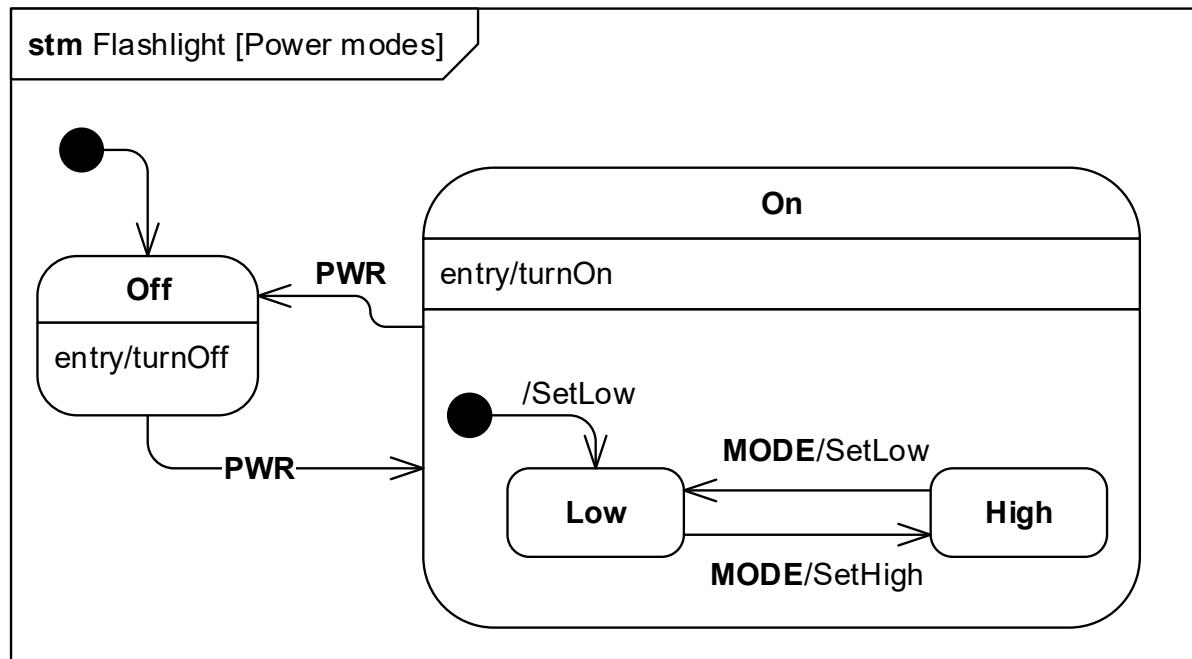
Exercise: Egg Timer 2000

- Create a state machine diagram for an egg timer:
 - The egg timer has four buttons: **MIN**, **SEC**, **START**, **STOP**
 - **MIN**, **SEC**: Increase time by 60 seconds and 1 second, respectively
 - **START**: Start countdown
 - **STOP**: If running: Stop countdown. If stopped: Clear time. If alarming: Stop alarm
 - Each second, there must be a *tick* event. If ET2000 is running, the remaining number of seconds shall be counted down by 1. If the timer expires, an alarm shall sound.
 - Ignore display updates etc. and concentrate on the setting, counting down and alarming.



States and substates (nested states)

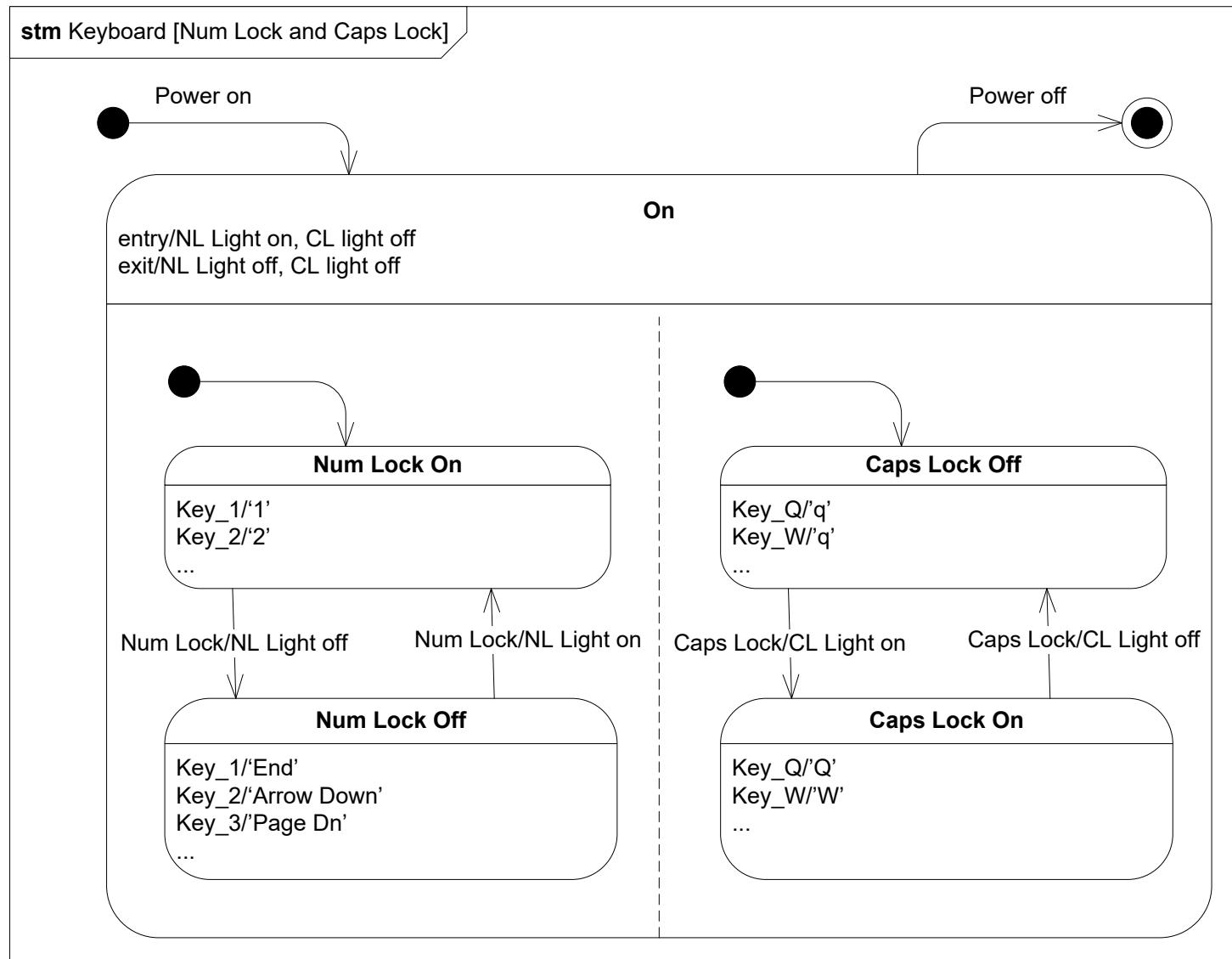
- A state may have substates.
- Example: Flashlight with **PWR** and **MODE** buttons



States with multiple regions

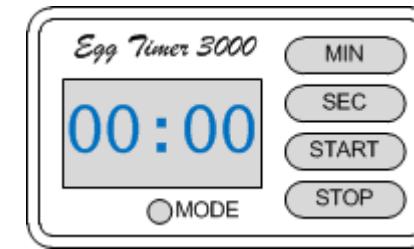
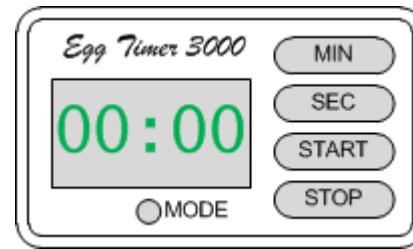
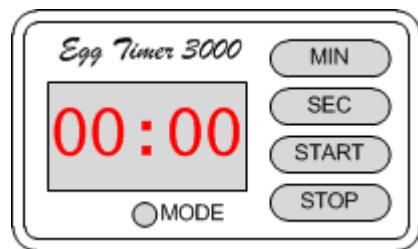
- A state may have multiple regions (aka. *orthogonal* or *independent substates*)
- If the enclosing state is active, each region will have exactly 1 active state
- State transitions in one region does not affect states in another region.
- State transitions can never transition the boundary between regions

States with multiple regions: Example



Exercise 2 : Pimped Egg Timer 3000

- PET3000 is like ET2000, but the display can be backlit with either red, green or blue light. This is controlled with the **MODE** button which toggles the light.
- Draw it's state machine diagram



Exercises

- SysML State Machines (konsol).pdf
- SysML State Machines (telefon).pdf

System Architecture and Design

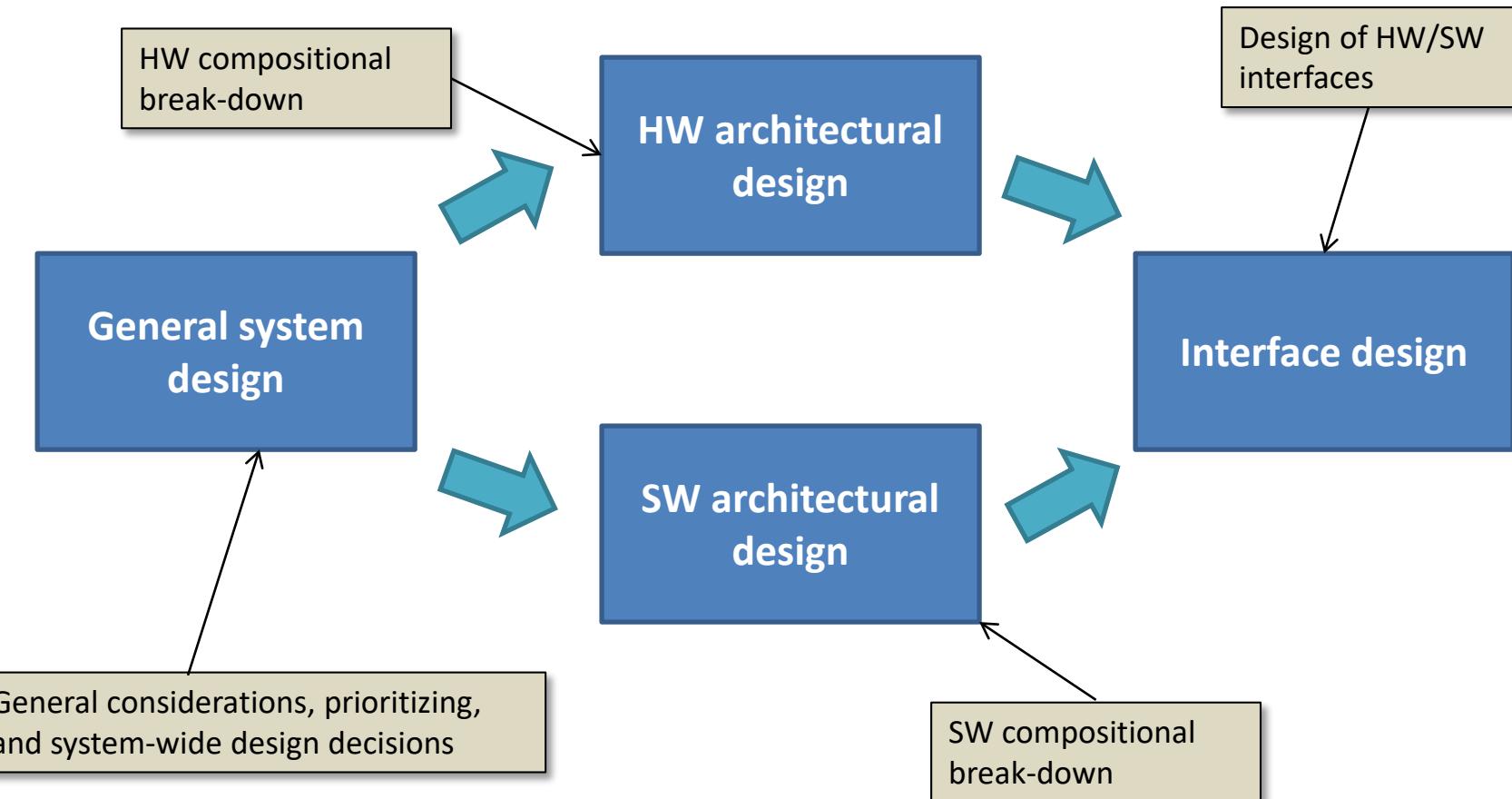
I2ISE

System Architectural Design

- Using our specification and System Design (BDD, IBD) as a starting point, we will now elaborate on the architectural design of the system
- The architectural design covers design decisions, priorities, HW and SW decomposition, etc. of our system
- This requires a number of different considerations which will be covered

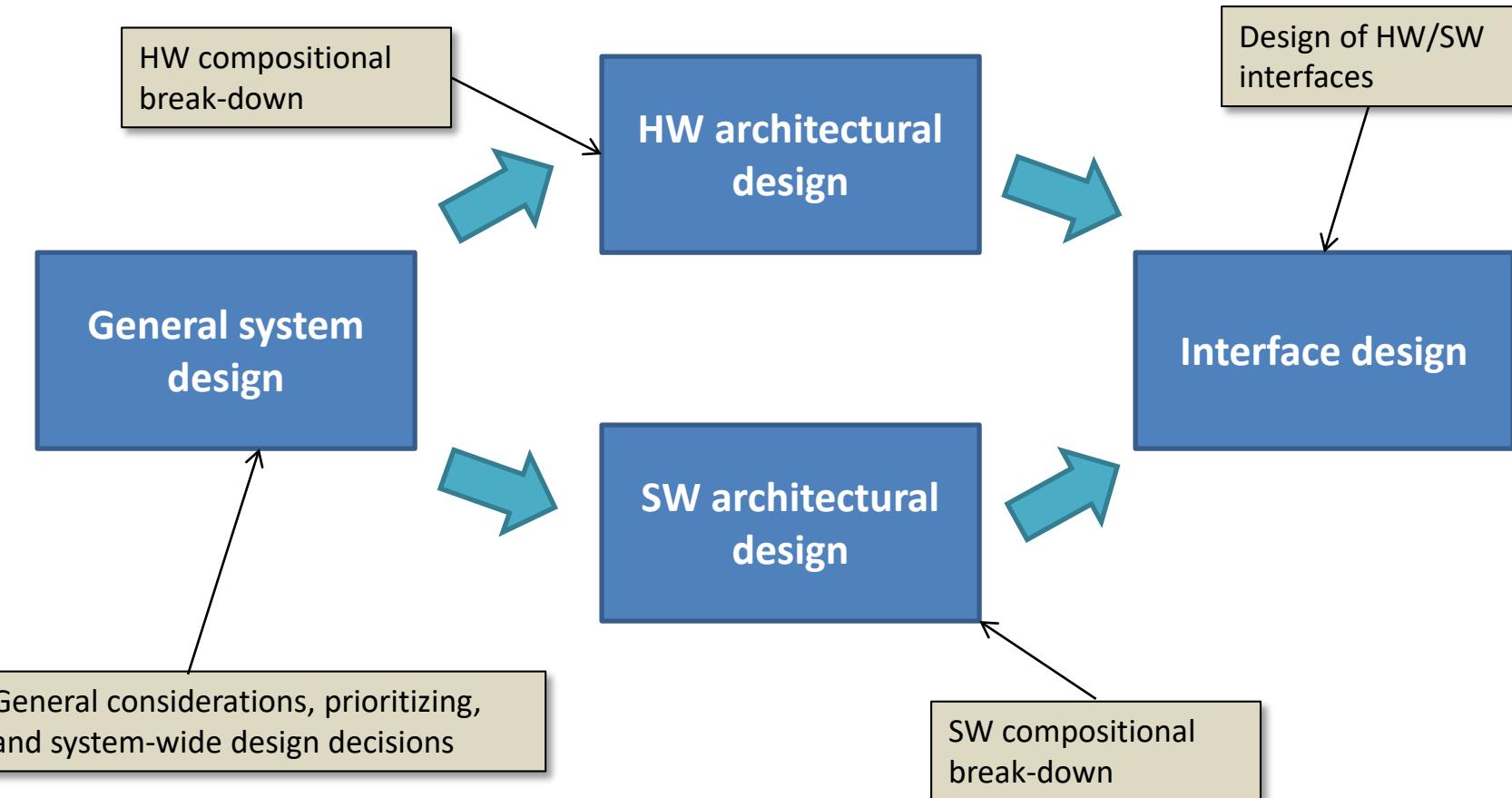
Architectural design activities - overview

- We will investigate 4 related architectural design activities:



General system design

- Today, we look at *General system design*



General system design

- In *general system design* we do design considerations
- There are important and hard choices to make



Fast, cheap, good – choose any two!

- But before we get to the hard choices, a couple of *design principles* that will help no regardless of choice

System design principles

- The system design principles are your new best friends!



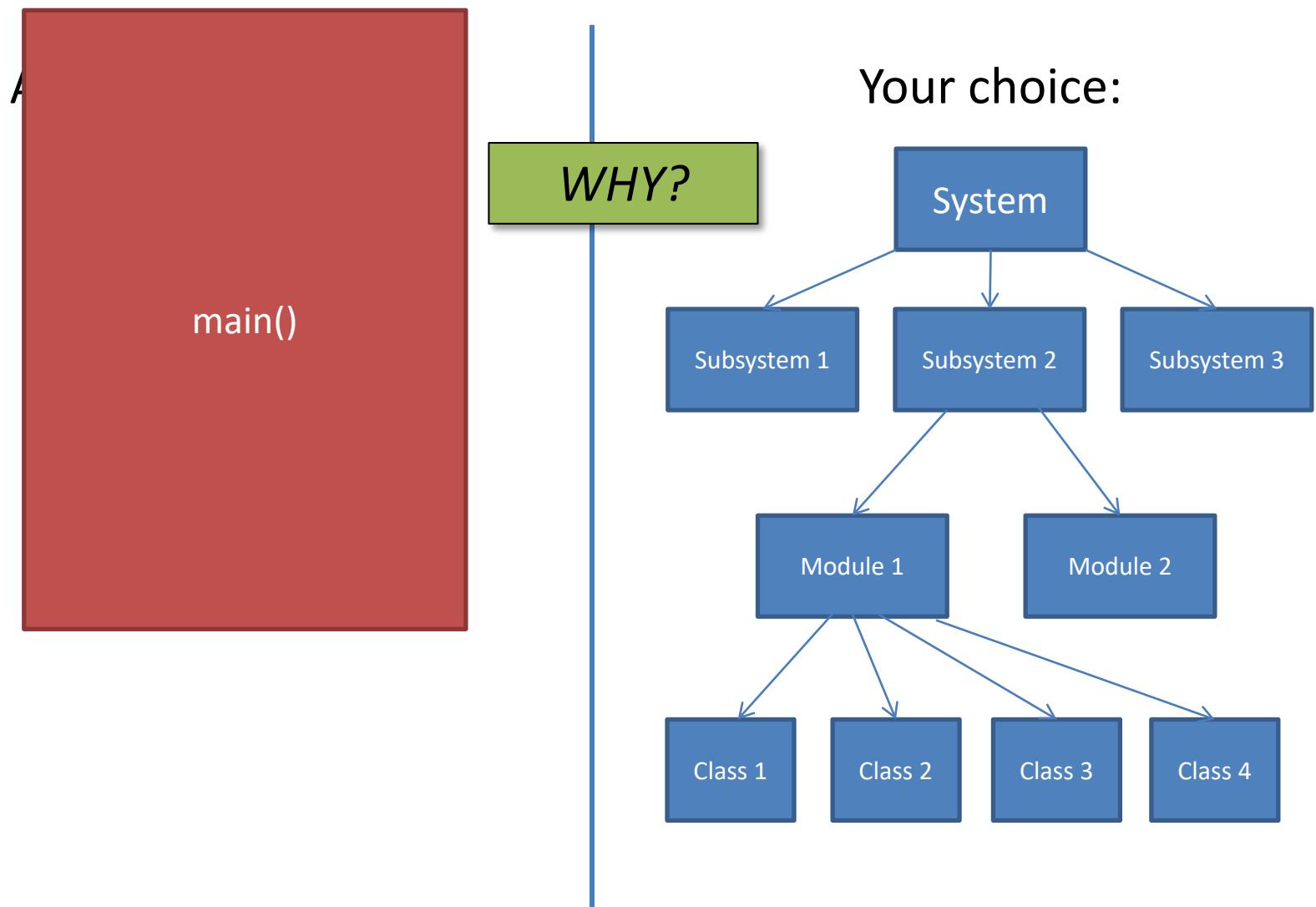
- They will help you construct a maintainable, scalable, easy-to-understand system
- The design principles will become second nature later in your studies. Right now, however, they are new and hard to understand and use.

System design principles

- The system design principles include:
 - *Decomposition*
 - *Low coupling (kobling/binding)*
 - *High cohesion (samhørighed)*
 - *Use abstractions*
 - *Re-use existing design solutions*
 - Ensure *testability*

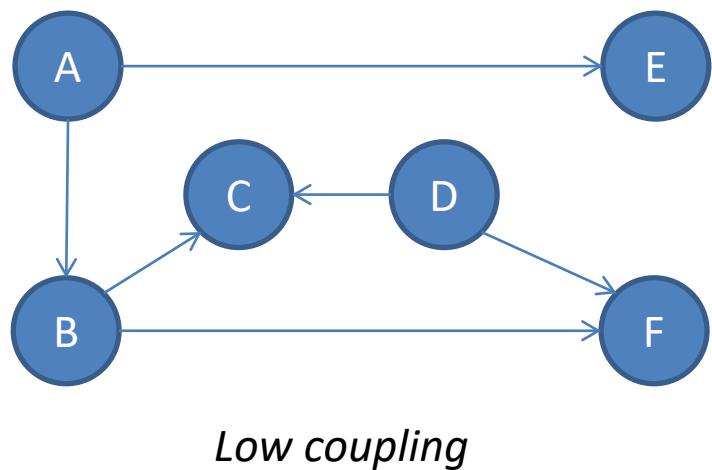
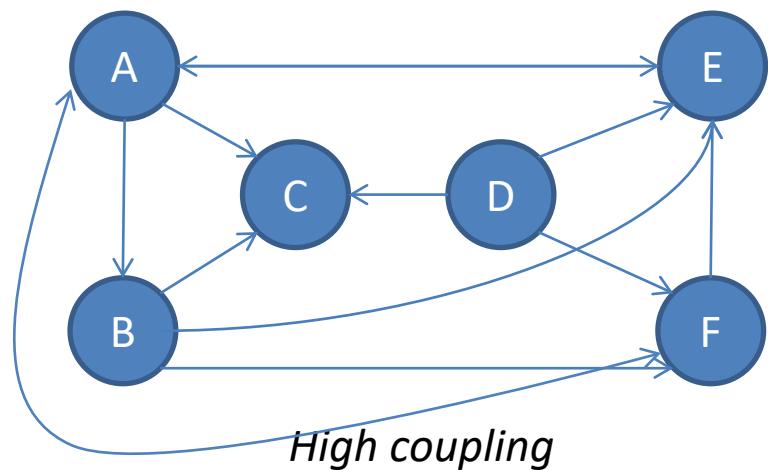


Design principles – Decomposition



Design principles – low coupling

- *Coupling* is a measure of how dependent a {SW | HW} module is of other modules



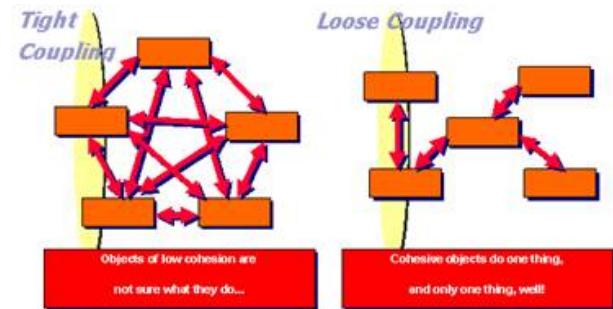
- Lowering the coupling entails a number of benefits – which?

High Coupling

- Difficult to debug during development
- Difficult to trace errors in running system
- Difficult to maintain
- Difficult to modify with new functionality
-

Advice

- Remove and reduce dependencies
- Minimize amount of information exchange between components
- Do not use global variables
- Keep design simple



Design principles – low coupling

- High coupling may be only indirectly evident:

```
void print(Report r)
{
    mySecretary.getTaskQueue().addPrintTask(new PrintTask(r));
}
```

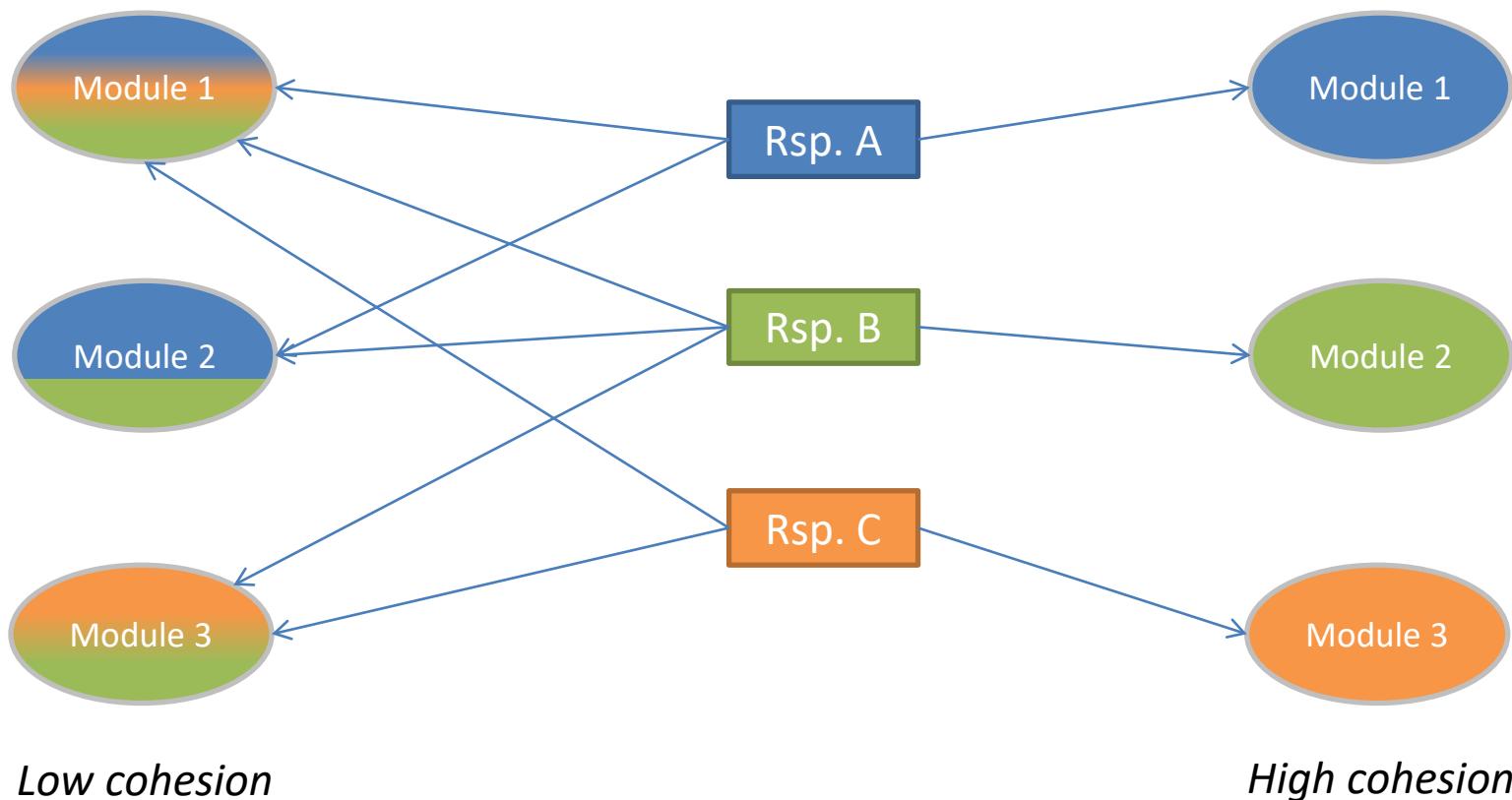
Any changes to the secretary, the task queue, or the print task will impact our implementation of `print()` ☹
E.g. renaming `PrintTask()` → `PrintJob()`: changes everywhere!

```
void print(Report r)
{
    mySecretary.print(r);
}
```

Only changes to the secretary will impact our implementation of `print()`. Whether she uses a task queue, stack, or whatever is not our concern 😊

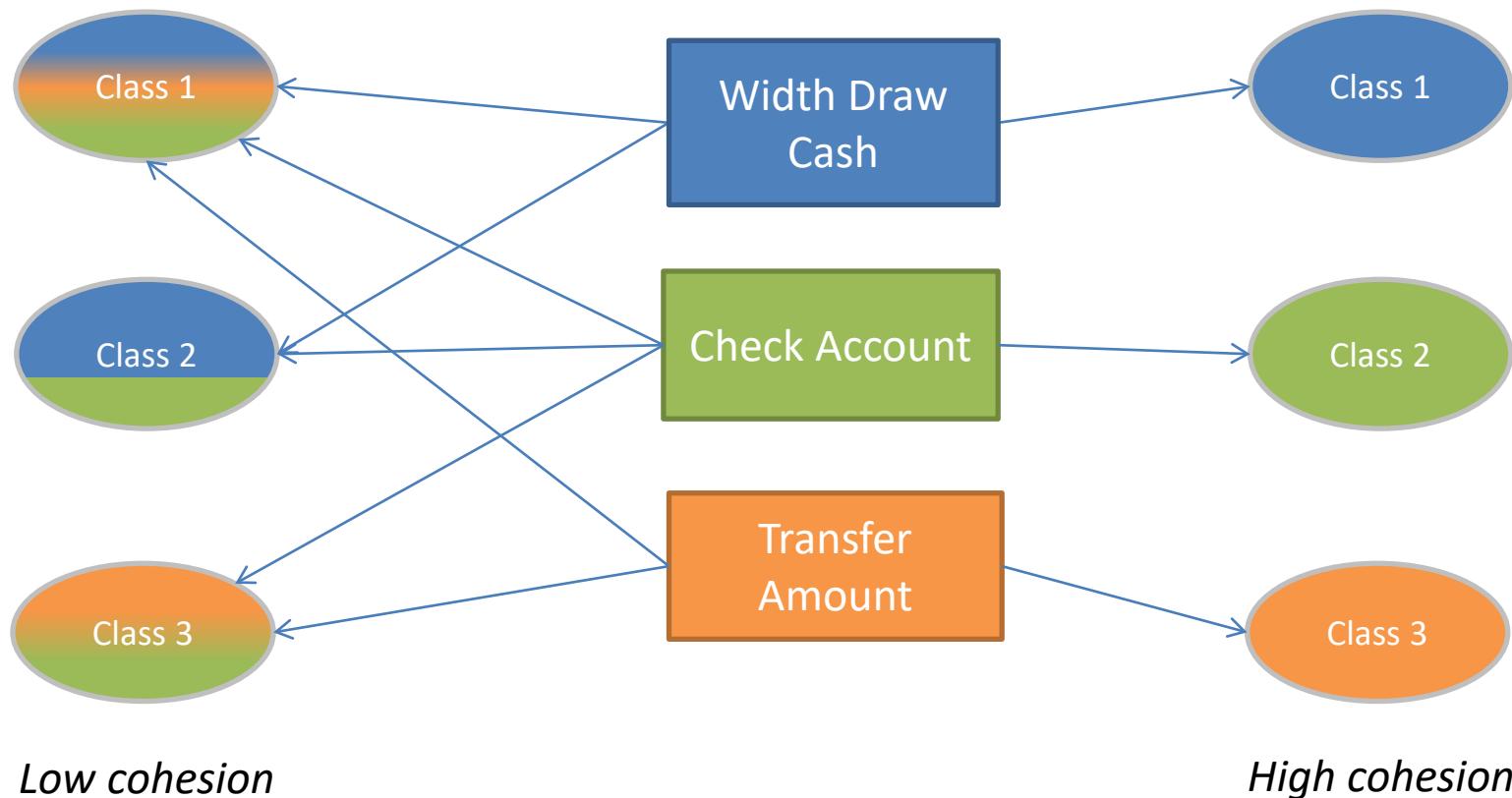
Design principles – high cohesion

- *Cohesion* is a measure of how well a given responsibility is encapsulated in a module or component



Design principles – Application Model (ATM)

- *Functionality specified by use cases, that is realized by controller classes, is an example of achieving high cohesion*

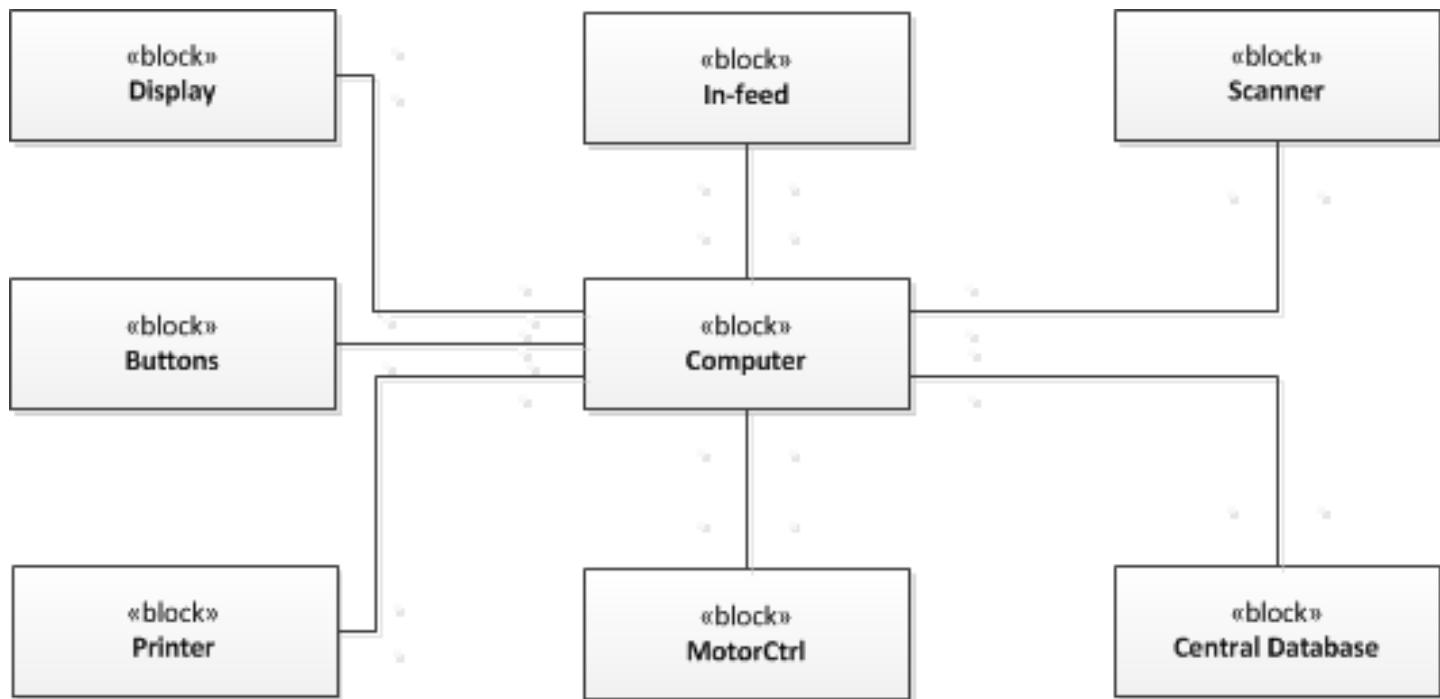


Types of Cohesion

- **Functional** – all elements contributes to execution of a specific task
- **Sequential** – output from one procedure becomes the input for the next
- **Communication** – procedures that operates on the same set of input data

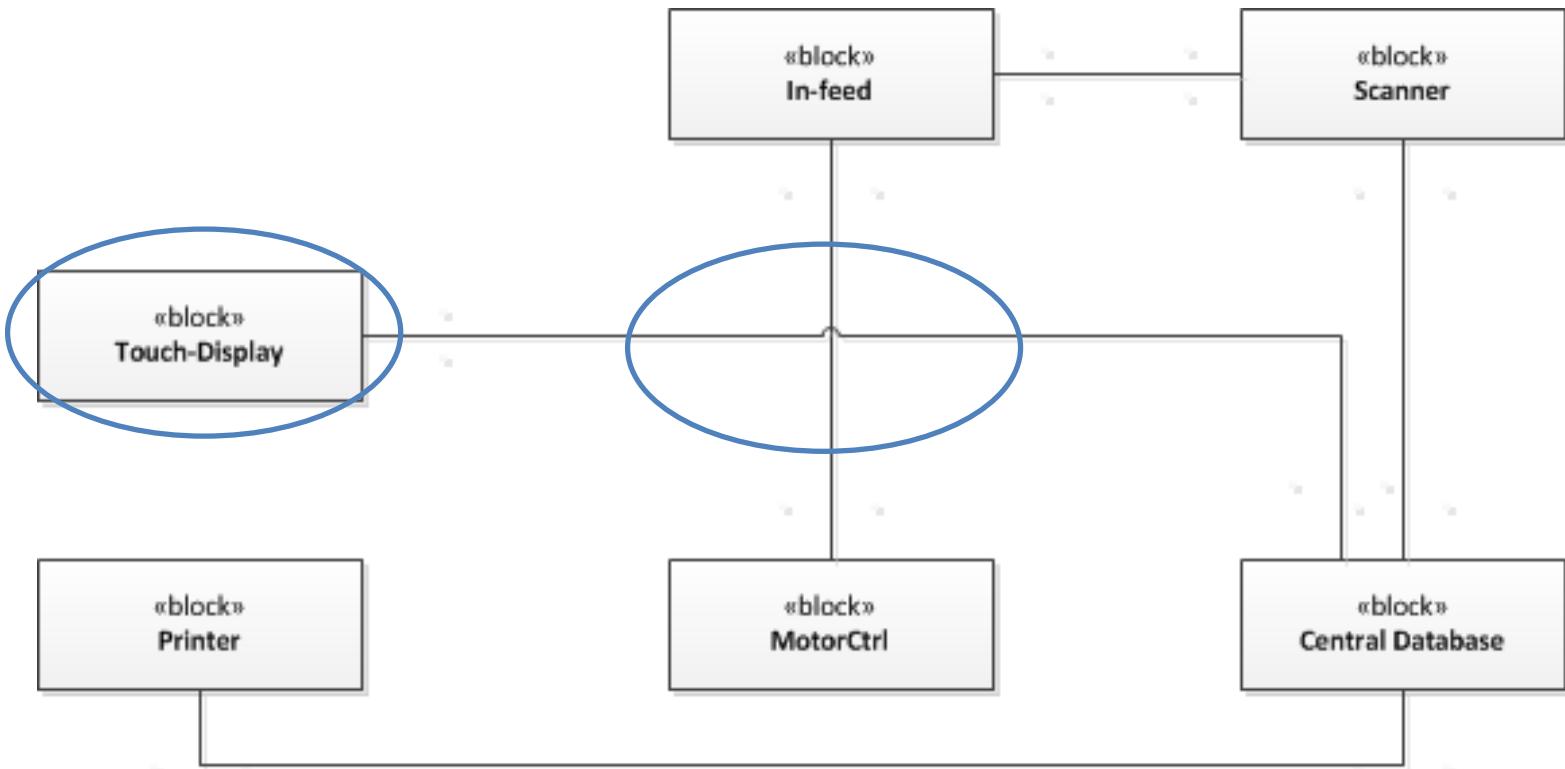
RVM Architecture 1

- Cohesion / Coupling ?



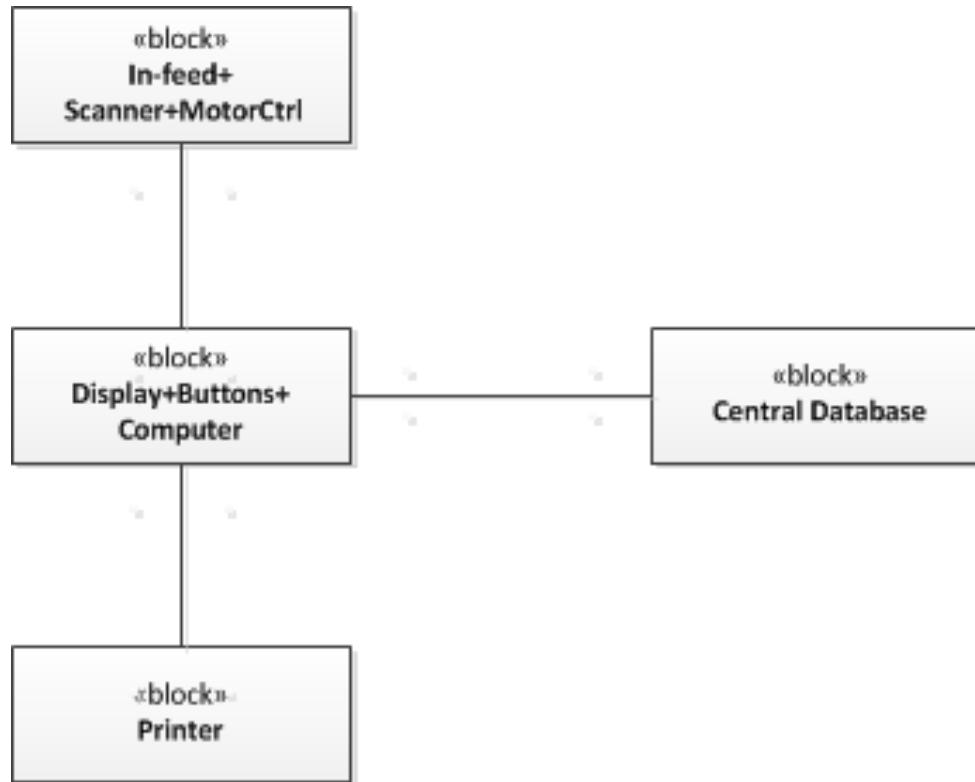
RVM Architecture 2

- Cohesion / Coupling ?
- Is the same functionality possible to realize as in RVM Architecture 1?



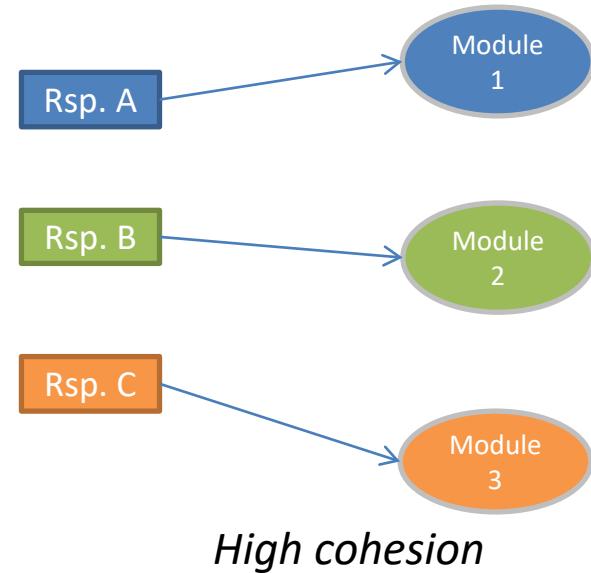
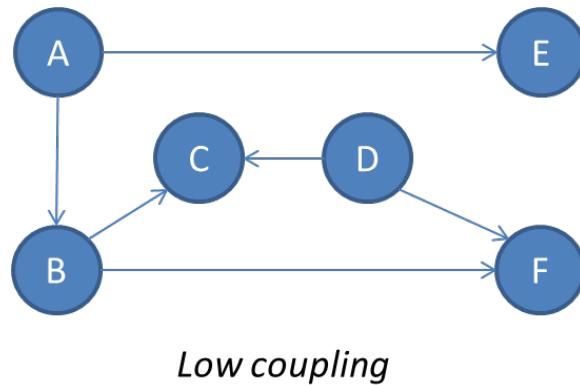
RVM Architecture 3

- Cohesion / Coupling ?
- What about errors and replacement of components?



Low coupling, high cohesion – your turn!

- 5 minutes: Discuss...
 - ...the benefits of high cohesion in a design
 - ...how low coupling supports high cohesion
 - ...how low cohesion hinders low coupling



Forklar kort hvilken betydning begreberne "coupling" og "cohesion" har i forhold til et godt arkitekturdesign.

- *Design principperne "coupling" (kobling) og "cohesion" (samhørighed) er vigtige for et godt design. Der skal være lav kobling mellem klasser/komponenter i designet og stor samhørighed i de enkelte klasser/komponenter. Den lave kobling gør at de enkelte komponenter er lettet at vedligeholde, udskifte, modificere og teste. Samhørigheden internet i klassen/komponenten kan omhandle emner som: funktionsmæssig sammenhæng, sammenhæng mellem sekvens af operationer, kommunikation og logisk sammenhæng.*

Abstractions

- Using *abstractions* help you achieve low coupling and high cohesion
 - Disregard *irrelevant details*, focus on only some aspects
 - Data or control abstraction
- Using abstractions properly will also increase cohesion
- An example: My abstraction of the garage that repaired my car's parking sensor



Abstractions – parking sensor

```
Car fixMyCar(Garage garage, Car car)
{
    Mechanic kian = garage.findMechanic();
    Lift lift = garage.getAvailableLift();
    kian.moveCarToLift(car, lift);
    lift.liftCar();
    kian.removeRearBumper(car);
    if(kian.inspectElectronicsUnderRearBumper(car) == BROKEN)
    {
        ParkingSensor ps =
            kian.getNewParkingSensor(garage.getStock());
        kian.install(ps, car);
    }
    else
        kian.cleanParkingSensorHeads(car);
    kian.attachRearBumper(car);
    lift.lowerCar();
    kian.moveCarToParkingLot(garage.getParkingLot());
}
return car;
}
```

Far too many details that I (as car owner) do not care about. I just want my car fixed ☹.

```
Car fixMyCar(Garage garage, Car car)
{
    garage.repair(car, "PARKING SENSOR BROKEN");
    return car;
}
```

Much better! Let the garage worry about *how* to fix my car - they can use Kian, Mary, a robot or ninja smoke for all I care. I just want my car fixed ☺

Automation – Production



Airbus A380 – Flight deck



Design for test – Ensure testability

- Plan how to test hardware components and software classes at the same time as designing the system
 - V-model
- Unit and component test
 - Interfaces
 - Functionality
 - Test cases: stimuli and expected response
- Integration test
 - Top-down or Bottom-up
 - Stubs and drivers

Meanwhile, in the system design cave...



System design - activities

- We are going to make some tough decisions. These include:

List of decisions:

1. Prioritize design criteria
2. Architectural design strategy / strategies
3. Data storage strategies
4. Software control strategies
5. Initiation/termination strategies
6. Error handling
7. Self-test and backup functions
8. ...

- The choices we make will impact the system architecture as a whole.
- We will investigate some of these in the following

Prioritizing design criteria

List of decisions:

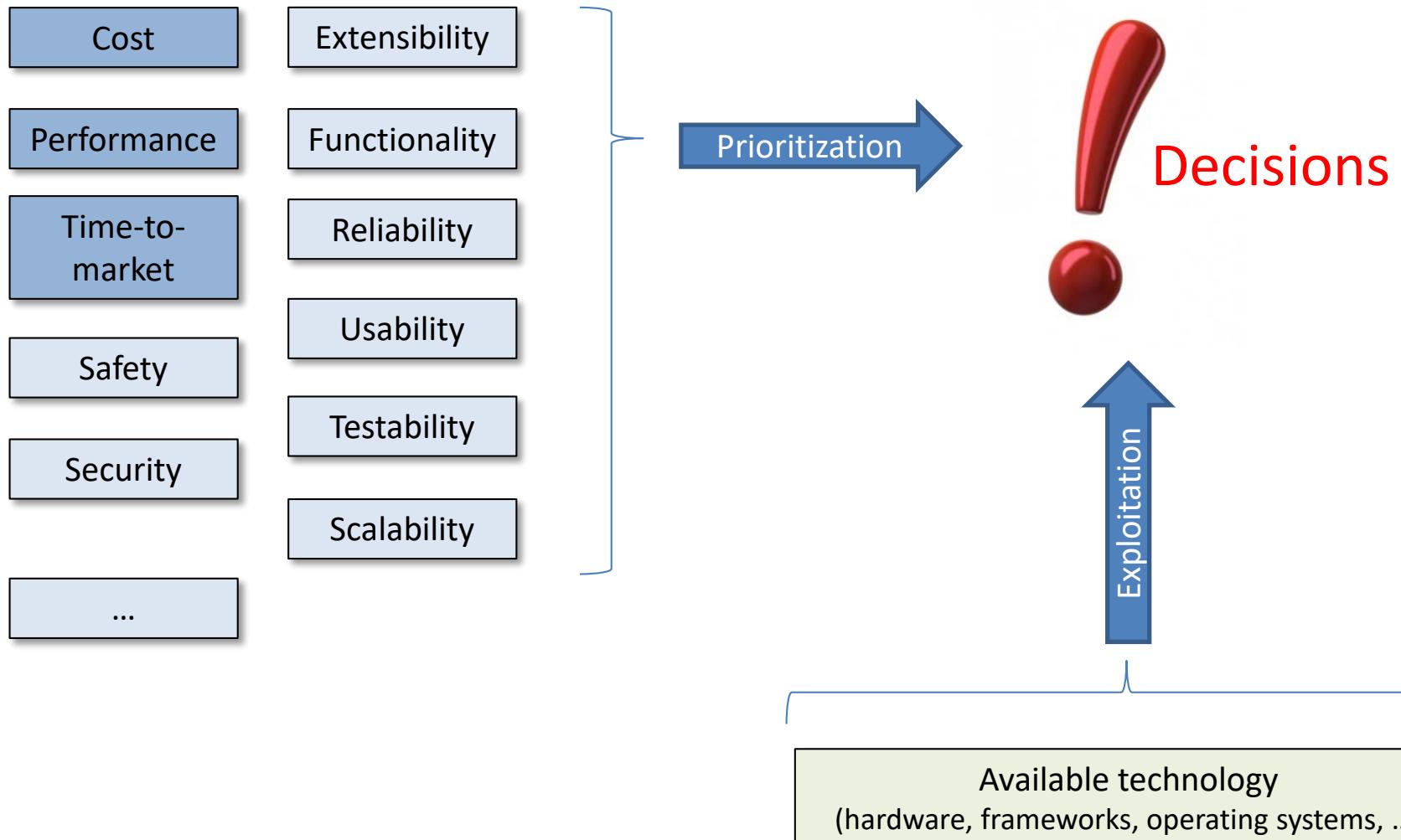
1. **Prioritize design criteria**
2. Architectural design strategy / strategies
3. Data storage strategies
4. ...

- A system's selected design criteria are the foundation upon which design decisions are later made
- Welcome to the real (imperfect) world!
 - Design criteria are diverse and contradicting - "*Fast, cheap, good...*"
- The selected design criteria are driven by the *intended market* and the *existing technology*
- Some example criteria on next slide

Prioritizing design criteria

List of decisions:

1. **Prioritize design criteria**
2. Architectural design strategy / strategies
3. Data storage strategies
4. ...

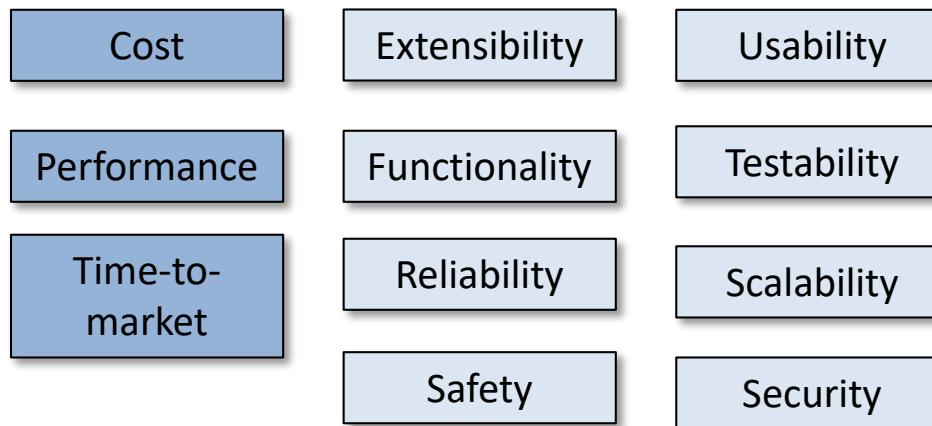


Prioritizing design criteria - your turn!

List of decisions:

1. **Prioritize design criteria**
2. Architectural design strategy / strategies
3. Data storage strategies
4. ...

- 10 minutes: Select top-4 design criteria if you are making...
 - An iPhone accessory
 - A nuclear power plant
 - A Reverse Vending Machine
 - "Slusesystem"



Architectural design strategy

List of decisions:

1. Prioritize design criteria
2. **Architectural design strategy / strategies**
3. Data storage strategies
4. ...

- An architectural design strategy is a strategy for the design of the system. This includes...
 - Selection of layering
 - Deciding the use of framework(s)
 - Network technologies
 - Database management
 - ...
- Let's take a look!

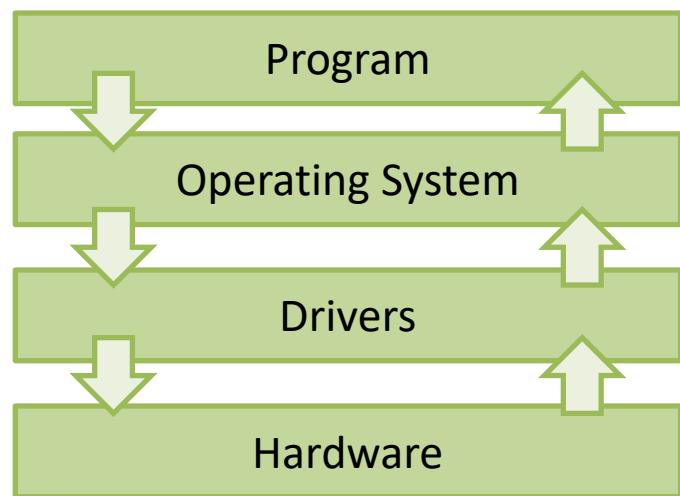
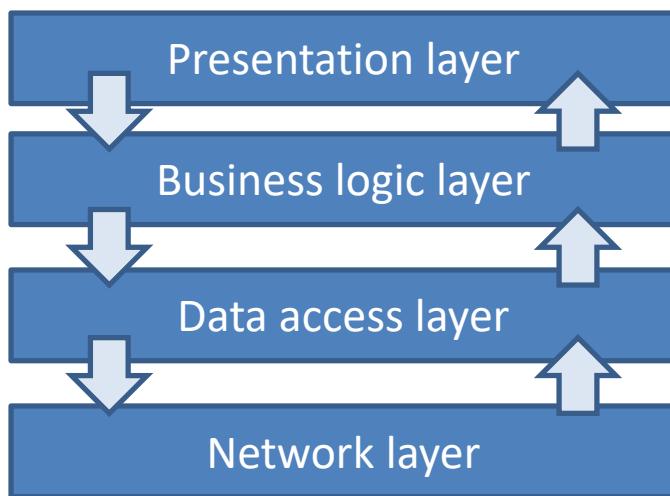
Architectural design strategy

- Layering

List of decisions:

1. Prioritize design criteria
2. **Architectural design strategy / strategies**
3. Data storage strategies
4. ...

- Layering the system is one of the most effective ways of achieving low coupling on a system-wide scale.



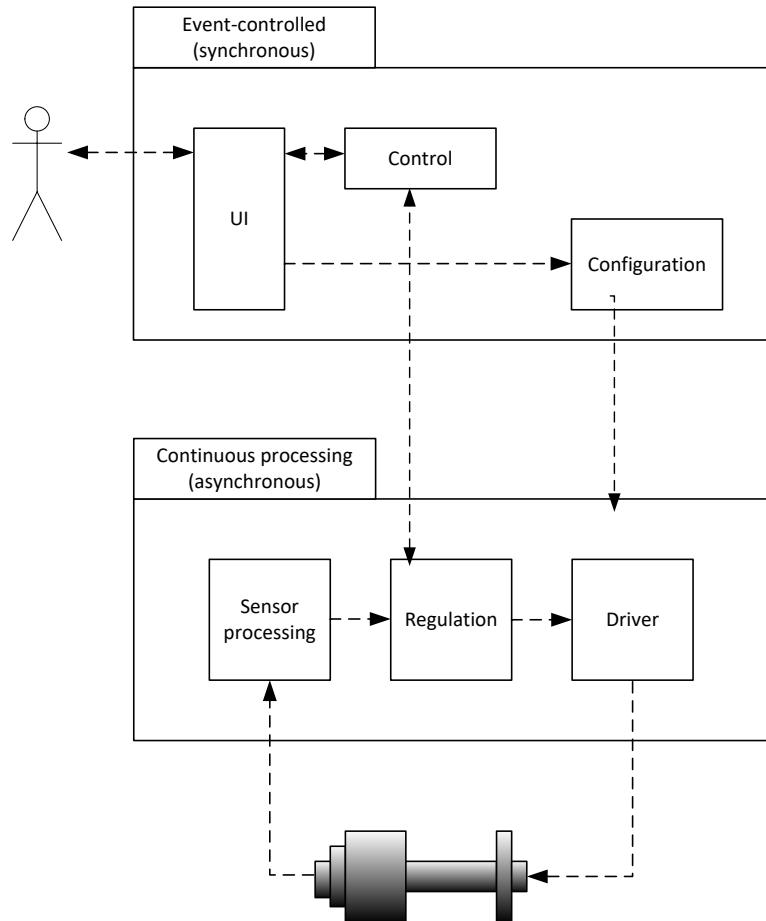
Architectural design strategy

- Half-sync, half-async

- For continuous (e.g. control) systems, *half-sync, half-async* can be a fine approach

List of decisions:

1. Prioritize design criteria
2. **Architectural design strategy / strategies**
3. Data storage strategies
4. ...



Frameworks

List of decisions:

1. Prioritize design criteria
2. **Architectural design strategy / strategies**
3. Data storage strategies
4. ...

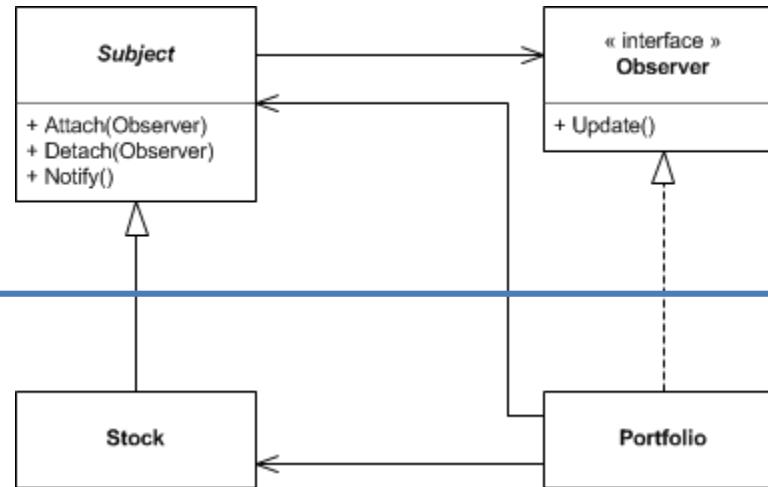
- Frameworks are ready-made software systems which you instantiate, typically by extension
 - Declare inheritance from a framework class
 - Implement framework (abstract) methods
- Frameworks provide a lot of functionality and decoupling, and typically hides irrelevant details

Frameworks – example: A stock market

List of decisions:

1. Prioritize design criteria
2. **Architectural design strategy / strategies**
3. Data storage strategies
4. ...

Framework



Application

```
Stock::setValue(double v)
{
    value = v;
    Notify();
}
```

```
Portfolio::addStock(Stock s)
{
    s.Attach(this);
}
```

```
Portfolio::Update()
{
    // Do something with the stock
}
```

Initiation/termination strategies

List of decisions:

4. ...
5. **Initiation/termination strategies**
6. Error handling
7. Self-test and backup functions

- You should also consider how the system is initiated and terminated
- Initiation:
 - Start order?
 - Access to configuration data?
 - Power-On Self-Test?
- Termination:
 - Termination order?
 - Configuration storage?

Error handling

List of decisions:

4. ...
5. Initiation/termination strategies
- 6. Error handling**
7. Self-test and backup functions

- Hardware in system → System will eventually fail
- Error handling is critical to get into the system from day 1
- You need a *strategy* for this.
 - How do you *detect* errors?
 - How do you *handle* detected errors?

Some strategies for error detection and handling

List of decisions:

4. ...
5. Initiation/termination strategies
6. **Error handling**
7. Self-test and backup functions

- Detection:
 - Watchdogs – detect software deadlocks
 - Voting systems
 - Self tests / self-diagnostics
 - Power-On Self-Test (POST)
 - Continuous Built-In Test (CBIT)
- Handling
 - User intervention
 - Limp mode
 - Redundant systems

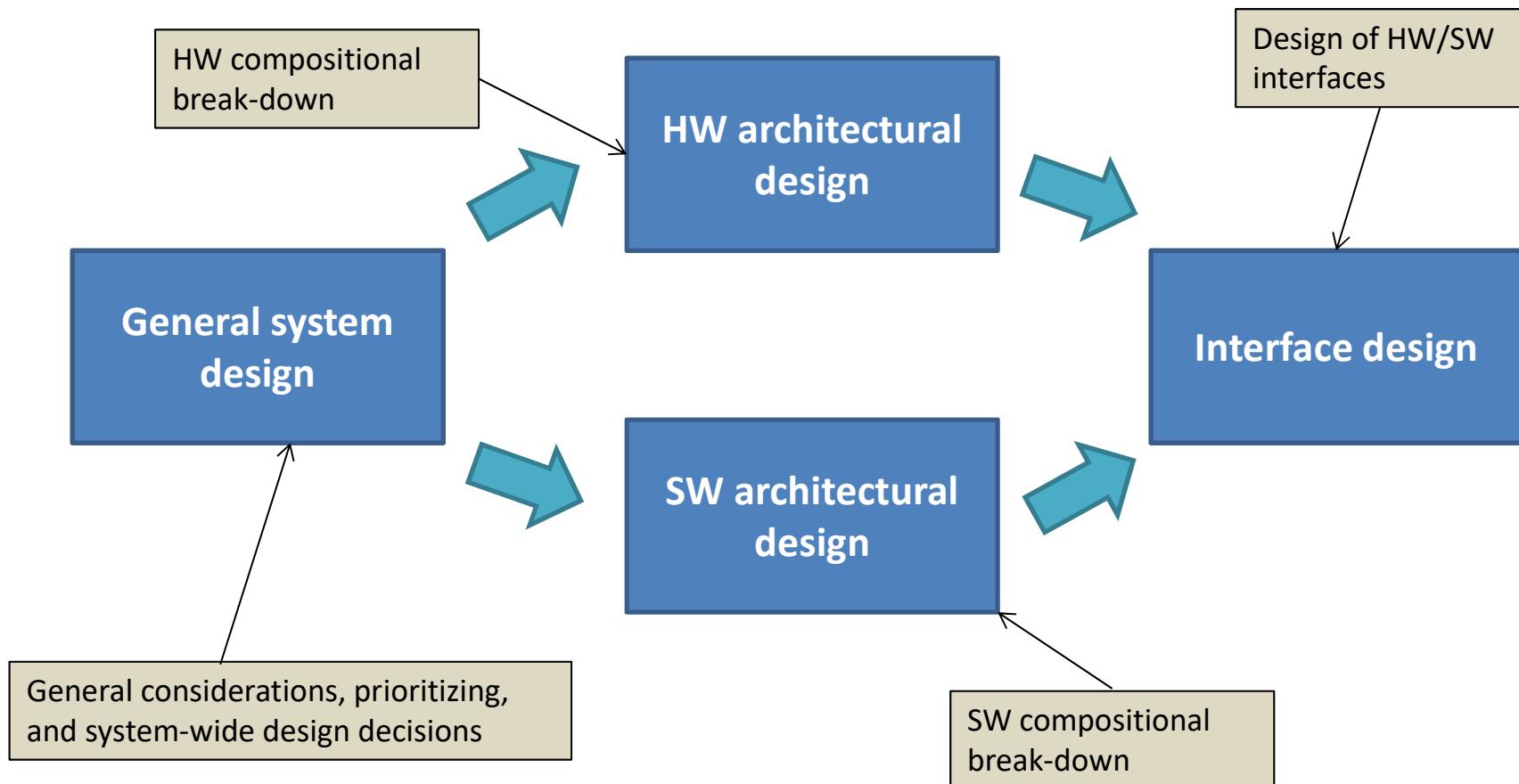
System Design and Interfaces (SysML and Hardware)

I2ISE

It is essential to know the specification of interfaces being able to design, test and develop a system.

HW/SW architectural design

- Today, we look at *HW architectural design and interfaces*



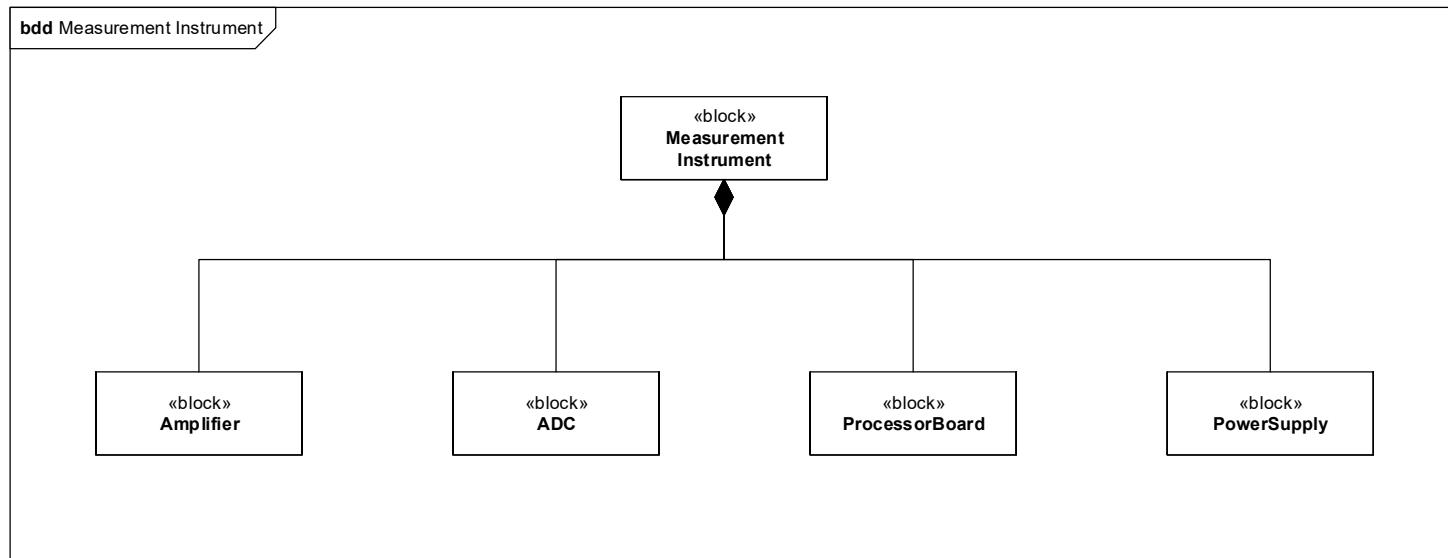
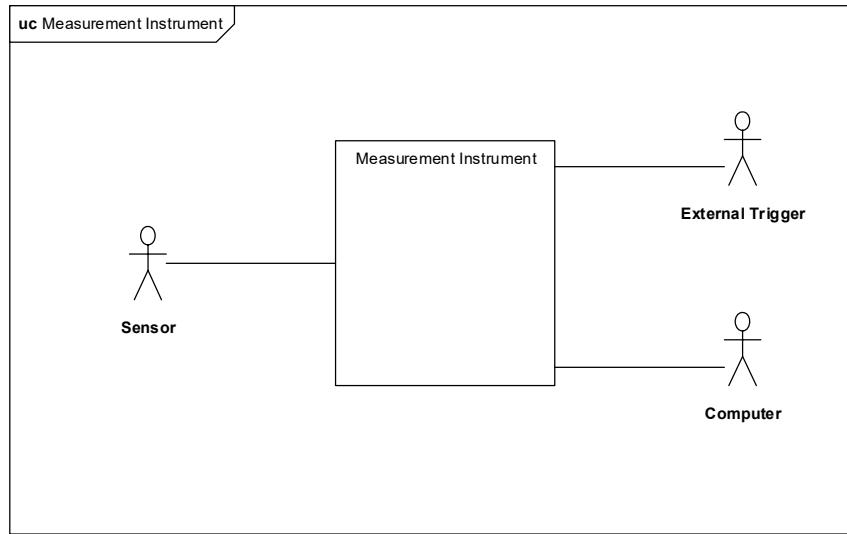
Interfaces

- Today, we will look at *interfaces*:
 - **Interfaces in SysML**
 - **Specifying HW interfaces in detail**
 - *Specifying SW interfaces – later in course*
 - *Specifying protocols – later in course*
 - ...

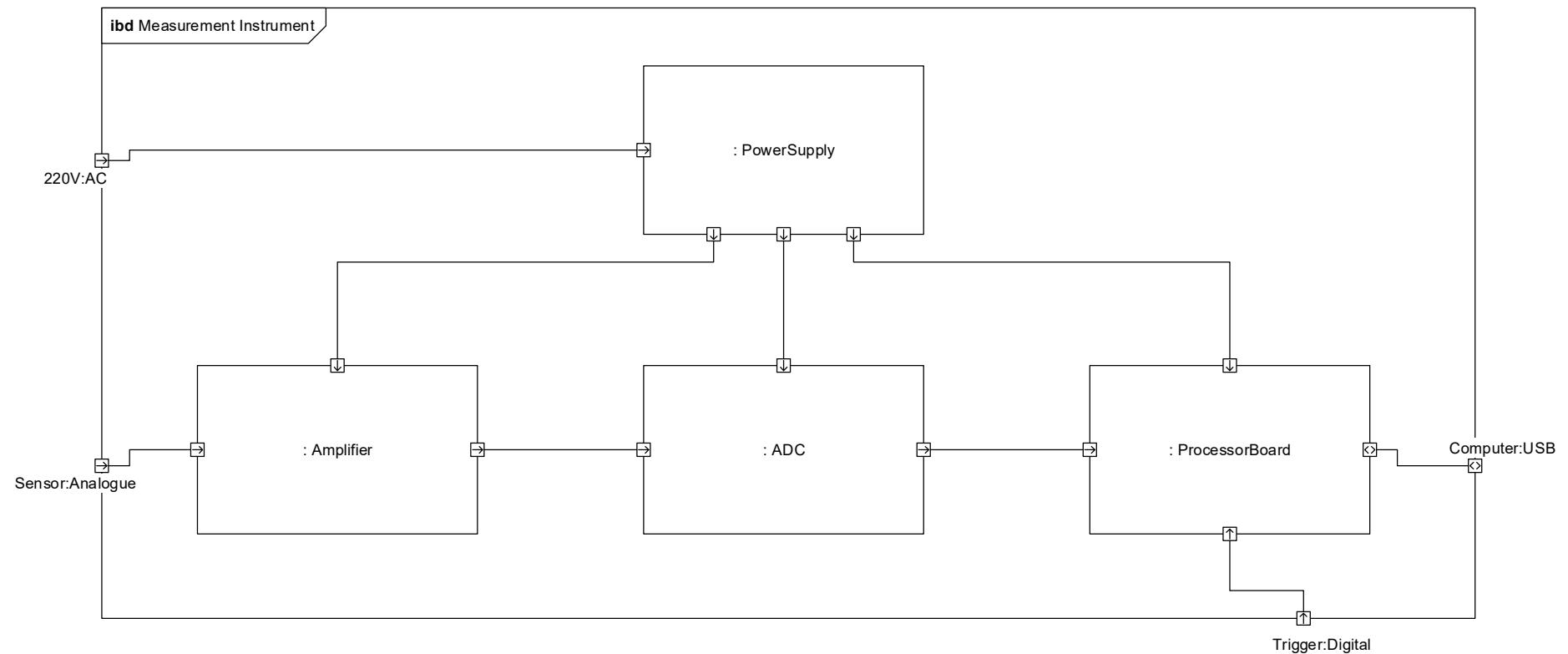
How to specify interfaces using SysML

1. Start with context and BDD and IBD diagrams
2. Define external ports on the IBD diagram
3. Define internal ports on the IBD diagram
4. Describe functionality for every block
5. Specify requirements to interfaces between parts - describe electrical requirements for all ports of all blocks in a table – ensure they fit together

1. Measurement Instrument – Context /BDD



2. Measurement Instrument – IBD External Ports



2. External Ports Requirements

Name of Block	Description of function	Port Name	Type	Port Specification
Measurement Instrument	Instrument to measure low voltage sensor signals. The sensor signal is digitized and data stored in memory whenever the input trigger signal is high. Measured sensor data can be transfer to a connect computer over the USB port.	220V	AC	200 – 250 V RMS, 50 Hz Input current limiter of 100 mA
		Sensor	Analogue	Differential Input Signal Voltage Range -100 to +100 uV peak Impedance 50 Ohm
		Trigger	Digital	5 V trigger input Low when < 0.8 V High when > 2.0 V
		Computer	USB	USB 2.0

Examples of different **type** categories

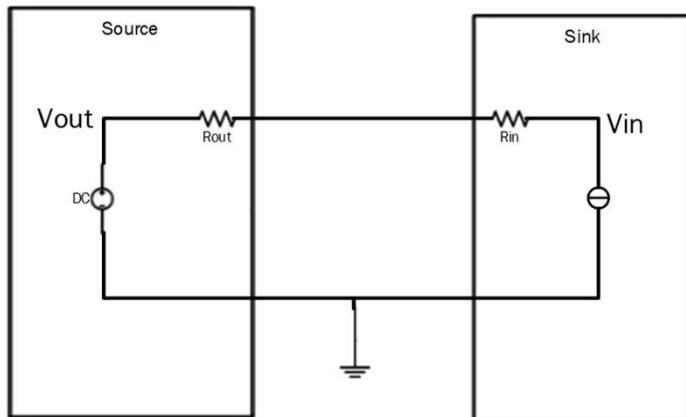
(Mainly used in the course)

Signals (Electrical)	Standards (Physical, Protocol)	Network (Protocol)	Information (Software)	Supply (Power)	Others
Analogue	USB	Ethernet	File	DC	Force
Digital	RS232	Wireless	Image	AC	Light
	HDMI	Internet	String		Sound
	SPI	Profinet	Barcode		Noise
	I2C		Bytes		Liquid
	TTL		Data		
	CMOS		Bool		

Signals (Electrical)

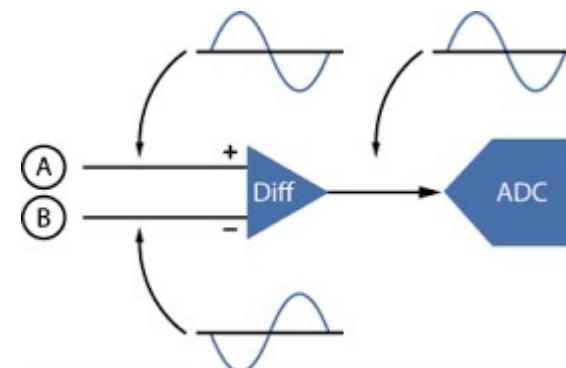
- Output voltage with tolerances and maximum current
- Input voltage with tolerances and maximum current
- Output and input impedance has to fit together

Single Ended Interface

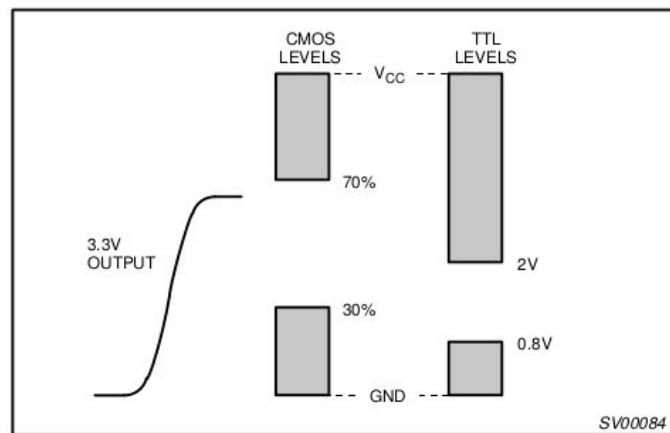


$R_{in} = R_{out}$ is best

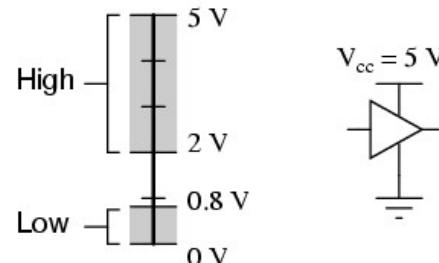
Differential Input



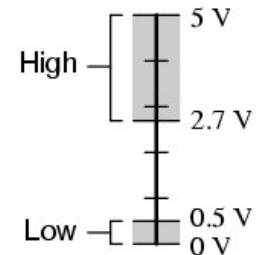
Interface with Digital Signals (Standards TTL or CMOS)



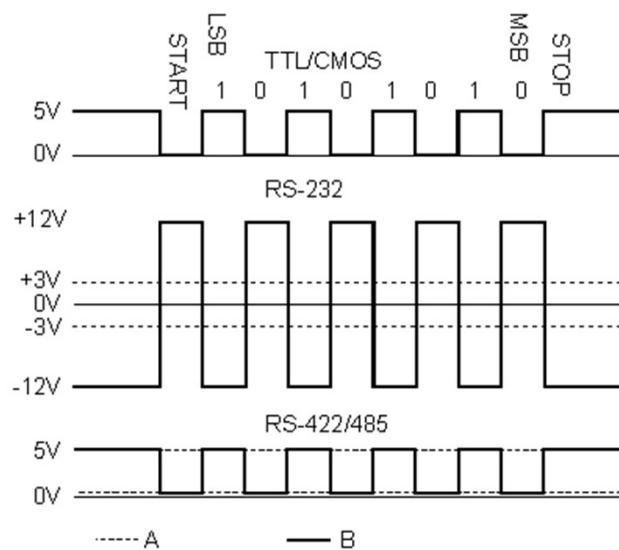
Acceptable TTL gate input signal levels



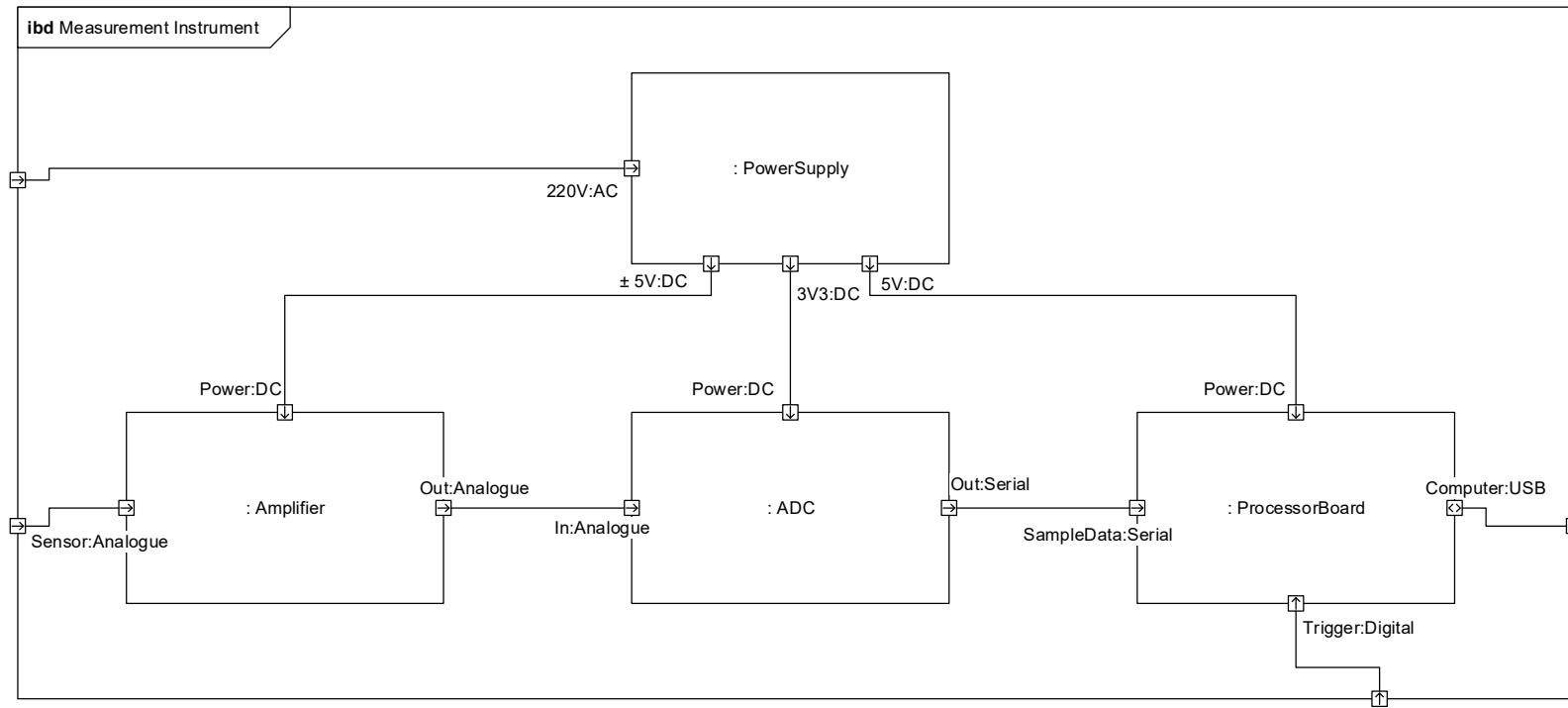
Acceptable TTL gate output signal levels



ASCII "U" = 85 Decimal = 55 Hexadecimal = 01010101 Binary



3. Internal Connections – Ports <name:type>



4-6. Example of block description and ports (Power Supply, Amplifier)

Name of Block	Description of function	Port Name	Type	Port Specification
Power Supply	Converts input AC power to internal DC power supplies	220V	AC	200 – 250 V RMS, 50 Hz Input current limiter of 100 mA
		±5V	DC	Dual Supply Voltage Tolerance ±0.2 V, Max. 250 mA
		3V3	DC	Single Supply Voltage Tolerance ±0.3 V, Max. 250 mA
		5V	DC	Single Supply Voltage Tolerance ±0.2 V, Max. 500 mA
Amplifier	Amplifies sensor input signal <ul style="list-style-type: none"> • 5000 times amplification • Frequency range 0 – 3 kHz • Signal to Noise Ratio better than 65 dBFS 	Power	DC	±5V, Tolerance ±0.3 V, Max. 200 mA
		Sensor	Analogue	Differential Input Signal Voltage Range -100 to +100 uV peak Impedance 50 Ohm
		Out	Analogue	Single Ended Output Signal Voltage Range –500 to +500 mV peak Impedance 500 Ohm

Your turn!

- Specify requirements to **ADC and ProcessorBoard**
 - Specify requirement to all ports with ?

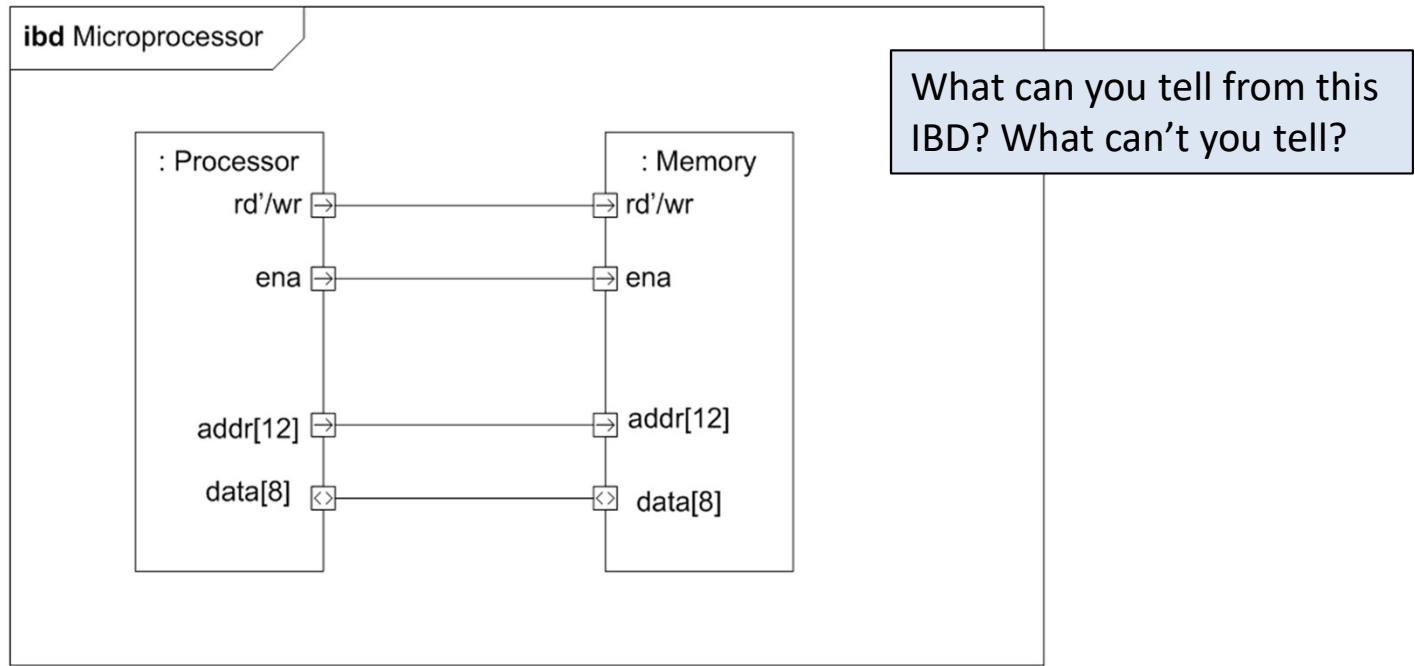
Name of Block	Description of function	Port Name	Type	Port Specification
ADC	Analogue to digital converter <ul style="list-style-type: none">• 8 kHz sample rate• 24 bits sample	Power	DC	?
		In	Analogue	?
		Out	Serial	SPI or I2S
Processor Board	Collects digitized sensor signals and store data in memory when trigger input is high. Possible to transfer sensor data over USB port. <ul style="list-style-type: none">• Memory 1 GByte• Processor ADI BF706	Power	DC	?
		Sample Data	Serial	SPI or I2S
		Trigger	Digital	?
		Computer	USB	USB 2.0

Questions

- Verify that the internal DC power interfaces are correct?
- Can you see any problems with the port specifications?
(Processor Board -> Power:DC - max. 600 mA!)
- Verify that the amplifier output fits with the ADC input?
- Can you see any problems with the interface?
(ADC -> In:Analouge – 5 Ohm!)

Interfaces: Specifying in detail

- SysML interface descriptions using flow specifications are "fine".



- However, at some point, the interface must be described in complete and unambiguous detail.

Specifying in detail: Example

- All information related to timing etc. are absent, so we need a timing diagram to create the HW-SW interface

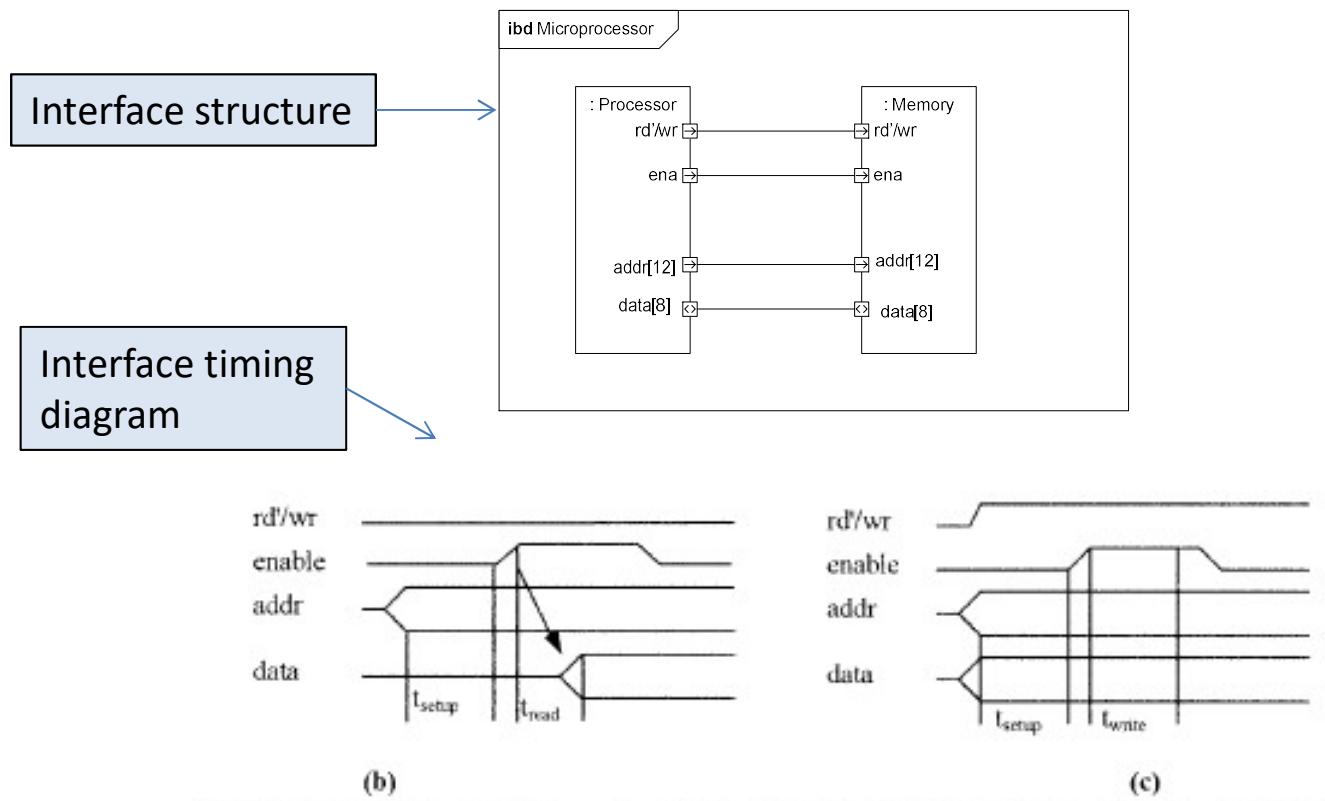
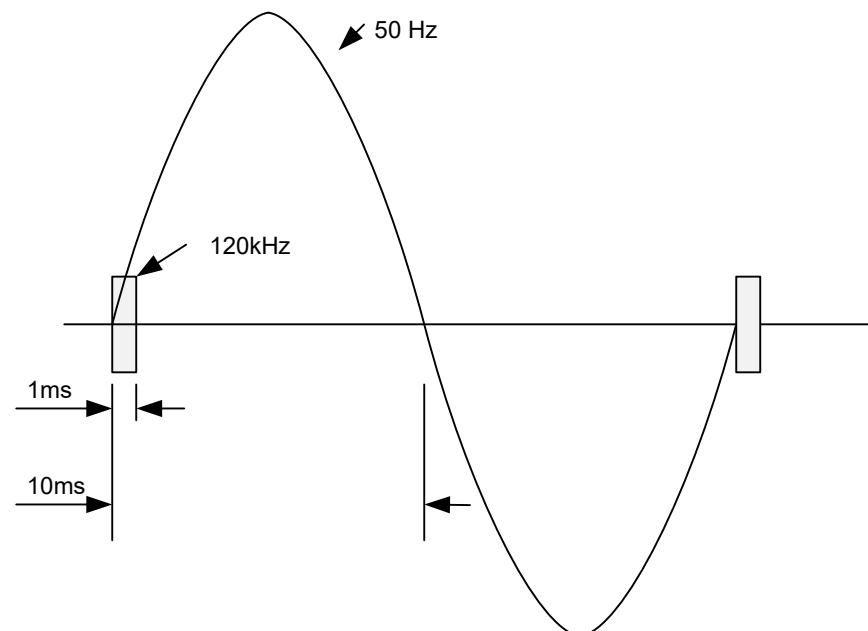


Figure 6.1: A simple bus example: (a) bus structure, (b) read protocol, (c) write protocol.

Specifying in detail: Timing diagram

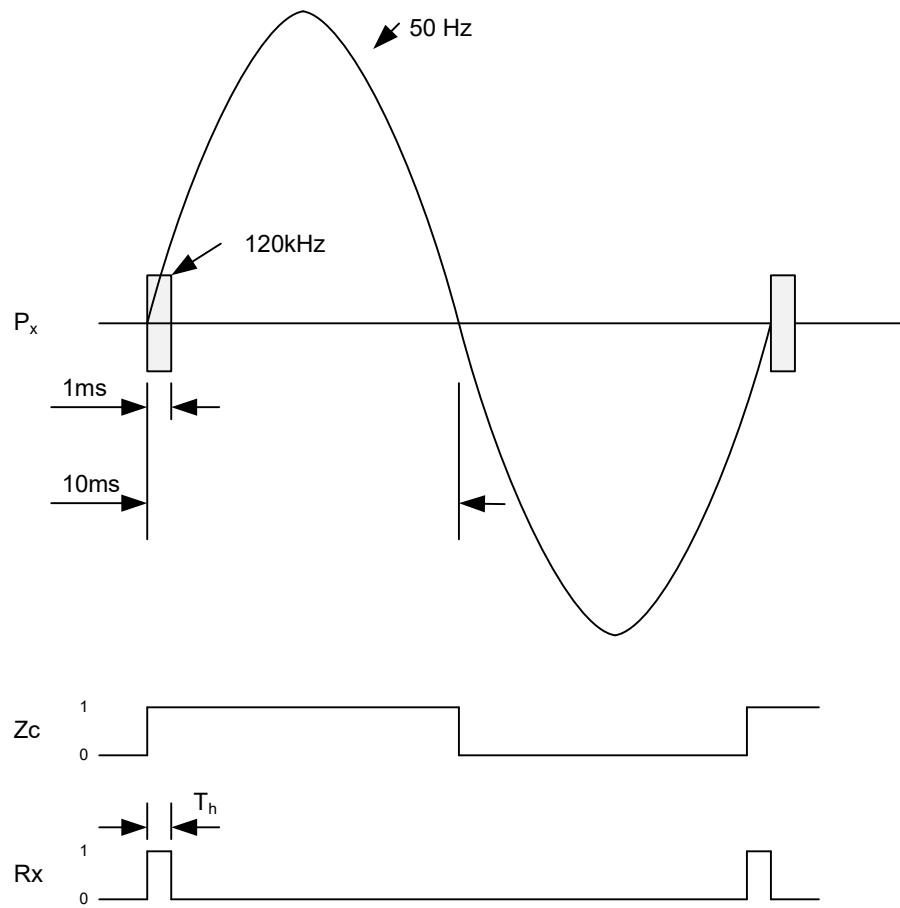
Your turn

- Specify a timing diagram for an X.10 receiver, including:
 - The 50Hz power signal P_x
 - A signal that toggles when a zero crossing in 50Hz signal is detected (Z_c)
 - A signal which is active whenever 120kHz signal is detected (Rx)
 - Requirements for hold time (T_h)

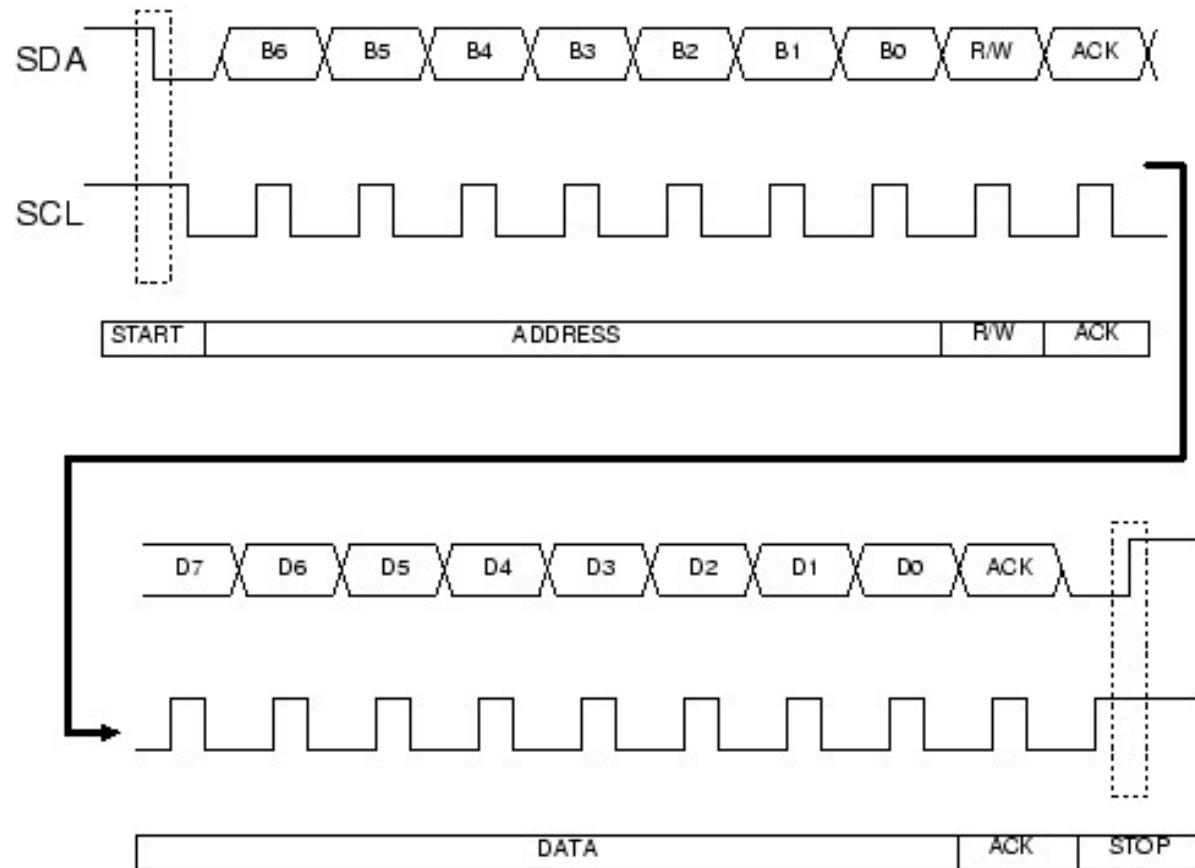


Specifying in detail: Timing diagram

Your turn



Specifying in detail: Another example: I²C



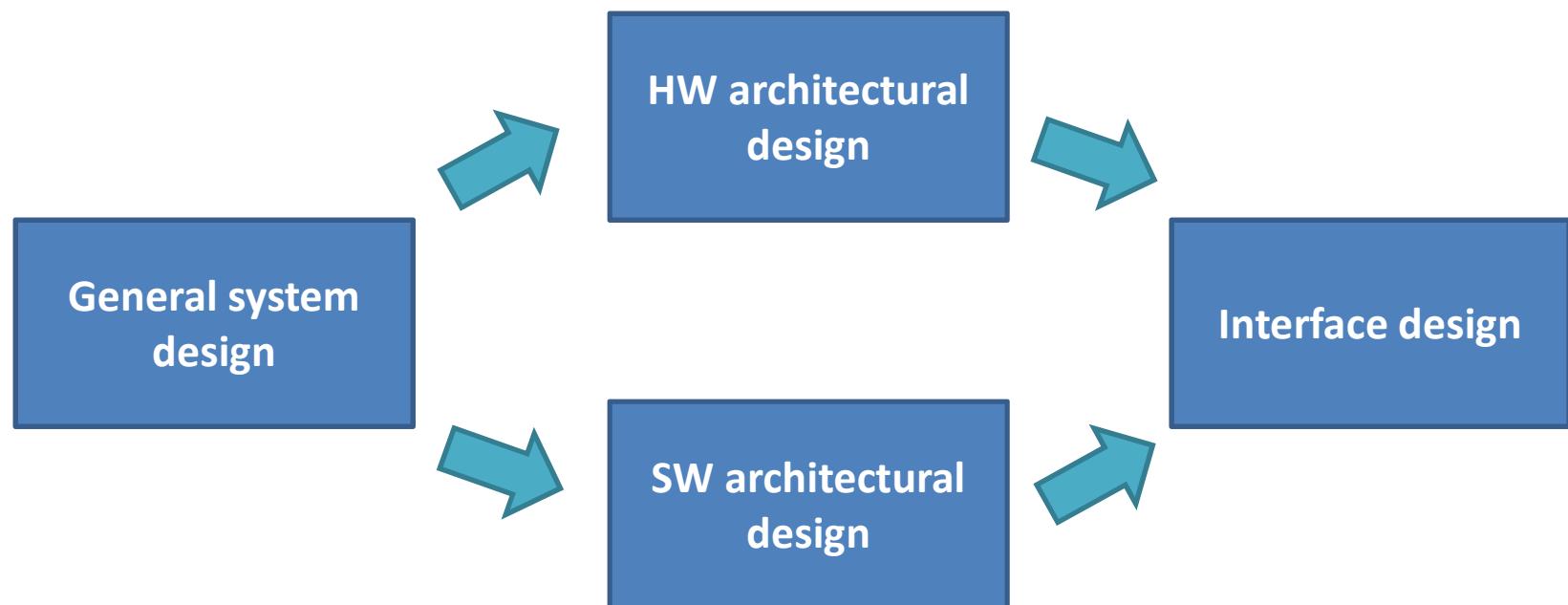
What you can't read from this is...

Specifying in detail: More details

- Specifying a hardware interface in detail will also require a load of other things to be specified:
 - Physical signals and boundaries
 - Inputs and outputs
 - Voltage and frequency limits
 - Standards
 - ...

System sample rates	
Internal sample rate	192 and 176.4 via Dual Wire (optional Digital Card required) and
	96, 88.2, 64, 48, 44.1 or 32 kHz
AIR Masters only	
I/O Connectors	XLR (2 channels AES/EBU in) 3 x RJ45 proprietary TC LINK
Formats	AES/EBU (24 bit)
Word clock input	BNC, 75 ohm, 0.6 to 10 Vpp
Display	2 x 16 character dot matrix
Operation	Menu system / four buttons
Analog input option	
Input connectors	XLR balanced (pin 2+, pin 3-)
Impedance	10/3 k Ohm (Balanced/unbalanced)
Selectable full scale input level	+9, +15, +21, +27 dBu
Dynamic Range	> 113 dB typ. (unweighted), BW: 20-20kHz
THD+N	< -105 dB typ. @ 1 kHz, -3 dBFS
Crosstalk	< -120 dB, 20 Hz to 20 kHz
A to D Conversion	24 bit (Dual bit delta sigma sampling at 4.1/5.6/6.1/6.1 MHz)
AIR Slaves only	
I/O Connectors	2 x RJ45 proprietary TC LINK

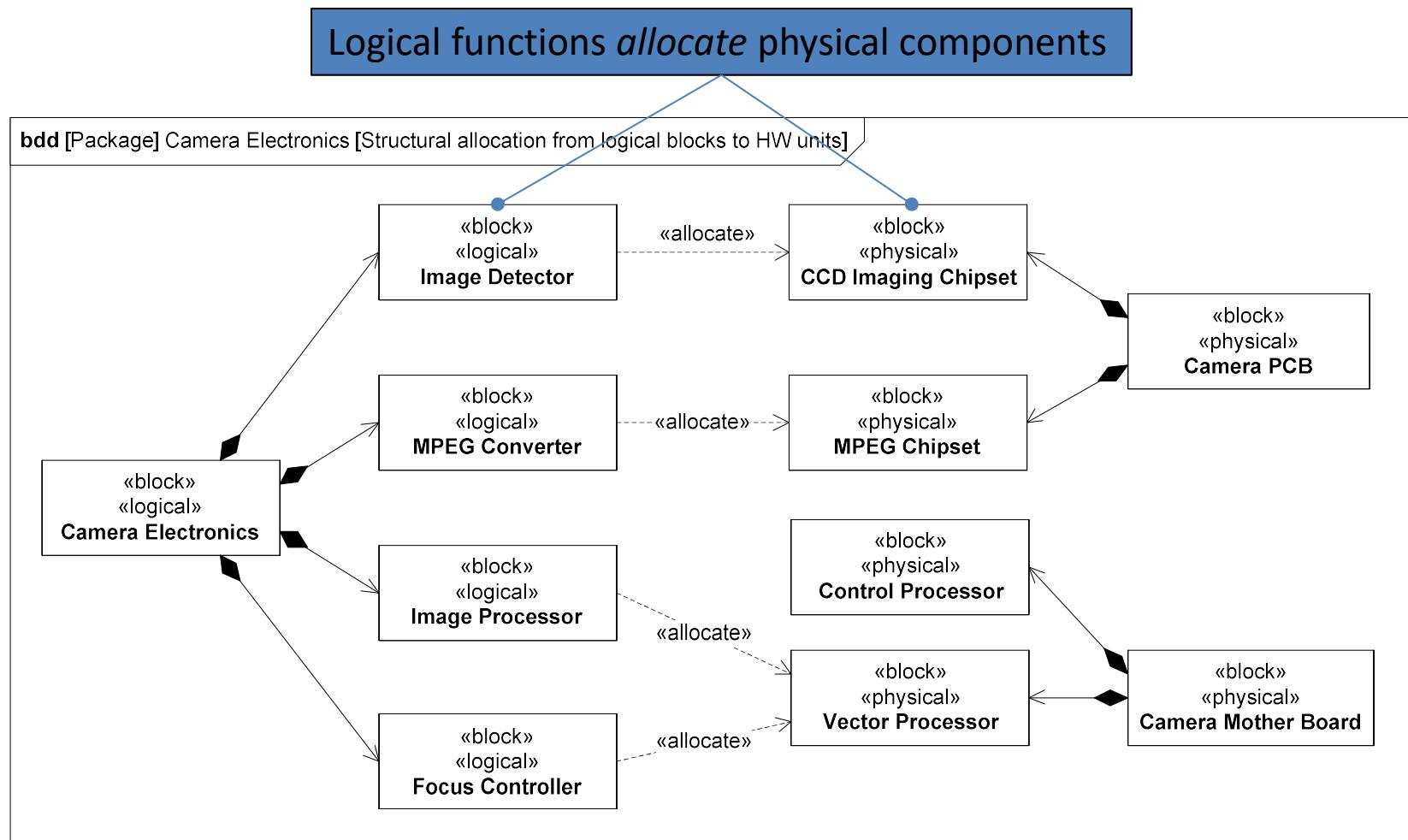
Hardware architectural design



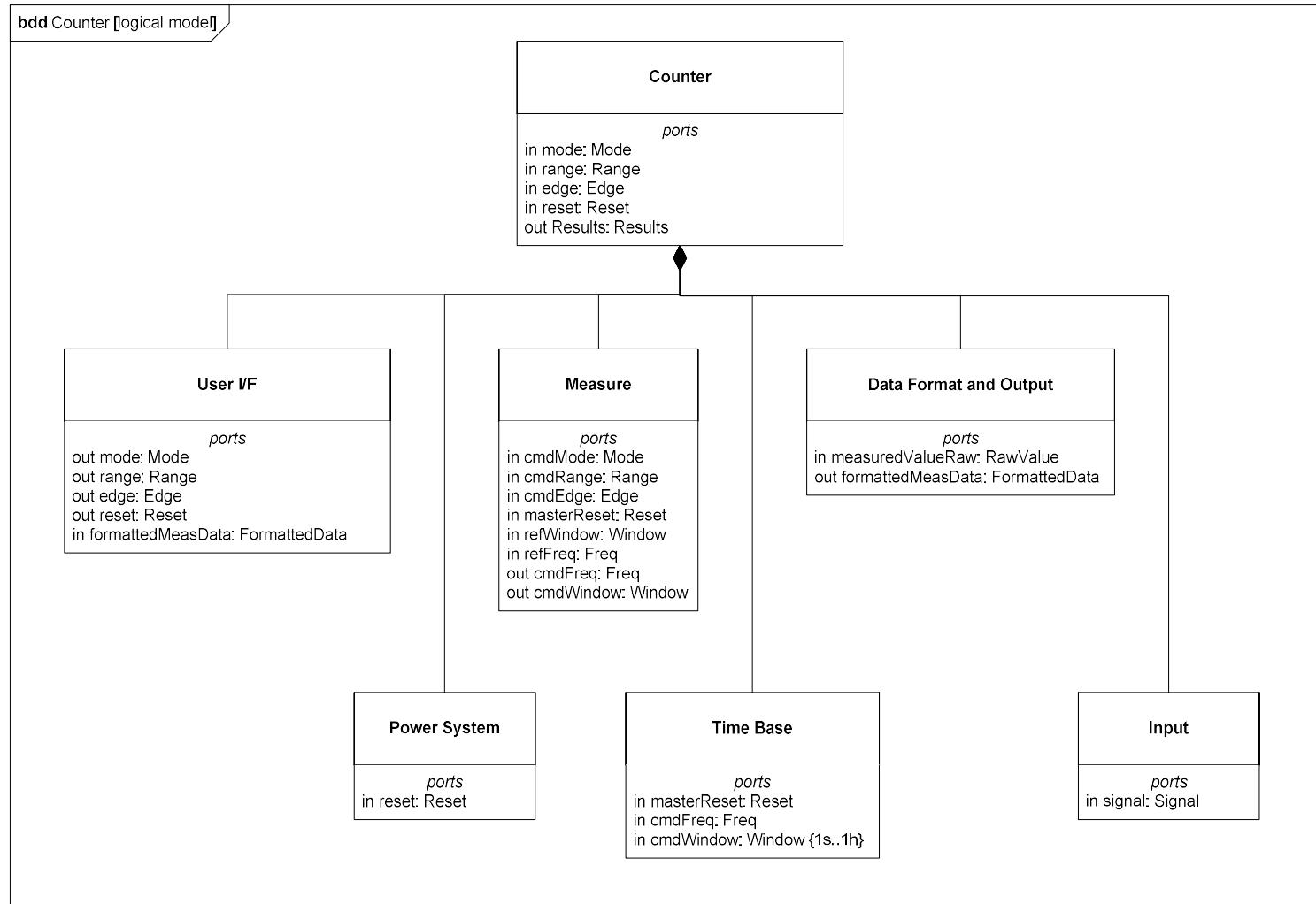
HW architectural design

- An attempt at a "cookbook" for HW architectural design:
 1. Create a *logical model* of the system (logical blocks)
 2. Investigate the *logical* interfaces
 3. Create a *HW model* of the system (physical blocks)
 4. Allocate the logical blocks to the *physical blocks*
 5. Define the *physical* interface between the blocks and to the environment

bdd: Logical to physical



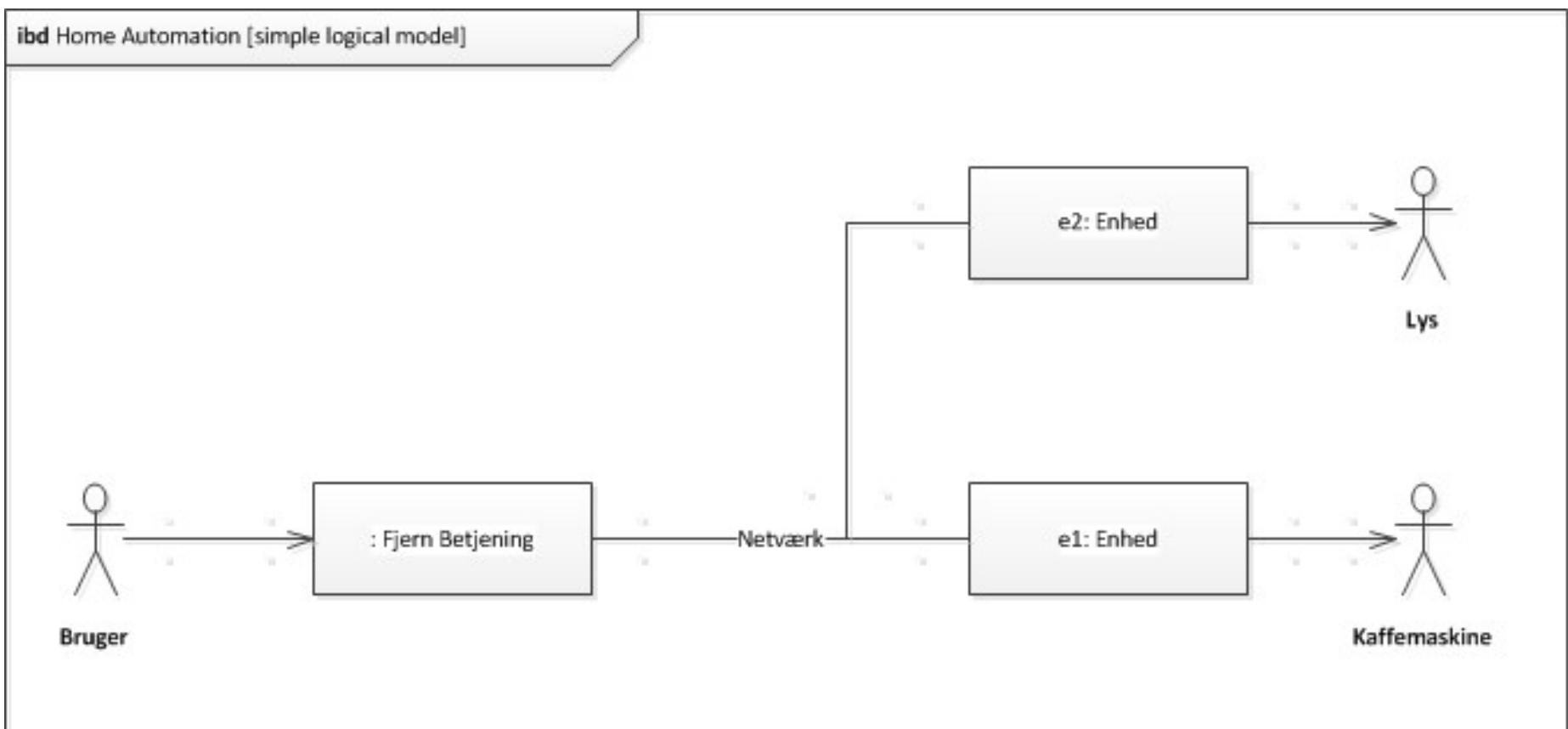
Counter example – *logical blocks and interfaces*



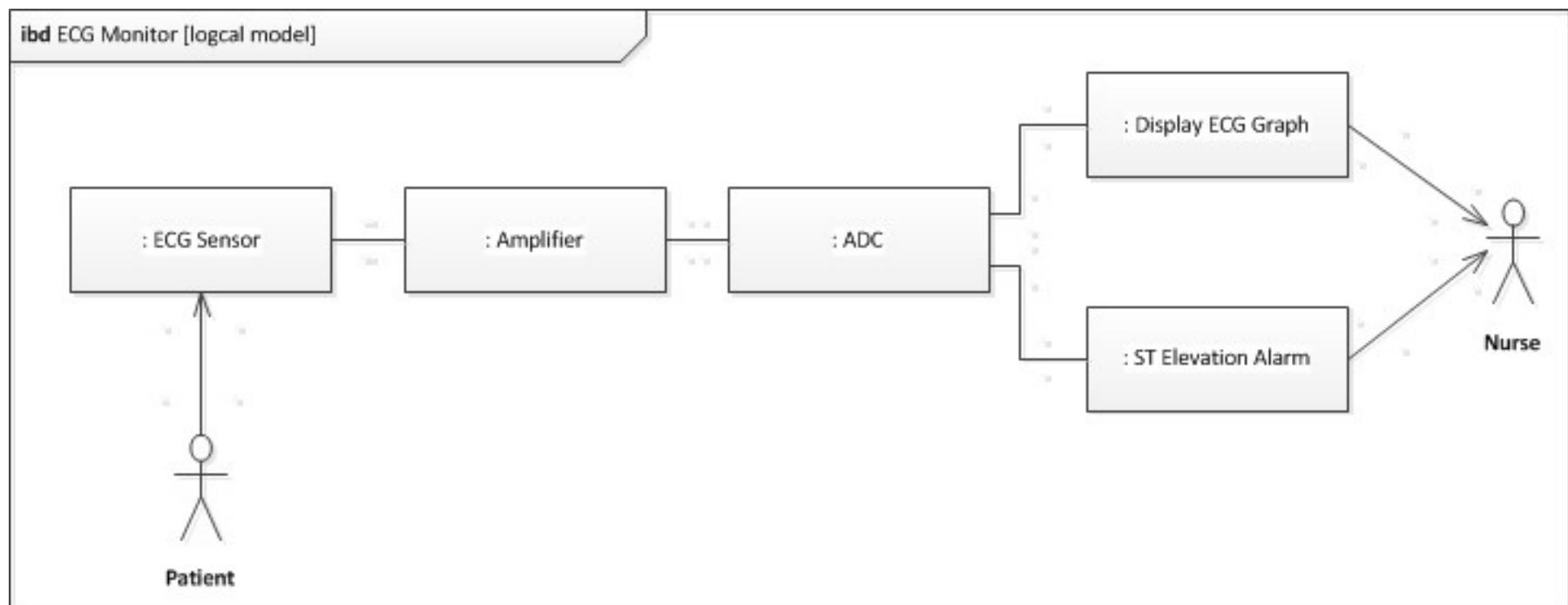
Logical structure of your semester project?

- Spend the few minutes discussing a logical structure for your semester project
 - Logical blocks?
 - Logical system interfaces?
 - Logical internal interfaces?

Home Automation



ECG Monitor

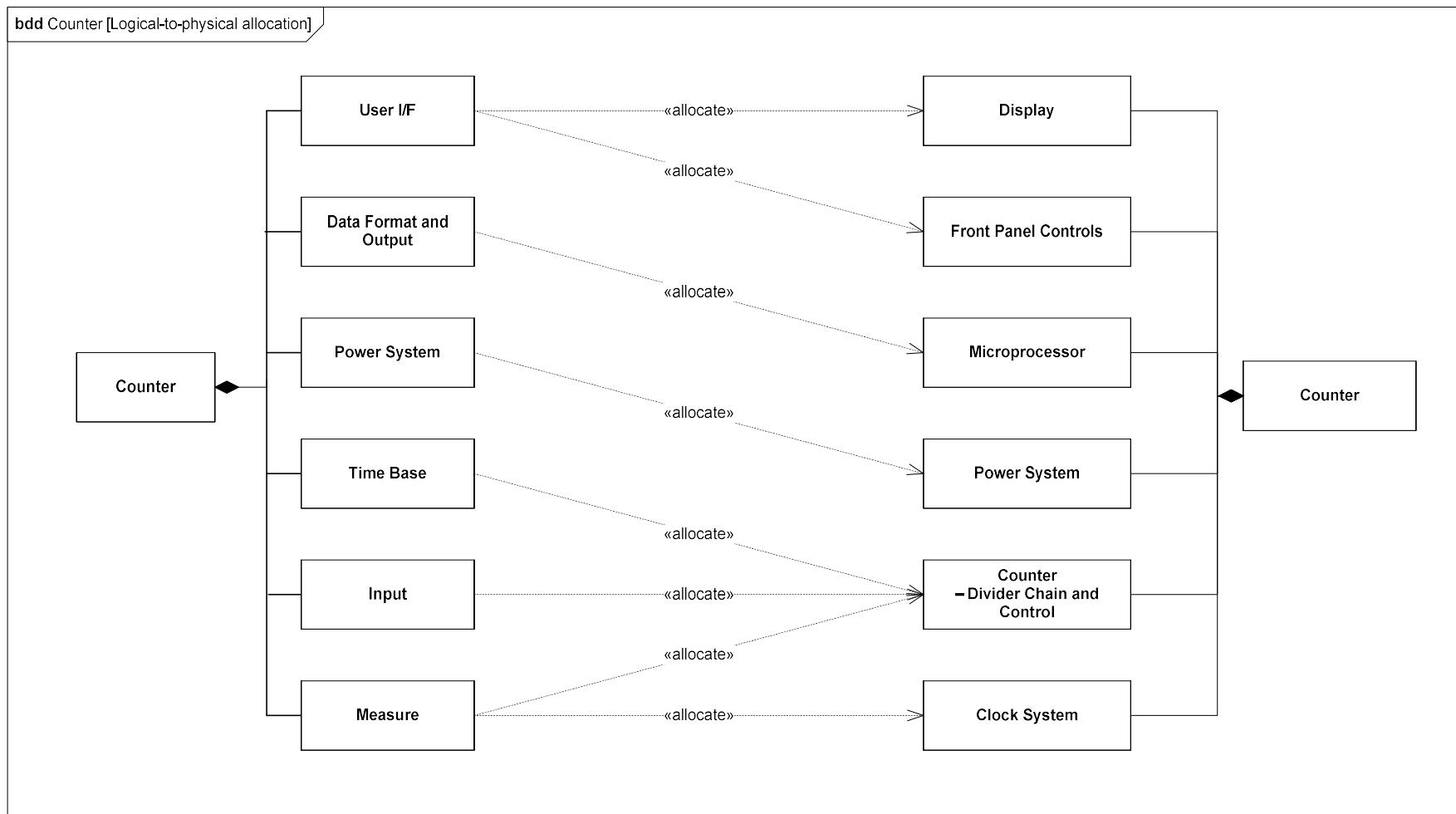


Mapping logical blocks to physical blocks

- From the logical system structure and the requirements¹, we derive a HW architecture with a suitable set of HW blocks
 - Processors, peripherals, buses, etc.
- Then, we (iteratively) allocate the logical blocks , i.e. *functions*, to the physical blocks, i.e. *hardware*

1: And from available, required or desired HW, experience, cost, and a score of other sources...

Mapping logical blocks to physical blocks «allocates»



Physical structure of your semester project?

- Spend the few minutes discussing a pyhsical structure for your semester project
 - Physical blocks?
- Then, discuss how the logical blocks map to the physical blocks you have defined.

Quality Management

Introduction to Systems Engineering
I2ISE

Introduction

- What is *Quality Management*?
- Reviews
- Configuration control
- Subversion/Github

What is a Quality Management?

- Quality Management (QM) is a set of activities performed to ensure that quality is
 - Planned
 - Controlled
 - Assured
 - Improved
- A couple of activities/tools: Reviews , version control and Subversion/Github

Errors

- Test is good in finding errors implemented by mistake
- Review and configuration management is to prevent errors being implemented at all!

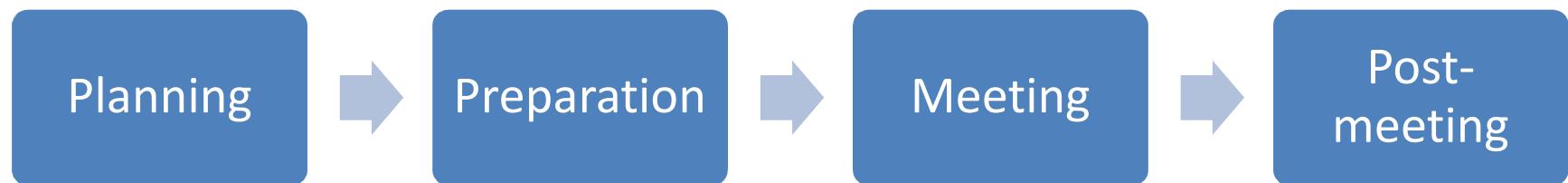
QM - reviews

- A *review* is the activity of looking through proposed work prior to its' commitment.
- You can (practically) review anything
 - Code, diagrams
 - Documents
 - Processes (e.g. *Scrum retrospective*)
 - ...
- For reviews to be effective, they must be *structured*

Reviews - goal

- The *goal* of a review: **Release of the item under review**
- **How is the goal supported by the review?**
 - By *constructive criticism* on the review item
 - By finding potential quality problems
 - By ensuring corrective action is taken
- **How is the goal obstructed by the review?**
 - By using it to prove you're smarter than everybody else
 - By establishing yourself as a leader
 - By pressing your preferred solution

Reviews – phases



Reviews – planning

- **What should be *planned*?**
 - *What* are we going to review?
 - Who are the reviewers?
 - *When, where* and *how* will the review take place?
 - How is the document and supplementary material *distributed*?
 - Who will *chair* the review meeting?



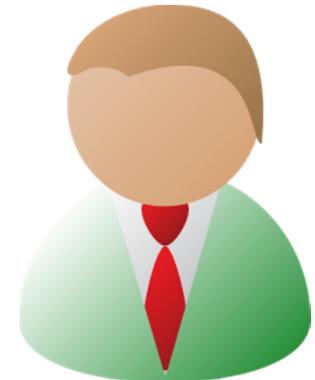
Reviews – preparation (owners)

- How should the *document owners* prepare?
 - Make sure the review item is frozen for review
 - Practicals: Book meeting room, ensure AV equipment is present, ...
 - Distribute review item etc. to review opponents along with agenda and venue
 - Define the desired *roles* (review leader, secretary)
 - ...
- How should the *reviewer* prepare?
 - Read the review item thoroughly – distribute roles (spelling, diagrams, ...)
 - Prepare overall and detailed critique
 - Find relevant sources etc.



The document owners' roles

- Review *leader* (chair) ensures...
 - agenda for review is issued and kept
 - everyone is heard during review
 - focus during the meeting
 - delegation of action items
- Review *secretary* ensures
 - distribution of review item, agenda, ...
 - creation and distribution of Minutes of Meeting (MoM)
 - changes are captured
 - review item is updated iaw. review (assigns action items)
 - sufficient coffee!



The reviewers' roles

- Some ground rules for the reviewers
 1. Be prepared
 2. Be friendly and empathic
 3. Behave
 4. Point to issues, not solutions
 5. Avoid discussions on style (taste)
 6. Stick to the subject matter



Review: The agenda

- The review meeting is best conducted along an agenda, e.g.
 1. Welcome, opening remarks (*chair*)
 2. General remarks (*opponents*)
 3. Detailed run-through of review item(*opponents*)
 4. Conclusion (*chair, secretary*)
 5. Actions to be taken - rework, corrections (*all*)
 6. Closing remarks (*all*)



Reviews – post-meeting

- Make sure Minutes of Meetings are distributed ASAP
- Make necessary corrections to the review item
- Decide on the action to take now: Call new review or release the review item



Configuration Management

- Another important activity is *Configuration Management*
- The purpose of configuration management is to
 - Capture the *baseline* of a given (version) of a product
 - Ensure that a given product can be re-created from scratch

Baseline!

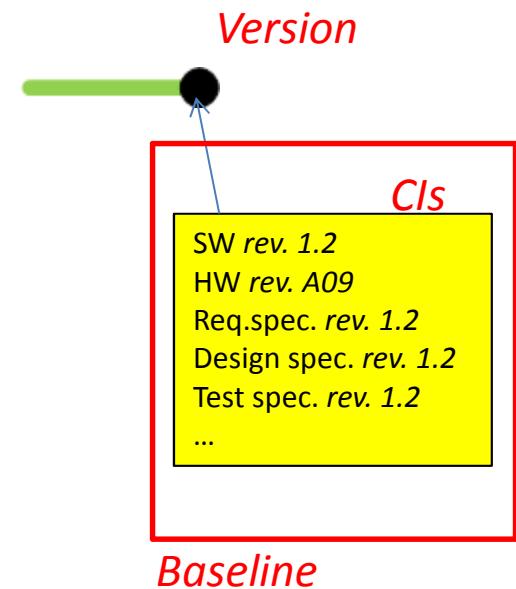
- Agreed-to information that defines and establishes the attributes of a product at a point in time and serves as the basis for defining change
- Goals
 - to handle changes related to baseline
 - to ensure documentation and product artifacts fits together
 - Where do I find documentation that fits with product release x.y.z?

Configuration Management



Configuration Management

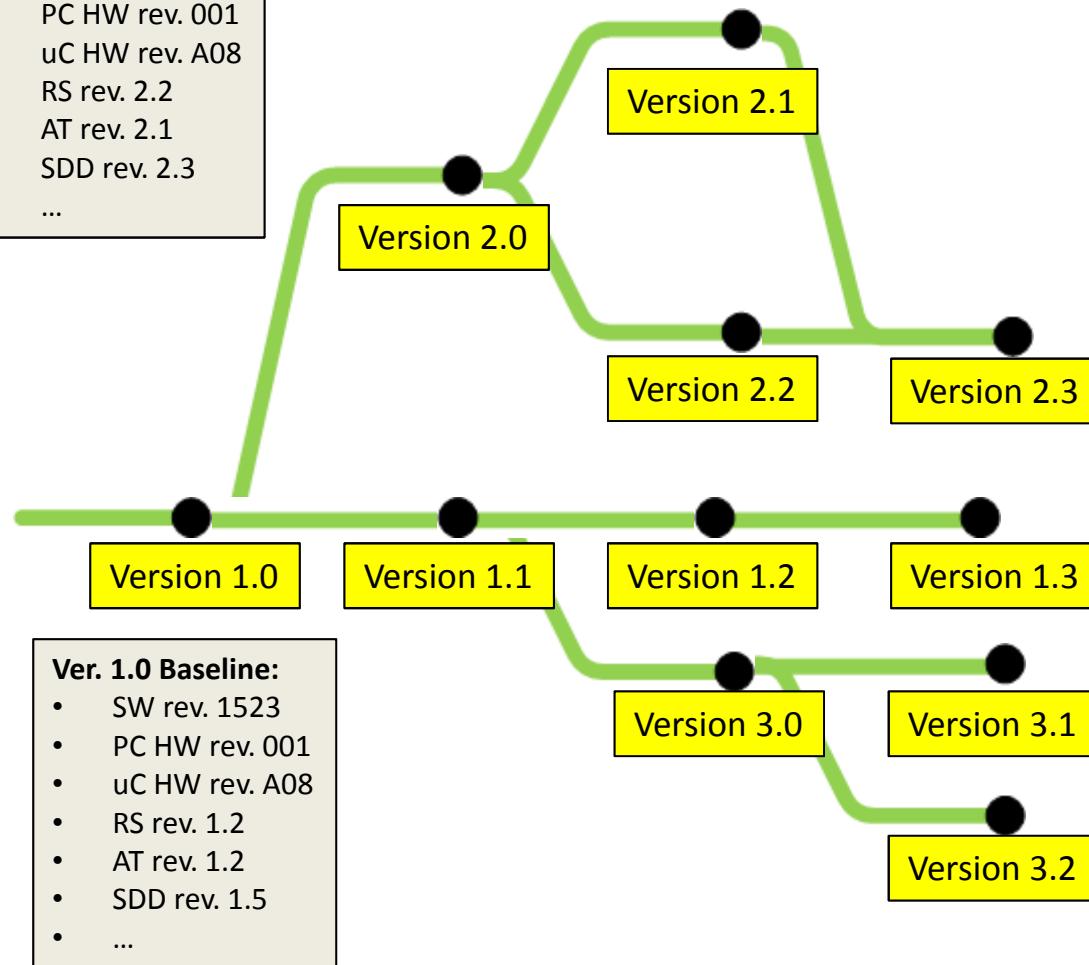
- What you need is *configuration management*
 - A way to control what *configuration items (CIs)* (documents, HW, SW, tools, ...) in what *revision* that apply to a given *version* of the product
- Collectively, this is known as a *baseline*
 - Typically defined by a top-level document that calls out the different versions of CIs
- The baseline must contain everything necessary to rebuild the version from scratch



Configuration Management

Ver. 2.0 Baseline:

- SW rev. 2301
- PC HW rev. 001
- uC HW rev. A08
- RS rev. 2.2
- AT rev. 2.1
- SDD rev. 2.3
- ...

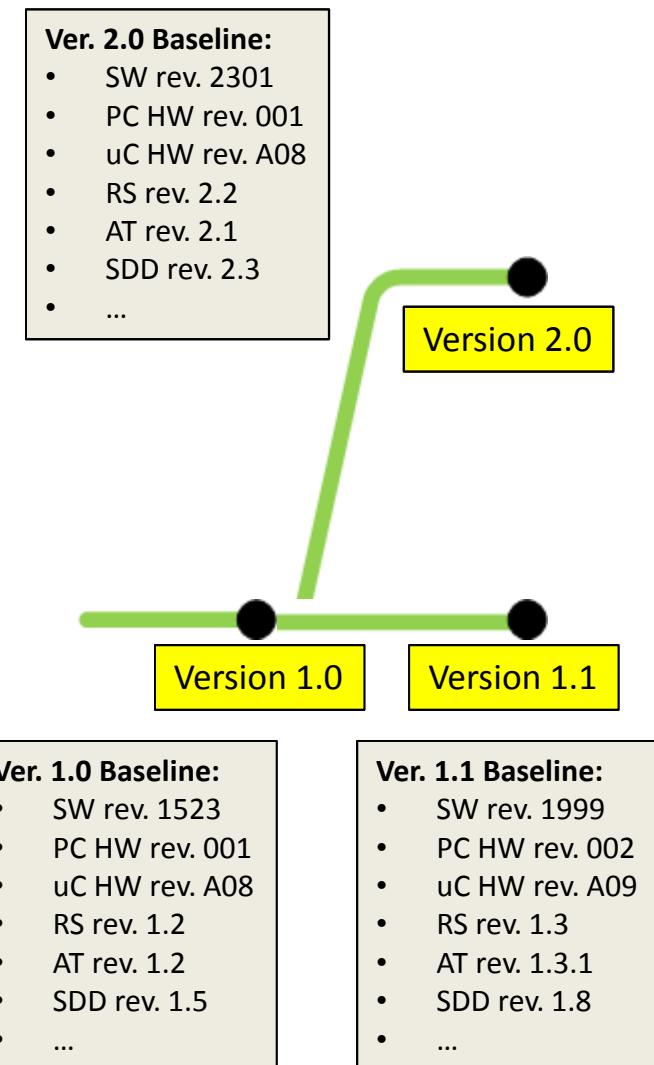


- Versions are defined
- Each *version* defines its own *baseline*



Configuration Management

- Note that...
 - *Versions* are defined at planning-time
 - *Baselines* are defined when a version is completed (released)



CM – version control systems

- There exists a variety of version control systems
 - Systems that allow you to get, update, submit, track and revert revisions of a document/source file
 - Examples: Git, Subversion, PVCS, Dropbox, ...
- Indispensable for a number of reasons:
 - Concurrent work
 - Versioning and revision control
- Example: Subversion (<http://subversion.tigris.org/>)

CM – Subversion

- *History:* All earlier revisions of a document are maintained
- *Availability:* Documents are securely accessible in a single place
- *Sharing:* Several people can contribute to a document

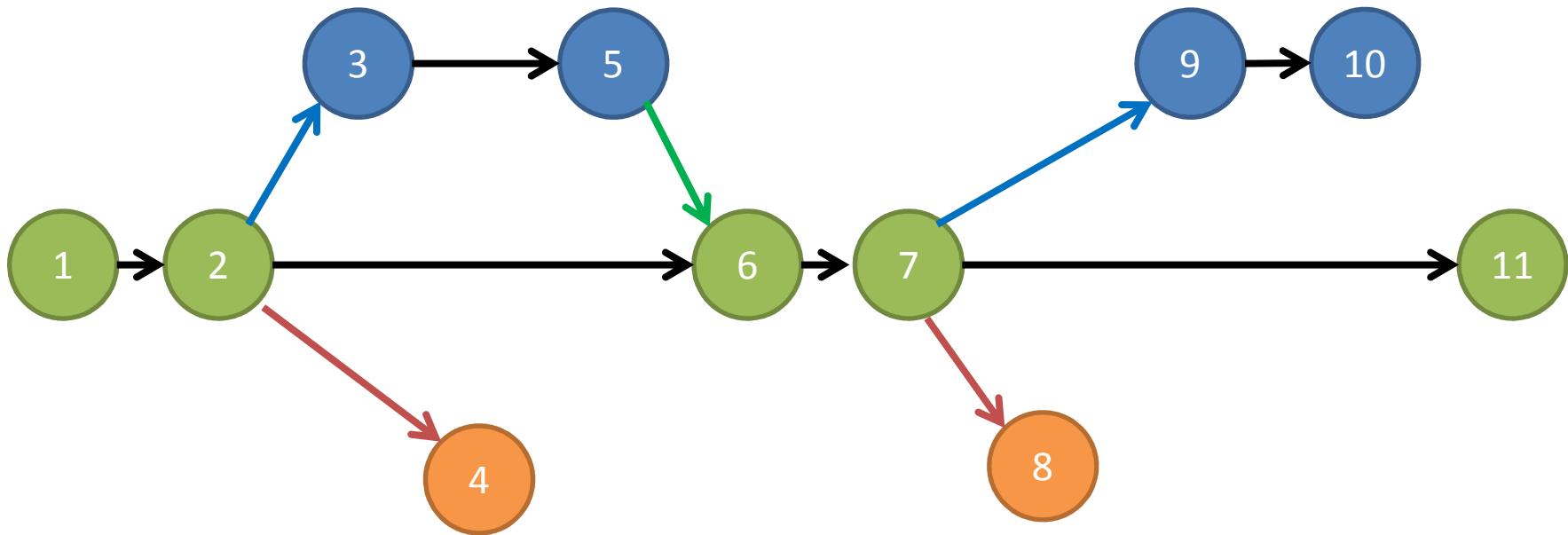
CM – Subversion basics

- Somewhere, someone (maybe you?) have created a *repository*
 - Ask Google how if you're interested
- First (and only once), you **check out** the repository
 1. Perform an **SVN checkout** operation – this will give you a *working copy* of the repository
- Before you start work, you **update** your working copy
 1. Perform an **svn update** operation
- You work on your working copy. When happy, you **commit** your changes
 1. Perform an **svn commit** operation on a file or folder

CM – Subversion basics

- If you make any new files, you can *add* them to the repository
 - Perform an **SVN add** operation
 - Note: Nothing changes until you *commit* your working copy
- If you regret your current changes you can *revert* them
 - Perform an **SVN revert** operation
- Other stuff:
 - Version history
 - Diff
 - Branching
 - Tagging
 - Merging

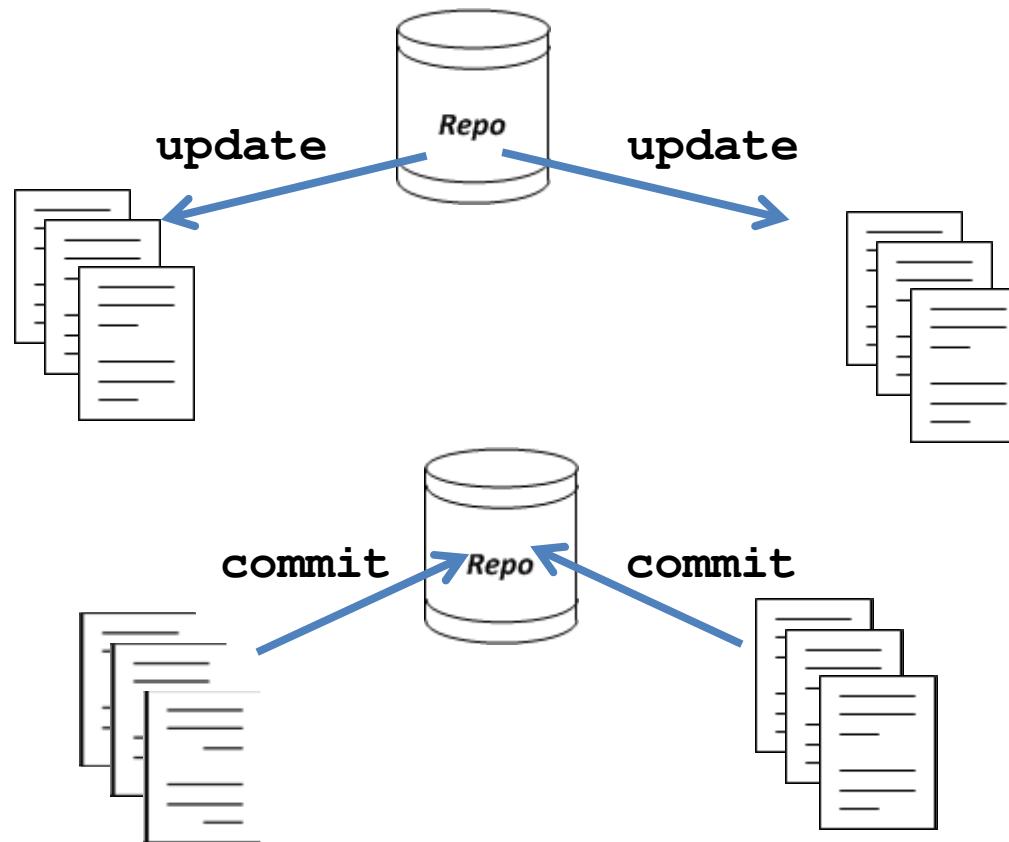
CM – Subversion



- Commit
- Tag
- Branch
- Merge

CM – Subversion: Multiple users

- Multiple users checkout/update → no problem!
- Multiple users commit → *usually* no problem!



Git - source code management system

- git clone (**checkout** a repository)
- git add (**add** new file to repository)
- git commit (**commit** changes to repository)
- git push (send changes to remote repository)
- git pull (**update** local repository to newest commit)

Github

- Web-based Git repository hosting
 - Web-based graphical interface (PC and mobile)
 - Incorporates bug tracking, feature request and task management
- <https://en.wikipedia.org/wiki/GitHub>
- Server: <https://github.com>
 - Desktop Client: <https://desktop.github.com/>
 - Repository:
 - <https://github.com/kimbjerge/education-ise>

Project Management

Introduction to Systems Engineering
I2ISE

Introduction

- Why do we need project management?
- Groups vs. teams
- Team roles and activities
- Maintaining a team – group AC
- Project planning and estimation
- Risk management

Why project management?

- Discussion: Your experiences on project work?
 - What is important for success?
 - What could be possible obstructions?



Project teams,
roles and activities



Groups vs teams

- What is the difference between a *group* and a *team*?

GROUP

Individual accountability

Meet to share information

Focus on *individual* goals

Produce *individual* work products

Define *individual* roles, responsibilities, and tasks

Concerned with *individual's* outcome and challenges

Purpose, goals, approach to work shaped by *manager*

TEAM

Individual and *mutual* accountability

Meet to discuss, make decisions, solve problems, planning

Focus on *team* goals

Produce *collective* work products

Define individual roles, responsibilities, and tasks *to help team do its work*

Concerned with *team* outcome and challenges

Purpose, goals, approach to work shaped by team leader *with team members*

Nobody is perfect – but a team can be

"A group is a matter of balance. Good team-members has strengths and competencies which cover the needs of the group – without doubling strengths and competencies already present. Strengths possessed by some team-members can compensated weaknesses in others. Nobody is perfect – but a team can be."



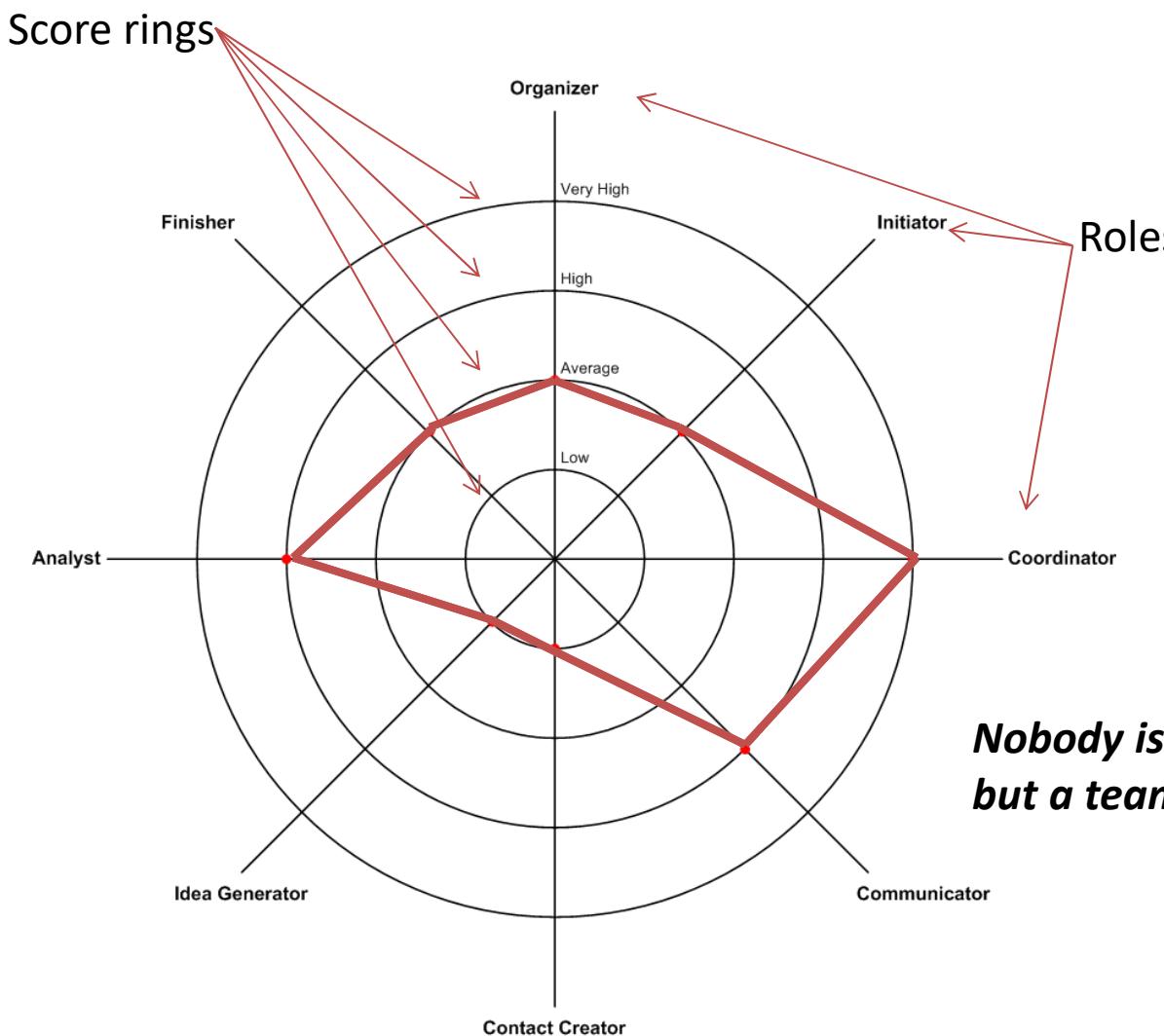
Dr. Meredith Belbin - <http://www.belbin.com/>

Belbin team roles

Type	Positive qualities	Allowable weaknesses
Organiser	Organizing, disciplined, turns ideas into practical actions. Hard working.	Less flexible Skeptical to unproven ideas
Analyst	Sober, strategic. Sees all options. Judges accurately. High intellect.	Lacks drive and ability to inspire others.
Idea generator	Dominating, high intellect. Creative, imaginative, unorthodox.	Ignores routine questions. Too focused on the special problems.
Finisher	Mindful, anxious. Finds errors and omissions. Delivers on time.	Inclined to worry unduly. Reluctant to delegate.
Coordinator	Stable, dominant. Good chairperson, clarifies goals, promotes decision-making, delegates well.	Can be seen as manipulative. Off loads personal work.
Communicator	Stable. Low dominance. Co-operative, mild, perceptive and diplomatic. Listens, averts friction.	Indecisive in crunch situations.
Contact creator	Stable, dominant, enthusiastic, communicative, develop contacts.	Over-optimistic. Loses interest once initial enthusiasm has passed.
Initiator	Impatient, dominant, challenging, dynamic, thrive on pressure.	Prone to provocation. Offends peoples feelings.

<http://www.persontests.dk/personlighedstests/belbin/>

Belbin team roles - Belbin chart



Traditional team roles

- Traditionally, in a team there are some well-known *roles*:
 - Project manager
 - Team members
 - Secretary
- All members assume (at least) one role
- With a role comes *tasks* and *responsibilities*

Team roles – project manager

- What is the project manager's tasks?
 - Manage expectations - internally (team) and externally (stakeholders)
 - Seek information – from team and stakeholders
 - Conduct planning - tasks, plans, manning, preferably with the team
 - Keep information level up - internally and externally
 - Display team culture and behaviour
 - Be the team lightning rod / shield
 - Report to steering committee
 - ...



Team roles – the team members

- What requirements are fair to have to team members?
 - There's no "I" in "team"
 - Responsible
 - Tolerant
 - Loyal to decisions
 - Self-reliant and self-driving
 - Honest
 - Display "due diligence"

Teams go through *phases*

Forming

team begins to discuss the task(s) and orientate towards a work plan

Storming

conflicts and tensions emerge - different work styles, expectations, ethics, ...

Norming

mutual trust and effective ways of working emerge

Performing

effective work patterns are producing the required results

Adizes speech on "What is a leader?"

Leadership part 1:

<http://www.youtube.com/watch?v=47laMl35kOk&feature=related>

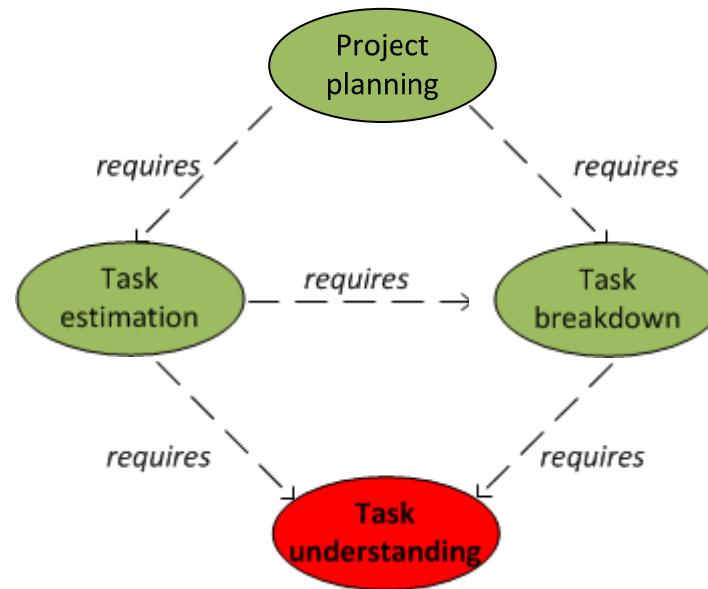
Leadership part 2:

<http://www.youtube.com/watch?v=tEIABc1Wbb4&feature=relmfu>

Project planning

Project planning

- To have a successful project, you will need a *plan*



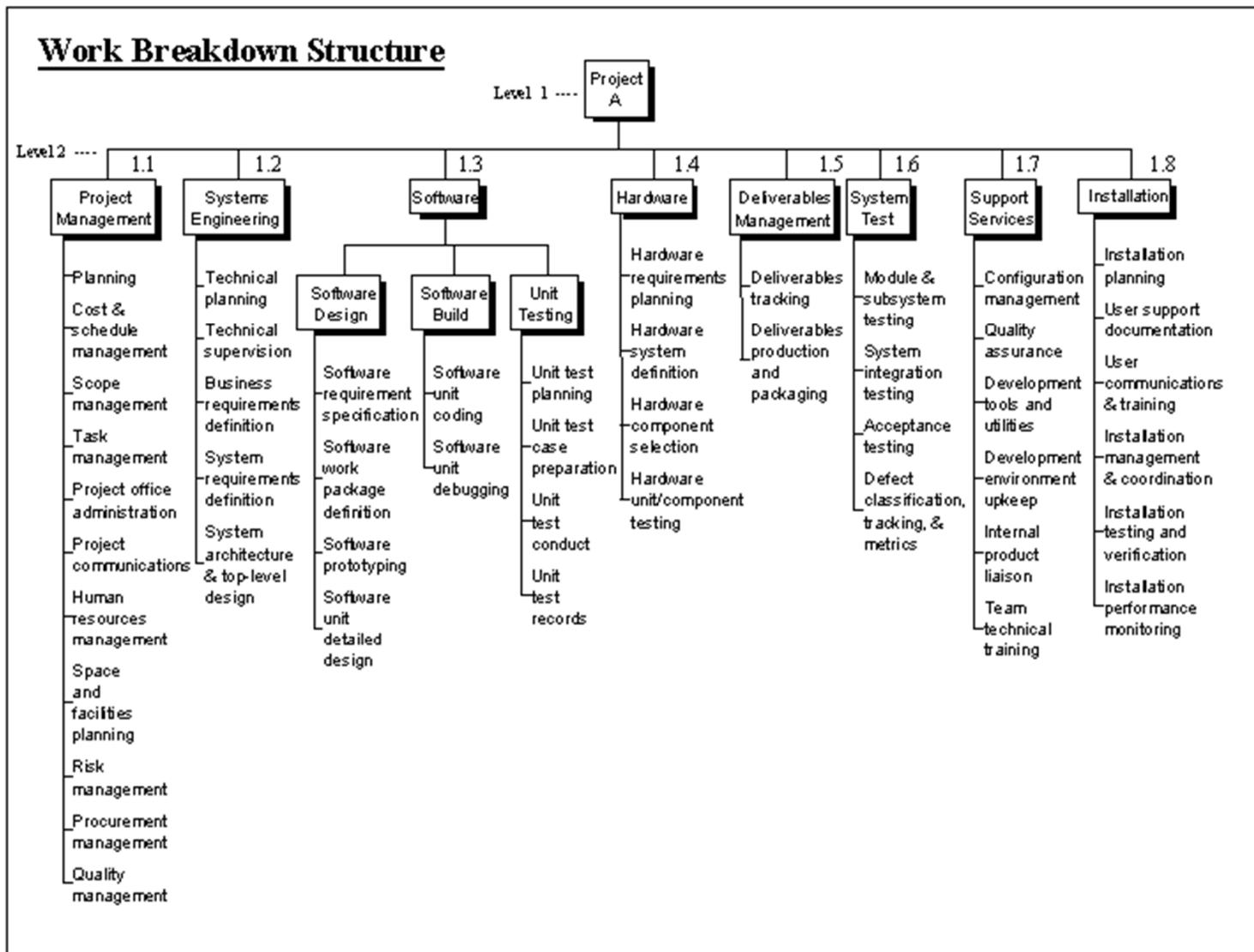
- Thus, to *plan*, we first need to *understand what to plan!*

Planning activities

Project planning is a ***continuous*** activity

- **Initially**
 - Break project down into manageable *work packages*
 - Identify *activities* and *milestones*
 - Make *estimates* (*Estimated Time to Complete* (ETC))
 - Allocate *resources*
 - Create the plan itself
- **Continuously**
 - Monitor project status and progress
 - Monitor time spent/remaining, compare with milestones
 - Adjust plan/scope of milestones, etc.

Project Work Breakdown Structure (WBS)

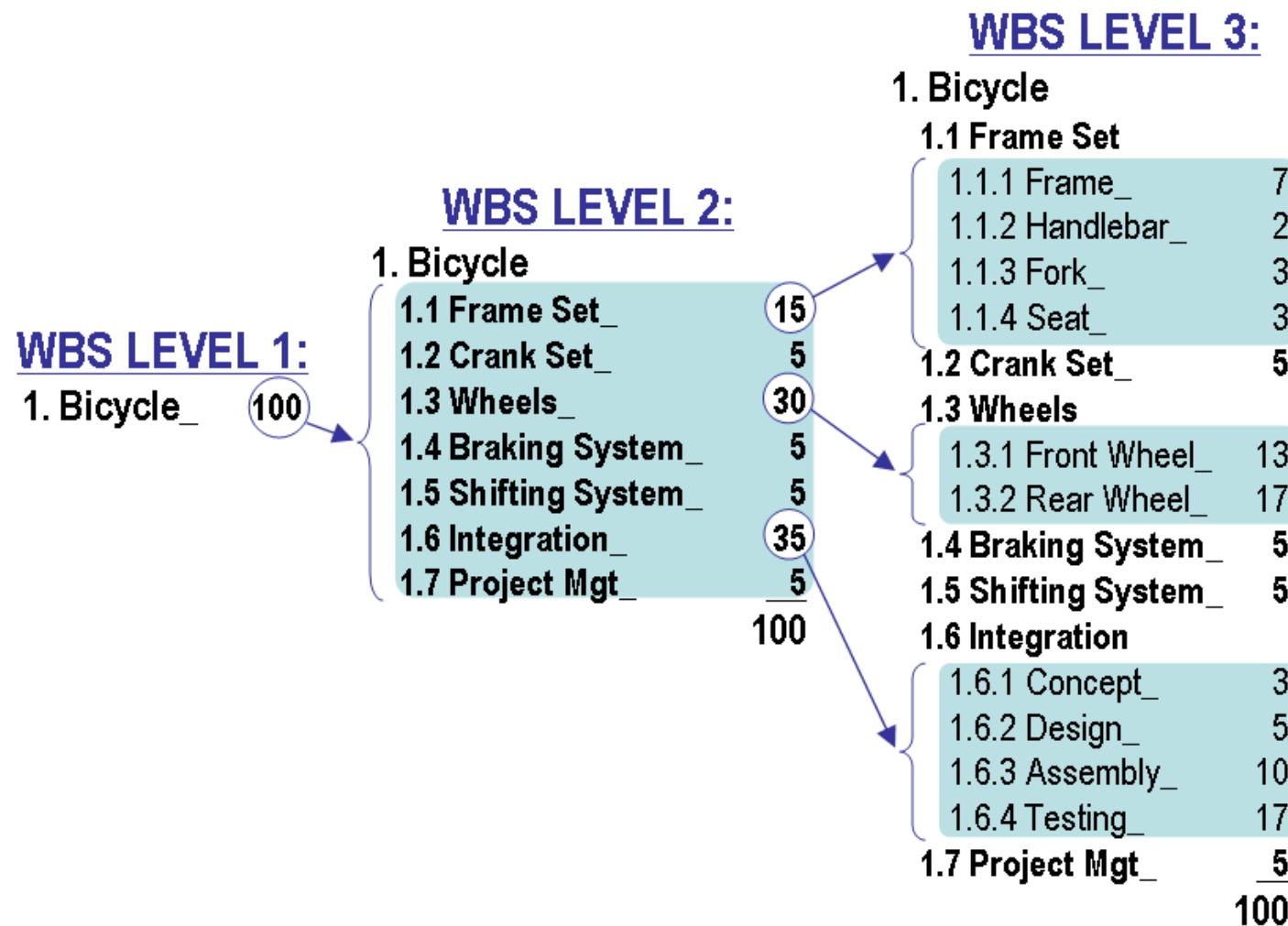


Source: <http://itsadeliverything.com/got-a-wbs-for-your-agile-project-sure-of-course>

Planning – WBS

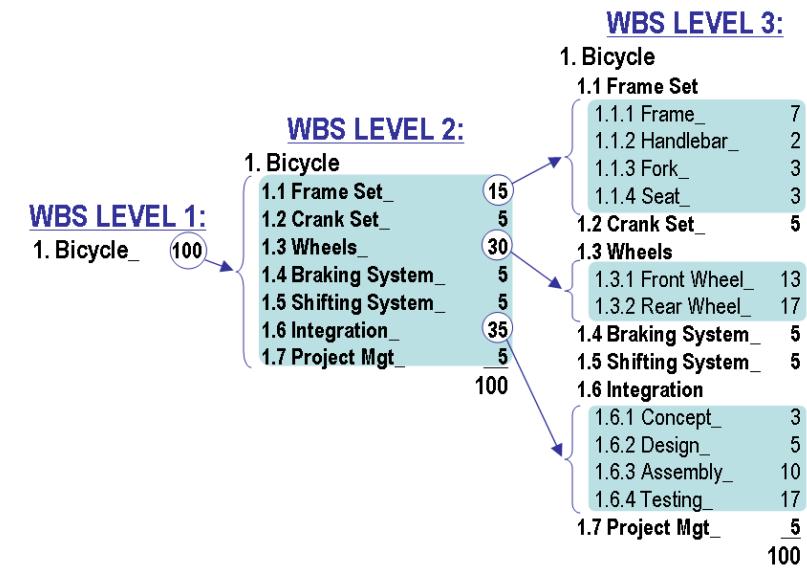
- The work in a project can be broken down in a *Work Breakdown Structure* (WBS)
 - A tree structure containing ever-finer divisions of work
- The WBS leaves should be manageable, well-defined, “estimatable” pieces of work
 - *Terminal elements or work packages (WPs)*
- The WBS is the basis of further planning, e.g. time, cost, manpower, dependencies, ...

Planning – WBS



WBS design principles

- 100% rule (recursive)
- Plan *outcomes*, not *actions*
 - This does *not* have to be physical products



Example

Introduction to Cartoon Heros (I2ICH1) project:

"Give a description on the three classical cartoon heroes, Superman, Batman, and Spiderman. Compare the three and conclude who would win if they got into a fight"



Example

Exercise 1: A WBS for the project

(Think planning, writing, reviewing, etc.)

"Give a description on the three classical cartoon heroes, Superman, Batman, and Spiderman. Compare the three and conclude who would win if they got into a fight"



I2ICH project: Example WBS

WP	Task Name
1	Plan detailed contents
2	Create document template
3	Write contents
4	Create artwork
5	Review and corret report
6	Finish report
7	Hand in report
8	Project Management

I2ICH project: Example WBS

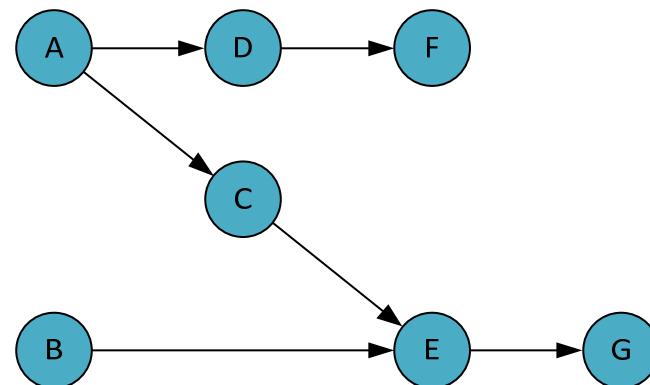
WP	Task Name
1	Plan detailed contents
1.1	Introduction
1.2	Presentation of characters
1.2.1	Spiderman section
1.2.2	Superman section
1.2.3	Batman section
1.3	Comparison
1.4	Conclusion
2	Create document template
3	Write contents
3.1	Introduction
3.2	Presentation of characters
3.2.1	Spiderman section
3.2.2	Superman section
3.2.3	Batman section
3.3	Comparison
3.4	Conclusion
4	Create artwork
4.1	Front page
4.2	Spiderman
4.3	Superman
4.4	Batman
5	Review and correct report
5.1	Review
5.2	Corrections
6	Finish report
6.1	Print contents
6.2	Print front page
6.3	Collect front page and contents
6.4	Bind report
7	Hand in report
8	Project Management

Planning

- Once the project is broken down, you can start to *estimate* and *schedule* your work
- One way to do this:
 - List your WBSs
 - Estimate time to complete (e.g. ETC = $\frac{P+4*N+O}{6}$)
 - Determine dependencies
 - "*C cannot start before A and B is complete...*"
 - Determine "critical path"
 - The path which, if delayed, delays the project as a whole

WBS, duration and predecessors

Activity	Predecessor	Time estimates			Expected time
		Opt. (O)	Normal (N)	Pess. (P)	
A	—	2	4	6	4.00
B	—	3	5	9	5.33
C	A	4	5	7	5.17
D	A	4	6	10	6.33
E	B, C	4	5	7	5.17
F	D	3	4	8	4.50
G	E	3	5	8	5.17



Your turn!

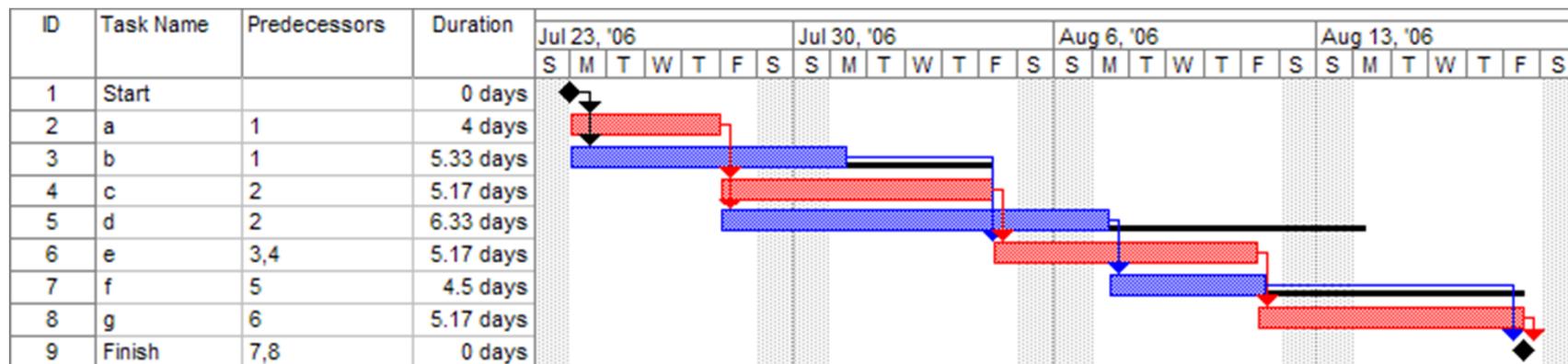
Exercise 2: Give some estimates on durations and determine dependencies for your WBS

"Give a description on the three classical cartoon heroes, Superman, Batman, and Spiderman. Compare the three and conclude who would win if they got into a fight"



Planning – Gantt chart

- With estimates in hand, you can do a *Gantt chart* to show dependencies, duration etc. of tasks
 - Graphical overview
 - Critical paths, milestones, etc.



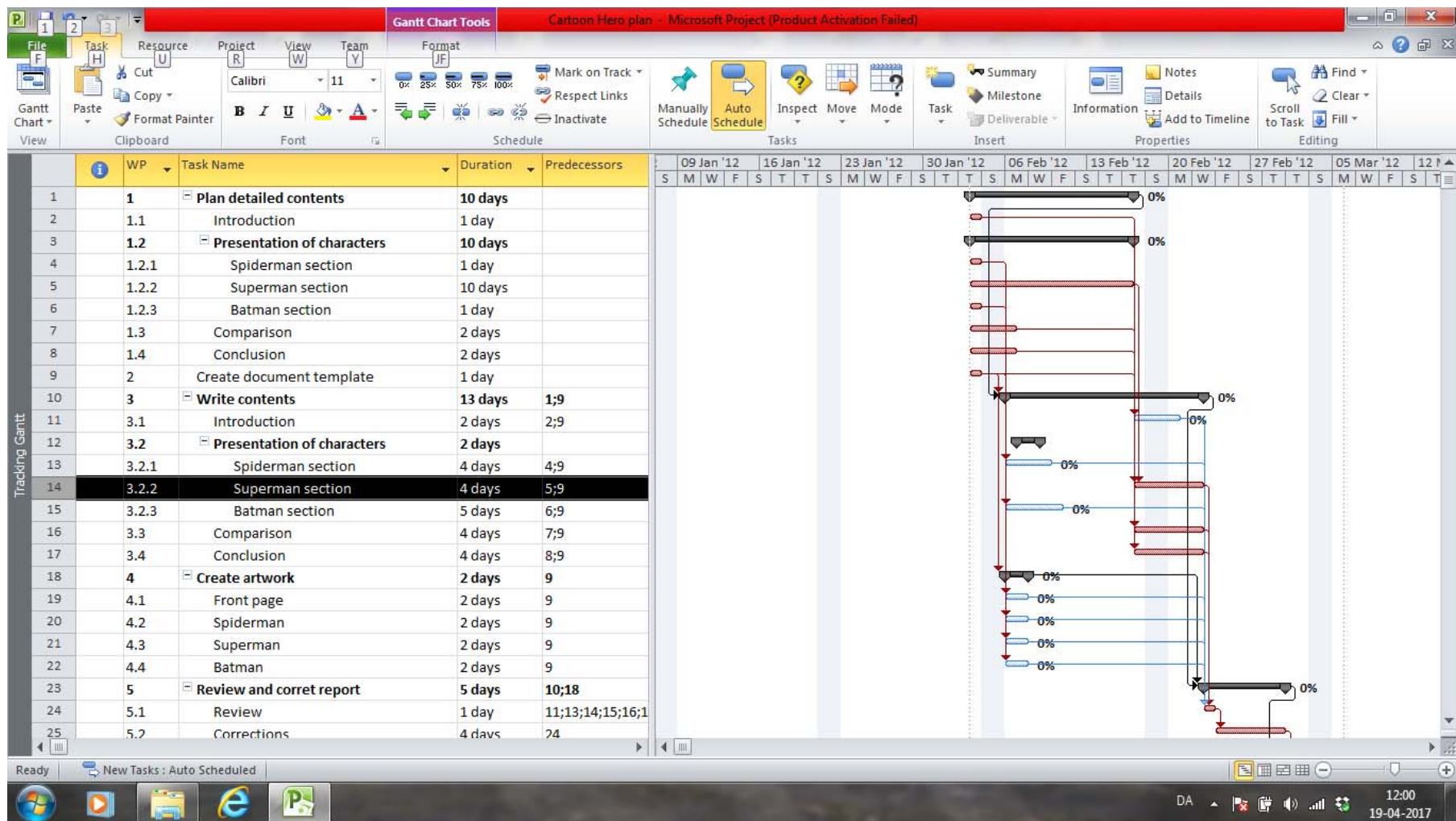
I2ICH project: Gantt chart

Exercise 3: Use the result of Exercise 2 to create a Gantt chart for your project

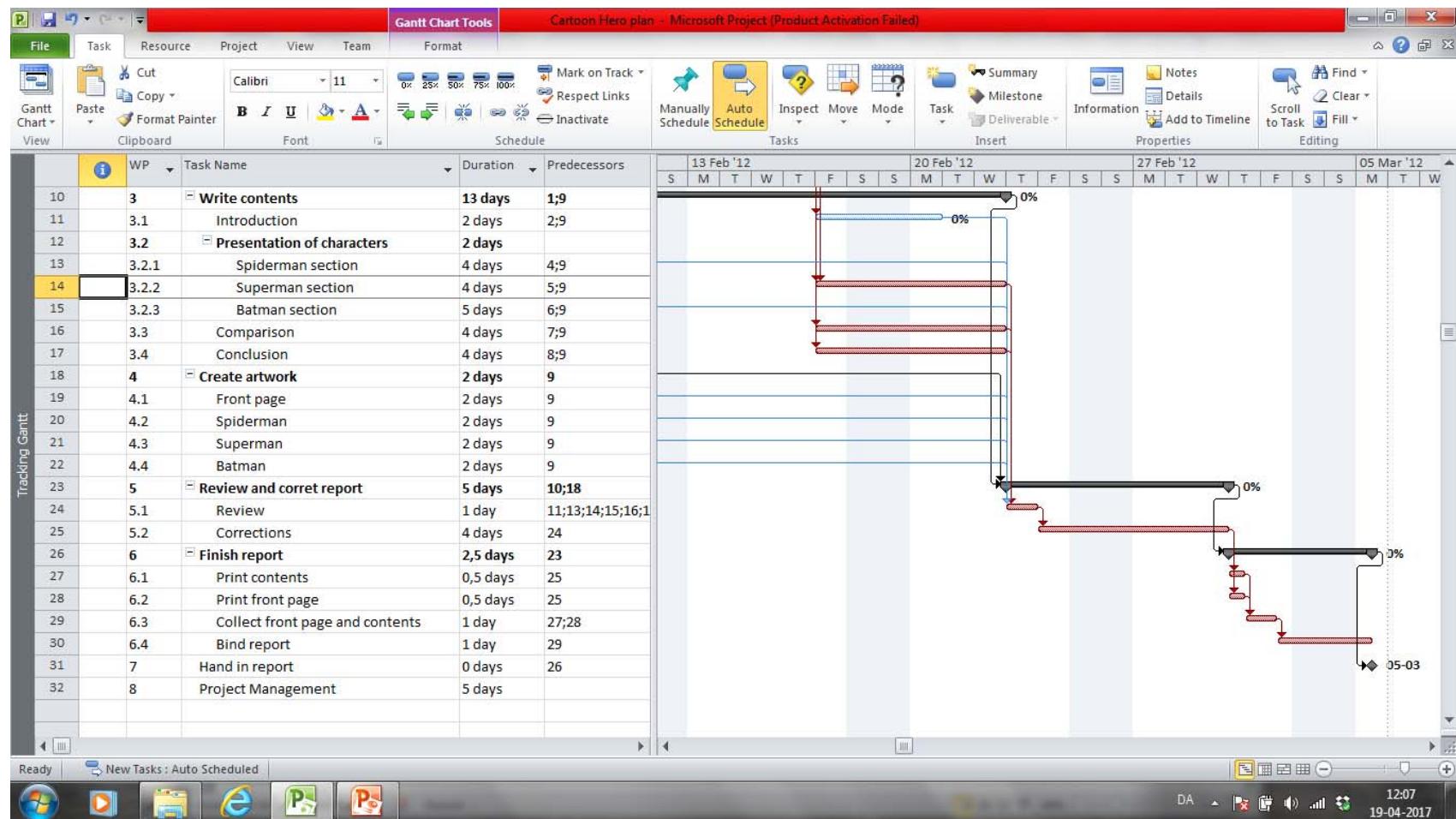
"Give a description on the three classical cartoon heroes, Superman, Batman, and Spiderman. Compare the three and conclude who would win if they got into a fight"



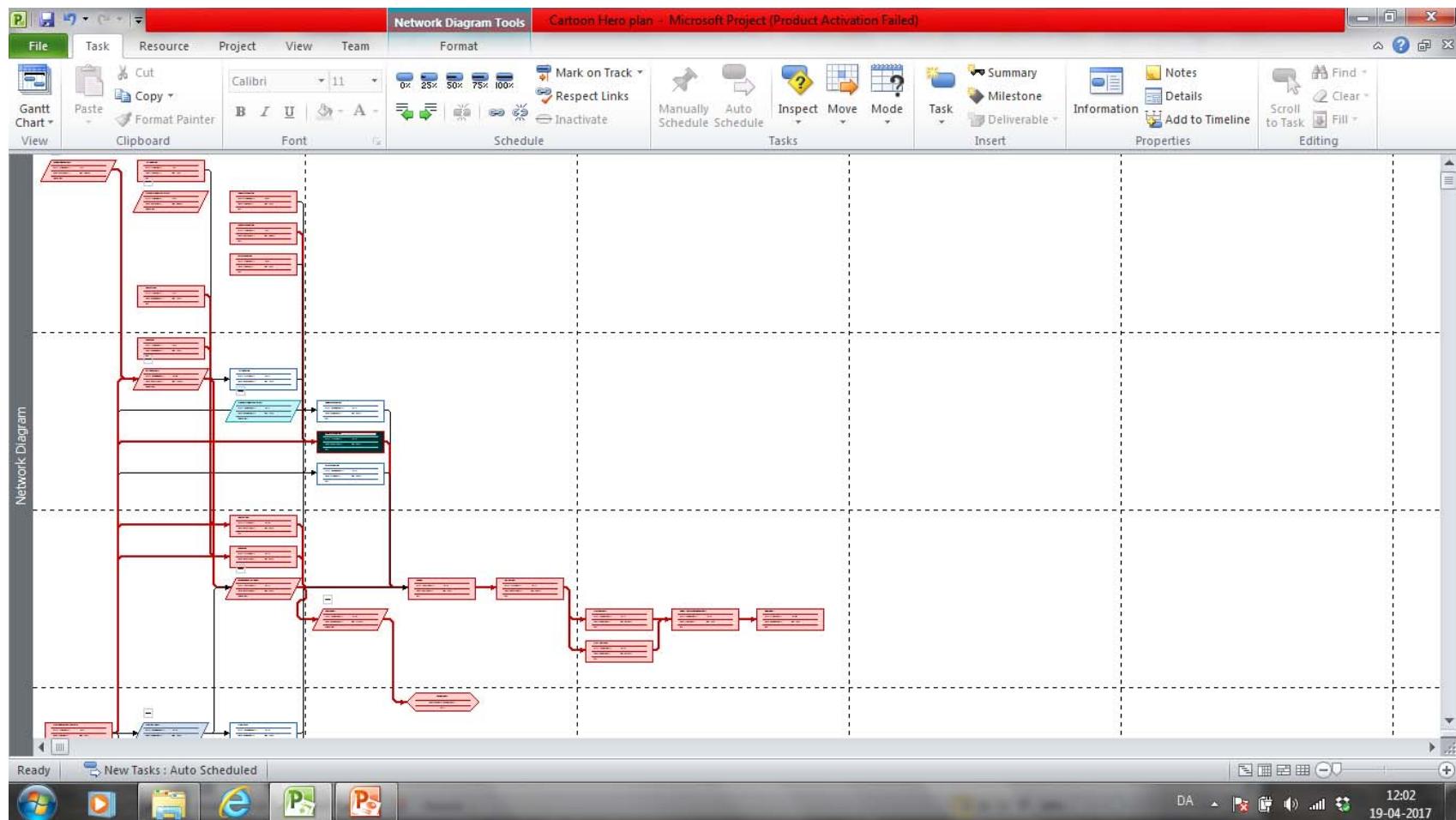
Gantt (1) – MS Project



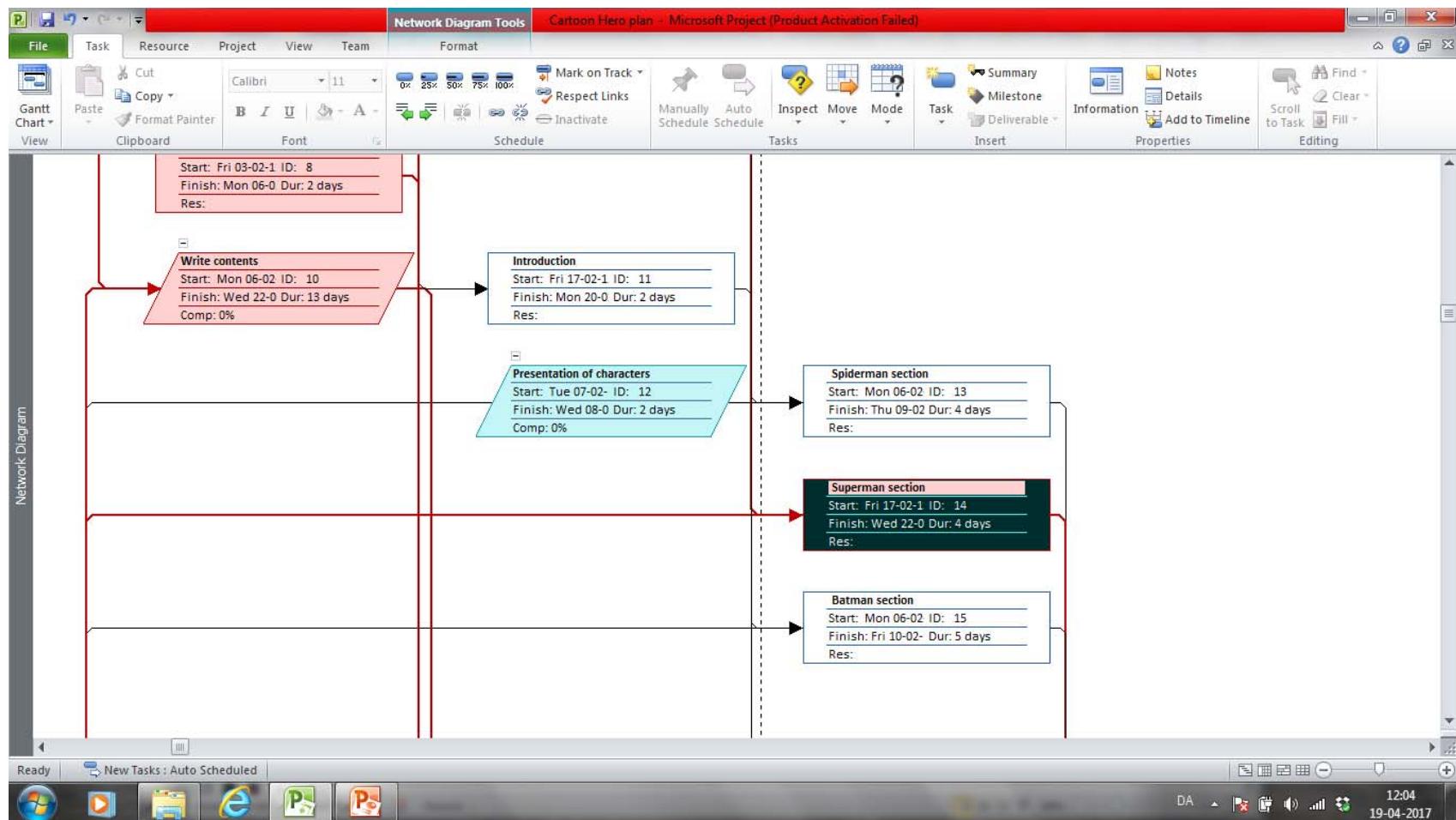
Gantt (2) – MS Project



Network (1) – MS Project



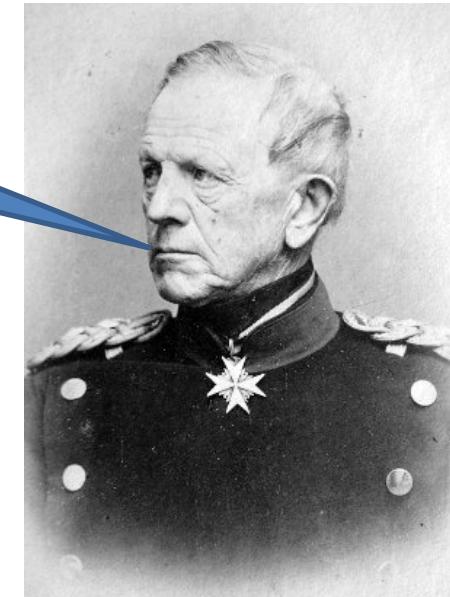
Network (2) – MS Project



Risk Management

- The project plan is objective, but idealized.

***No campaign plan survives
first contact with the enemy!***



*Helmuth Karl Bernhard von Moltke
German Field Marshal (1800-1891)*

Risk Management

- What can we do to handle risk in the project?
- Risks cannot be avoided, but some can be foreseen and planned for.
- Simple, effective risk mitigation tool:
 1. *Envision* the risk items
 2. *Evaluate* the risk items (probability × consequence)
 3. Make *risk mitigation / contingency plan* for each risk item

Identification of risks

- Project impact
 - Schedule, resources, people skills, estimates, tools
 - E.g. Loss of team member, problem with tool, missing important competences
- Product impact
 - Requirements, quality, performance, purchased components, technology
 - E.g. Poor requirements, low performance, mature technology
- Business impact
 - Organization, competitor, cost, world economic
 - E.g. New competing product, too expensive, change of organization

Risk Matrix

Risk Matrix		Probability of risk item		
Consequence of risk item	High	High	Medium	Low
	Medium	A	B	C
	Low	B	C	C

Risk Management

Description	Prob. 1-5	Conseq. 1-5	Impact 1-25	Risk Mitigation Plan
Members leave team	2	3	6	Mandatory monthly knowledge sharing via team meetings
Subsuppliers delayed	2	5	10	Formal agreement with reimbursement plan
Requirement changes	5	3	15	Frequent demonstrations of product to customer
...
...

- Risk Impact = Probability x Consequence (Highest)
- Extensions:
 - Identify cause
 - Separate *risk mitigation* from *contingency planning*

I2ICH project: Risk Matrix

Exercise 4: Create a risk matrix for your project

(Identify and evaluate risk items, like exceeding budget or plan)

"Give a description on the three classical cartoon heroes, Superman, Batman, and Spiderman. Compare the three and conclude who would win if they got into a fight"



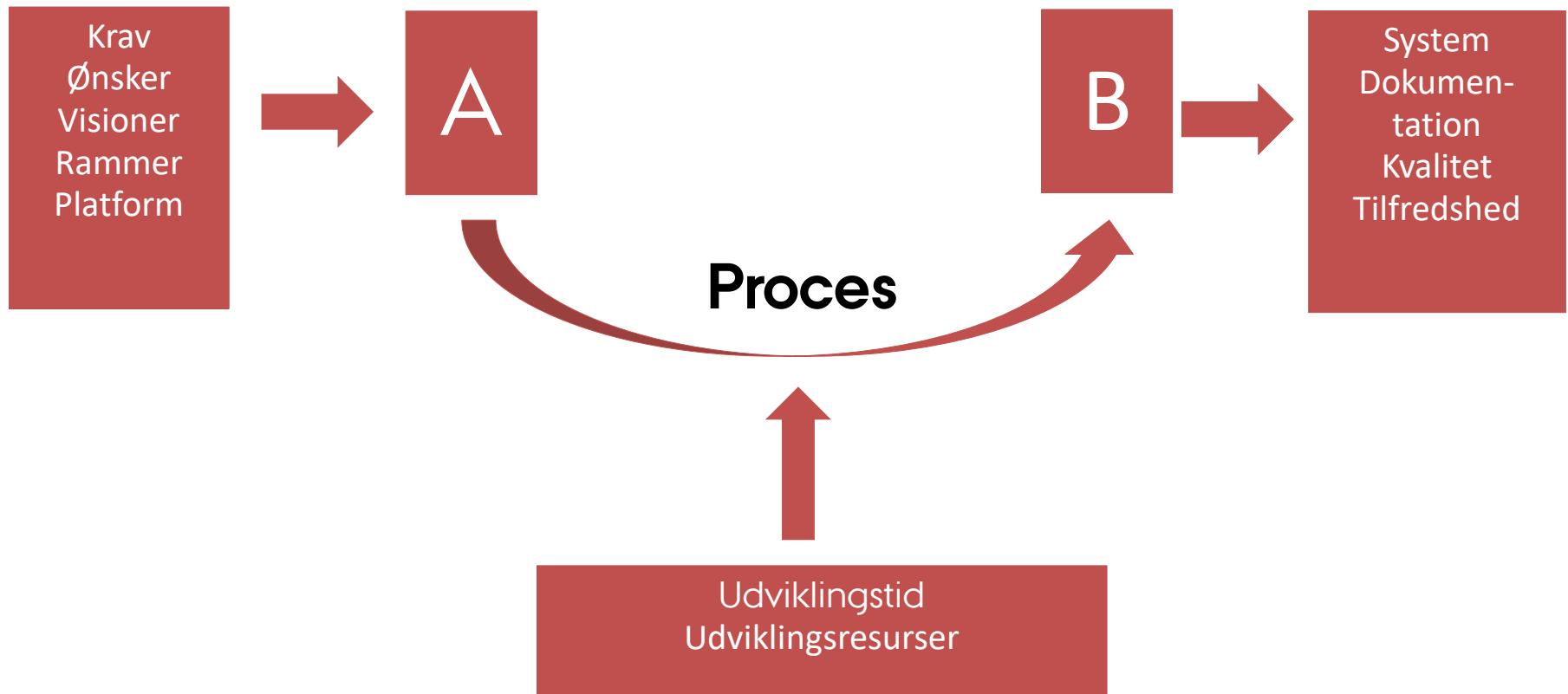
Systemdomæneanalyse og Domænemodeller

Introduktion til Systems Engineering
I2ISE

Introduktion

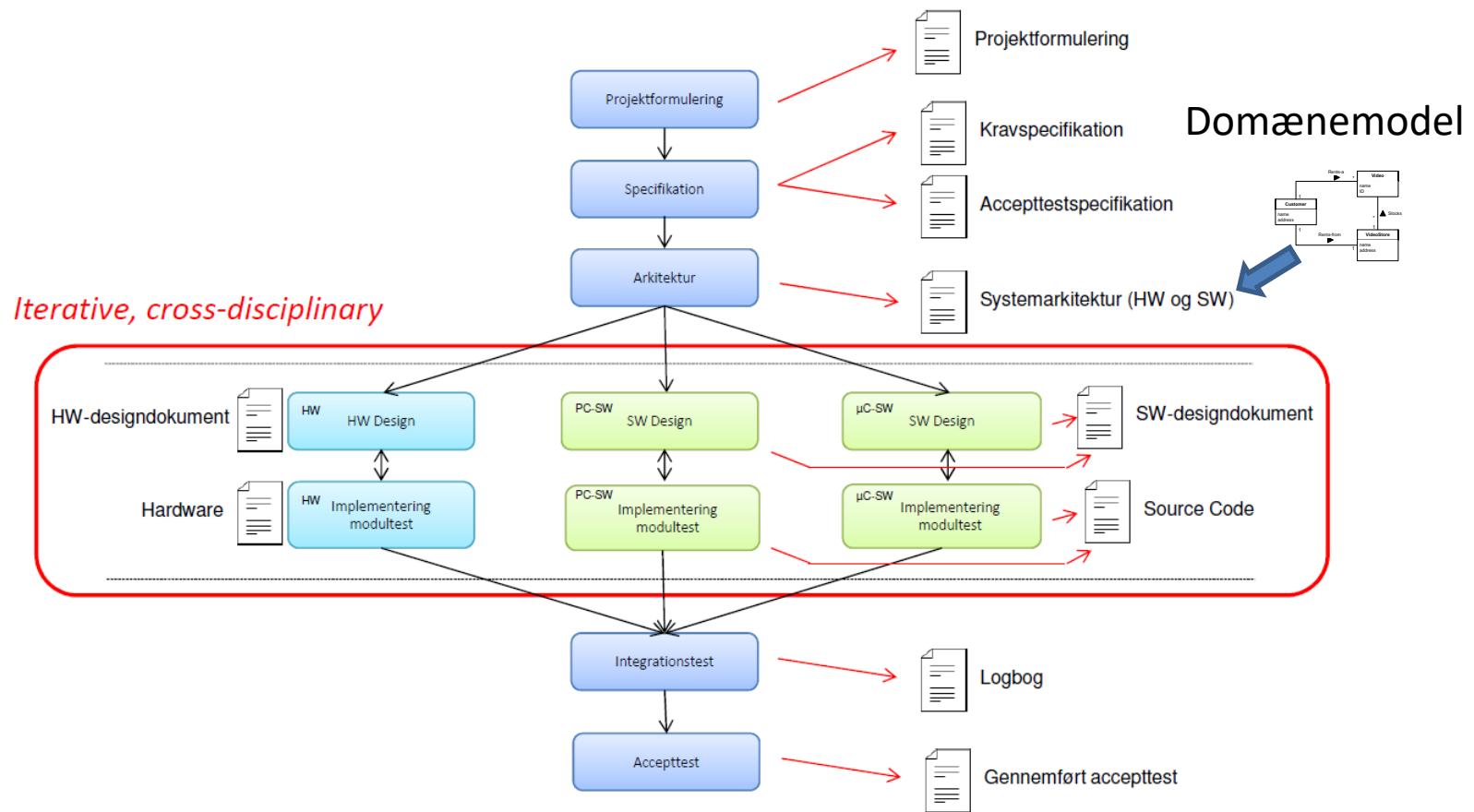
- Hvad er Systemdomæneanalyse (*System Domain Analysis*)?
- Hvad er en Domænemodel (*Domain Model*)?
- Hvorfor lave en Domænemodel?
- Hvordan laver man en Domænemodel?

Systems engineering – skematisk set

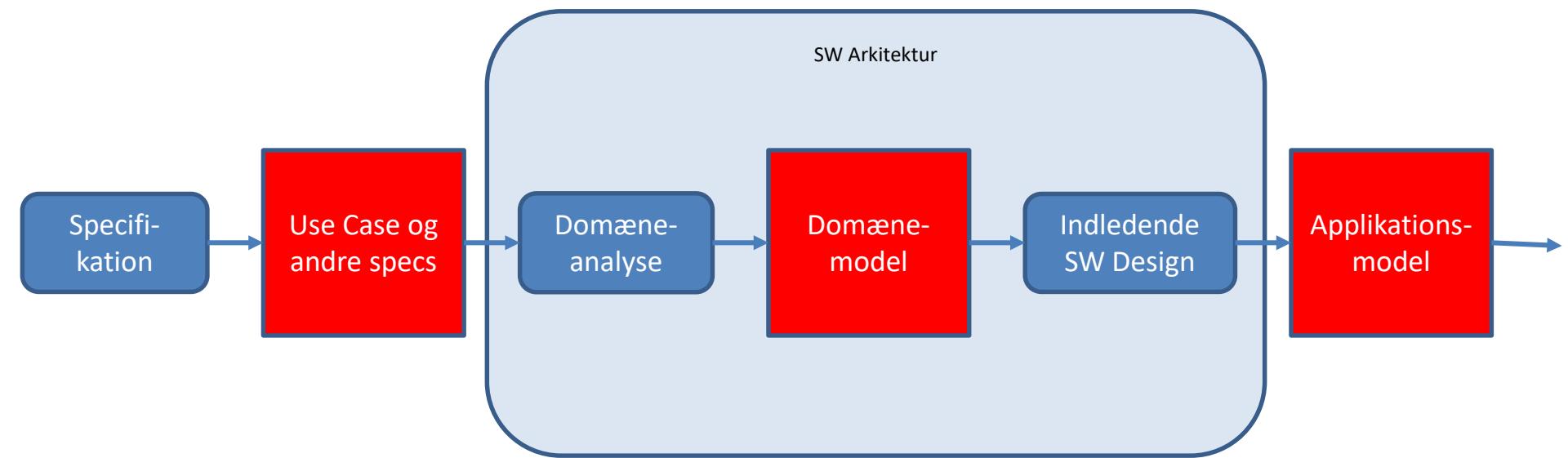


DM's plads i dokumenter og proces

The ASE Process



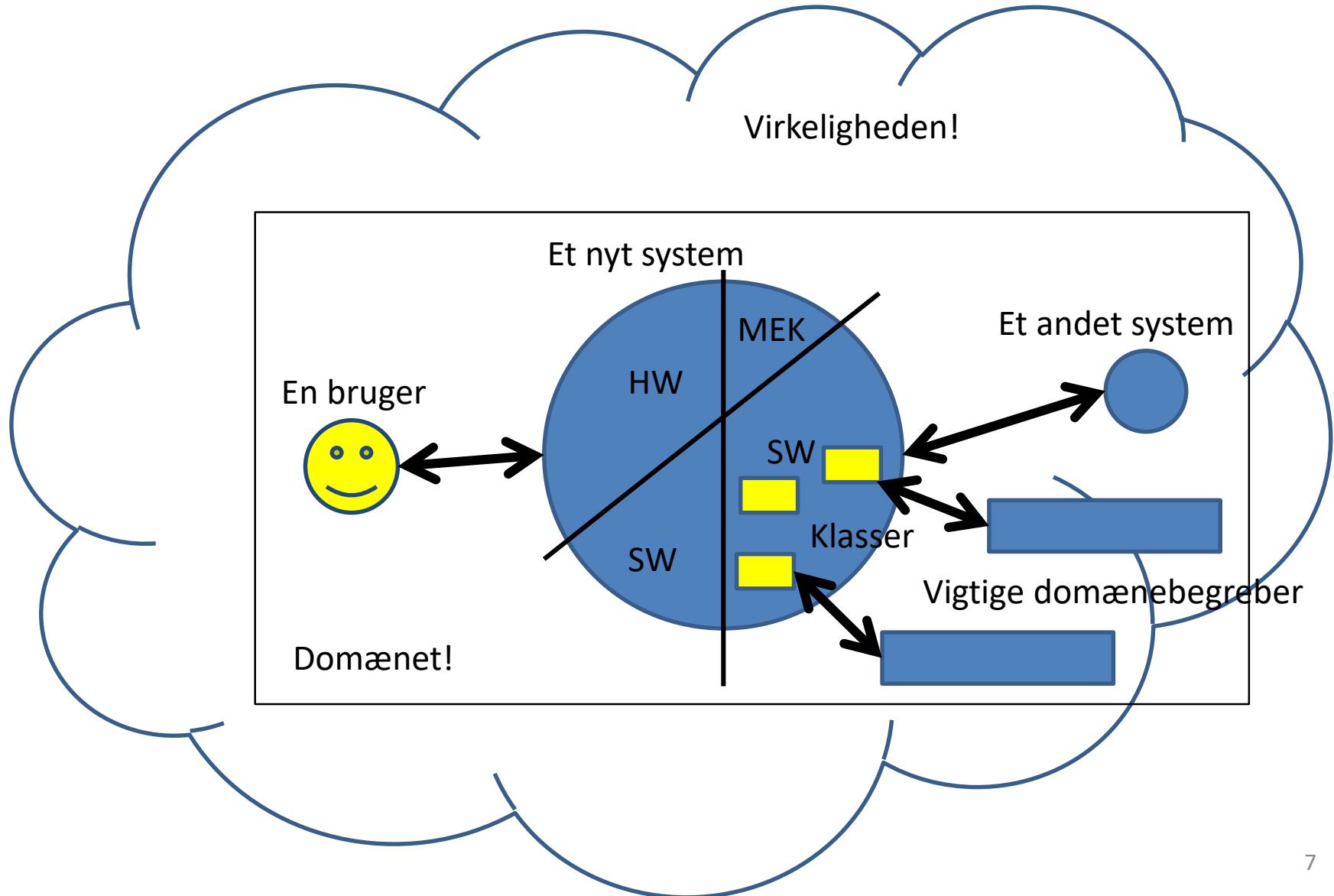
SW Arkitektur – fra UC til Design



Hvad er Systemdomæneanalyse?

- **Systemdomæneanalyse** er en aktivitet der udføres for at bestemme og afgrænse systemets **Domæne** i form af vigtige **begreber og relationer** mellem disse
- Resultatet af **Systemdomæneanalysen** er **Domænemodellen**

Virkeligheden og systemet



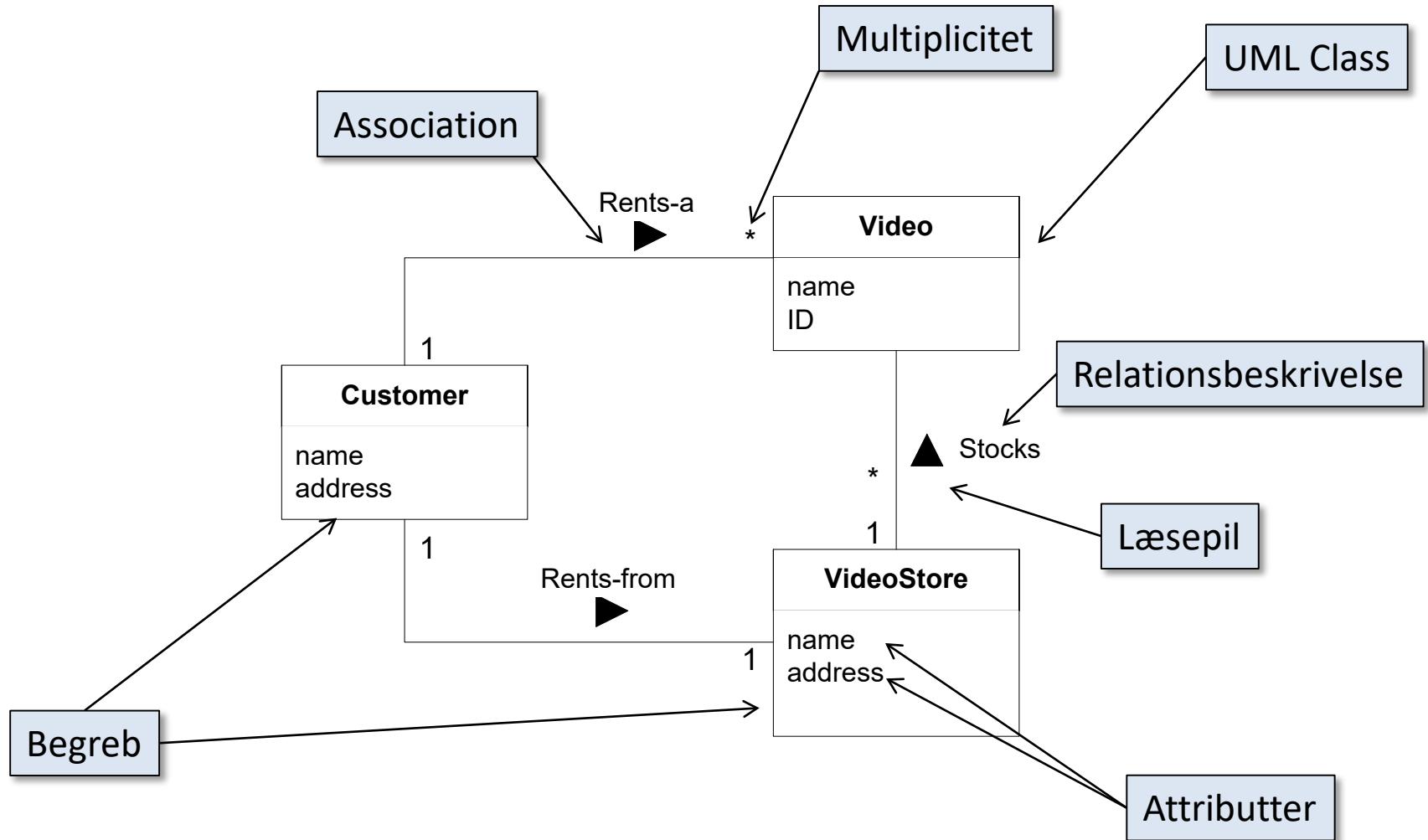
Hvad er en Domænemodel (og hvad er det ikke)? (1)

- Domænemodellen er en illustration af **vigtige, interessante, relevante, bemærkelsesværdige** (*noteworthy*) begreber i systemets domæne
 - Begreberne, deres relationer, multipliciteter og attributter
 - **Ikke** ansvar og funktioner
 - **Ikke** systemarkitektur
- Domænemodellen viser begreber fra virkeligheden, ikke SW begreber (endnu)
 - "Køretøj", "Betaling", "Kontantautomat"
 - **OK –** vigtige begreber fra virkeligheden
 - "SQLDatabase", "string"
 - **IKKE OK –** software (*implementerings-*) begreber

Hvad er en Domænemodel (og hvad er det ikke)? (2)

- Domænemodellen er en **visuel ordbog**, tegnet som et UML klassediagram
- Domænemodellen fortæller **en historie** som en slags **resume** af kravene

Domænemodel: Eksempel



Hvad er et begreb (*conceptual class*)?

- Et **begreb** som det bruges på Domænemodellen er en *ide*, en *ting* eller et *objekt*, tegnet som en **klasse**
- Dets **relationer** med andre begreber tegnes som **associationer**

Hvad gør et begreb eller en relation vigtig (*noteworthy*)?

- Det **vigtige begreber** er dem som er nødvendige for at **forstå** systemets funktion og eksistensberettigelse
- Et klassediagram er et statisk billede, der viser de relationer, der er vigtige over tid i systemets levetid
- De **vigtige** begreber, attributter og relationer at vise på Domænemodellen er dem, der er vigtige at **huske** over tid for systemet
- Dette kan bruges til både at **fuldstændiggøre**, men også **begrænse** Domænemodellen

Hvorfor lave en Domænemodel?

- Domænemodellen er **input** til at **vælge** og designe nogle af **klasserne** som skal indgå i det indledende (objektorienterede) **software design**
- Domænemodellen er dermed med til at understøtte det første svære skridt **fra krav til design** – det første skridt **fra "hvad" til "hvordan"**
- En Domænemodel kan bruges til at **tale med kunden/product owner** om systemet
- Den visuelle ordbog fastlægger **navnene** i den videre process, så **alle bruger samme navne**
- En **ny medarbejder** kan få et **hurtigt overblik** over systemets Domæne

Kogebog for Domænemodeller



Step 1: Find begreberne (med evt. attributter) i Domænet og tegn dem i et klassediagram

Step 2: Find relationerne og tegn dem som associationer med en tekst og evt. en pil

Step 3: Forsyn relationerne med multiplicitet

Step 4: Lav finpudsning med attributter/klasser, specifikke klasserelationer, og slet evt. unødvendige begreber

Find begreberne

Step 1.1: Find navneord i Use Cases og andre specifikationer

Step 1.2: Find begreber ved at lade sig inspirere af listen med kategorier af generelle begreber (afsnit 1.6.1)

Organiser dem på et klassediagram.

Nogle af dem er måske attributter i stedet for klasser

Step 1.1 Navneordsanalyse

- Marker navnenordene i UC og andre specifikationer
- Ikke alle navneord er nødvendigvis vigtige begreber
 - De kan være implementationsorienterede
 - De kan være for lavniveau og skal måske op på et højere abstraktionsniveau
 - De er måske slet ikke relevante for systemet og kan ignoreres
- Men da de kommer fra specifikationerne er de sandsynligvis ret relevante
- Nogle navneord kan måske med det samme udnævnes som attributter for andre begreber
- Hellere lidt for mange begreber i Domænemodellen end for få på nuværende tidspunkt

Step 1.2 Kategoriliste

- Listen står i afsnit 1.6.1 i pensumsartiklen til lektionen:

Generelt Begreb	Eksempler
Transaktion – en afsluttet forretningshændelse, som naturligt afgrænser en aktivitet, der skal kunne skelnes fra andre slags og andre af samme slags	Et Salg En Betaling En Tankning En Udcheckning på biblioteket/supermarkedet En Flyreservation
Transaktionslinje – en af flere linjer, der repræsenterer et begreb, der indgår i eller dækkes af transaktionen	En Varelinje på en bon ved et salg En Boglinje i en udcheckning på biblioteket
Fysisk genstand – en fysisk objekt, der fx matcher en transaktionslinje	Selve Varen Selve Bogen Et Lånerkort
...	...

Step 1.2 Kategoriliste

- Listen indeholder kategorier af begreber, der erfaringsmæssigt ofte optræder i systemer
- Listen er meget generel og nogle kategorier er måske ikke relevante for den slags system I er ved at lave
 - fx indlejrede/embeddede systemer
- Lad dig inspirere af listen for at se, om der er nogen begreber, der ikke er direkte nævnt som navneord, eller afledt heraf

Step 1 - afslutning

- Tegn dem på et klassediagram med gode navne og evt. attributter
- Med lidt **øvelse** har man
 - identifieret de vigtige begreber
 - ignoreret ikke vigtige begreber
 - indført attributter
- Og er klar til næste skridt

Kogebog for Domænemodeller 2



Step 1: Find begreberne (med evt. attributter) i Domænet og tegn dem i et klassediagram

Step 2: Find relationerne og tegn dem som associationer med en tekst og evt. en pil

Step 3: Forsyn relationerne med multiplicitet

Step 4: Lav finpudsning med attributter/klasser, specifikke klasserelationer, og slet evt. unødvendige begreber

Find relationerne

Step 2.1: Find udsagnsord i Use Cases og andre specifikationer

Step 2.2: Find relationer ved at lade sig inspirere af listen med kategorier af generelle relationer (afsnit 1.6.2)

Tegn dem ind på klassediagrammet som associationer
Giv dem navne/beskrivelser og evt. læsepile

Step 2.1 - udsagnsord

- Find de udsagnsord i UC og andre specifikationer, der forbinder begreber
- Identifier begreberne på dit klassediagram og forbind dem med en association
- Det er ikke sikkert at alle udsagnsord giver vigtige relationer

Step 2.2 Relationsliste

- Listen står i afsnit 1.6.1 i pensumsartiklen til lektionen:

Generel Relation	Eksempler
A er en transaktion som hænger sammen med en anden transaktion B	Salg – Betaling Reservation – Afbestilling
A er en transaktionslinje i transaktionen B	Faktura – Varelinje
A er et produkt eller en service for en transaktion eller transaktionslinje B	Varelinje – Vare Reservation – Selve flyveturen
...	...

Step 2.2 Relationsliste

- Listen indeholder kategorier af relationer, der erfaringsmæssigt ofte optræder i systemer
- Listen er meget generel og nogle kategorier er måske ikke relevante for den slags system I er ved at lave
 - fx indlejrede/embeddede systemer
- Lad dig inspirere af listen for at se, om der er nogen relationer, der ikke direkte er afledt af udsagnsord
 - fx mellem de begreber du fik fra kategorilisten

Step 2 - afslutning

- De fundne relationer har man nu tilføjet til klassediagrammet som associationer
- De har fået nogle gode navne, der beskriver formålet/begrundelsen for relationen
- De har fået læsepile, der hvor der kan være mulighed for misforståelser
- Er der nogen begreber, der ikke har nogen association, må man overveje, om der mangler en relation, eller om begrebet er irrelevant
- **Associationerne har IKKE fået pile i enderne – det venter vi med at gøre til design og implementation**

Kogebog for Domænemodeller 3

Step 1: Find begreberne (med evt. attributter) i Domænet og tegn dem i et klassediagram



Step 2: Find relationerne og tegn dem som associationer med en tekst og evt. en pil

Step 3: Forsyn relationerne med multiplicitet

Step 4: Lav finpudsning med attributter/klasser, specifikke klasserelationer, og slet evt. unødvendige begreber

Step 3 - multipliciteter

- Associationerne skal nu forsynes med multiplicitet
- Det er vigtigt at huske, at det er set fra systemets side – vi ser det fra én instans af systemet
- Der bruges typisk multipliciteterne "1", "*", "0..1", "1..*"
- Er der vigtigt, fx for product owner, at der er exakte værdier for antal, skriver man bare det, fx som "1..4" eller "3". Ellers er det typisk en implementationsdetalje,
- Er det en 1-til-1 association kan man udelade multiplicitet

Kogebog for Domænemodeller 4

Step 1: Find begreberne (med evt. attributter) i Domænet og tegn dem i et klassediagram

Step 2: Find relationerne og tegn dem som associationer med en tekst og evt. en pil



Step 3: Forsyn relationerne med multiplicitet

Step 4: Lav finpudsning med attributter/klasser, specifikke klasserelationer, og slet evt. unødvendige begreber

Step 4 - finafpudsning

- Nogle vigtige emner at tage stilling til her er:
- **Aktører**: Er en tændstiksmand nok, eller skal det være en klasse med attributter, evt. med stereotypen <<aktør>>?
- **Klasser-Attributter** – er der nogle klasser, der burde være attribut på en af de andre i stedet for?
- Giver det mening med et **System** begreb?
- **Komposition og Arv** – er der grund til at bruge dem?
- **Generel oprydning**
 - Se efter overflødige begreber
 - Se efter begreber der IKKE skal indgå i denne iteration af udviklingsforløbet

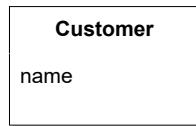
Aktør

- Er aktøren blot en anonym ekstern aktør bruges en tændstiksmand
- Er der brug for at huske noget om aktøren, skal den have en klasse
- IKKE Begge dele!

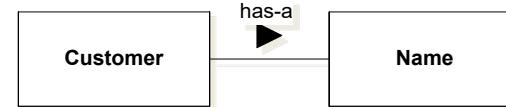


Attribut eller klasse?

- Guidelines
 - Hvis det fylder noget fysisk, er det et begreb
 - Hvis det har en kompleks struktur med underattributter (som er vigtige for dette system), er det et begreb
 - Hvis det har aktiviteter, er det et begreb
 - Hvis det er simpelt, med en veldefineret værdimængde, er det en attribut



eller



?

Praktiske råd til Systemdomæneanalysen

- Brug et whiteboard og/eller PostIts(sticky notes, gule sedler) eller papir & blyant eller et diagrammeringsprogram direkte
- Arbejd sammen!
- Tag et foto af whiteboardet (gem de gule sedler) eller af papiret eller gem den første version af diagrammet, når I er færdige med den "kreative proces"
- Så kan man rentegne og arrangere Domænemodellen i et diagrammeringsprogram, så andre kan forstå det, og det er nemmere at arbejde videre med det
- Vi forventer at finde jeres Domænemodel i Systemarkitekturdokumentet for jeres semesterprojekt(er) i letlæselig form ☺

Øvelse i dag, og som hjemmearbejde:

Tankstation

- Lav en Domænemodel for tankstationen, baseret på Use Casen ”Optank Bil” (se den fulde tekst på Brightspace):
 1. Identificer vigtige begreber med metoderne beskrevet under Step 1
 2. Lav et UML Class Diagram med begreberne og evt. attributter
 3. Identificer vigtige relationer med metoderne beskrevet under Step 2 og indfør dem på klassediagrammet
 4. Sæt multipliciteter, hvor nødvendigt
 5. Udfør finpudsning med attributter og anden oprydning



Øvelse næste lektion:

Poultry Galore

- Lav en Domænemodel for det nye pakkesystem for fabrikken "Poultry Galore" (se den fulde tekst på Brightspace):
 1. Identificer vigtige begreber med metoderne beskrevet under Step 1
 2. Lav et UML Class Diagram med begreberne og evt. attributter
 3. Identificer vigtige relationer med metoderne beskrevet under Step 2 og indfør dem på klassediagrammet
 4. Sæt multipliciteter, hvor nødvendigt
 5. Udfør finpudsning med attributter og anden oprydning



Eksempel til illustration af Systemdomæneanalyse og Domænemodeller

Som et gennemarbejdet eksempel på arbejdsprocessen med at udføre Systemdomæneanalysen og komme frem til Domænemodellen, har jeg valgt den forhadte Selvbetjeningskasse hos Føtex. Den kan se sådan ud:



Et Use Case for dette system er udarbejdet, og heraf ses et uddrag:



Man har valgt at betragte disse to dele af et naturligt indkøbsforløb som to separate UC, man kunne enten samle dem i en, eller forbinde dem med en <<extends>> eller <<includes>> afhængighed.

Der er også lavet Fully Dressed Use Cases. For overskuelighedens skyld vises her blot hovedscenarierne, uden extensions. Dem kunne der jo sagtens tænkes nogle stykker af.

Her for Use Case Skanning af Varer

1. Selvbetjeningskassen anmoder kunden om at skanne vare
2. Kunden placerer vare foran skanner
3. Systemet skanner varens stregkode
4. Systemet finder varens pris i varedatabasen
5. Vare med pris tilføjes til en vareliste som vises på skærmen
6. Kunden lægger vare i pose på vægten ved siden af skanner
7. Punkterne 1-6 gentages indtil alle varer er skannet
8. Kunden vælger afslut på skærmen

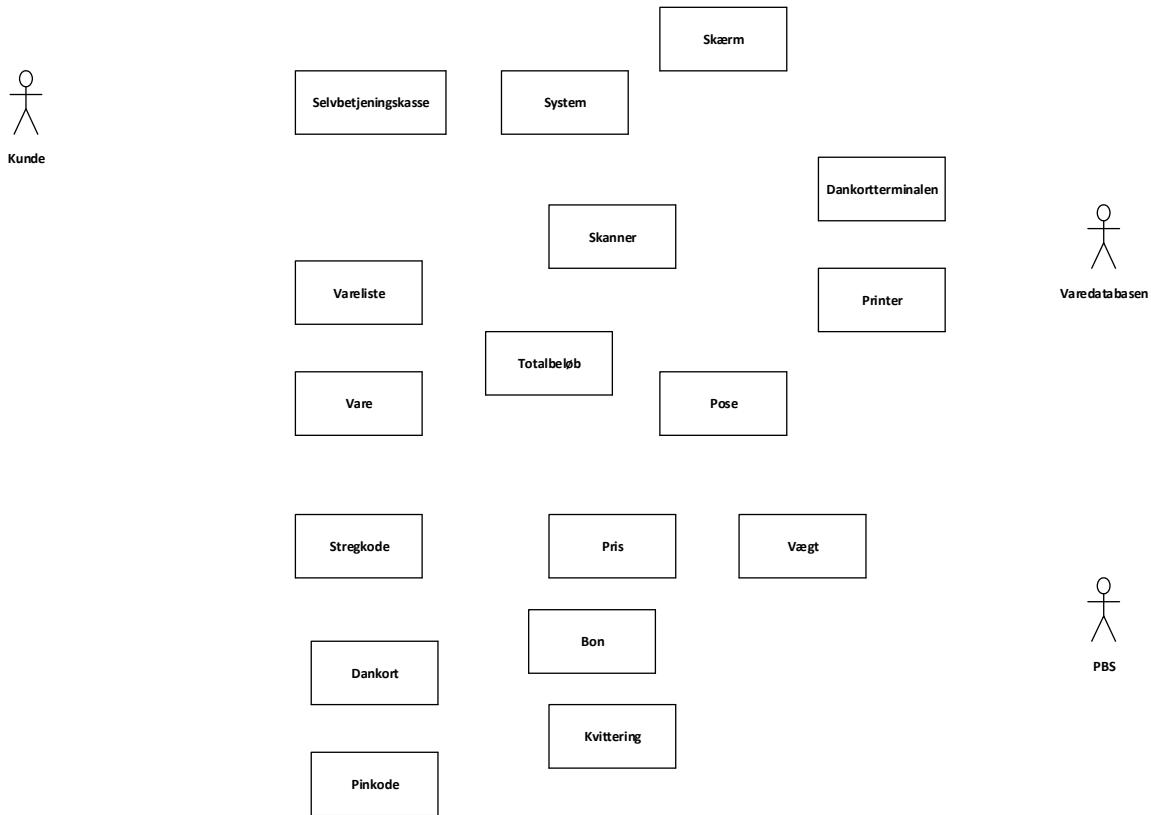
Og her for Use Case betaling af varer.

1. Kunden vælger betal med dankort på beløbet
2. Kunden indsætter kort i dankortterminalen
3. System viser det totale beløb og anmoder om pinkode
4. Kunden indtaster pinkode
5. Kort og pinkode valideres mod PBS
6. Printer udskriver bon med vareliste og kvittering

Skridt 1.1

I dette skridt tager vi fat på jagten på navneord i de UC vi skal til at behandle på dette tidspunkt af projektet.

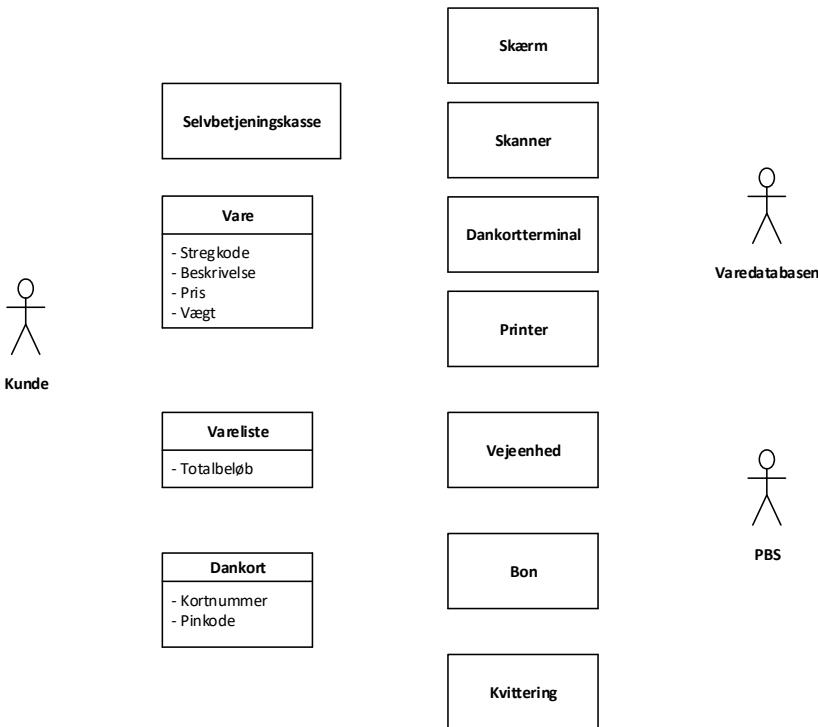
Det giver dette diagram



Her er valgt næsten bevidstløst at give alle fundne navneord en klasse, og aktørerne er blot angivet som aktører.

Finpudsning af Skridt 1.1

Vi kan lige så godt tage fat på forenklinger med det samme, for at få gjort det store klassediagram lidt mere overskueligt.



Her har jeg indset, at Selvbetjeningskasse og System må være det samme, så System er fjernet. Nogle af begreberne kan ret nemt indsese, at de er simple data, der kunne være attributter for Vare, Vareliste og Dankort.

Pose kan jeg ikke indse har nogen særlig betydning i alt dette – det kan være jeg tager fejl, men der står ikke noget i UC om, at der er en extension om man starter med eller uden pose. Så den har jeg fjernet.

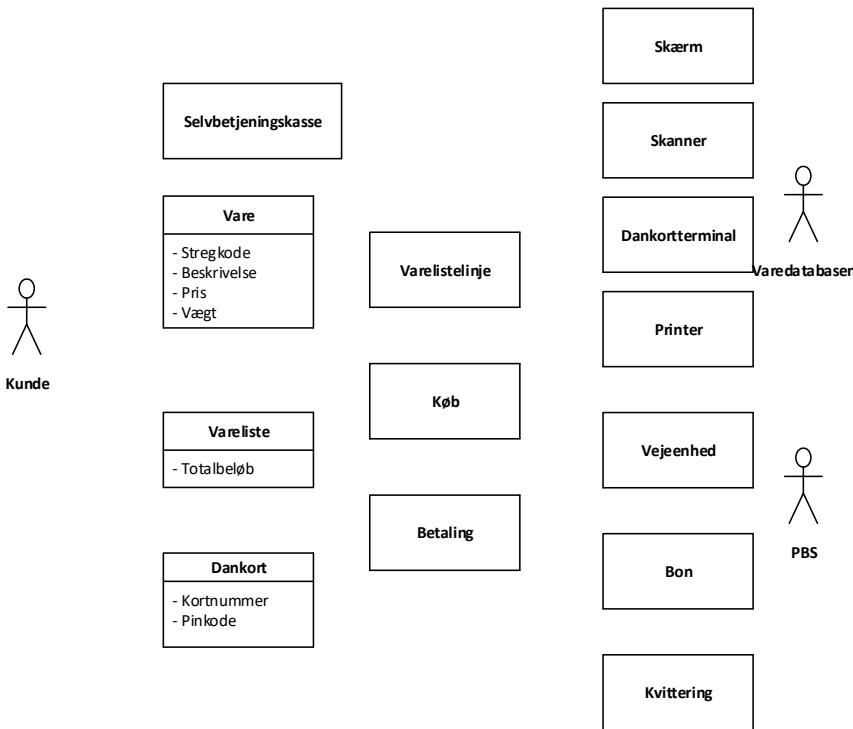
Jeg har også tænkt lidt over hvilke attributter der ellers kunne være relevante, så jeg har tilføjet Vægt (per enhed) og Beskrivelse til Vare.

Til Dankort ville det også være naturligt at holde styr på Kortnummeret.

Da jeg nu har opfundet attributten Vægt for Vare, har jeg omdøbt "Vægt" til Vejeenhed, for at undgå misforståelser. Dette skal man selvfølgelig være enige med projektets kunde om.

Skritt 1.2

For at sikre mig, at det hele nu er med, lader jeg mig inspirere af listen over kategorier af generelle begreber som ofte ses i IT-systemer, og tænker efter, om der skulle være nogen af dem at finde i systemet.

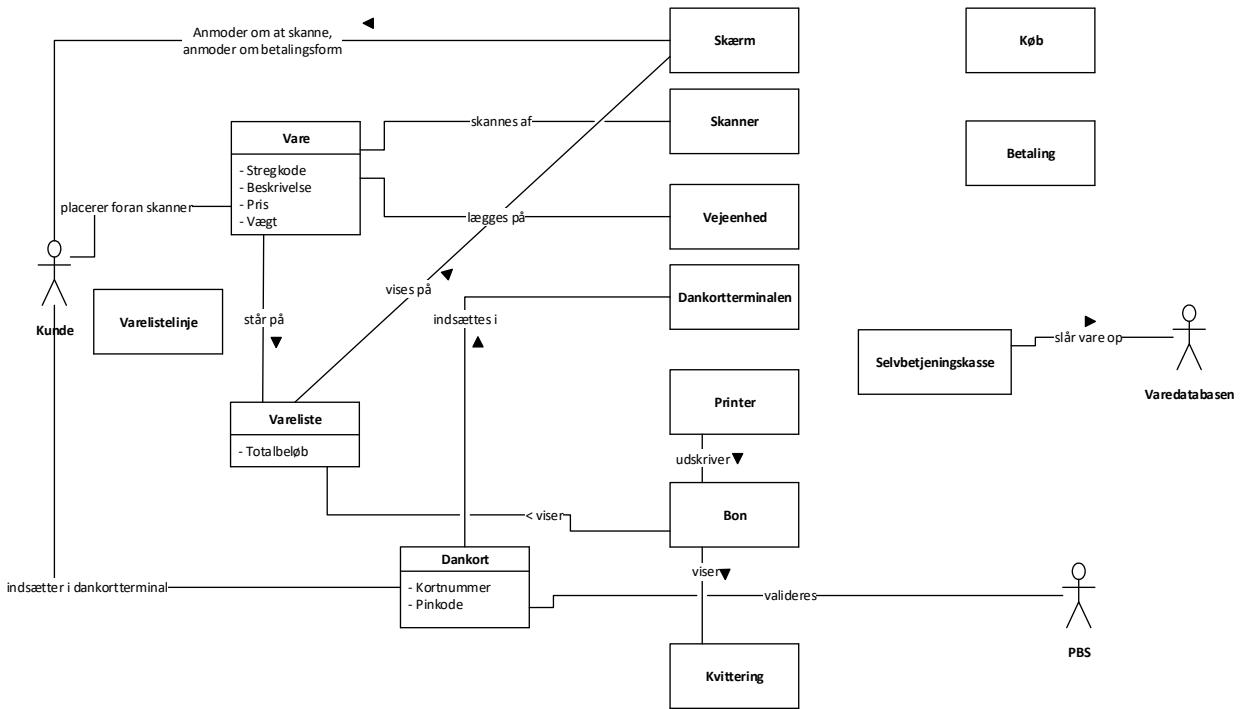


Her bliver jeg inspireret til at indføre transaktionerne **Køb** og **Betaling**, samt transaktionslinjen **Varelistelinje**. En **Varelistelinje** er ikke det samme som en **Vare**, for der kan være forskellige attributter, og jeg har set, at Føtex' kasseboner samler flere varer af samme slags til én linje, så det vil jeg gerne give mulighed for.

Skridt 2.1

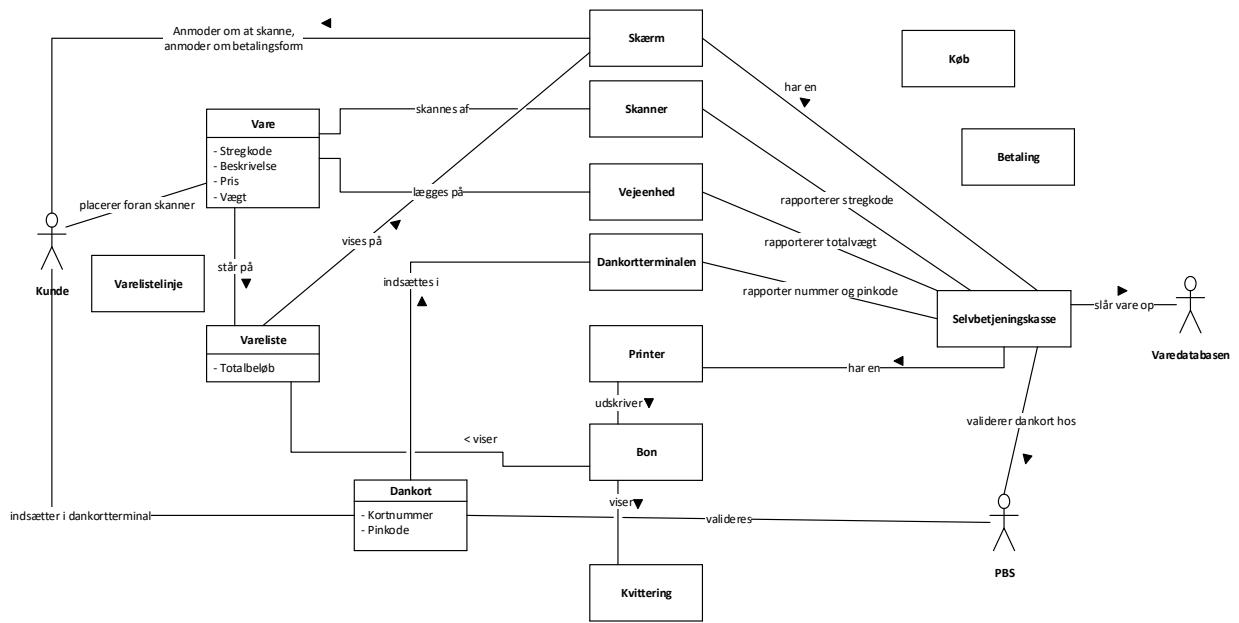
Nu skal vi i gang med at finde relationerne. Som første skridt kigger jeg efter udsagnsord i UC, og ser om jeg kan bruge dem som relationer.

Dette er første udskrift:



Jeg har her kun taget de udsagnsord, jeg kunne finde og bruge i UC, evt. med lidt omformulering.

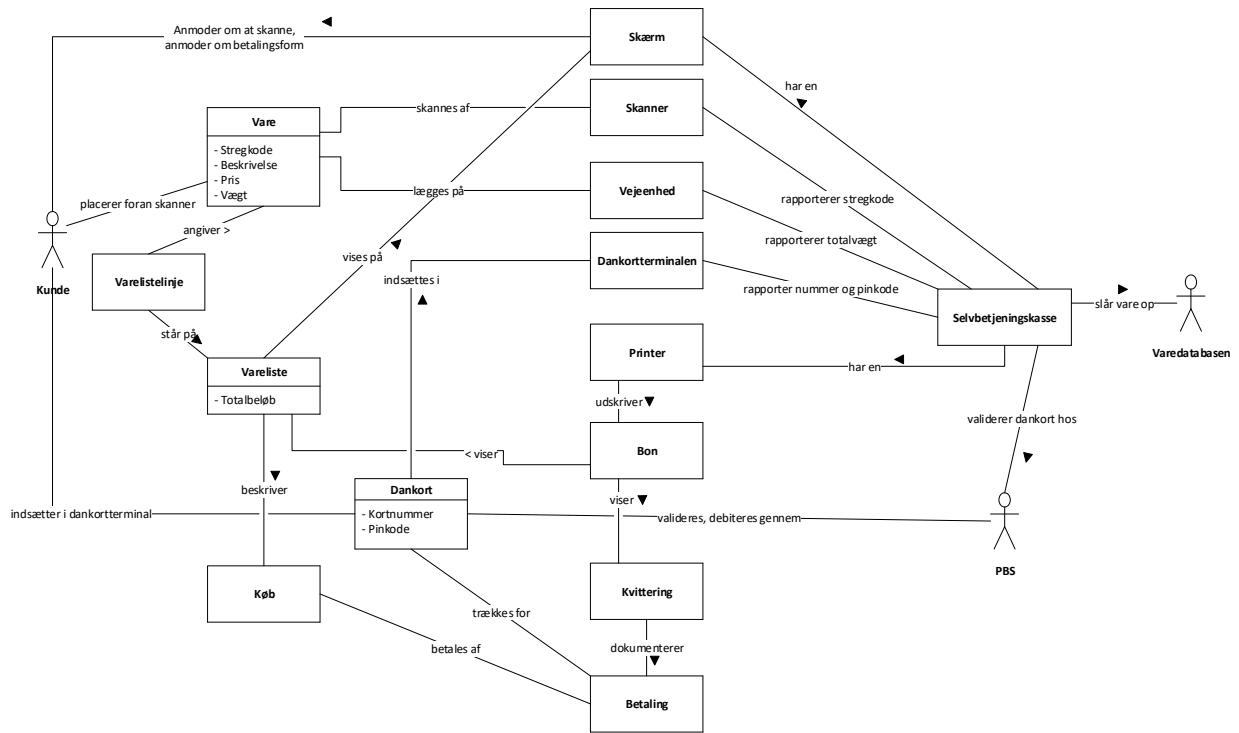
Men der er jo stadig mange løsthængende begreber, så jeg prøver at se, om jeg kan finde nogle flere.



Her er det specielt "systembegrebet" Selvbetjeningskasse der kan bruges til at hænge nogle af de andre begreber op på. Der hvor jeg synes det giver mening, har jeg ændret "har en" relationen til at have et lidt mere sigende navn, der kan fortælle lidt om relationens rolle.

Skridt 2.2

Men der er stadig nogen begreber, der hænger og flagrer. Nu lader jeg mig inspirere af listen over generelle relationskategorier.

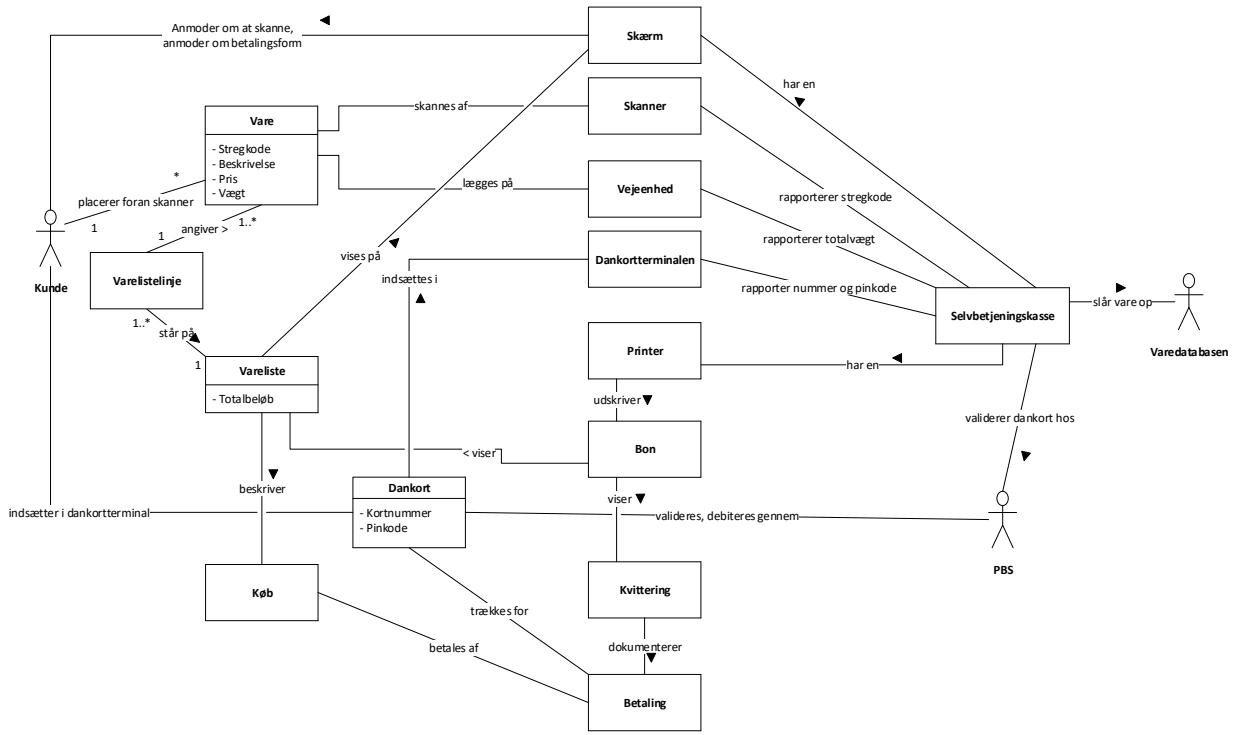


En Varelistelinje står selvfølgelig på Varelisten, og en Vare er et produkt der hænger sammen med sådan Varelistelinje.

Betaling dækker Køb. Køb er beskrevet af Vareliste. Kvitteringsdelen af Bon er et transaktionsbevis for Betaling. Varelistedelen af Bon er et transaktionsbevis for Køb.

Skridt 3.

Nu skal der multiplicitetsannotationer på.



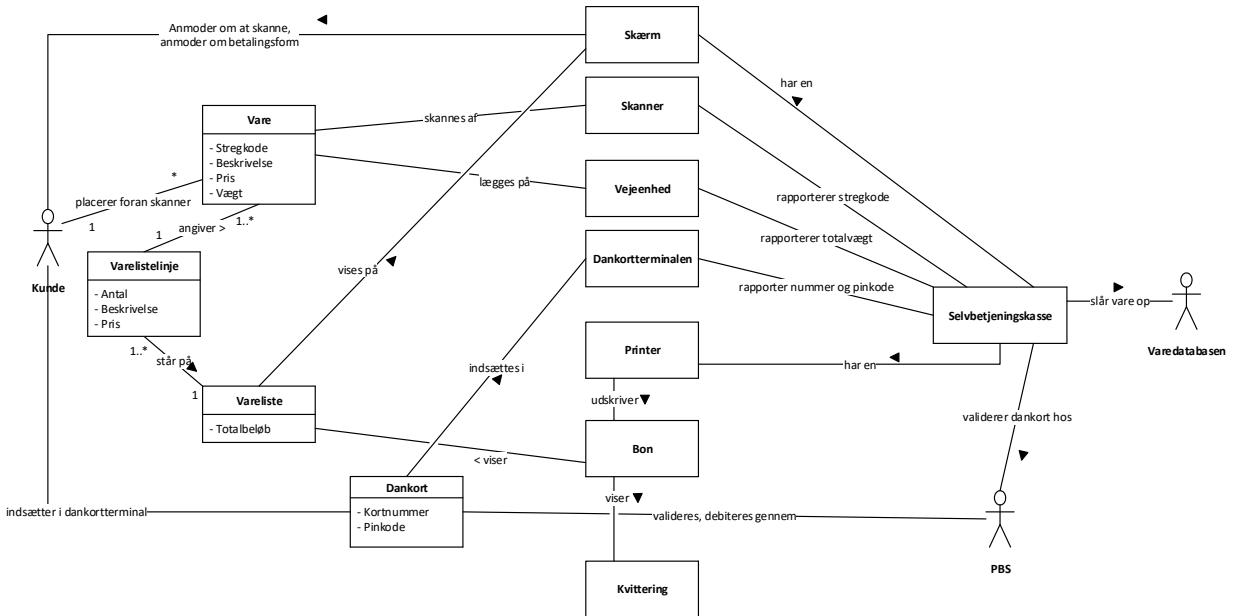
Der er ikke ret mange af relationerne, der ikke er 1-til-1. Tænk på, at vi jo kun ser på selve Selvbetjeningskassen, og ikke hele Føtex, hvor der kan være mange af dem. Vi interesserer os fx derfor ikke for at der er mange Selvbetjeningskasser der anvender Varedatabasen – det kunne man have brug for, når man skal lave Domænemodellen for Varedatabasen.

Der er 1 eller flere Varelistelinjer på en Vareliste, og hver Varelistelinje kan dække over 1 eller flere fysiske Varer (jf. min tidligere observation over Føtexboner.)

Kunden skanner ifølge UC flere varer, derfor er der "*" på "placerer foran skanner".

Skridt 4

Nu er det tid til lidt finpudsning.



Jeg har fjernet Køb og Betaling igen fra Domænemodellen. Der er ingen steder i specifikationerne for dette system – Selvbetjeningskassen – der står noget om, at de skal gemmes et eller andet sted, og huskes, når Kunden er gået sin vej. Så er der ingen grund til at bekymre sig om dem nu. De kan altid komme på i en de næste iterationer, hvis der kommer krav om det.

En anden kandidat til at bliver fjernet eller ændret, ud fra samme argumentation, er Vejeenheden. Der står ikke noget om, at den målte totale vægt bliver sammenlignet med den teoretiske ud fra Varernes samlede Vægt og at en uoverensstemmelse giver alarm. Så den kunne fjernes eller omdøbes. Men jeg tror mest på, at den står i en extension i UC. Hvordan sikrer Føtex sig ellers, at jeg ikke snyder?

Til gengæld ville jeg gerne uddybe, hvorfor Vare og Varelistelinje er forskellige – og har derfor indført de attributter, som jeg forestiller mig, der er brug for, når den skal skrives på Bonen.

Vedr. andre klasserelationer end association, kunne man måske forestille sig at tegne Bonens relation til Vareliste og Kvittering som komposition. Men dels indgår Varelisten også i andre relationer, og dels er Bon nok nærmere en fysisk ting, end et objekt.

Der er lidt tendenser til, at der gået lidt BDD i dette, fordi jeg har angivet at Selvbetjeningskasse har en Skærm, en Scanner, en Dankortterminal, en Vejeenhed og en Printer. Men dels er disse begreber nævnt i specifikationen, og dels er der mange andre detaljer i Domænemodellen, der gør at jeg har masser af input til mit efterfølgende softwaredesign, i form af information om systemet.

Indholdsfortegnelse

1. DOMÆNEMODELLER	2
1.1. Domænemodellens plads i udviklingsforløbet	2
1.2. Hvorfor skal man lave en Domænemodel?	2
1.3. Hvad er Systemdomæneanalyse og Domænemodellen?	2
1.4. Domænemodellens rolle som kommunikationsmiddel	4
1.4.1 Domænemodellen udad til	5
1.4.2 Domænemodellen indad til	5
1.5. Hvad er begreber (concepts, conceptual classes)	5
1.6. Metode	6
1.6.1 Skridt 1. Find begreberne	6
1.6.2 Skridt 2. Find relationerne	8
1.6.3 Skridt 3 Find multipliciterne	9
1.6.4 Skridt 4 Finpudsning	10
1.7. Hvad er en Domænemodel IKKE!	11

1. DOMÆNEMODELLER

1.1. Domænemodellens plads i udviklingsforløbet.

Forudgående aktiviteter: Kravspecifikation, hvor der er arbejdet meget med, HVAD systemet skal gøre.

Input: Fra arbejdet med Kravspecifikationen vil man have dokumenter, der specificerer funktionelle krav, fx Fully Dressed Use Cases (UC). Under Systemdomæneanalyse kan det være relevant med input fra dialog med: Kunden, Slutbrugere, Udviklere, Stakeholders, Product Owner.

Output: Domænemodellen – nemlig et eller flere klassediagrammer i den specielle Domænemodelnotation.

Efterfølgende aktiviteter: Detaljeret SW systemdesign – Applikationsmodellen

I ASE modellen: En tidlig del af arkitekturaktiviteterne.

I V-modellen: En tidlig del af Systemdesign.

Systemdomæneanalyesen er en del af analysen og designet for Systemarkitekturen, hvor de første skridt tages fra HVAD systemet skal gøre frem mod HVORDAN det skal gøre det.

1.2. Hvorfor skal man lave en Domænemodel?

Med opståelsen af begrebet objektorienteret analyse, design og programmering i begyndelsen af 1990'erne, opstod et alternativ til den på den tid fremherskende indstilling: programmer består af aktiviteter, der skal behandle nogen data. Altså: aktiviteter først – og så data.

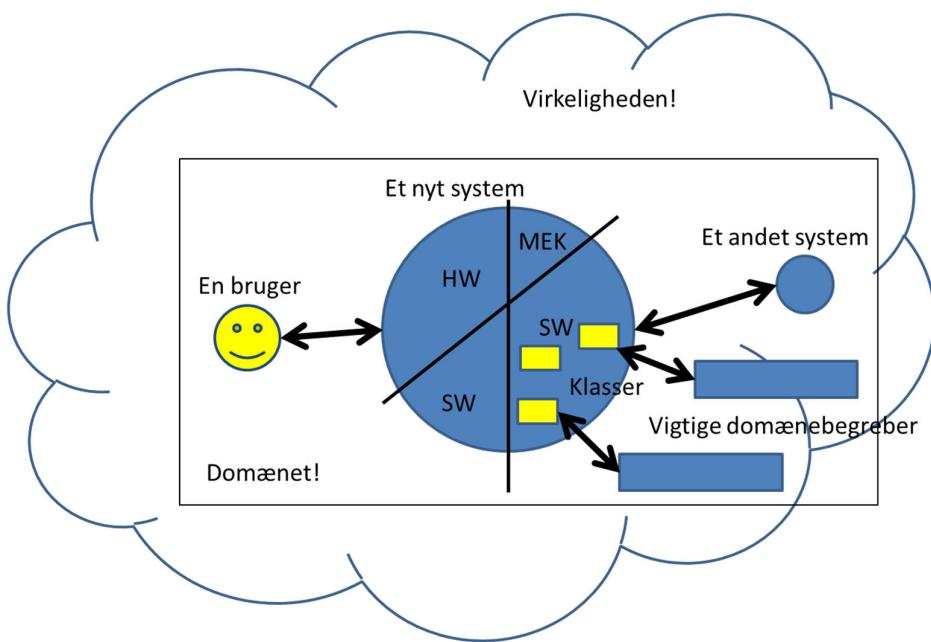
Filosofien er, at IT-systemer, der skal interagere med objekter og begreber, der findes i den virkelighed – udenfor systemet – der skal have nytte af systemet, konstrueres bedst, hvis de afspejler objekter og begreber i den måde softwaren er struktureret på. Altså: Data og begreber først – og så aktiviter.

Der er derfor brug for en metode hvormed man kan gå på jagt efter de objekter og begreber, der skal tages i betragtning – det er hvad denne artikel handler om.

1.3. Hvad er Systemdomæneanalyse og Domænemodellen?

Der er først og fremmest brug for en opdeling af begreberne i virkeligheden omkring systemet i dem, der skal tages hensyn til af systemet – og dem der **IKKE skal!** Den del af begreberne, der **SKAL** med, kaldes for **Domænet**, se nedenstående figur.

Det er først og fremmest den information som systemet skal huske og have styr på, der er de mest interessante og relevante objekter og begreber og relationer, at finde frem i **Domænet**.



Figur 1 Virkeligheden, Systemet og dets Domæne

Aktiviteten at afgrænse og beskrive Domænet kaldes **Systemdomæneanalyse** (*Systems Domain Analysis*).

Når vi skal designe og kode systemet, har vi også brug for at vide hvordan disse begreber forholder sig til hinanden. Hvad har de med hinanden at gøre, hvor mange er der af dem og hvad er der af karakteristika, der kendtegner dem, og som er vigtige at kende, når vi skal behandle dem i systemet?

Disse forhold skal vi også gå på jagt efter i Systemdomæneanalysen. Systemets og Domænets plads i Virkeligheden kan ses på Figur 1.

Når vi er færdige med analysen, skal vi gerne have et resultat, der er fastholdt som et dokument, vi kan bruge, når vi går videre i softwareudviklingen. Dette dokument kaldes for **Domænemodellen** (*Domain Model*).

Det vil være godt, at alle er enige om Domænemodellen, at den er let at forstå af dem, som skal bruge den i det videre arbejde, og at den er rimelig komplet og relevant. Den skal kommunikere resultatet af Systemdomæneanalysen, så den kan diskuteres, reviewes, godkendes og bruges i det videre udviklingsforløb.

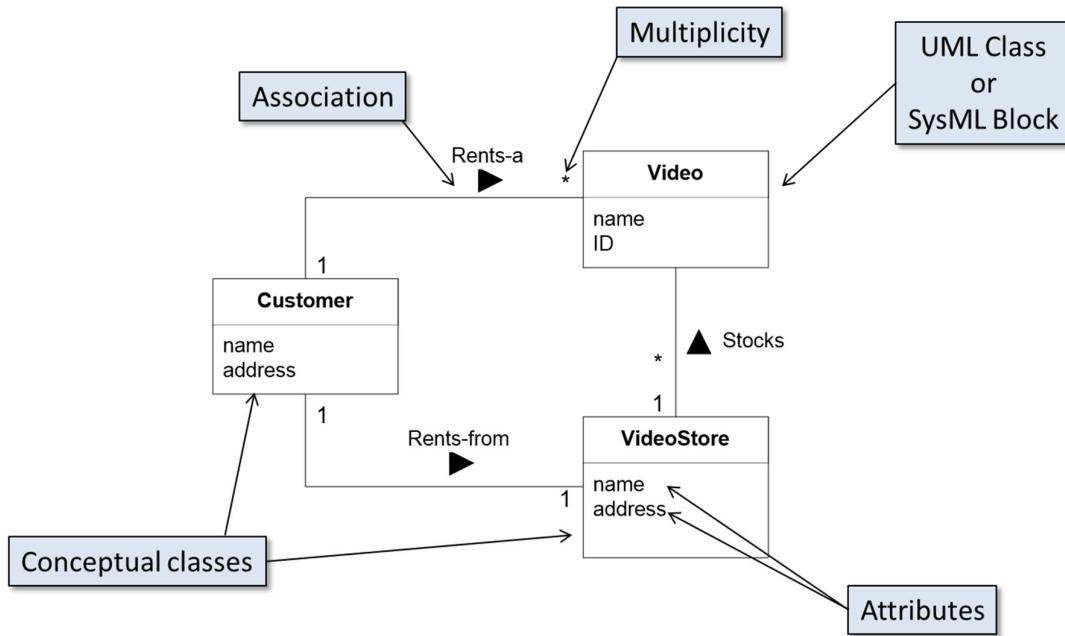
Hvordan kan man gøre det?

Ligesom man kan vælge at fastholde kravene til systemet i ren tekst – side op og side ned – viser erfaringen, at en tegning siger mere end tusind ord – en grafisk præsentation har mange fordele som kommunikationsværktøj.

Med SysML – og som en del af dette UML – som værktøj, har vi mulighed for at lave et diagram (eller flere), der kan opfylde disse ønsker.

Domænemodellen laves som et klassediagram (eller flere), hvor hvert begreb får et klassestencil.

Her er et eksempel (Figur 2):



Figur 2 (Del af) et eksempel på en Domænemodel

Relationerne mellem begreberne tegnes som klasserelationer – typisk som en association med en forklarende tekst og evt. multipliciteter, som illustrerer, at der i virkelighedens verden er en relation mellem begreberne i hver ende af associationen, som kan være vigtig for det system, vi skal designe og implementere.

Klasserne kan være forsynet med attributter, som vi ved Systemdomæneanalysen er kommet frem til, er vigtige for det endelige system at kende og huske.

Derimod har klasserne ikke metoder. Begreber i virkeligheden kan ikke "kalde metoder" på andre begreber – de er jo ikke software. Klasserne på Domænemodellen er heller ikke klasser i vores kode – endnu. Måske bliver de til klasser i det endelige system – det afgør vi først i de senere faser af softwareudviklingen, nemlig når vi skal lave Applikationsmodellen. Skulle det være tilfældet, har vi allerede nu et godt navn for en sådan klasse.

Når man ser på eksemplet, vil man lægge mærke til, at associationerne ikke har pile for enderne, men derimod er der pile på de forklarende tekster – så man kan se, hvilken vej den skal læses/forstås.

Man vil også lægge mærke til en anden ting: attributterne har ikke typer tilknyttet.

Begge disse forhold er en afspejling af, at vi ikke ønsker at fastlægge implementationsdetaljer på dette tidspunkt af udviklingsprocessen. Det kan kræve en nærmere analyse at afgøre om et tal skal gemmes som heltal eller kommatal. Eller om de endelige klasser rent faktisk har brug for at implementere en association, den ene vej eller den anden vej, begge veje, eller måske slet ikke. Men vi ved, at de har en relation i virkelighedens verden.

1.4. Domænemodellens rolle som kommunikationsmiddel

Domænemodellen er først og fremmest et kommunikationsmiddel, der skal bruges af forskellige deltagere i og omkring et projekt. Det kan have mindst to kommunikationsretninger:

1. Udad fra projektet, til kunder, product owner og brugere.
2. Indad til projektet, til udviklere, designere, projektleder, SCRUM master og testere

1.4.1 Domænemodellen udad til

Domænemodellen som vi beskriver den i denne artikel, vil være i form af klassediagrammer med visse specielle notationer og begrænsninger i forhold til et fuldt UML klassediagram.

Denne type diagram bør sagtens kunne læses og forstås af eksterne interesserter uden speciel teknisk indsigt, således at man kan tale med dem om Domænemodellen.

Den kan bruges til at afklare, om man nu har forstået specifikationerne korrekt, fordi at de er et tegnet resume af specifikationer og Use Cases.

Den kan derfor betragtes som et "rigt billede" i en stringent notation, som kan bruges i en dialog med de eksterne stakeholders til projektet.

Det er imidlertid ikke disse egenskaber for Domænemodellen vi lægger mest vægt på i dette kursus og i denne artikel.

1.4.2 Domænemodellen indad til

Domænemodellen er lavet i et sprog, UML, som alle, der deltager i implementationen, skal kunne forstå.

For dem vil Domænemodellen være som en visuel ordbog, der giver navn til de vigtige begreber, der skal arbejdes med, så alle bruger samme navne. Og de vigtige relationer mellem begreberne er identificeret.

Disse navne kan også bruges til de klasser og objekter, som senere vil blive designet og anvendt i implementeringen. Ligesom relationerne vil indgå i overvejelserne, om de skal designes og implementeres med.

Dermed er Domænemodellen med til at danne bro mellem virkeligheden og den implementerede kode, man kan sige at Domænemodellen formindsker det gab, der er mellem virkeligheden – systemets domæne – og de elementer, der skal repræsentere virkeligheden i systemet.

Den beskriver en afgrænsning af systemet og dets domæne, så man ved, hvad vi skal have med, og hvad vi ikke skal have med.

Det er disse egenskaber, vi lægger mest vægt på i dette kursus og i denne artikel: Domænemodellen som det første skridt til at hjælpe os i gang med at designe softwaren – det første skridt fra HVAD til HVORDAN.

En Domænemodel bliver nok aldrig færdig. Hvis vi arbejder med den systematisk i projektet, og gerne med projektets stakeholderne, kan den nå et fornuftigt niveau, specielt med lidt øvelse.

1.5. Hvad er begreber (concepts, conceptual classes, begrebsklasser)

Et begreb, som vi er ude efter i Systemdomæneanalysen er en ide, en abstrakt ting eller et fysisk objekt.

Man må nogle gange strække sin tankegang lidt, for at på "øje" på tingene, og hæve sig op på et lidt højere abstraktionsniveau, end hvis vi begrænsede "Virkeligheden" til kun at være den fysiske verden, vi kan røre ved.

Er et "Salg" en ting? Nej, det er en abstrakt ide, men det er sådanne ting som vores software nogen gange skal beskæftige sig med.

En SMS eller en e-mail – er det en ting? Vi kan ikke røre ved dem, men vi kan tale om dem. Så det må være en ting, bare abstrakt.

For at undgå det lidt tunge ord begrebsklasser – direkte oversat fra engelsk – bruger jeg i denne artikel ordet begreb.

1.6. Metode.

Der ikke er nogen Domænemodel compiler – både desværre og heldigvis! Som ovenfor nævnt, er Domænemodellen et arbejdsredskab og et kommunikationsværktøj, ikke et design- og implementeringsværktøj. Det er derfor de mennesker, der skal læse, forstå og bruge Domænemodellen der er "compileren".

At opstille en Domænemodel med et godt valg af begreber fra domænet og deres relationer, er en aktivitet der kræver øvelse og erfaring.

I det følgende vil jeg beskrive en metode, der kan bruges til at arbejde sig frem imod en god Domænemodel.

1.6.1 Skridt 1. Find begreberne

Første skridt er, tro mod den beskrevne objektorienterede filosofi, at finde begreberne. De følgende afsnit giver en praktisk metode til at komme i gang.

Skridt 1.1: Navneordsanalyse i specifikationer og Use Cases.

I de forrige faser af systemudviklingen, er der brugt meget tid på at specificere, hvad Systemet skal gøre. Ved arbejdet med krav og specifikationer er der kommet en masse tekstbeskrivelser og – ikke mindst – nogle detaljerede Fully Dressed Use Cases.

Første skridt er simpelthen at tage disse frem, og gennemgå dem for navneord. Begreber er ting, koncepter og ideer, der har et navn, ellers kunne vi ikke skrive om dem i specifikationer.

Tag fat med en markeringspen eller en blyant og marker alle navneord i teksterne. Det vil gøre arbejdet overskueligt, hvis man tro mod de iterative principper, tager en eller et par Use Case af gangen, nemlig dem som vi gerne vil starte med i de første iterationer/sprints af systemudviklingen.

Skriv dem så ned på en liste – eller måske endnu bedre – giv straks hvert navneord en gul seddel på en tavle, eller et klassesymbol på et klassediagram.

Der skal ikke bare kigges i aktivitetsdelen (hovedscenariet med extensions) i FDUC. Aktørerne skal også med, de er jo også begreber, der indgår i domænet (se ovenstående tegning).

Det er ikke sikkert at alle navneord er nyttige for at lave en tilstrækkelig men overskuelig Domænemodel. Første filtrering vil være at sortere ting fra, der er for tekniske eller for meget rettet mod implementeringsdetaljer. Og andre ting skal man måske erstatte med et begreb, der er mere abstrakt, end et håndfast ord, der står i specifikationerne.

Et eksempel kunne være "magnetstriben" på et betalingskort. Denne striben indeholder data, fx kortets nummer eller identitet (id), og det er dem, der er mere interessante, når vi skal finde frem til de væsentlige begreber i domænet. Som et andet eksempel, vil vi ofte ligeglade med, om stregkoden på en vare står på en etiket eller er i det ene eller det andet stregkodeformat, det er ting, der bør stå i de ikke-funktionelle krav til systemet.

Nogle af navneordene vil måske dække over begreber af simpel karakter. Så er de måske kandidater til at være en attribut på en af de andre begreber. Gør det med det samme på den gule seddel eller klasseSymbolet for det andet begreb.

Det kræver øvelse at finde det rigtige niveau, og måske også et par gennemløb af alle skridtene i dette afsnit. Når det er sagt, så hellere lidt for mange begreber og detaljer i første skud, de kan altid fjernes senere.

Skridt 1.2: ofte sete IT begreber (kategorier)

Erfaringen med at udvikle systemer viser, at der er begreber der er nyttige i sammenhænge, man genkender mellem mange forskellige systemer.

Måske gemmer der sig bag beskrivelserne i specifikationer og Use Cases begreber på højere abstraktionsniveau, som det vil være relevant at fremdrage.

Eller der kan være nogen begreber, man ikke har fået med. Eller måske er der slet ikke nogen Use Cases eller særlig gode Specifikationer ☺ !

Her er en liste af kategorier med eksempler på sådanne "ofte sete IT begreber".

Generelt Begreb	Eksempler
Transaktion – en afsluttet forretningshændelse, som naturligt afgrænses en aktivitet, der skal kunne skelnes fra andre slags og andre af samme slags	Et Salg En Betaling En Tankning En Udcheckning på biblioteket/supermarkedet En Flyreservation
Transaktionslinje – en af flere linjer, der repræsenterer et begreb, der indgår i eller dækkes af transaktionen	En Varelinje på en bon ved et salg En Boglinje i en udcheckning på biblioteket
Fysisk genstand – en fysisk objekt, der fx matcher en transaktionslinje	Selve Varen Selve Bogen Et Lånerkort
Service – en service eller aktivitet, som er en "vare" i forbindelse med fx en transaktion	Selve Flyveturen (<i>Flight</i> , med rute og tidspunkt) Selve retten til et Sæde
Register – hvor registreres Transaktionen	Hovedbog Kasseapparat Systemet Dagsjournal
Fysisk sted for Transaktion eller Service	Bibliotek Lufthavn Sæde
Hændelse som vi ønsker at registrere (behøver ikke at være en Transaktion)	Et Salg En Betaling En Flyvetur (<i>Flight</i> , med rute og tidspunkt)
Katalog	Bibliotekets Kartotek Supermarkedets Ugeavis Flyveplanen Køreplanen

Generelt Begreb	Eksempler
Fysisk container – som kan have flere fysiske ting i sig	Lagerrum Container Restaurationsbord
Eksternt System	Flykontrollen Betalingscentralen
Aktør	Kunde Kasseassistent Mekaniker Stewardesse
Beskrivelse – generisk information, som er vigtig at bevare når alle objekter er væk	Videobeskrivelse Bogbeskrivelse Varebeskrivelse
Transaktionsbevis	Bon Kvittering Udlånseddelen Reservationsbekræftelse
Betalingsmiddel	Kontanter Kort Kredit Point Mærker

Brug denne liste af kategorier til at overveje, om der kan findes nogle flere begreber, som måske ikke direkte fremgår af specifikationerne, måske fordi de er underforståede, og dermed ikke står der eksplisit.

Dette skridt kræver selvfølgelig også erfaring med udvikling af systemer for at gøre det godt, men det er med til at gøre vores Domænemodel mere komplet.

1.6.2 Skridt 2. Find relationerne

Nu har vi fundet et sæt af begreber, som vi mener er relevante for at kunne beskrive vore system. De ligger nu spredt ud foran os på tavlen eller på et klassediagram.

For at gøre disse klasser til en Domænemodel skal vi nu i gang med at beskrive, hvilke relationer de har til hinanden.

Lad os igen gå praktisk frem.

Skridt 2.1: udsagnsordsanalyse i specifikationer og Use Cases

I specifikationerne vil der stå en masse udsagnsord (verber), der beskriver hvordan begreberne agerer eller forholder sig til og med hinanden.

Dem kan man nu markere eller fremhæve i teksterne, gerne på en sådan måde at de kan skelnes fra navneordene. Man kan også bruge teksten som en checkliste, og for hvert udsagnsord, man finder, tegne en relation i form af en association imellem de begreber, der indgår, og skrive udsagnsordet på associationen. Bagetter kan man så markere udsagnsordet, når det er behandlet.

For overskuelighedens skyld beskriver man nogle gange flere relationer på den samme associationsforbindelse, ved at tilføje flere tekster. Men der skal kun stå ét udsagnsord i hver tekst. Måske skal der vælges

mellem aktiv eller passiv, og måske er man nødt til at tilføje et 3. begreb og/eller et forholdsord. Grundled og genstandsled (eller hensynsled) vil fremgå automatisk af, at disse står i hver deres ende af associationen, derfor skal de IKKE stå i teksten på associationen.

For at gøre Domænemodellen nemmere at læse, tilføjer man næsten altid en læsepil, der gør det sikrere at forstå relationen. Denne kan indsættes som et pilehoved parallelt med teksten, brug fx flowpilene kendt fra IBD-diagrammerne, eller man kan indlejre det i teksten, evt. ved at bruge tegnene <,>,^ og v. Hvis relationens læseretningen falder sammen med den almindelige læseretning fra venstre til højre, kan læsepilen udelades.

Men der er IKKE pile i enden af associationerne! Retningspile på associationer er implementationsdetaljer for rigtige klasser i det endelige program, der fortæller noget om at den ene klasse skal kende den anden for at kunne kalde dens metoder, el. lign. Så langt er vi slet ikke endnu.

Skridt 2.2: ofte sete IT strukturer

Ligesom for begreber, kan man også lave en checkliste over relationer, man kender fra mange forskellige systemer.

Her er en liste med nogle kategorier af "ofte sete IT strukturer" med eksempler.

Generel Relation	Eksempler
A er en transaktion som hænger sammen med en anden transaktion B	Salg – Betaling Reservation – Afbestilling
A er en transaktionslinje i transaktionen B	Faktura – Varelinje
A er et produkt eller en service for en transaktion eller transaktionslinje B	Varelinje – Vare Reservation – Selve flyveturen
A er en fysisk eller logisk del af B	Kasseapparat – Pengeskuffe Fly – Sæde
A er fysisk eller logisk indeholdt i B	Container – Vare
A er en beskrivelse for B	Flyrute – Selve flyveturen Bogbeskrivelse - Bogeksemplar
A er en rolle/aktør relateret til aktionen B	Salg – Kunde
A huskes/registreres/rapporteres i B	Kasseapparat – Salg Journal – Konsultation
A er medlem af B	Firma – Ansat Supermarked – Kasseassistent
A er en underorganisation af B	Firma – Afdeling
A bruger eller ejer B	Kasseapparat – Kasseassistent Kaffeautomat – Firma

Brug dem til at tænke over, om der er nogle relationer, man ikke lige har fået med, måske fordi de er underforståede i specifikationerne. Tag dem med på tegningen, for at gøre Domænemodellen mere komplet.

1.6.3 Skridt 3 Find multipliciteterne

En vigtig detalje i mange relationer er hvor mange instanser af begreberne, der indgår.

Her bruges multiplicitetsnotationer på associationerne, så man kan se, når man læser relationen, om hvor mange, der er i hver ende af relationen.

I de fleste tilfælde vil det være nok med at angive at der er én instans i en ende, skrevet som et ettal "1", eller at der er "nul, en eller mange", skrevet som en stjerne "*". Det kan være nødvendigt at angive muligheden for at der ikke er nogen instans i den ene ende af relationen, så bruger man "0..1", eller at der er "mindst en", så bruger man "1..*".

Er det vigtigt, ud fra specifikationen, at angive et nøjagtigt antal eller maksimum antal, gør man bare det.

Er det en 1-til-1 relation, behøver man ikke at skrive multipliciteter på associationen.

Alle relationerne gennemgås og opdateres med multiplicitet ud fra ovenstående retningslinjer.

1.6.4 Skridt 4 Finpudsning

Attribut eller selvstændigt begreb?

Domænemodellen kan blive mindre kompleks, men måske også mindre dækkende, ved at vælge at lade begreber fra Domænet blot være attributter til et andet begreb.

Følgende retningslinjer kan man bruge til at tage sin beslutning om et begreb er en attribut, eller skal vises som sit eget begreb.

- Hvis det fylder noget fysisk, er det et begreb
- Hvis det har en kompleks struktur med underattributter (som er vigtige for dette system), er det et begreb
- Hvis det har aktiviteter, er det et begreb
- Hvis det er simpelt, med en veldefineret værdimængde, er det en attribut

Man kan fx spørge sig selv, om man kan tænke på begrebet X som et tal eller en simpel tekst i den virkelige verden. Hvis ikke, er det et begreb i sig selv, ikke en attribut.

Domænemodellen gennemgås, også for de attributter, man allerede har oprettet i skridt 1.1 og 1.2.

Systembegreb

Ud fra inputtet til Systemdomæneanalysen kan man til tider stå med en aktivitet – et udsagnsord, der ikke har noget naturlig tilhørsforhold til en relation mellem de fundne begreber. Her kan det være nyttigt at bruge et Systembegreb, og indføre det på Domænemodellen. Specielt hvis det står direkte nævnt i specifikationerne. Se også nedenfor.

Aktør – tændstiksmand eller klasse?

Hvis systemet skal huske noget om en bestemt Aktør, fx navn eller CPRnr, kan man tegne aktørerne som en klasse, med de attributter, der er nødvendige for systemets funktion. Evt. kan man markere klassen med stereotypen <<aktør>>. Ellers er det fint, at tegne Aktørerne som en tændstiksmand.

Komposition, aggregering og arv i Domænemodellen

Der er sjældent brug for andre klasserelationer end association i det klassediagram, som Domænemodellen er. Her er et par kommentarer om de andre. Ofte er disse mere udspecifiserede relationer implementeringsdetaljer, der ikke skal/bør vises på Domænemodellen.

Aggregering er en "har en" relation, uden ejerskab. Den er en svag relation, og kan ligeså godt tegnes som en association, uden at dybere viden går tabt. Hvis man kun kan finde på at kalde relationen "har en", som er en ret intetsigende betegnelse på Domænemodellen, kan man prøve at vende retningen, og se, om der ikke er en bedre beskrivelse af relationen.

Komposition er en "har en" relation, hvor levetiden for det underordnede begreb er stærkt sammenkoblet med ejeren, ligeså stærkt som en attribut. Kun hvis det underordnede begreb ikke kan bruges/ses alene eller har en relation til andre begreber på Domænemodellen, og levetiden er stærkt forbundet, er der grund til at bruge komposition. Som med Aggregering kan man prøve at beskrive relationen "set fra den anden ende".

Arv er en "er en" relation, en generalisering, hvor en eller flere attributter og relationer er fælles for en række begreber, og basisklassen vil derfor være et mere abstrakt begreb. Arv kan bruges til at forenkle Domænemodellen, men brug den kun, hvis de afledte klasse adskiller sig fra hinanden ved forskellige relationer, som man gerne vil have med i Domænemodellen. Er det blot simple attributter el. lign. der adskiller dem, bør man identificere og bruge den abstrakte basisklasse med passende attributter, evt. en "type" attribut. Så har man lavet en forenkling, uden at tilføje yderligere begreber og relationer.

Oprydning

På dette tidspunkt kan også overveje, om der er kommet for mange begreber med, fx fra skridt 1.2, som man ved nærmere overvejelse indser, ikke har betydning for dette system. Så kan man jo slette dem igen, samt deres relationer, eller finde ud af om andre begreber skal overtage relationerne, fordi det slettede begreb er en del af det andet begreb.

1.7. Hvad er en Domænemodel IKKE!

En Domænemodel er IKKE et hardwaresystemdesign eller en softwaresystemarkitektur. Disse ting beskrives bedre med andre diagrammer, fx BDD og IBD for hardware og Applikationsmodellen for software.

Den beskriver først og fremmest de dele af virkeligheden, der er interessante i forhold til vores system, og hvordan de interagerer med hinanden og med systemet.

Er der underdele af systemet med i Domænemodellen, skal det være fordi de fx er synlige udefra, og en aktør interagerer med den specifikke del af systemet.

Den indbyggede CPU, eller netværkskortet, eller andre (hardware) implementationsdetaljer optræder IKKE i Domænemodellen. Man kan fx skjule dem i en System klasse, specielt hvis der i specifikationerne står noget om hvad systemet gør, uden at det er vigtigt hvilke dele af systemet, der gør det.

Domænemodellen er heller ikke en dynamisk flowmodel, der angiver hvordan information eller materiale flyder mellem forskellige dele af systemet. Den slags information skal beskrives på IBD'erne og Applikationsmodellerne.

Med andre ord, pas på med at Domænemodellen ikke kun indeholder hardware komponenter, for så bliver modellen let bare det samme som et IBD diagram. Kontroller at der er objekter med, der beskriver information som systemet skal huske på, det er de objekter, som er mest interessante for softwaren.

Applikationsmodeller

I2ISE

UCs er vigtige!

Applikationsmodellen – slår bro over kløften

- Vi har brugt masser af tid til at skrive UCs og lave Domænemodel(ler)
- I dag får vi nytte af dem!
- Vi vil bruge dem til at slå bro over kløften mellem **Hvad** systemet skal gøre (krav/specifikationer) og **Hvordan** det skal gøres (design)
- Vi vil bruge UCs som *design drivers* – input til designprocessen
- Så:

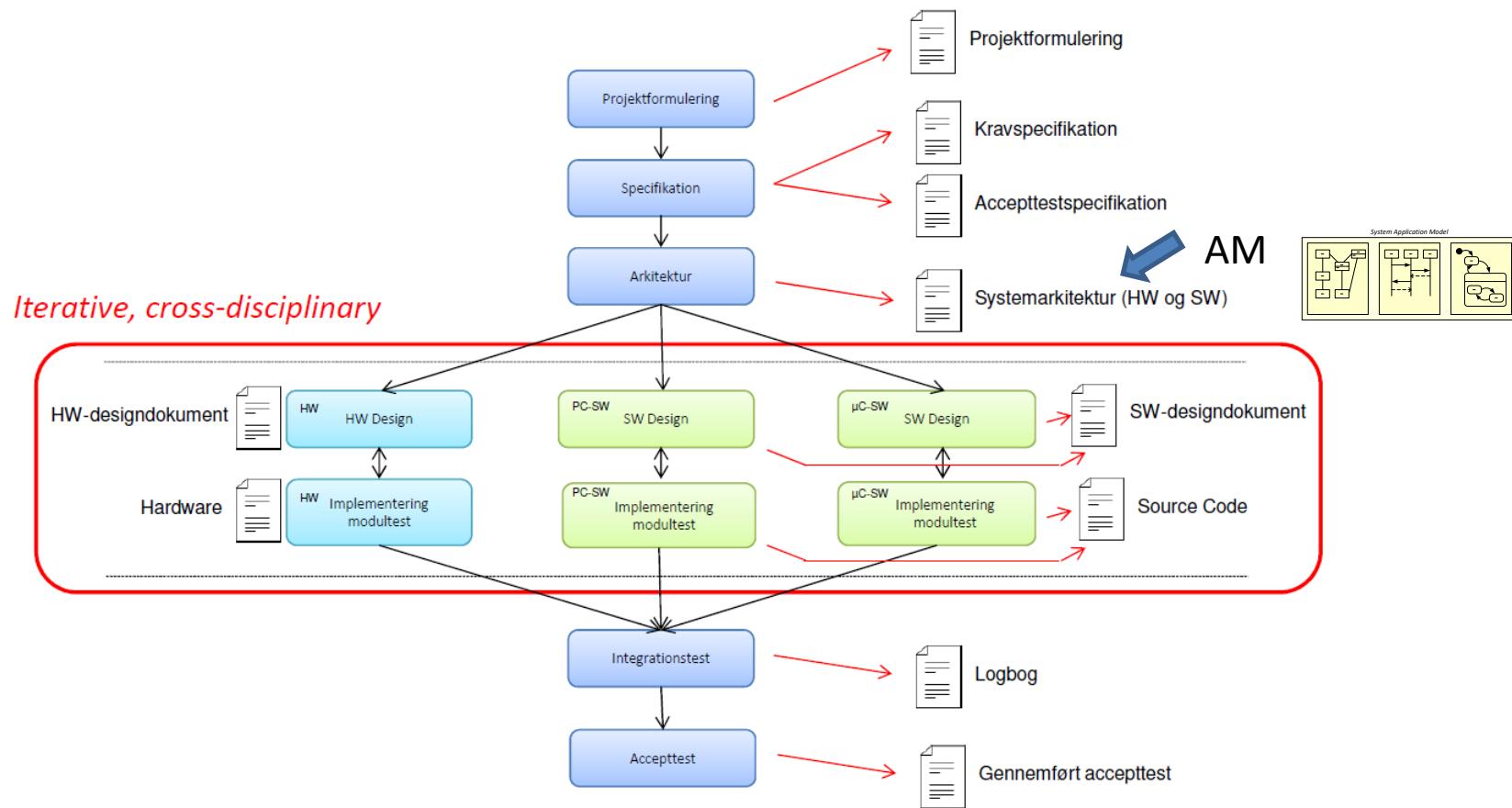
UCs er vigtige!

Hvad er Applikationsmodellen?

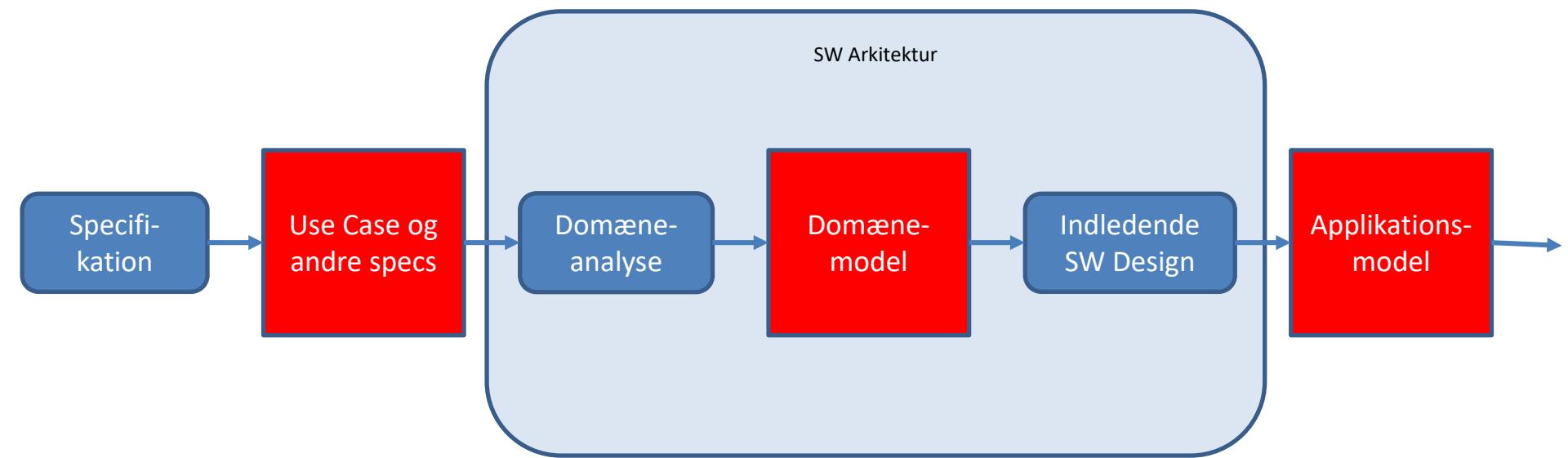
- *Applikationsmodellen* – AM – er første skridt i designprocessen!
- Den vil pege på relevante klasser/moduler som designet bygges op på!
- Den vil beskrive hvordan disse interagerer
- Applikationsmodellen er en del af SW Design afsnittet i Systemarkitektur-dokumentet.

AM's plads i dokumenterne

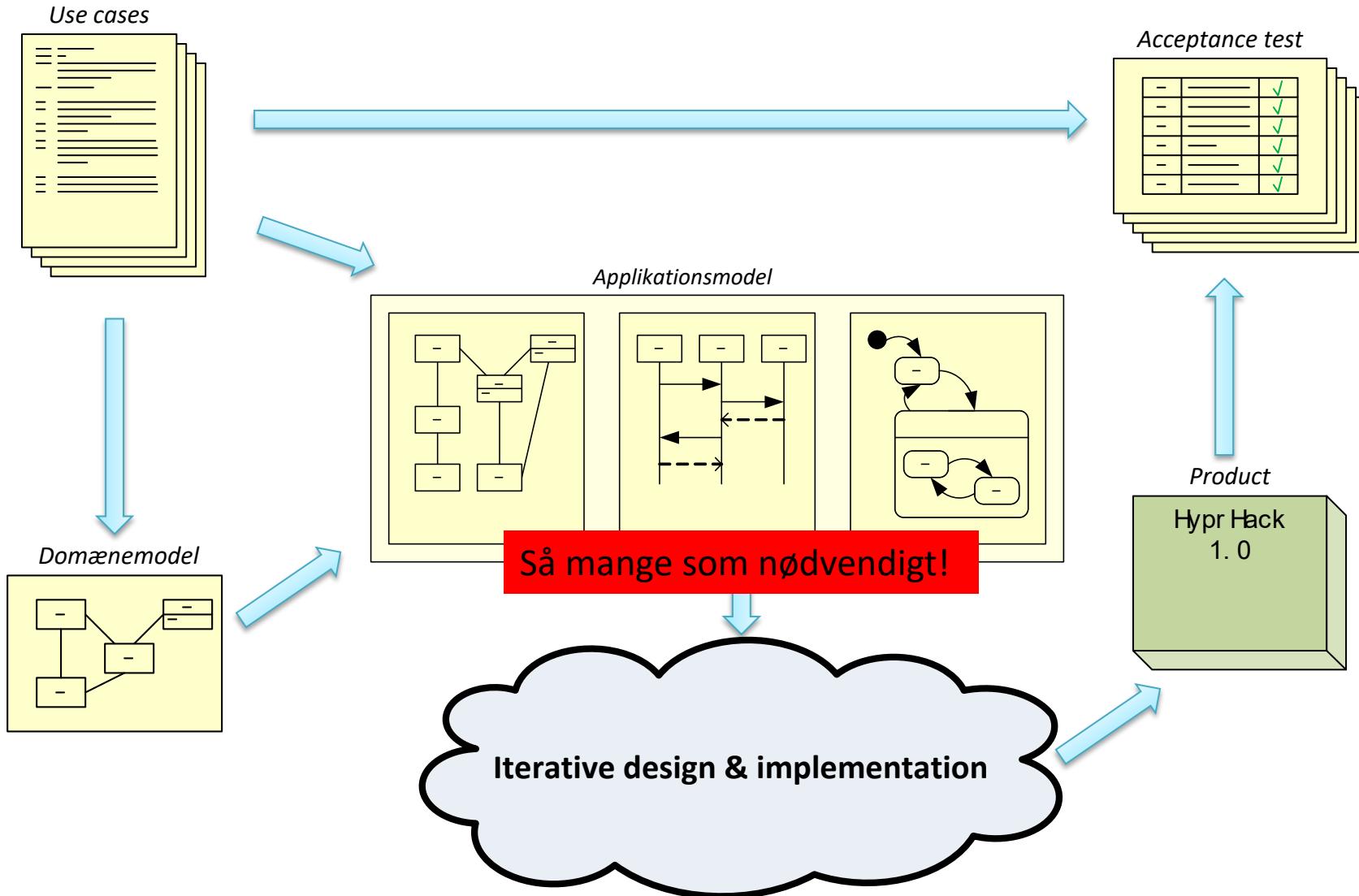
The ASE Process



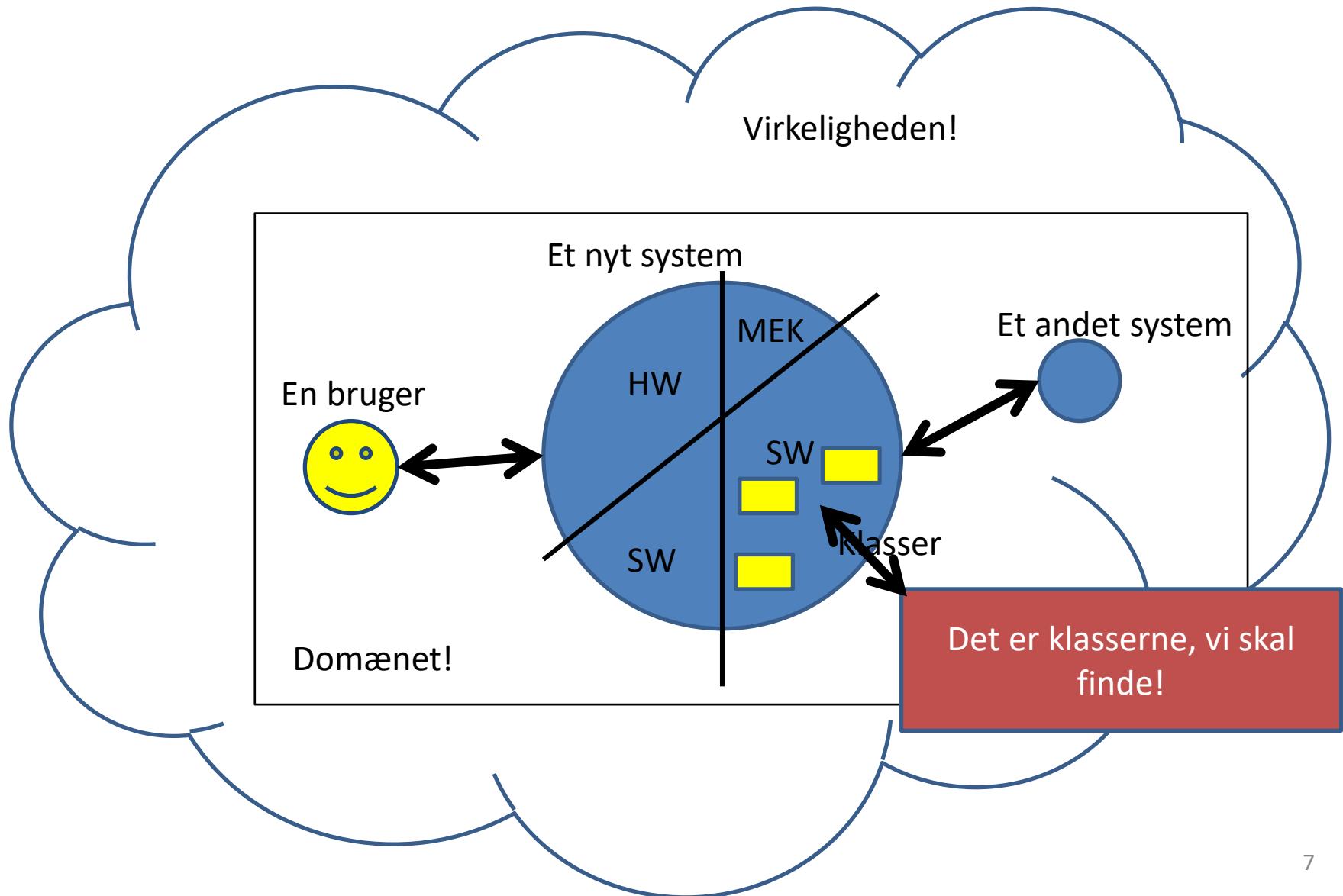
SW Arkitektur – fra UC til Design



Applikationsmodellens plads i det store billede



Virkeligheden og systemet



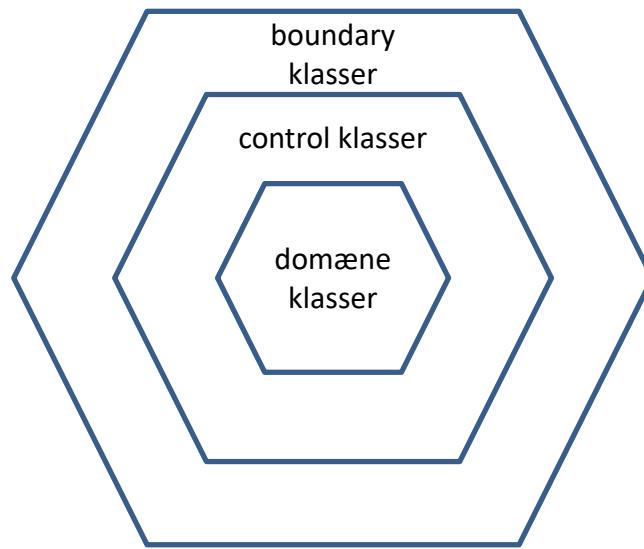
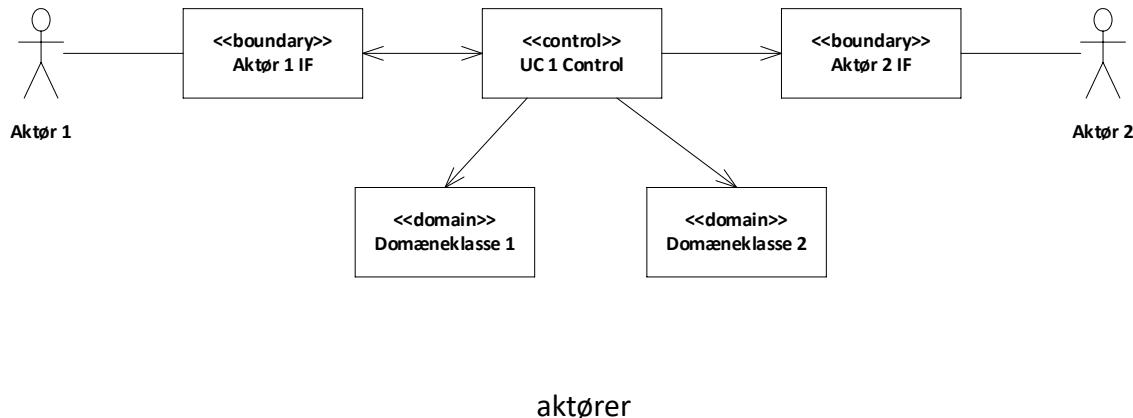
Arkitektur – hvad er det?

- Vi mangler en god ide til at organisere disse klasser!
- Det kaldes Arkitektur
- Vi har ganske vist Domæneklasserne
- Men hvordan får vi styret afviklingen af UC?
- Det bør Domæneklasserne ikke vide noget om!
- Og Domæneklasserne bør ikke vide noget om Hardware og andre interfaces til omverdenen

Vores arkitektur

- ... hedder *Entity-Control-Boundary* pattern
- Dette er et velkendt og velforprøvet *Architectural Pattern*
- Et *pattern* er et mønster, som man kan genkende i mange godt designede og godt fungerende applikationer
- Vi vil her kalde *Entity* klasser for *Domain* klasser

Domain – Control - Boundary



Applikationsmodellen

- Applikationsmodellen er sammensat af følgende 3 typer af diagrammer:
 - *Klassediagram* (cd) for strukturen (statisk)
 - *Sekvensdiagrammer* (SEQ) og
 - *Tilstandsdiagrammer* (STM) for aktiviteter (dynamisk)
- Der laves et tilstrækkeligt antal **sæt** af disse til at beskrive **alle UCs**
- UC bruges til at konstruere dem

Applikationsmodellen – Step 1

- Applikationsmodellen opbygges skridt for skridt, hvor hvert skridt styres af én UC

Step 1.1: Vælg den næste fully-dressed UC til at designe for
(hvordan?)

Er der tvivl, er en klasse boundary klasse før den er domænekasse

Step 1.2: Identifier alle involverede **aktører** i UC →
Boundary klasser

Step 1.3: Identifier **relevante klasser** i Domænemodellen som er involveret i UC → **Domain klasser**

Step 1.4: Tilføj én UC *control* → **Control klasse**

Nogle hvad for nogle klasser?

- Applikationsmodellen består af 3 forskellige klassetyper:
Boundary, *domain*, og *control* klasser
- *Boundary* klasser repræsenterer UC *aktører*
 - De er aktørernes interface **til systemet** (UI, protokol, ...)
 - De gør systemet **synligt for aktørerne**
 - Indeholder ingen "business logic" – dvs. ingen styring af UC
 - Mindst 1 per aktør, deles mellem de UCs der har samme aktører
 - Bør forsynes med stereotypen «boundary»
- *Domain* klasser repræsenterer systemets *domæne*
 - Data, domæne-specifik viden, konfigurationer, etc.
 - 0, 1 eller flere, deles mellem de UCs der bruger samme begreber
 - Kan forsynes med stereotypen <>domain<>

Nogle hvad for nogle klasser?

- Applikationsmodellen består af 3 forskellige klassetyper:
Boundary, *domain*, og *control* klasser
- *Control klassen* indeholder UC'ens *business logic*
 - Den styrer ("executes") UC'en ved at interagere med *boundary* og *domain* klasserne
 - Den skal have navn efter UC'en
 - Typisk er der 1 per UC eller 1 som deles mellem nogle få UCs
 - Bør forsynes med stereotypen «control»

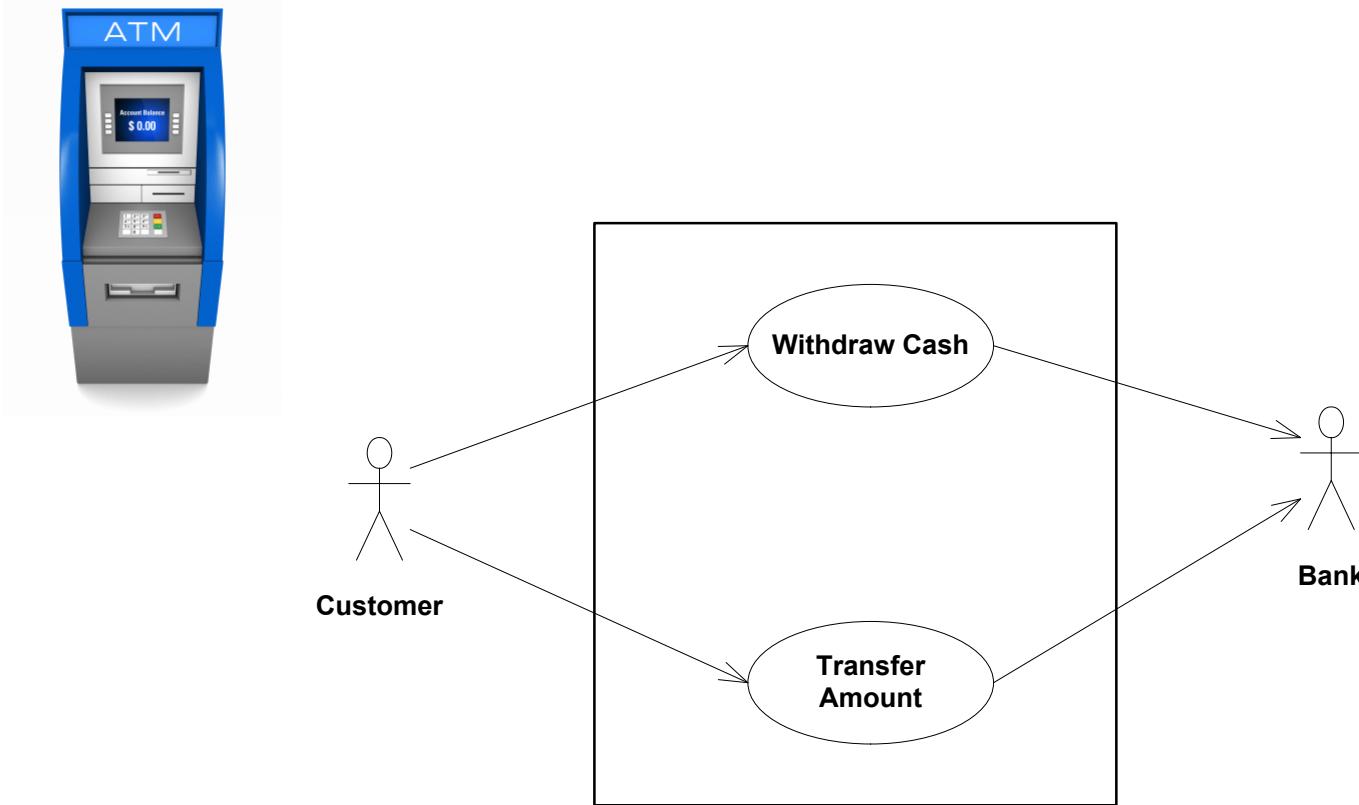
Q&A Domænemodeller

- Gå til padlet, link på BB

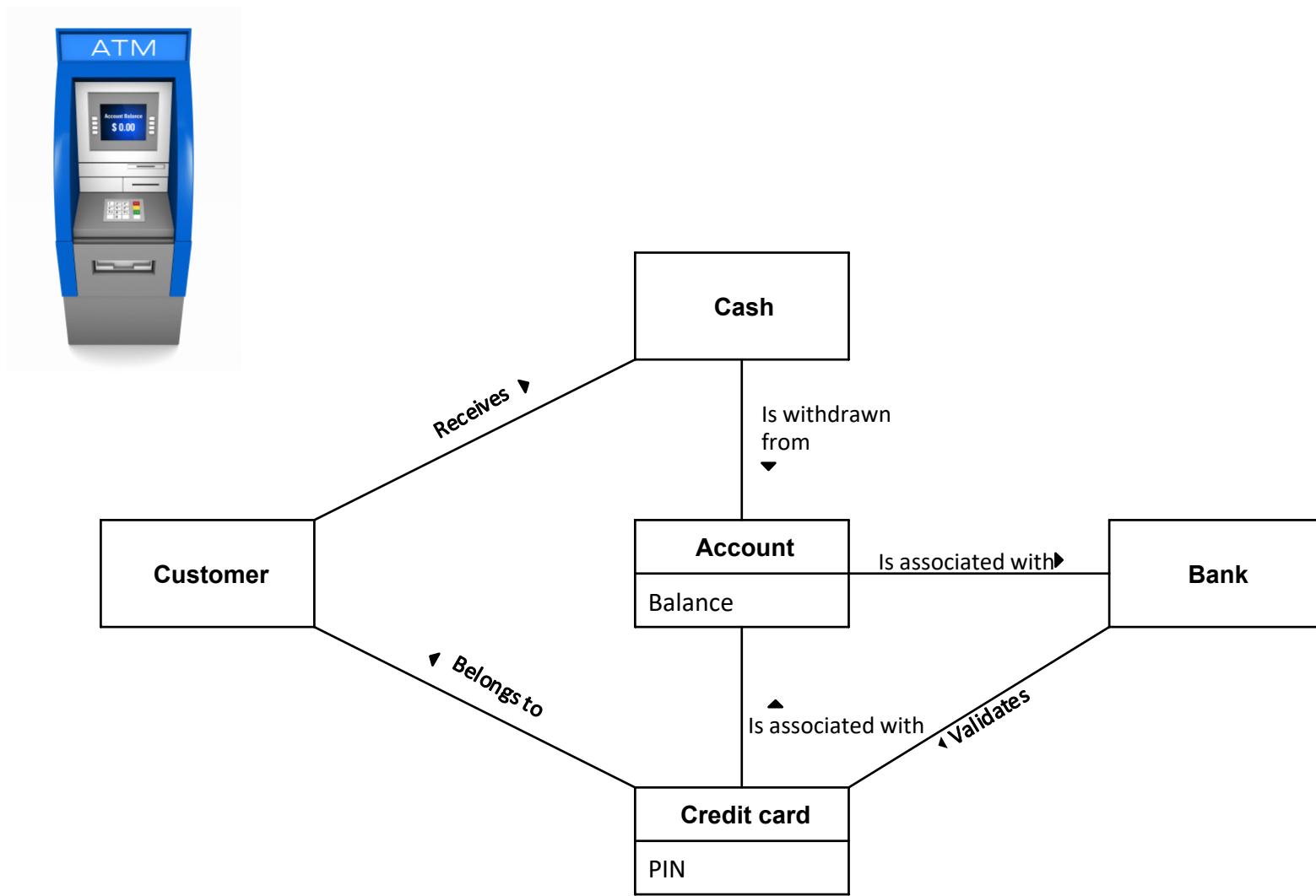
Kontant-/Bankautomaten (ATM – Automatic Teller Machine)



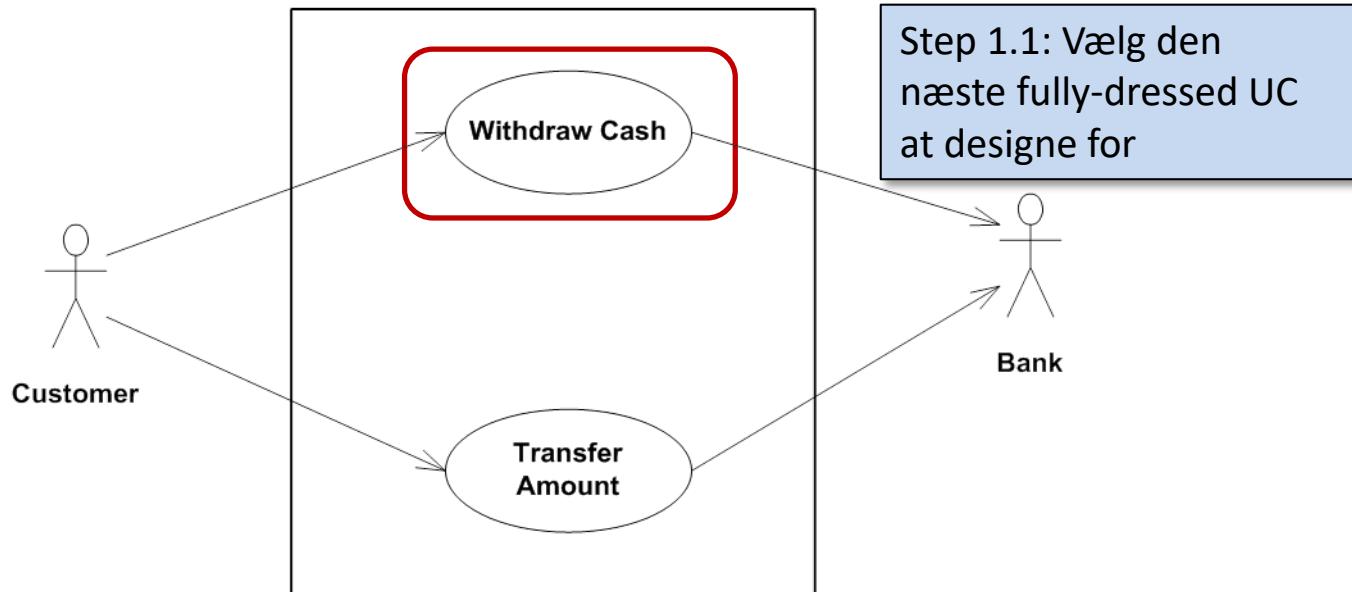
ATM Use Cases



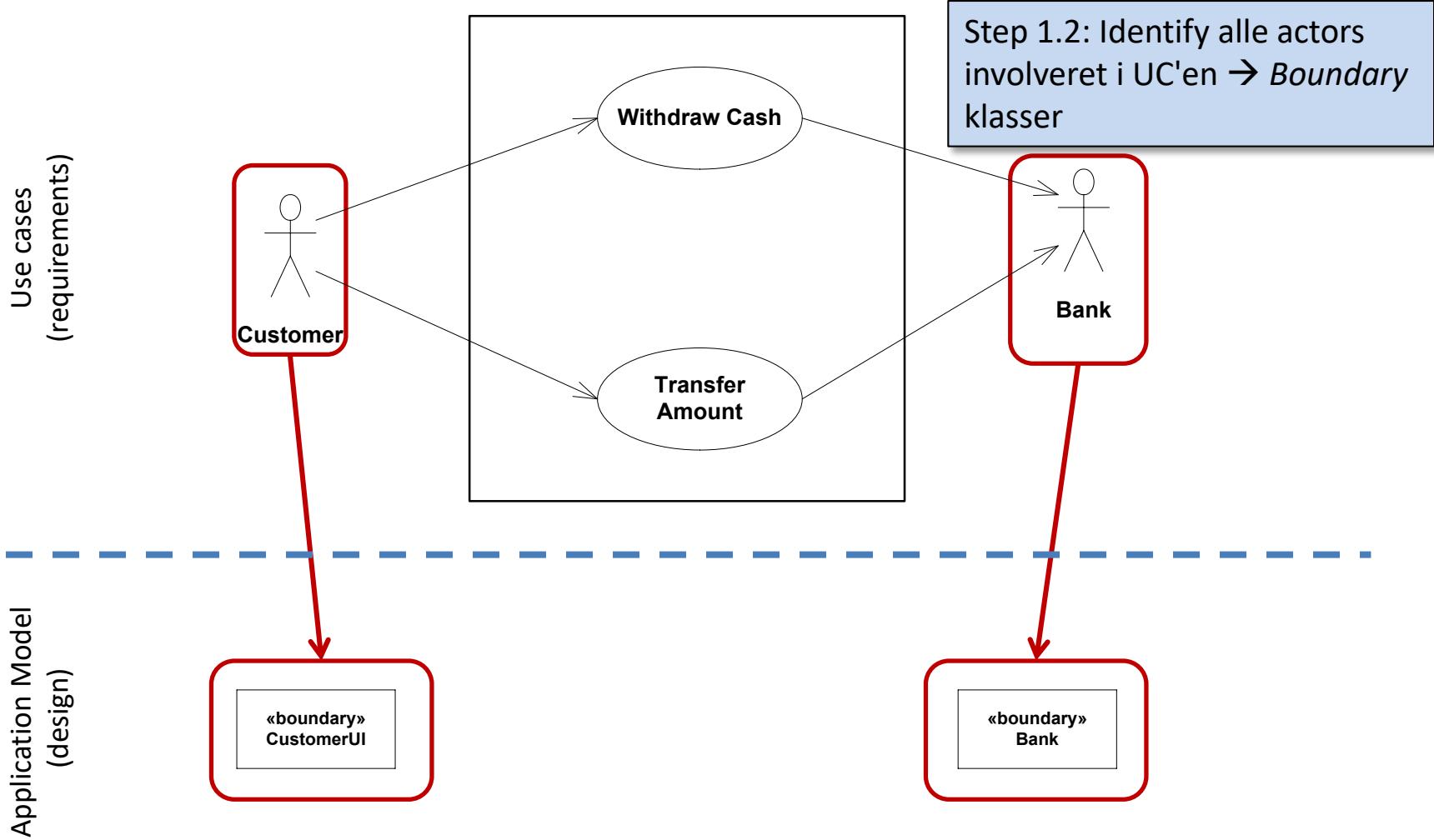
ATM Domænemodel



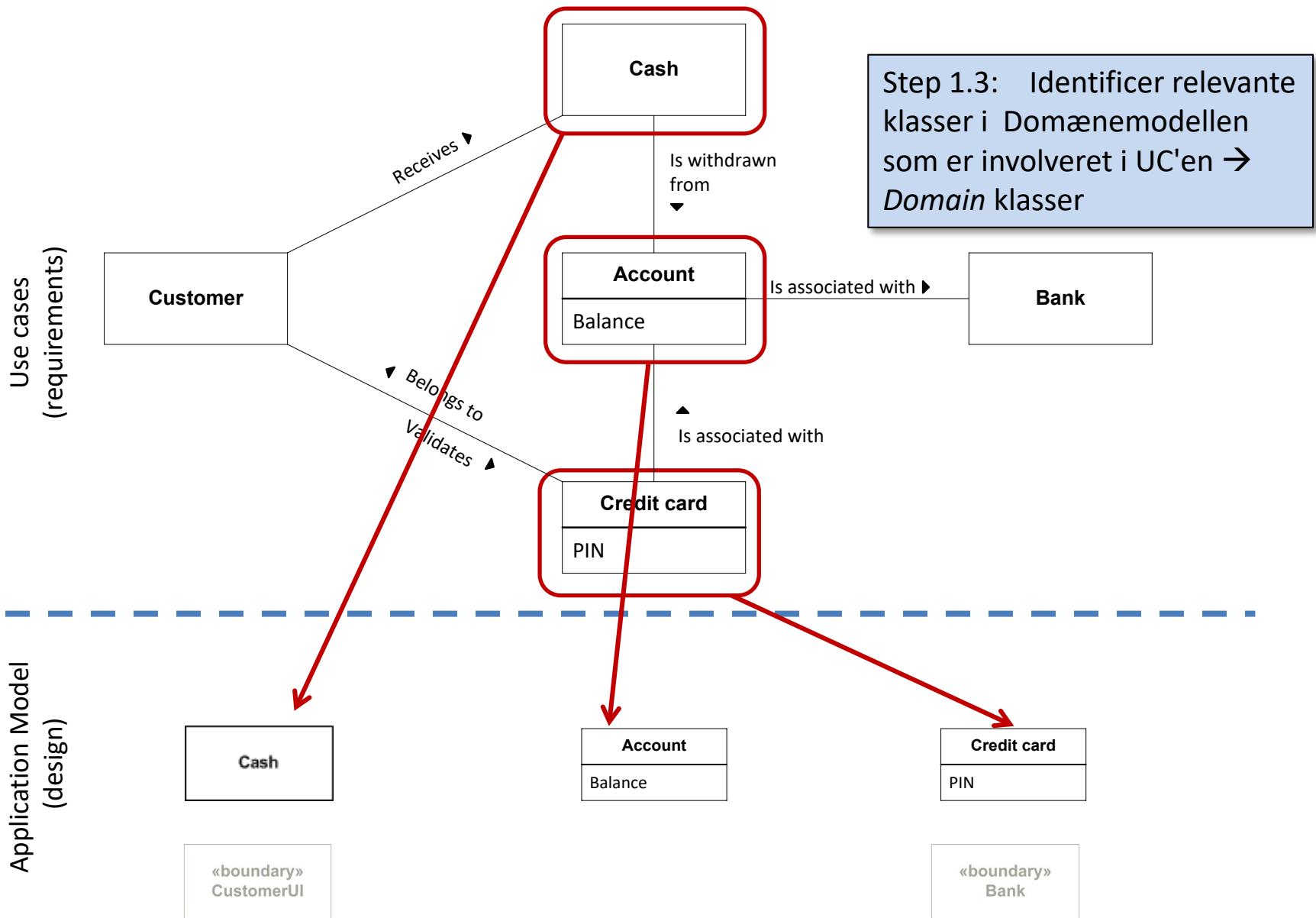
ATM step 1.1: Vælg den næste UC



ATM step 1.2: Actors -> *boundary* klasser

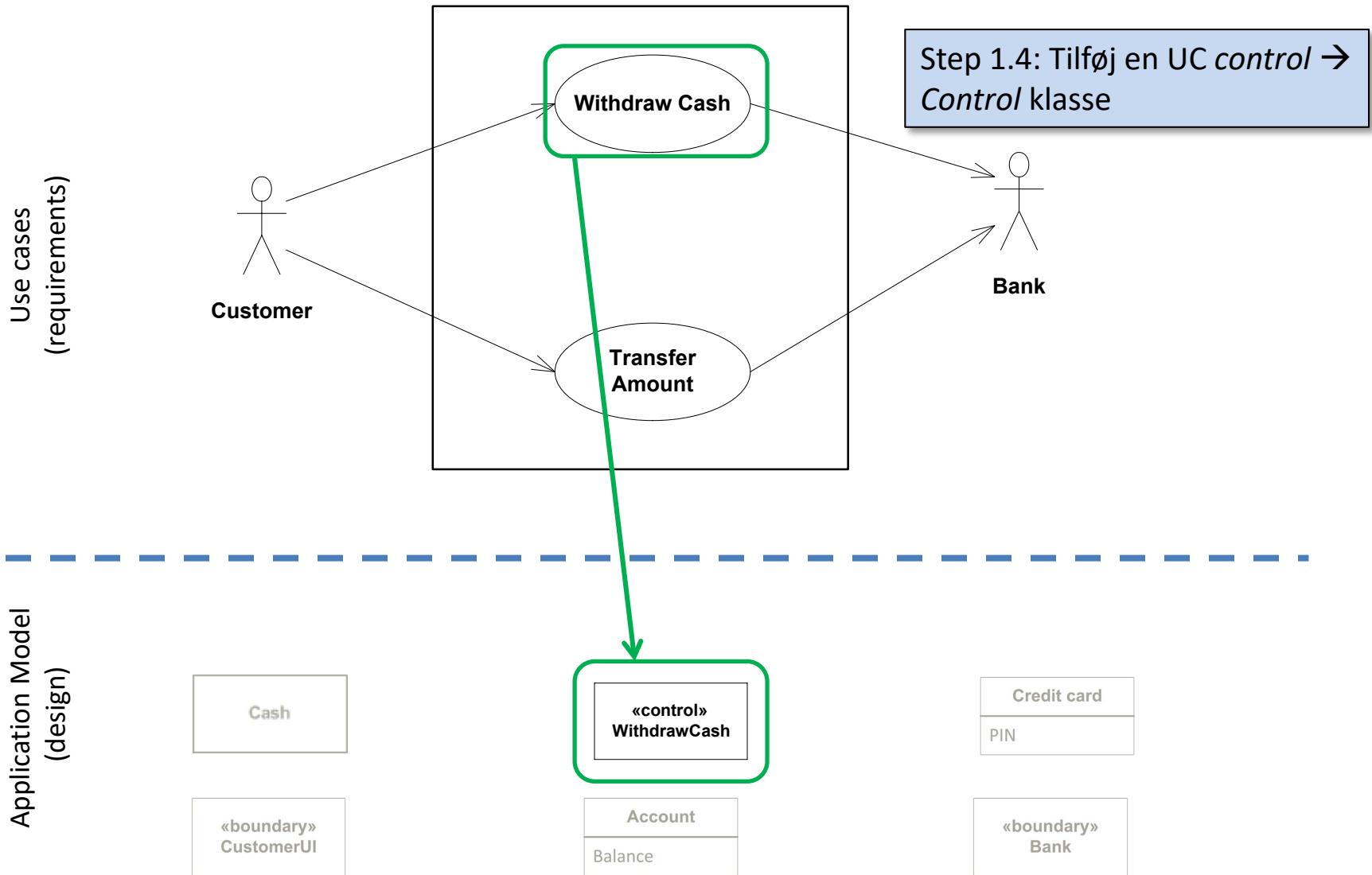


ATM step 1.3: *Domain klasser*



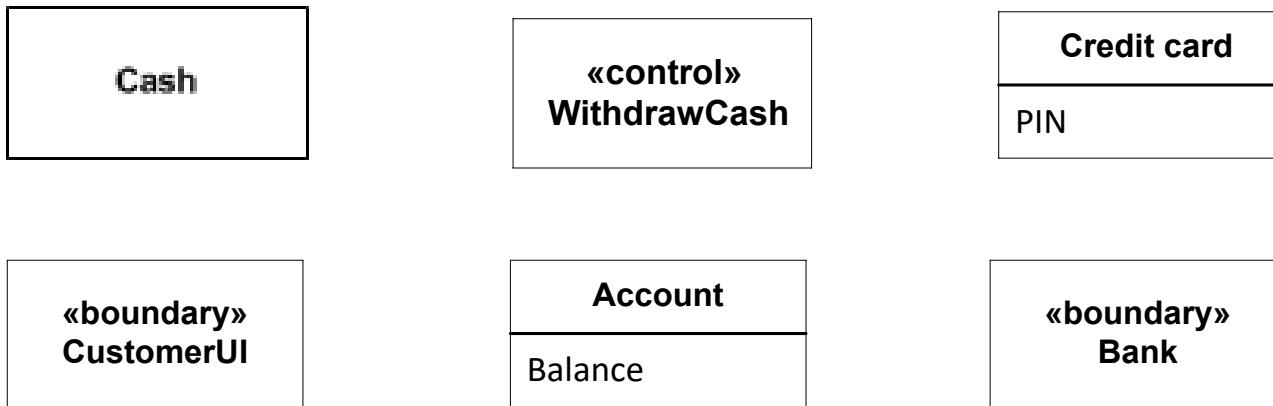
ATM step 1.4

UC control -> Control class



Step 1 færdigt – så langt, så godt

- Vi er nu færdige med Step 1 og har identificeret 6 kandidater som SW klasser for vores indledende design
- For at komme så langt, brugte vi vores *use case* og vores *Domænemodel*

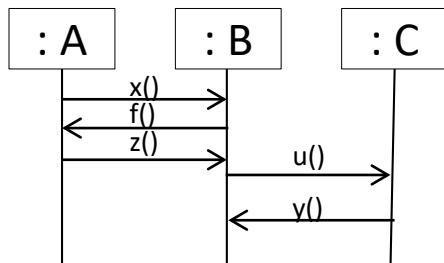


- Nu skal vi tilføje aktiviteter – det er Step 2

Principperne for step 2.1-4: Gå igennem hovedscenariet for UC og opdater løbende

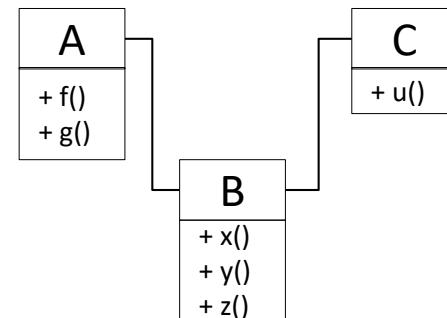
Sekvensdiagram:

- Tilføj kald af objekternes metoder



Klassediagram:

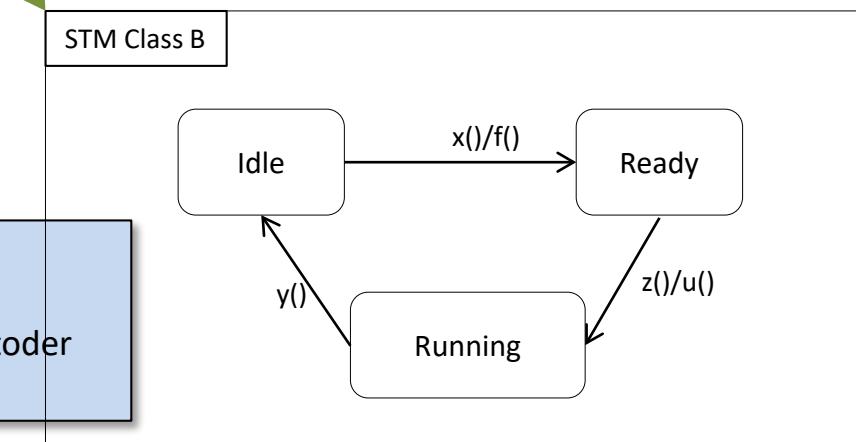
- Tilføj metoder til klasserne
- Opdater associationer



Opdateres parallelt!

Tilstands/aktionsdiagram:

- Lav det for de klasser der har tilstande
- Triggerne til transitionerne er kald til klassens metoder
- Aktionerne er kald til andre klassers metoder



Applikationsmodellen – Step 2

- Samarbejdet mellem klasserne udledes nu fra UC

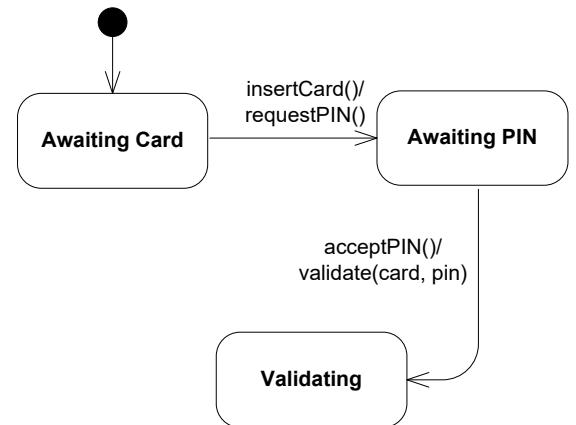
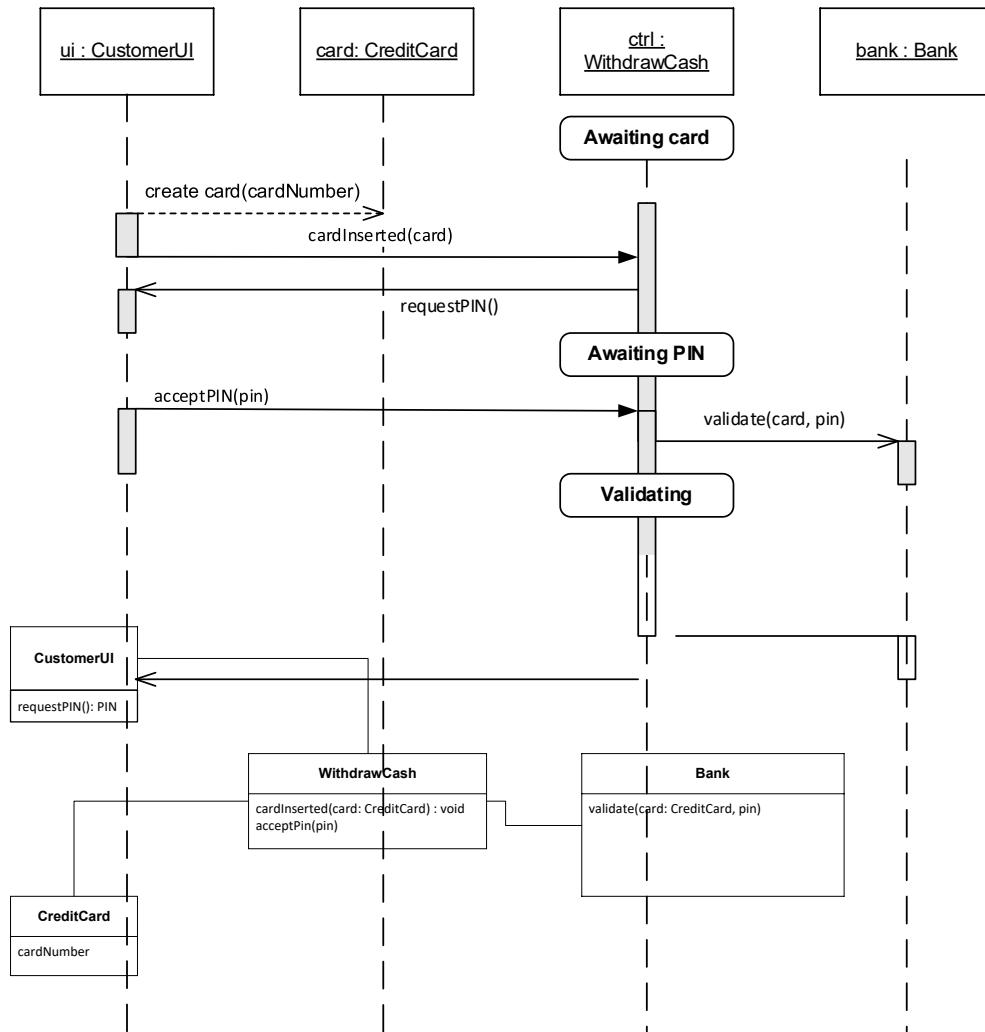
- Step 2.1: Gennemgå UC's hovedscenarie skridt-for-skridt og udtænk hvordan klasserne kan samarbejde for at udføre skridtet
- Step 2.2: Opdater sekvens- og klassediagrammet for at beskrive samarbejdet (metoder, associationer, attributter)
- Step 2.3: Hold øje med, om der er state-baserede aktiviteter og opdater STMs for disse klasser (tilstade, triggere, overgange, aktioner)
(Step 2.3 springes over hvis der ikke er nogen tilstandsbaserede klasser)
- Step 2.4: Verificer at diagrammerne passer med UC (postconditions, test)
- Step 2.5: Gentag 2.1 – 2.4 for alle UC extentioner. Finpuds modellen.

- Alle 3 diagrammer (cd, SEQ, STM) opdateres *parallel/samtidigt* under dette arbejde

Steps 2.1-2.4 for UC Withdraw Money

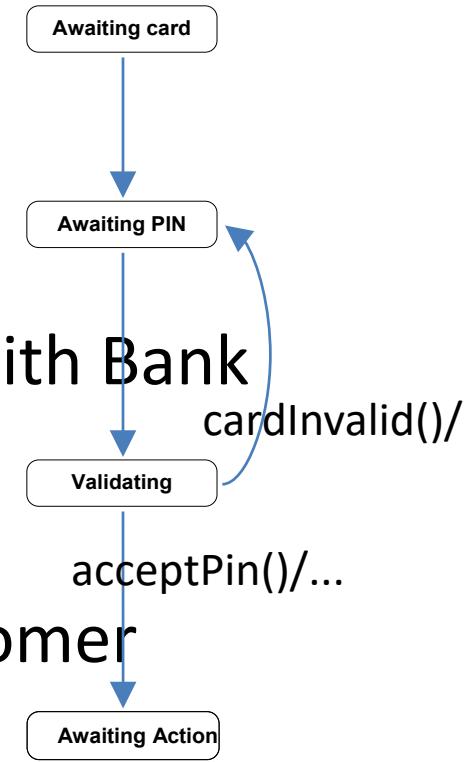
Main scenario:

- 1. Customer inserts credit card in System
- 2. System requests Customer's PIN code
- 3. Customer enters PIN code
- 4. System validates card info and PIN code with Bank



Find STM fra hovedscenariet med extensions

1. Customer inserts credit card in System
2. System requests Customer's PIN code
3. Customer enters PIN code
4. System validates card info and PIN code with **Bank**
5. Bank validates card
[Ext. 5.1: Invalid PIN entered]
6. System requests desired action from customer
7. Customer selects “Withdraw Cash”
8. ...



Applikationsmodellen – hvad nu?

- Fortsæt med næste UC
- Efterhånden som man tilføjer flere UC'er vil man opdage at man kan genbruge nogle af de allerede fundne klasser
 - *Domain* og *boundary* klasser dukker ofte op igen
 - Forskellige *domain* klasser er måske så nært beslægtede, at de kan slås sammen til en
 - Nogle gange kan også *control* klasser slås sammen
- At tage det rigtige valg mellem genbrug, slå sammen eller introducere nye klasser kommer med erfaringen

Hvad så NU?

Applikationsmodellen bruges til det første
kodedesign

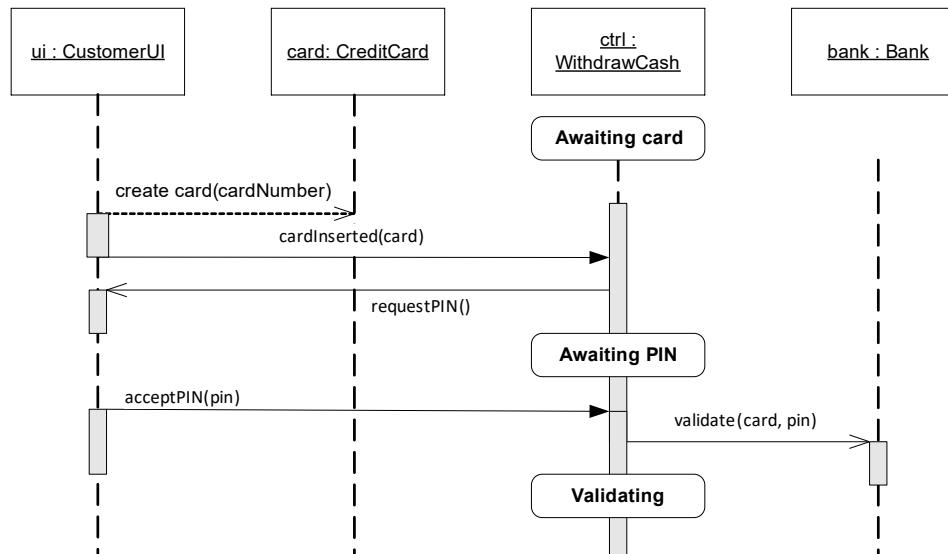
Mere om dette i lektion 21 og 22

Så er det jeres tur: Gør Applikationsmodellen for UC *Withdraw Cash* færdig!

- Den fulde tekst for UC Withdraw Cash findes på BlackBoard. I skal:
 - Gøre Applikationsmodellen færdig for hovedscenariet for UC'en
 - Udbygge Applikationsmodellen med alle extensions for UC'en
- Udgangspunktet er de 3 diagrammer vi har lavet her ved gennemgangen – ses nedenfor
- Arbejd videre nu og hjemmearbejde til næste gang

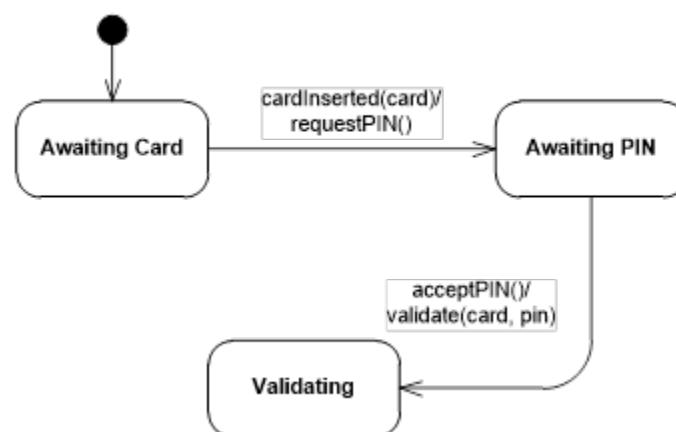
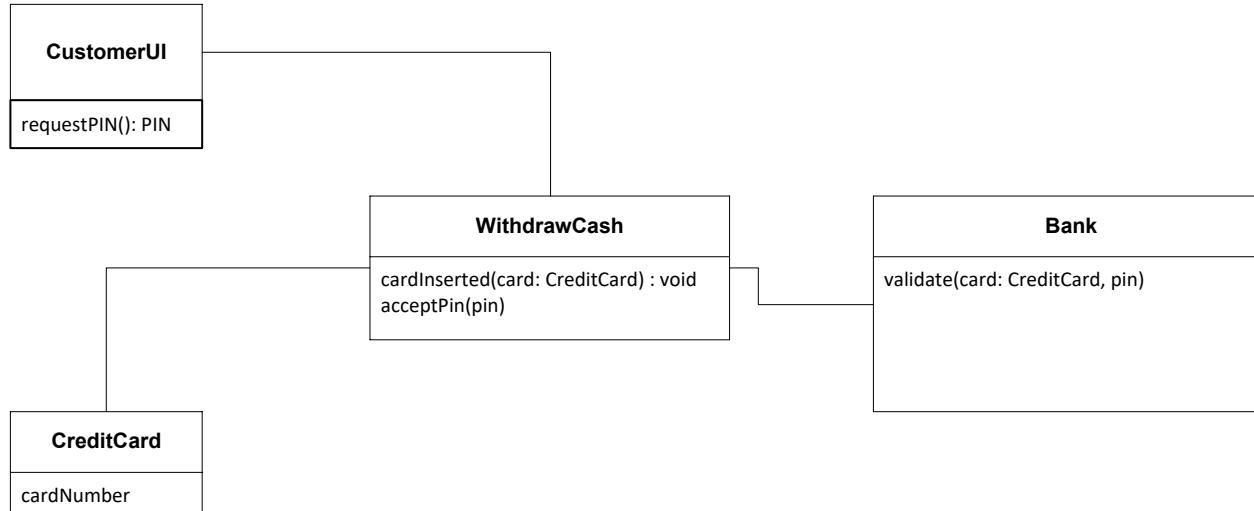
Start øvelsen

- Starten på Sekvensdiagrammet



Start øvelsen

- Starten på klassediagrammet og tilstandsdiagrammet



Applikationsmodeller

Del 2

Find de **rigtige** *boundary* klasser

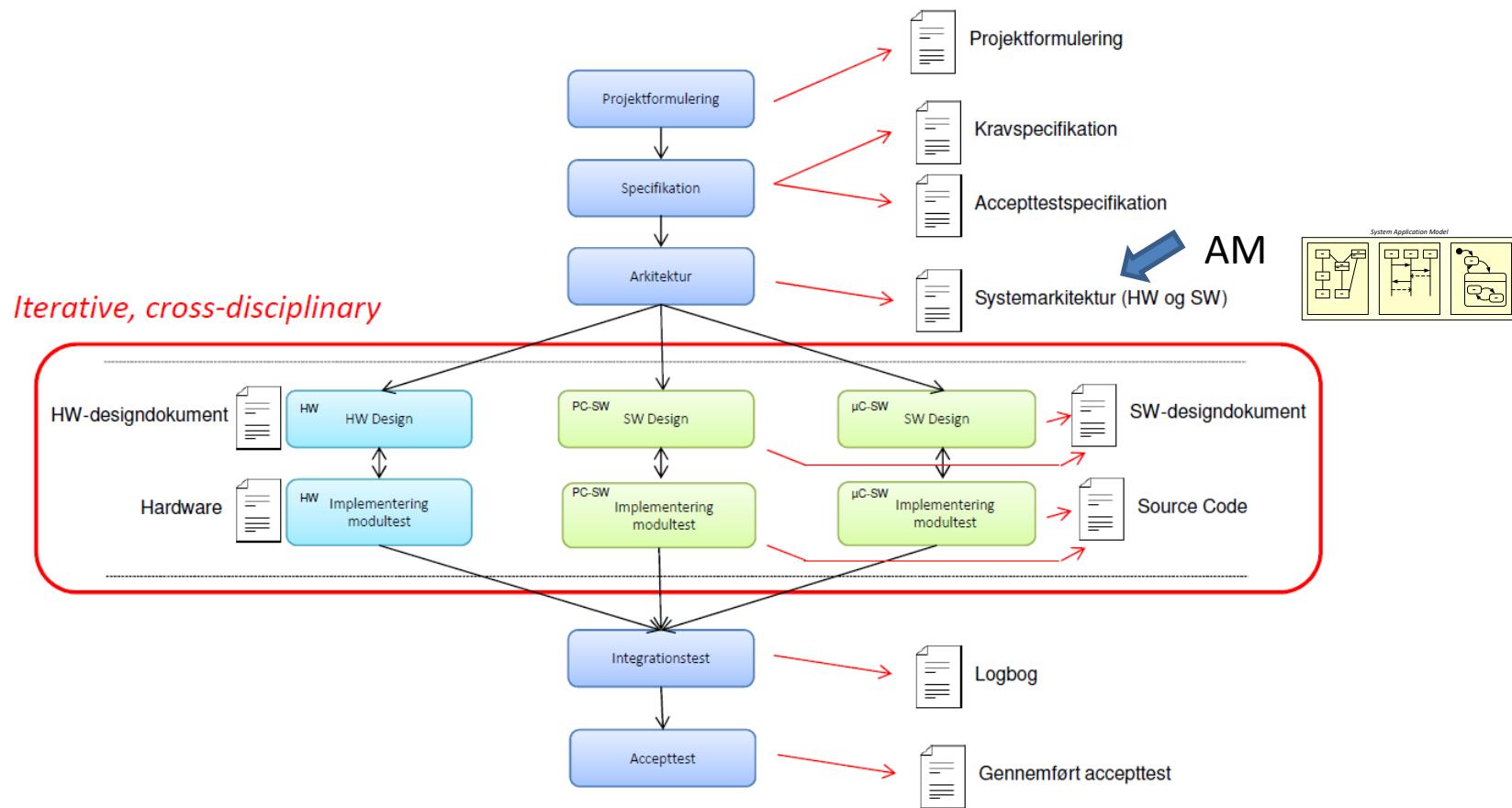
I2ISE

Dagens program

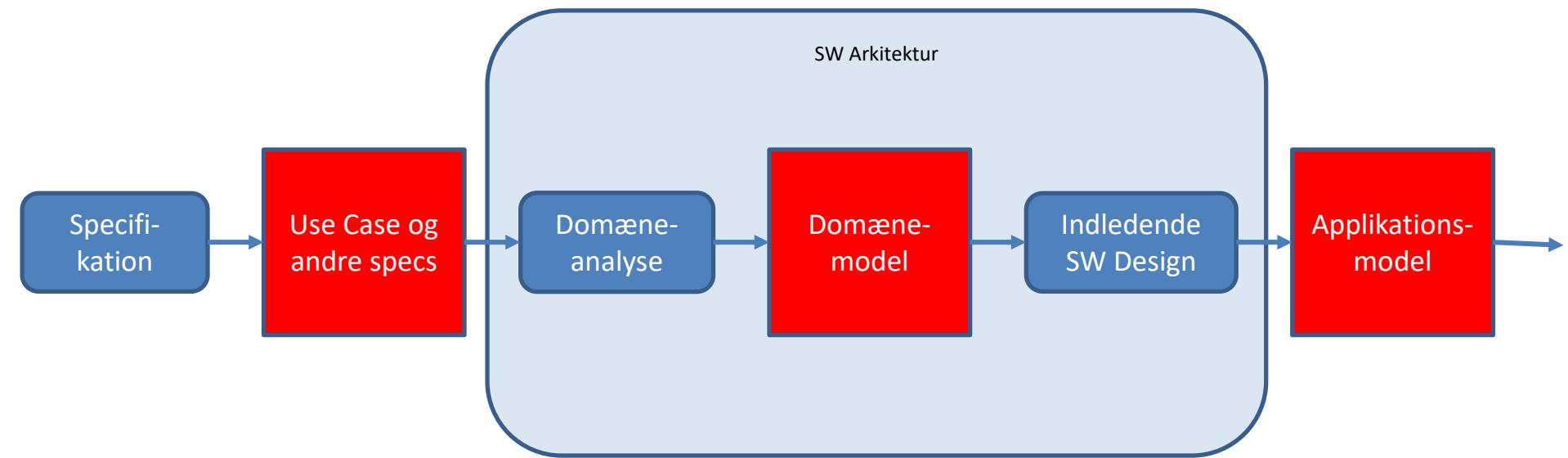
- Resumé
- Find nogle gode boundary klasser
- Regler og guide lines for applikationsmodeller
- Øvelse

AM's plads i dokumenterne

The ASE Process



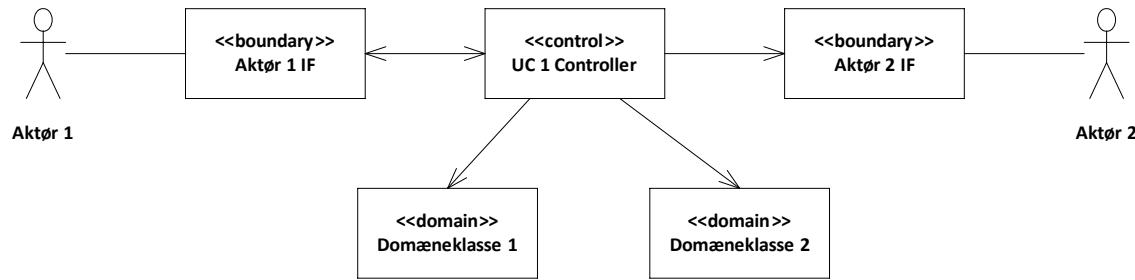
SW Arkitektur – fra UC til Design



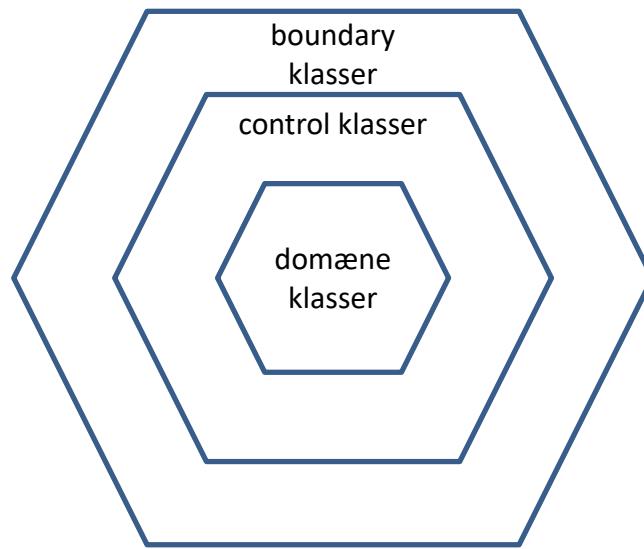
Vores arkitektur

- ... hedder *Entity-Control-Boundary pattern*
- Dette er et velkendt og velforprøvet *Architectural Pattern*
- Et *pattern* er et mønster, som man kan genkende i mange godt designede og godt fungerende applikationer
- Vi vil her kalde *Entity* klasser for *Domain* klasser

Domain – Control - Boundary



aktører



Applikationsmodellen

- Applikationsmodellen er sammensat af følgende 3 typer af diagrammer:
 - *Klassediagram* (cd) for strukturen (statisk)
 - *Sekvensdiagrammer* (SEQ) og
 - *Tilstandsdiagrammer* (STM) for aktiviteter (dynamisk)
- Der laves et tilstrækkeligt antal **sæt** af disse til at beskrive **alle UCs**
- UC bruges til at konstruere dem

Nogle hvad for nogle klasser?

- Applikationsmodellen består af 3 forskellige klassetyper:
Boundary, *domain*, og *control* klasser
- *Boundary* klasser repræsenterer UC *aktører*
 - De er aktørernes interface **til systemet** (UI, protokol, ...)
 - De gør systemet **synligt for aktørerne**
 - Indeholder ingen "business logic" – dvs. ingen styring af UC
 - Mindst 1 per aktør, deles mellem de UCs der har samme aktører
 - Bør forsynes med stereotypen «boundary»
- *Domain* klasser repræsenterer systemets *domæne*
 - Data, domæne-specifik viden, konfigurationer, etc.
 - 0, 1 eller flere, deles mellem de UCs der bruger samme begreber
 - Kan forsynes med stereotypen <>domain<>

Nogle hvad for nogle klasser?

- Applikationsmodellen består af 3 forskellige klassetyper:
Boundary, *domain*, og *control* klasser
- *Control klassen* indeholder UC'ens *business logic*
 - Den styrer ("executes") UC'en ved at interagere med *boundary* og *domain* klasserne
 - Den skal have navn efter UC'en
 - Typisk er der 1 per UC eller 1 som deles mellem nogle få UCs
 - Bør forsynes med stereotypen «control» or «controller»

Boundary klasser og hardware interfaces

- **Boundary klasserne** er den del af softwaren, der tager sig af interfaces til **Actors**
- Softwaren kører på computeren/microcontrolleren
- Så må interfaces til **Actors** være interfaces på computeren/microcontrolleren!
- Derfor: Led efter hardware interfaces der er involveret med **Actors**!
- For hver af dem: Lav en **boundary klasse**!

Applikationsmodellen – Step 1

Version 2!

- Applikationsmodellen opbygges skridt for skridt, hvor hvert skridt styres af én UC

Step 1.1: Vælg den næste fully-dressed UC til at designe for (**hvordan?**)

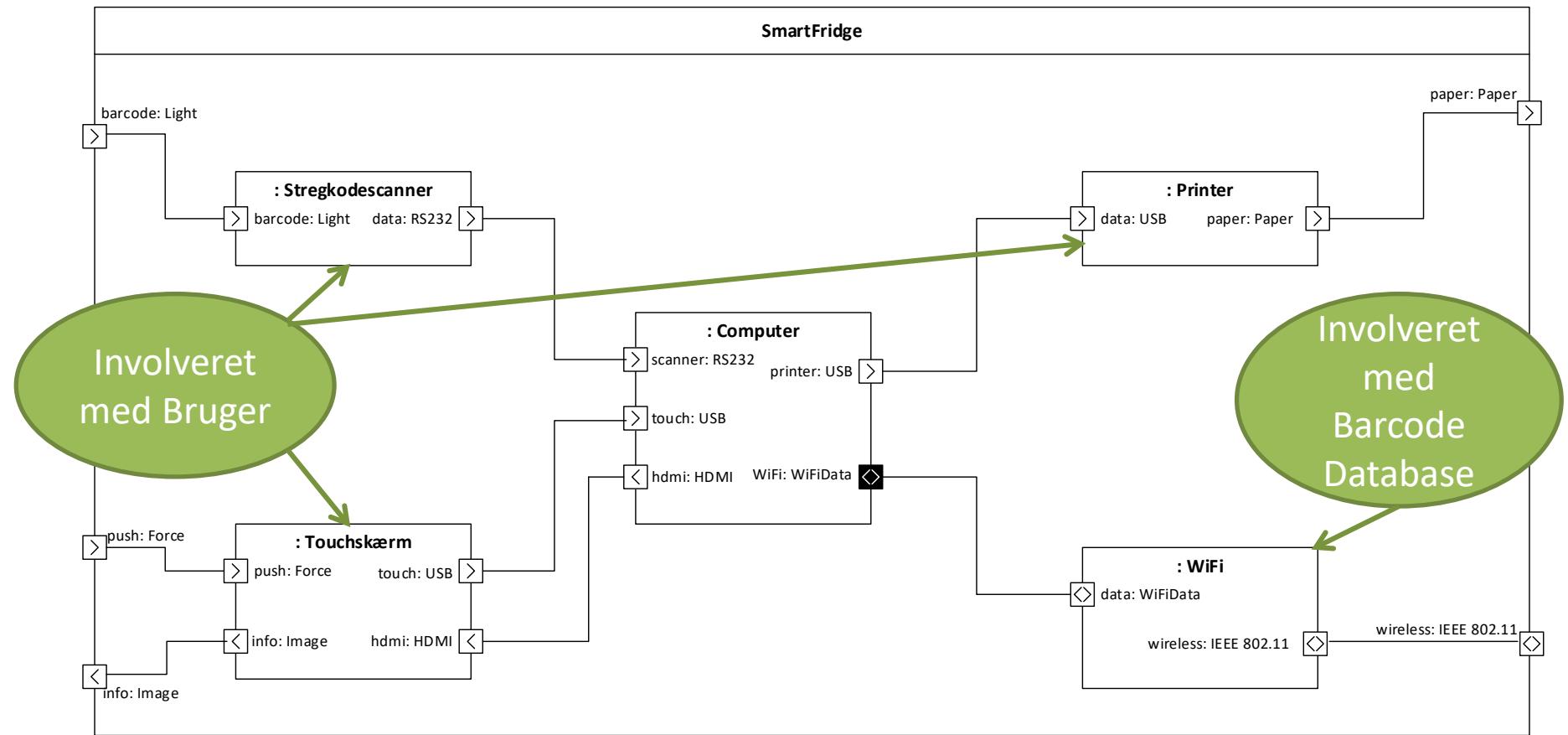
Step 1.2a: Identifier alle involverede **aktører** i UC →
Boundary klasser

Step 1.2b: **Hvis man har et IBD som definerer de faktiske hardware interfaces: opsplit *Boundary klasserne* i relevante *hardware interface Boundary klasser*!**

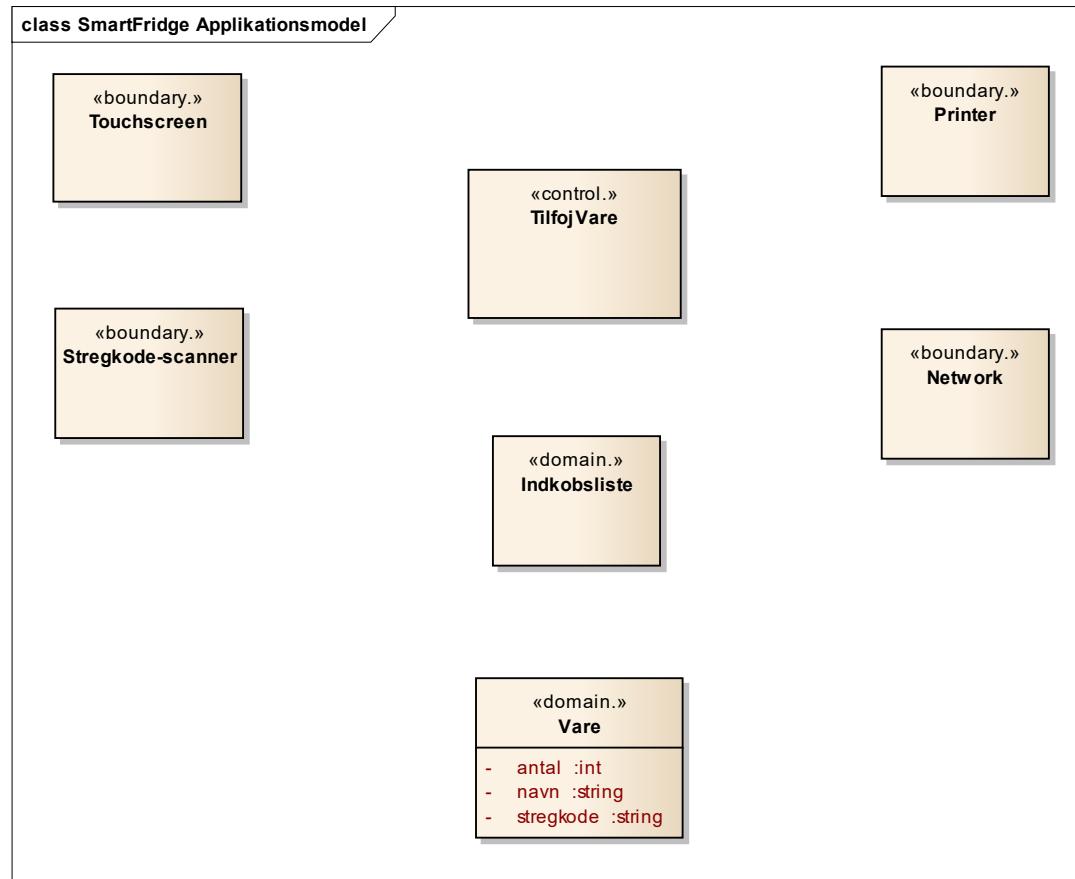
Step 1.3: Identifier **relevante klasser** i Domænemodellen som er involveret i UC → **Domain klasser**

Step 1.4: Tilføj én UC *control* → **Control klasse**

Find Boundaryklasser



1. Version SAM klassediagram



Applikationsmodellen er en Softwaremodel!

- Derfor skal vi finde metoderne på klasserne
- Alle messages mellem objekterne på sekvensdiagrammet er metodekald! Derfor har de "()"
- Udtænk
 - et godt navn
 - parametre og parametertyper
 - returværdi (for synkrone kald som ikke er void)
 - synkron/asynkron

Interrupts vs. polling

Applikationsmodellen – Step 2

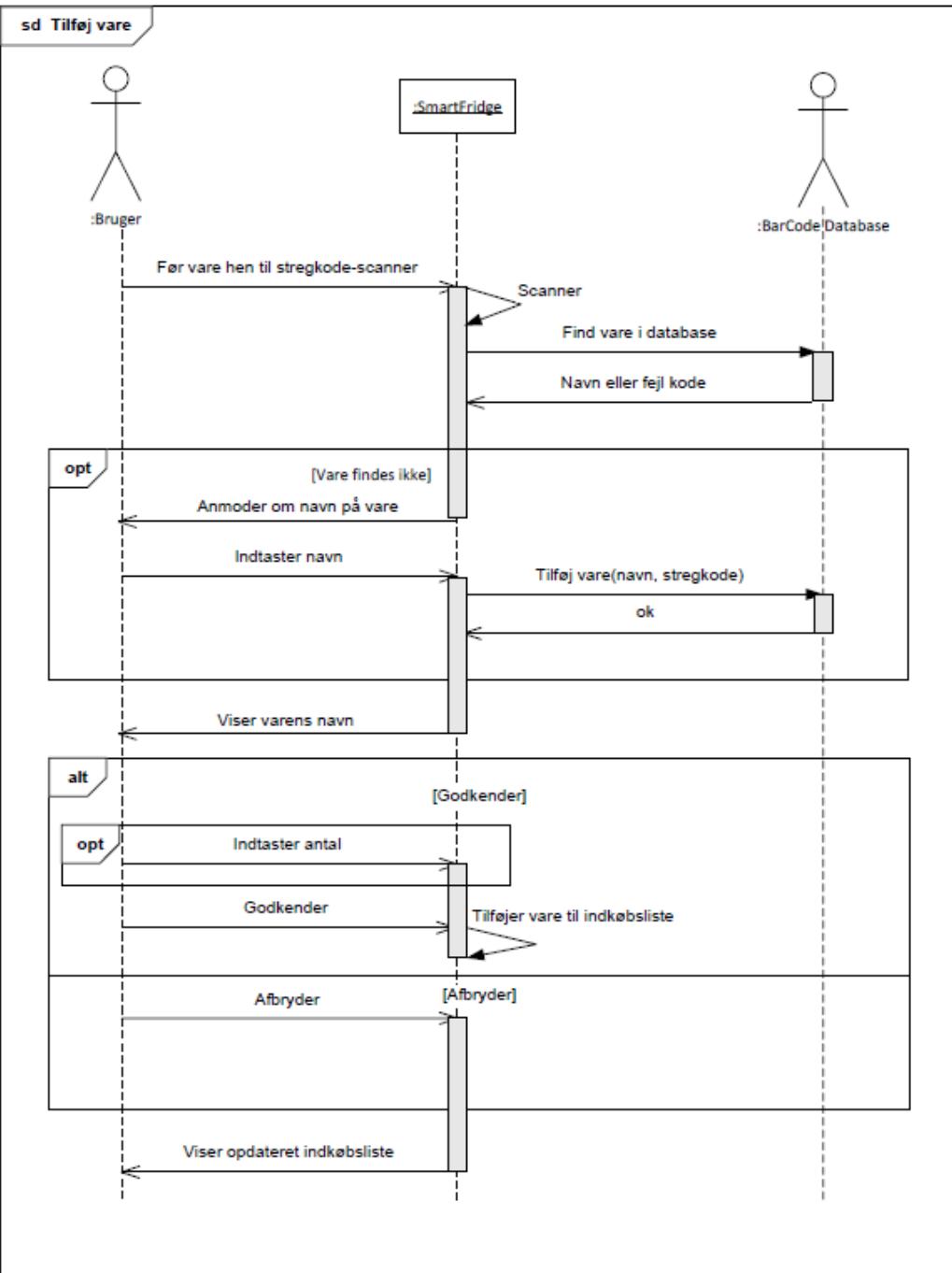
- Samarbejdet mellem klasserne udledes nu fra UC

- Step 2.1: Gennemgå UC's hovedscenarie skridt-for-skridt og udtænk hvordan klasserne kan samarbejde for at udføre skridtet
Hvis man er heldig, har man et System SEQ, der viser UC – brug det!
- Step 2.2: Opdater sekvens- og klassediagrammet for at beskrive samarbejdet (metoder, associationer, attributter)
- Step 2.3: Hold øje med, om der er state-baserede aktiviteter og opdater STMs for disse klasser (tilstande, triggere, overgange, aktioner)
(Step 2.3 springes over hvis der ikke er nogen tilstandsbaserede klasser)
- Step 2.4: Verificer at diagrammerne passer med UC (postconditions, test)
- Step 2.5: Gentag 2.1 – 2.4 for alle UC extentions. Finpuds modellen.

- Alle 3 diagrammer (cd, SEQ, STM) opdateres *parallel/samtidigt* under dette arbejde

Hovedscenarie

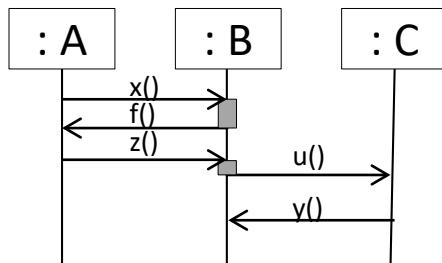
Hovedscenarie	<ol style="list-style-type: none">1. Bruger fører en vare hen til systemets stregkode-scanner2. Systemet scanner varens stregkode3. Systemet sender varens stregkode til BCDB4. BCDB returnerer varens navn til Systemet <i>[Extension 1: Stregkoden findes ikke i BCDB]</i>5. Systemet viser varens navn6. Bruger redigerer og godkender antallet af varer <i>[Extension 2: Bruger afbryder tilføjelsen af en vare]</i><ol style="list-style-type: none">1. Systemet tilføjer varens navn og antal til indkøbslisten2. Systemet viser en opdateret indkøbsliste
----------------------	--



Principperne for step 2.1-4: Gå igennem hovedscenariet for UC og opdater løbende

Sekvensdiagram:

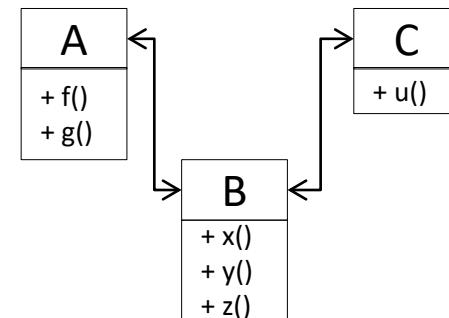
- Tilføj kald af objekternes metoder



Opdateres parallelt!

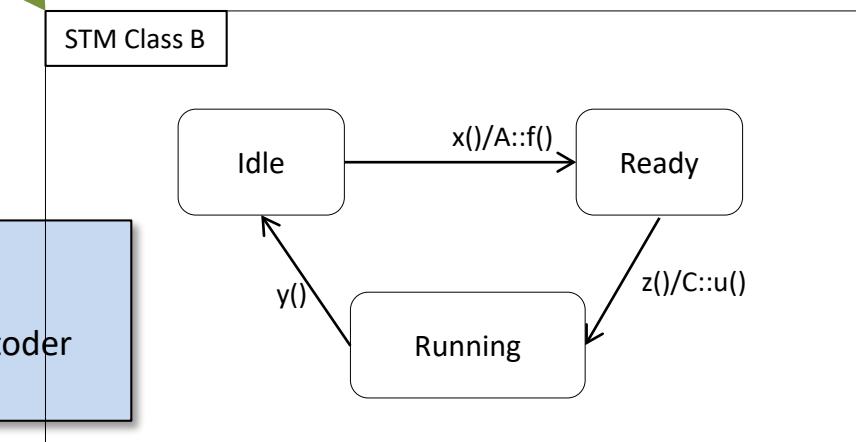
Klassediagram:

- Tilføj metoder til klasserne
- Opdater associationer



Tilstands/aktionsdiagram:

- Lav det for de klasser der har tilstande
- Triggerne til transitionerne er kald til klassens metoder
- Aktionerne er kald til andre klassers metoder



Kommunikationsregler

- Det er *control* klassen der tager alle logiske beslutninger – derfor gælder:
 - *Boundary* klasser kalder **KUN** til *control* klassen/r!
 - (og evt. nødvendige *domain* klasser der bruges som parametre (*Data Transfer Objects - DTO*))!
 - *Boundary* klasser kalder **IKKE direkte** til andre *boundary* klasser!
 - (medmindre man har en lagdelt *boundary* struktur)!
 - *Domain* klasser kalder **IKKE** til *boundary* klasser!
 - *Domain* klasser starter **IKKE** noget på eget initiativ!

Kommunikationsregler

Principielle kommunikationsregler i arkitekturen

	Til Boundary	Til Domain	Til Control
Fra Boundary	Nej!	Nej!	Ja!
Fra Domain	Nej!	Nej!	Nej!
Fra Control	Ja!	Ja!	Ja!

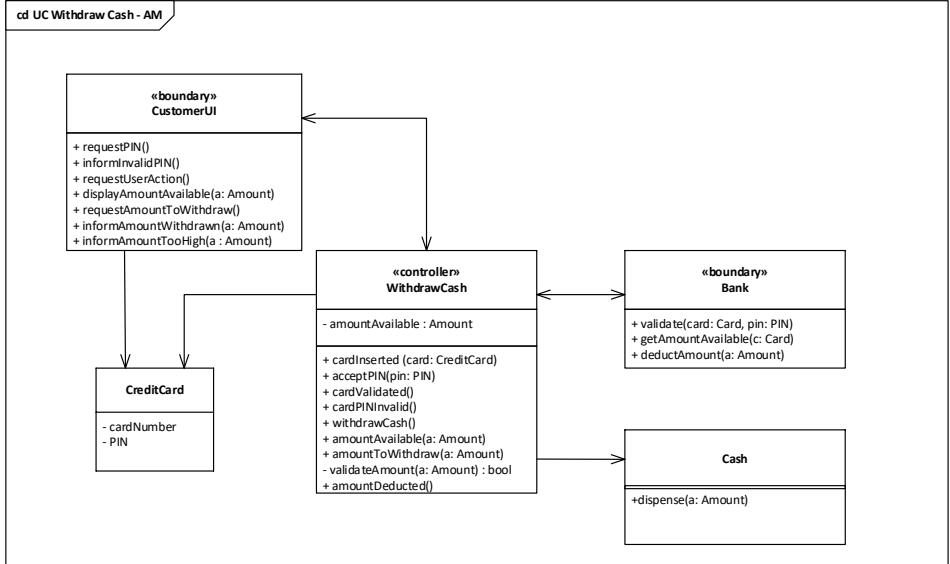
Pragmatiske kommunikationsregler i arkitekturen

	Til Boundary	Til Domain	Til Control
Fra Boundary	(Lagdelt I->I og O->O)	(Data Transfer Object)	Ja!
Fra Domain	Nej!	(Komposition)	Nej!
Fra Control	Ja!	Ja!	Ja!

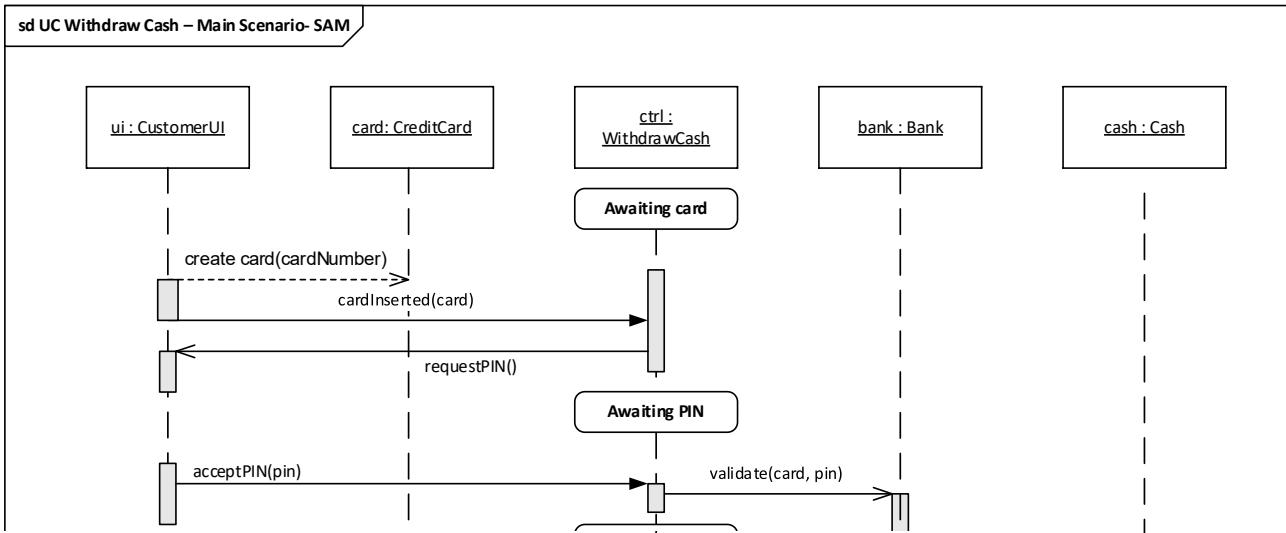
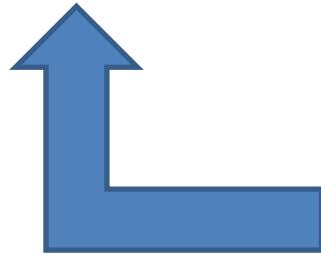
Designregler

- Alle klasser på sekvensdiagrammet, skal også være på klassediagrammet (men ikke nødvendigvis omvendt)
- Alle metoder skal tilknyttes den klasse på klassediagrammet, der står for enden af pilen på sekvensdiagrammet
 - Det er et metodekald fra den ene klasse til den anden
- Alle associationer skal pege samme vej på klassediagrammet, som de gør på sekvensdiagrammet
 - For at den ene klasse kan kalde den anden, skal den have en association til den
- Associationer kan være tovejs, hvis begge klasser kalder den anden i løbet af UC

Designregler

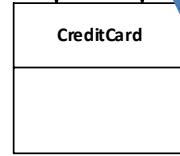
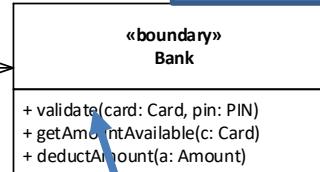
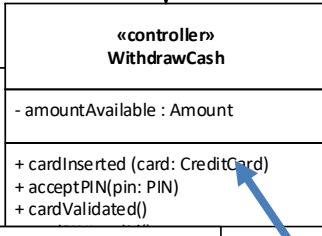
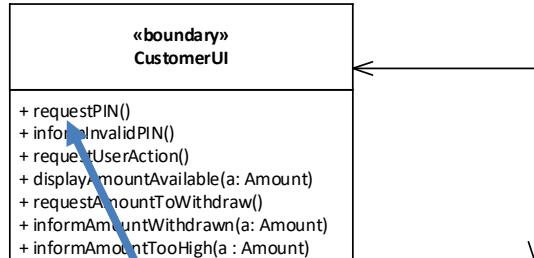


Alle klasser på sekvensdiagrammet
skal også være på
klassediagrammet!



Designregler

cd UC Withdraw Cash - AM



sd UC Withdraw Cash – Main Scenario- SAM

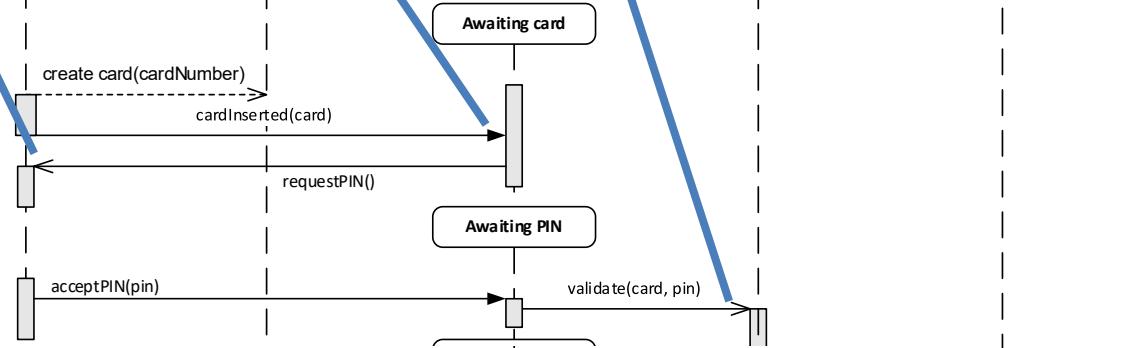
ui : CustomerUI

card: CreditCard

ctrl : WithdrawCash

bank : Bank

cash : Cash



Alle metoder skal tilknyttes den klasse på klassediagrammet, der står for enden af pilen på sekvensdiagrammet

Designregler

cd UC Withdraw Cash - AM



```
+ requestPIN()  
+ informInvalidPIN()  
+ requestUserAction()  
+ displayAmountAvailable(a: Amount)  
+ requestAmountToWithdraw()  
+ informAmountWithdrawn(a : Amount)  
+ informAmountTooHigh(a : Amount)
```

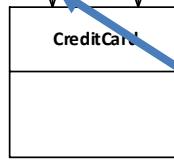


```
amountAvailable : Amount  
  
+ cardInserted(card: CreditCard)  
+ acceptPIN(pin: PIN)  
+ cardValidated()
```

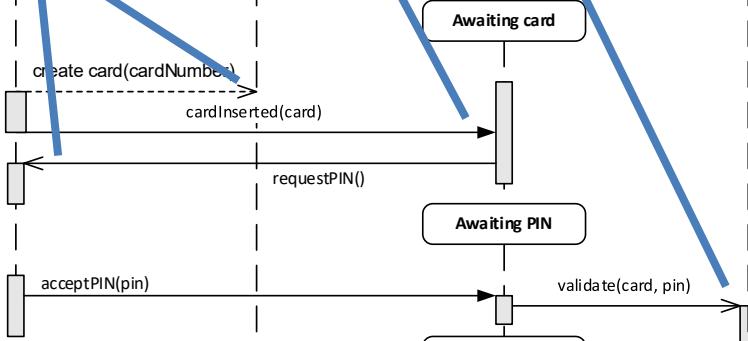
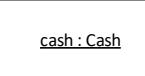
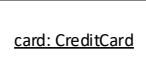
Alle associationer skal pege samme
vej på klassediagrammet, som de
gør på sekvensdiagrammet



```
+ validate(card: Card, pin: PIN)  
+ getAmountAvailable(c: Card)  
- deductAmount(a: Amount)
```



sd UC Withdraw Cash – Main Scenario- SAM



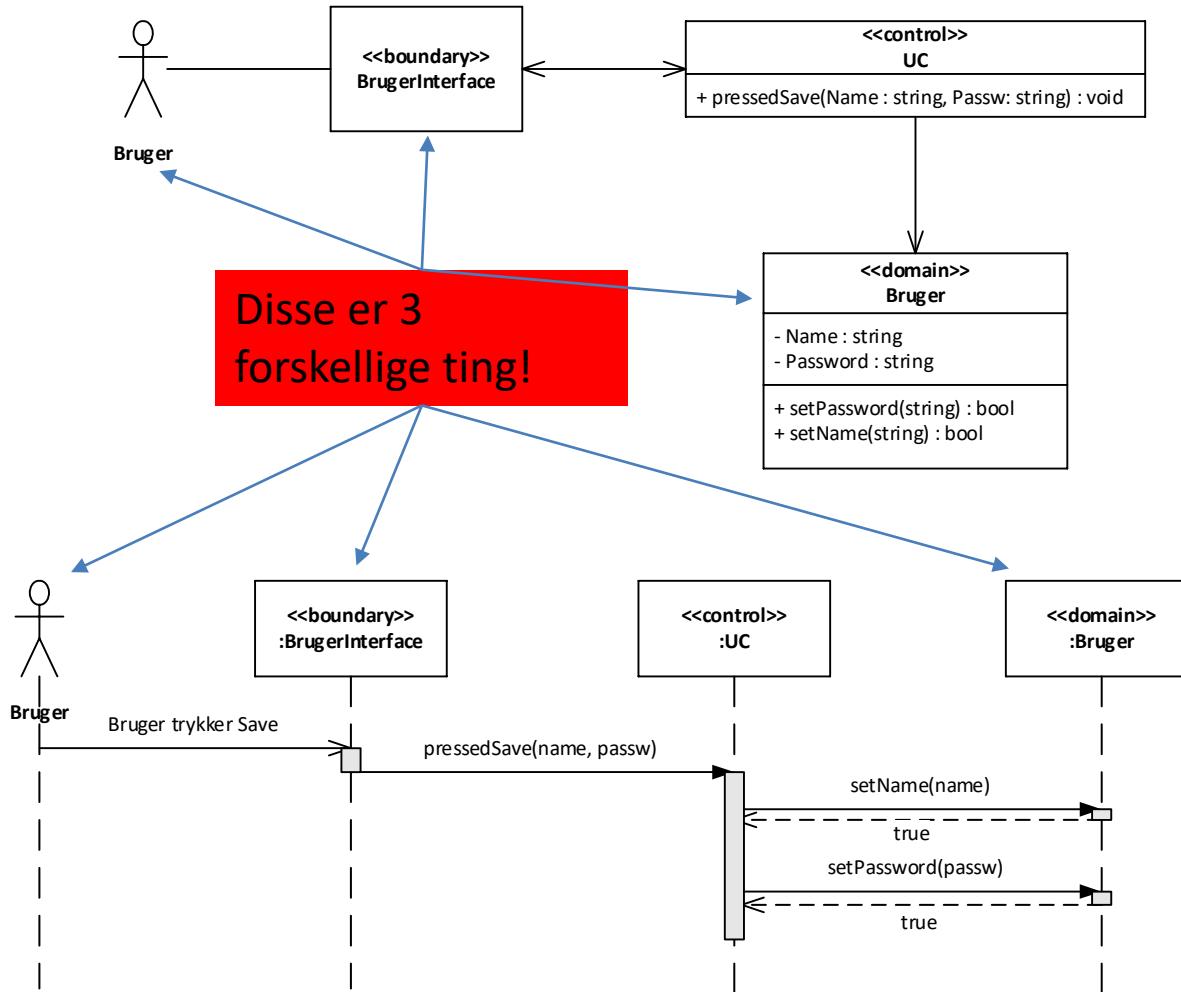
Guidelines "control"

- ***Control klassen*** er IKKE det samme som **hardware controlleren** eller **μ-controlleren** eller **"control unit"** på BDD/IBD!
- ***Control klassen*** er en del af **softwaren** som kører på **CPUen** i disse hardware controllers!
- ***Control klassen*** har navn efter den UC, den udfører!

Guideline Actor

- Man må gerne bruge en UC **Actor** (tændstiksmand) på Applikationsmodellens sekvensdiagram – **MEN:**
 - Den faktiske **Actor** og **boundary klassen/r** for aktøren og den **domain klasse** som bruges til at gemme attributter for aktøren – er 3 forskellige ting!
 - En faktisk **Actor** har **IKKE** metoder og kan **IKKE** kalde **metoder** – det kan kun **boundary klassen/r for** aktøren!
 - Ikke alle tegneværktøjer kan tegne messages uden () på sekvensdiagrammer :-)

Guideline Actor



Opgave:

- Udfør Step 2.1-2.5 for SmartFridge, mens I overholder designregler og andre guidelines

Applikationsmodeller

Part 3

Systemer med subsystemer

I2ISE

Dagens emner

- Resume regler og guide lines
- Applikationsmodeller for systemer med subsystemer

Kommunikationsregler

- Det er *control* klassen der tager alle logiske beslutninger – derfor gælder:
 - *Boundary* klasser kalder **KUN** til *control* klassen/r!
 - (og evt. nødvendige *domain* klasser der bruges som parametre (*Data Transfer Objects - DTO*))!
 - *Boundary* klasser kalder **IKKE direkte** til andre *boundary* klasser!
 - (medmindre man har en lagdelt *boundary* struktur)!
 - *Domain* klasser kalder **IKKE** til *boundary* klasser!
 - *Domain* klasser kalder **IKKE** noget på eget initiativ!

Kommunikationsregler

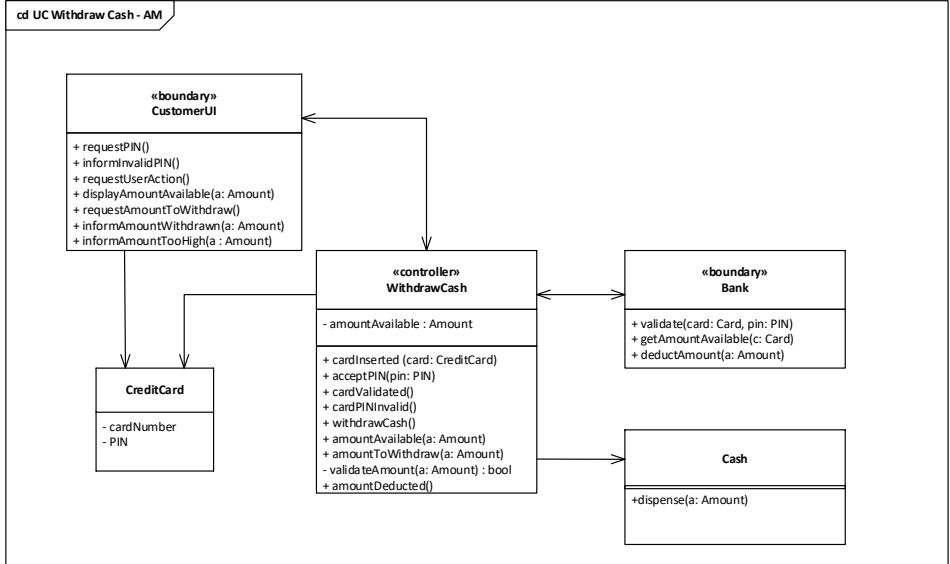
Principielle kommunikationsregler i arkitekturen

	Til Boundary	Til Domain	Til Control
Fra Boundary	Nej!	Nej!	Ja!
Fra Domain	Nej!	Nej!	Nej!
Fra Control	Ja!	Ja!	Ja!

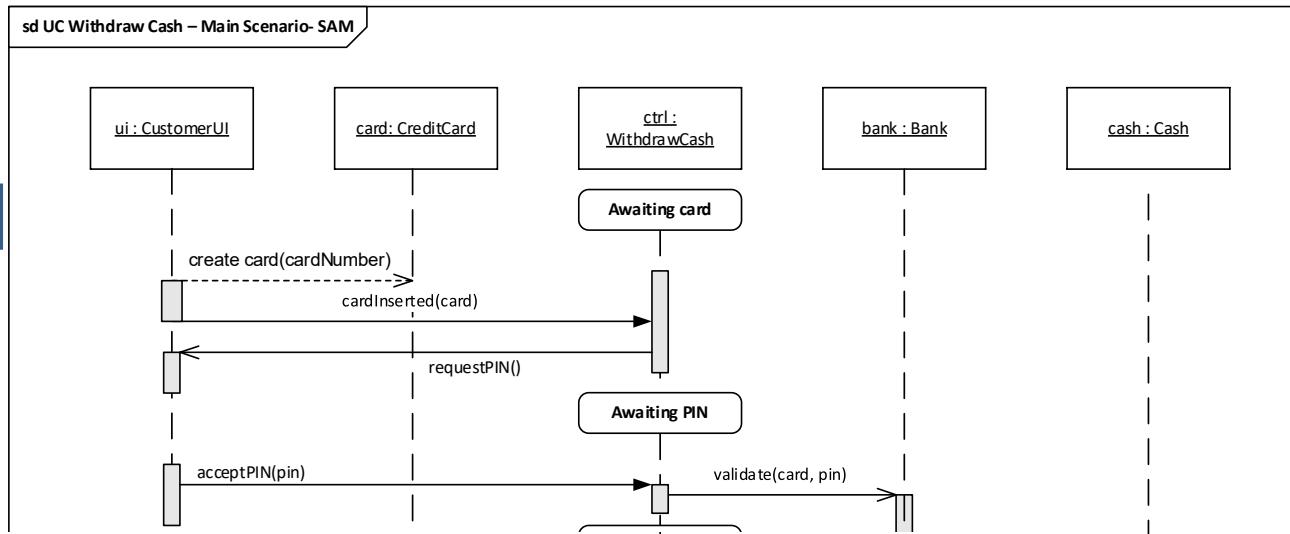
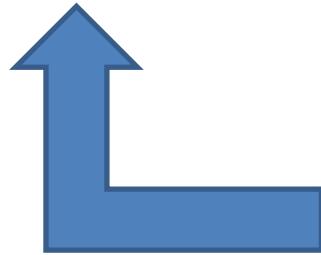
Pragmatiske kommunikationsregler i arkitekturen

	Til Boundary	Til Domain	Til Control
Fra Boundary	(Lagdelt I->I og O->O)	(Data Transfer Object)	Ja!
Fra Domain	Nej!	(Komposition)	Nej!
Fra Control	Ja!	Ja!	Ja!

Designregler

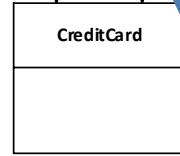
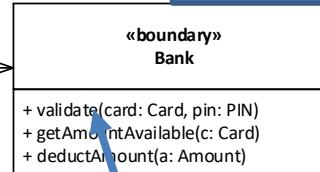
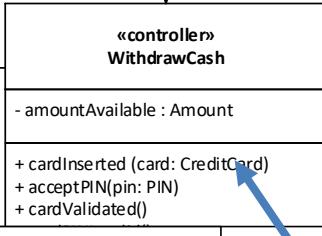
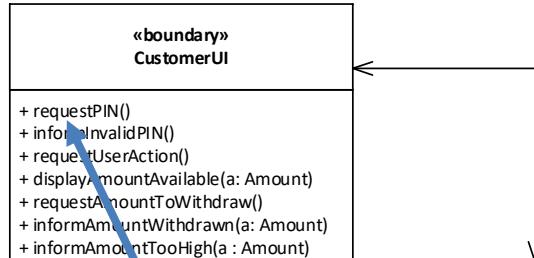


Alle klasser på sekvensdiagrammet
skal også være på
klassediagrammet!



Designregler

cd UC Withdraw Cash - AM



sd UC Withdraw Cash – Main Scenario- SAM

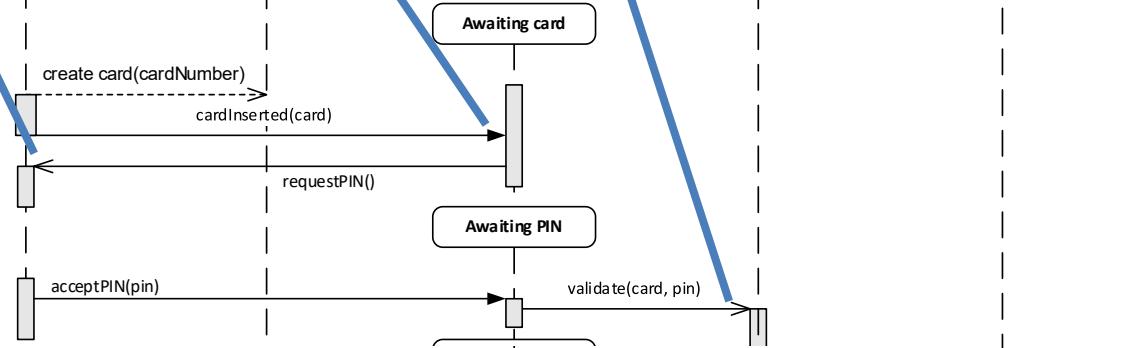
ui : CustomerUI

card: CreditCard

ctrl : WithdrawCash

bank : Bank

cash : Cash



Alle metoder skal tilknyttes den klasse på klassediagrammet, der står for enden af pilen på sekvensdiagrammet

Designregler

cd UC Withdraw Cash - AM



```
+ requestPIN()  
+ informInvalidPIN()  
+ requestUserAction()  
+ displayAmountAvailable(a: Amount)  
+ requestAmountToWithdraw()  
+ informAmountWithdrawn(a : Amount)  
+ informAmountTooHigh(a : Amount)
```

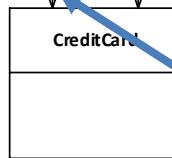


```
amountAvailable : Amount  
  
+ cardInserted(card: CreditCard)  
+ acceptPIN(pin: PIN)  
+ cardValidated()
```

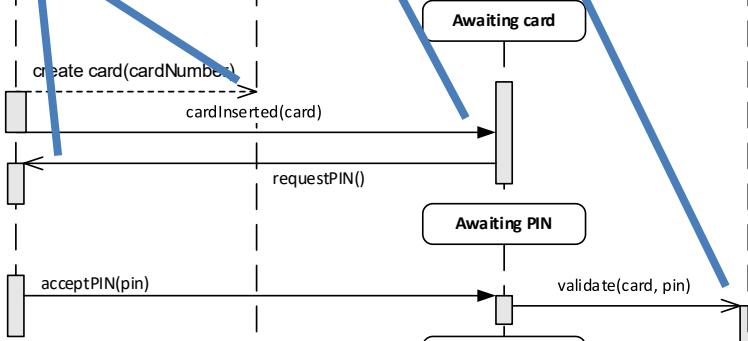
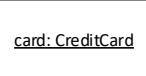
Alle associationer skal pege samme
vej på klassediagrammet, som de
gør på sekvensdiagrammet



```
+ validate(card: Card, pin: PIN)  
+ getAmountAvailable(c: Card)  
- deductAmount(a: Amount)
```



sd UC Withdraw Cash – Main Scenario- SAM



Designregler

- Alle klasser på sekvensdiagrammet, skal også være på klassediagrammet (men ikke nødvendigvis omvendt)
- Alle metoder skal tilknyttes den klasse på klassediagrammet, der står for enden af pilen på sekvensdiagrammet
 - Det er et metodekald fra den ene klasse til den anden
- Alle associationer skal pege samme vej på klassediagrammet, som de gør på sekvensdiagrammet
 - For at den ene klasse kan kalde den anden, skal den have en association til den
- Associationer kan være tovejs, hvis begge klasser kalder den anden i løbet af UC

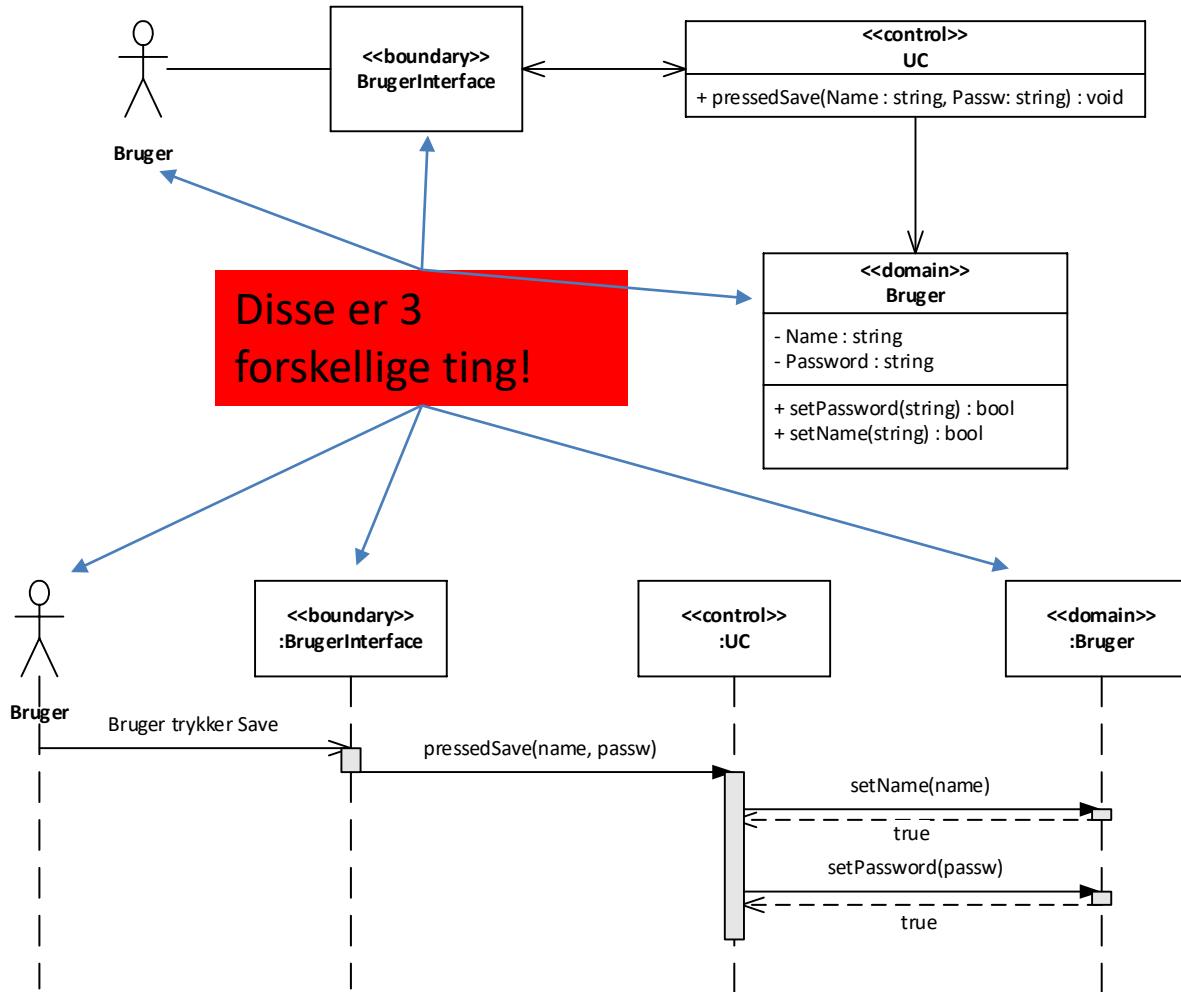
Guidelines "control"

- ***Control klassen*** er IKKE det samme som **hardware controlleren** eller **μ-controlleren** eller **"control unit"** på BDD/IBD!
- ***Control klassen*** er en del af **softwaren** som kører på **CPUen** i disse hardware controllers!
- ***Control klassen*** har navn efter den UC, den udfører!

Guideline Actor

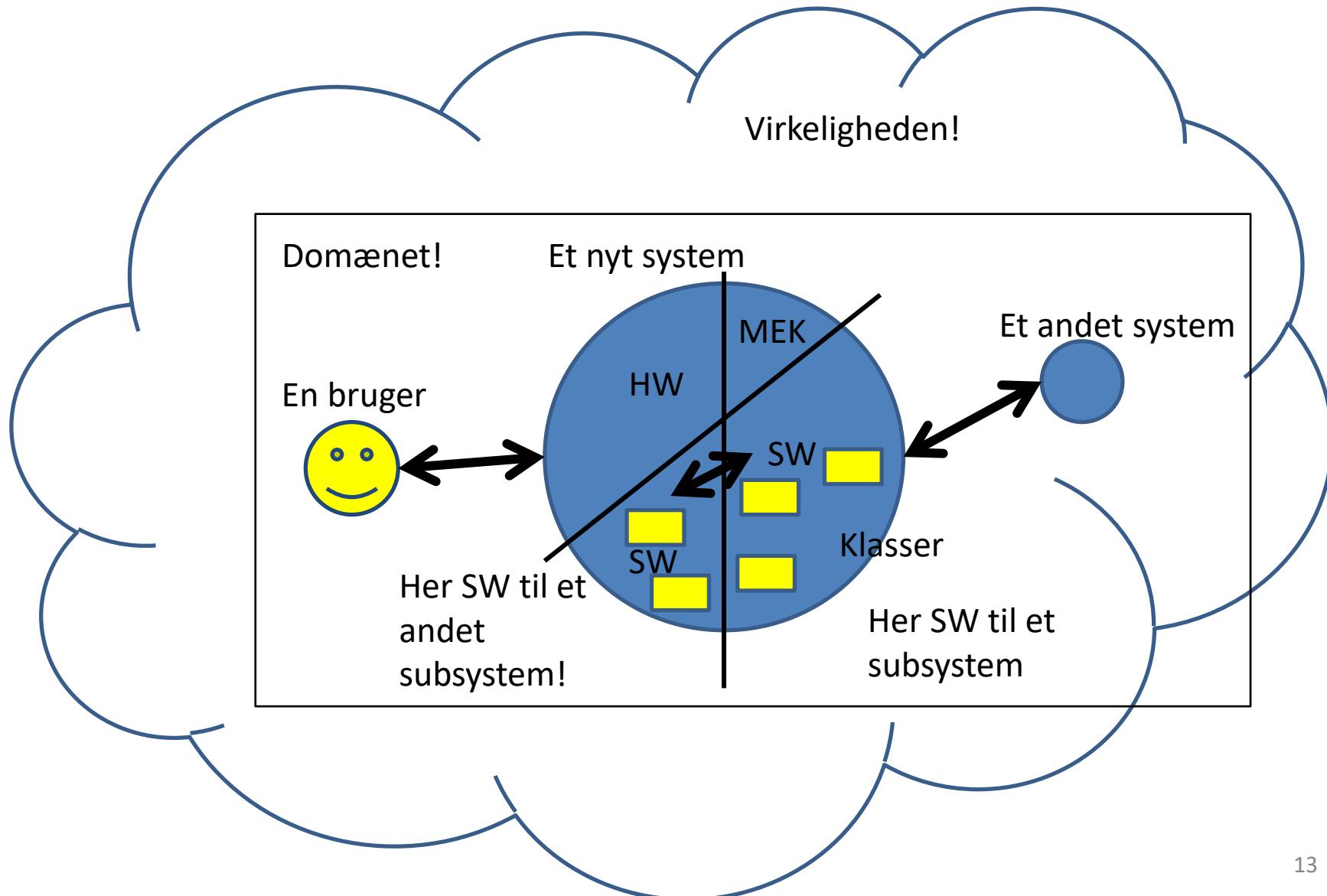
- Man må gerne bruge en UC **Actor** (tændstiksmand) på Applikationsmodellens sekvensdiagram – **MEN:**
 - Den faktiske **Actor** og **boundary klassen/r** for aktøren og den **domain klasse** som bruges til at gemme attributter for aktøren – er 3 forskellige ting!
 - En faktisk **Actor** har **IKKE** metoder og kan **IKKE** kalde **metoder** – det kan kun **boundary klassen/r for** aktøren!
 - Ikke alle tegneværktøjer kan tegne messages uden () på sekvensdiagrammer :-)

Guideline Actor



- Løsning Smartfridge

Virkeligheden og systemet – med subsystemer!



Applikationsmodellen – Step 1

Version 3!

- Applikationsmodellen opbygges skridt for skridt, hvor hvert skridt styres af én UC
- **Én for hvert subsystem!**

Step 1.1: Vælg den næste fully-dressed UC til at designe for

Step 1.2a: Identifier alle involverede **aktører og subsystemer** i UC **for dette subsystem** → **Boundary** klasser

Step 1.2b: Hvis man har et IBD som definerer de faktiske hardware interfaces, **også mellem subsystemerne**: opsplit **Boundary** klasserne i relevante **hardware interface Boundary** klasser!

Step 1.3: Identifier **relevante klasser i Domænemodellen** som er involveret i UC → **Domain** klasser

Step 1.4: Tilføj én UC *control* → **Control** klasse

Applikationsmodellen – Step 2

Version 3

- Samarbejdet mellem klasserne udledes nu fra UC **og fra System sekvensdiagrammer for UC**
- **For hvert subsystem!**

Step 2.1: Gennemgå UC's hovedscenarie skridt-for-skridt **og/eller System sekvensdiagrammet for UC** og udtænk hvordan klasserne kan samarbejde for at udføre skridtet!
Man skal kun se på de skridt/messages, der involverer det pågældende subsystem!

Step 2.2: Opdater sekvens- og klassediagrammet for at beskrive samarbejdet (metoder, associationer, attributter)

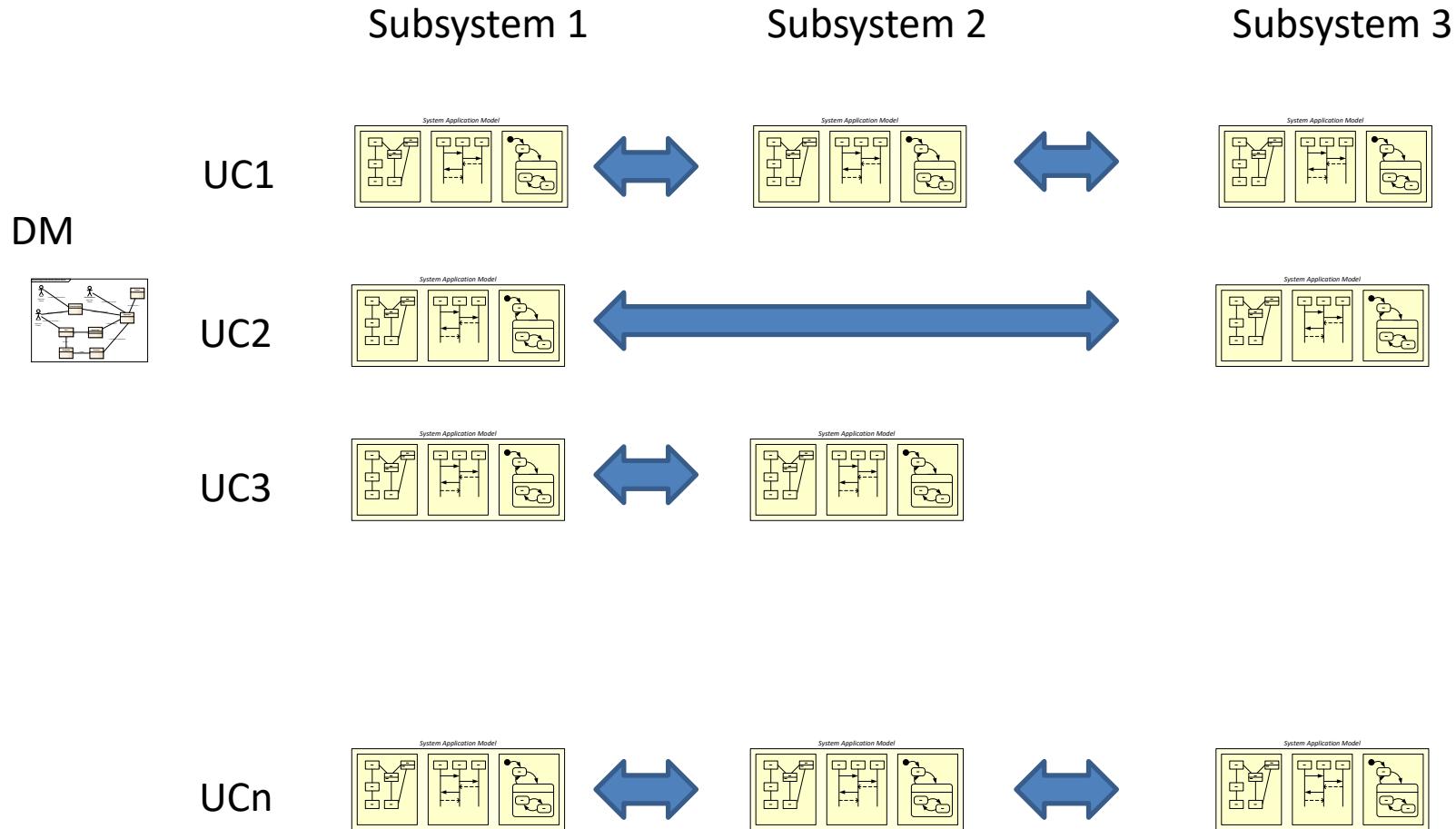
Step 2.3: Hold øje med, om der er state-baserede aktiviteter og opdater STMs for disse klasser (tilstade, triggere, overgange, aktioner)
(Step 2.3 springes over hvis der ikke er nogen tilstandsbaserede klasser)

Step 2.4: Verificer at diagrammerne passer med UC (postconditions, test)

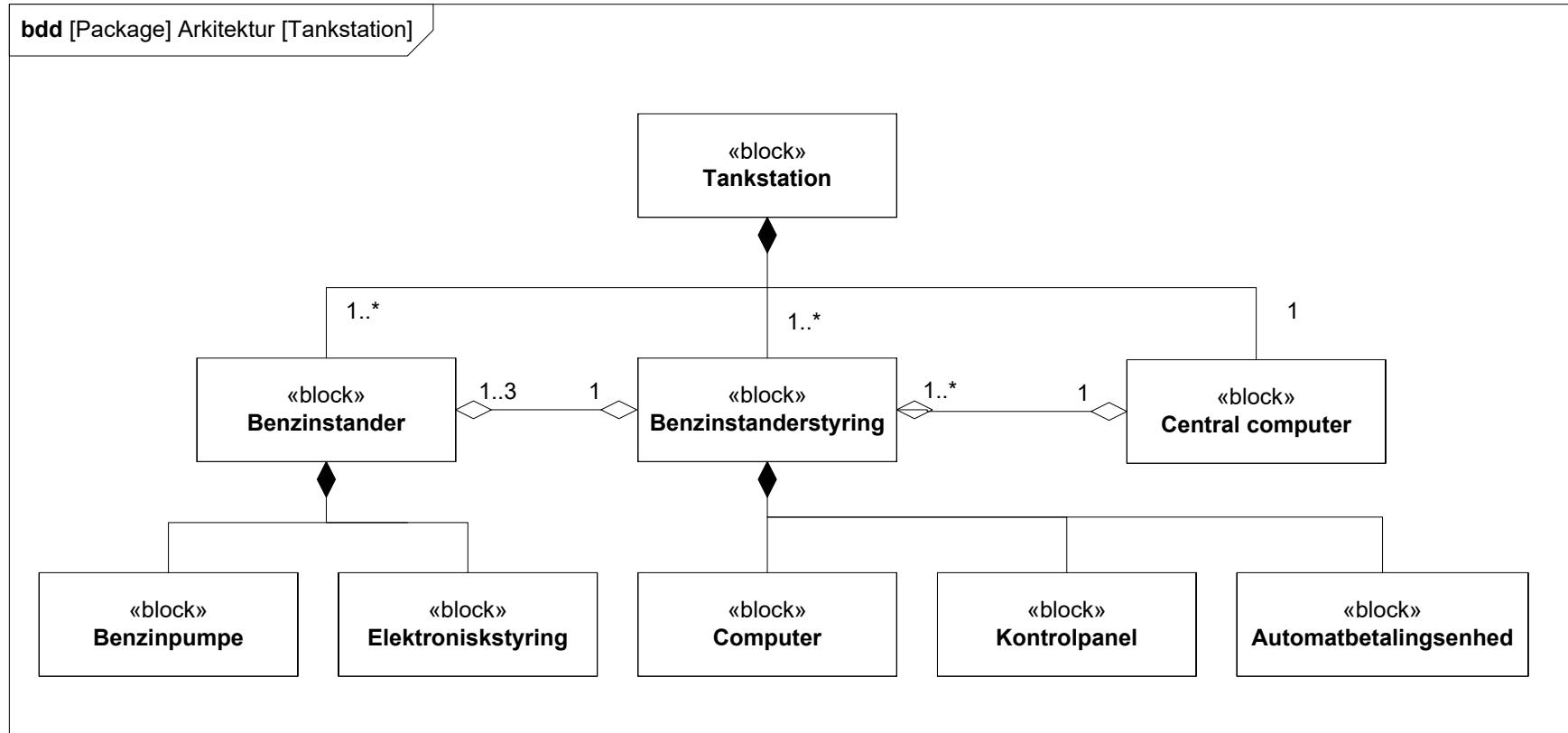
Step 2.5: Gentag 2.1 – 2.4 for alle UC extentions. Finpuds modellen.

- Alle 3 diagrammer (cd, SEQ, STM) opdateres *parallel/samtidigt* **for et subsystem af gangen** under dette arbejde

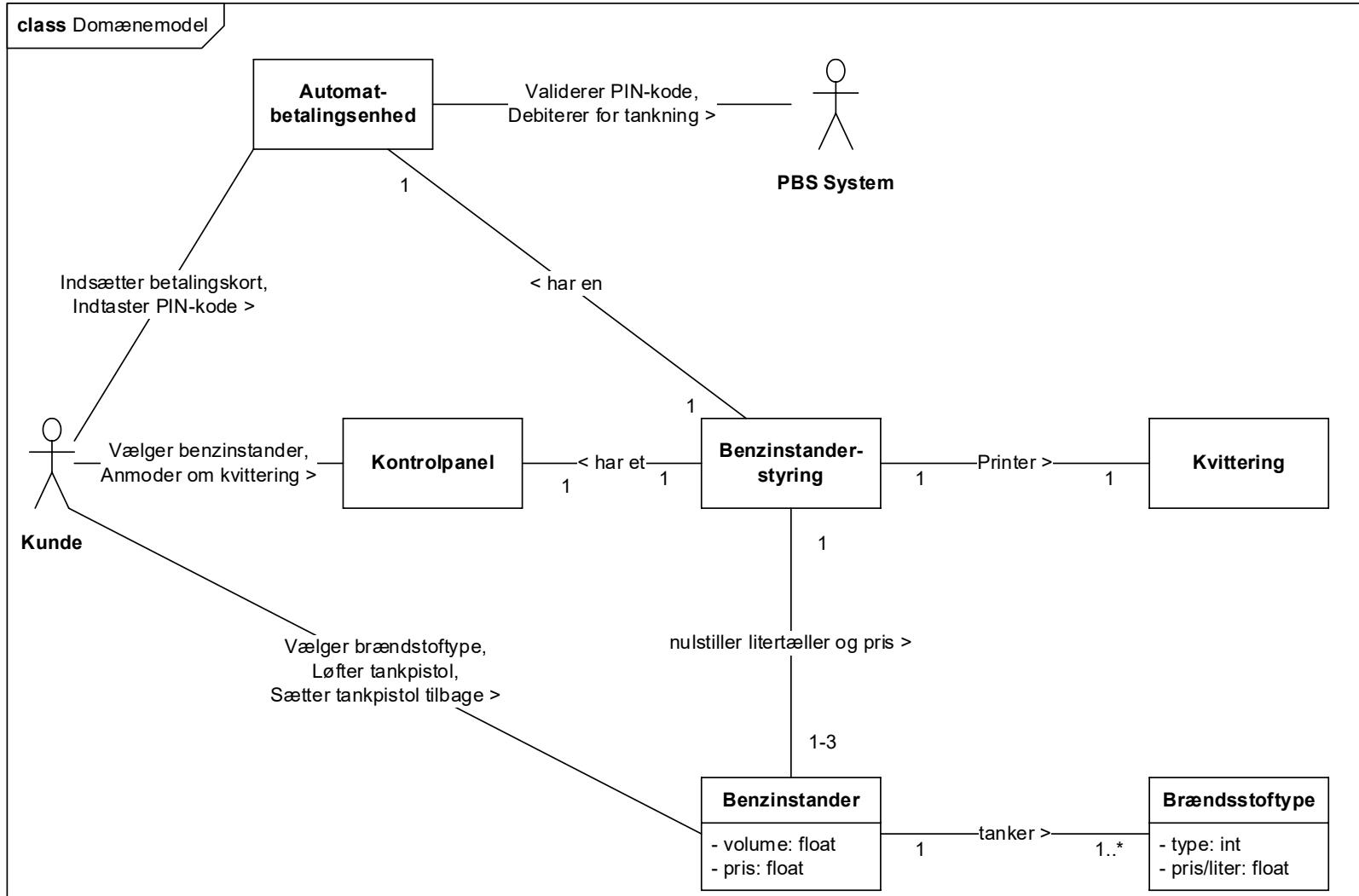
Applikationsmodeller for samarbejdende subsystemer



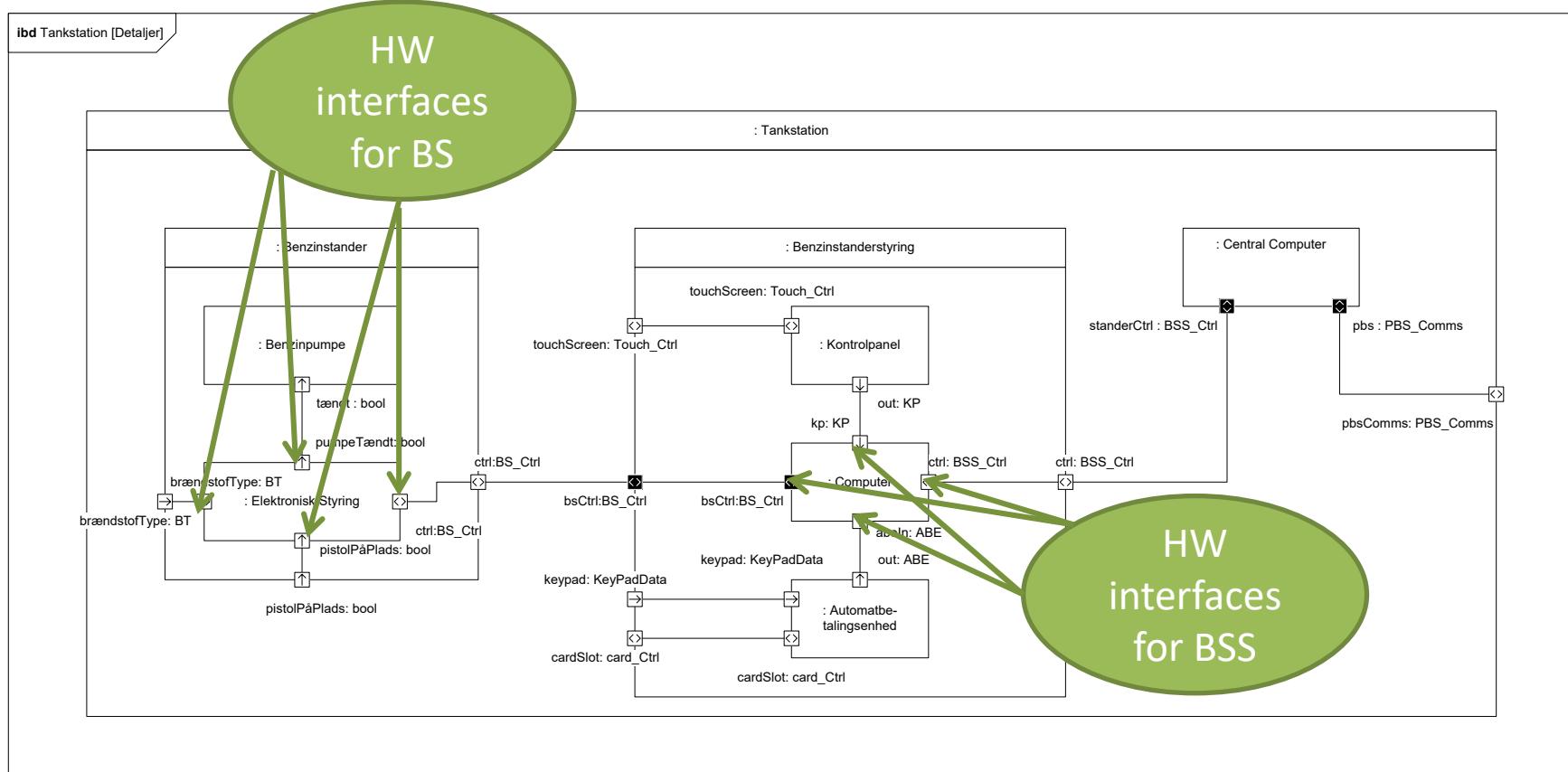
System med subsystemer



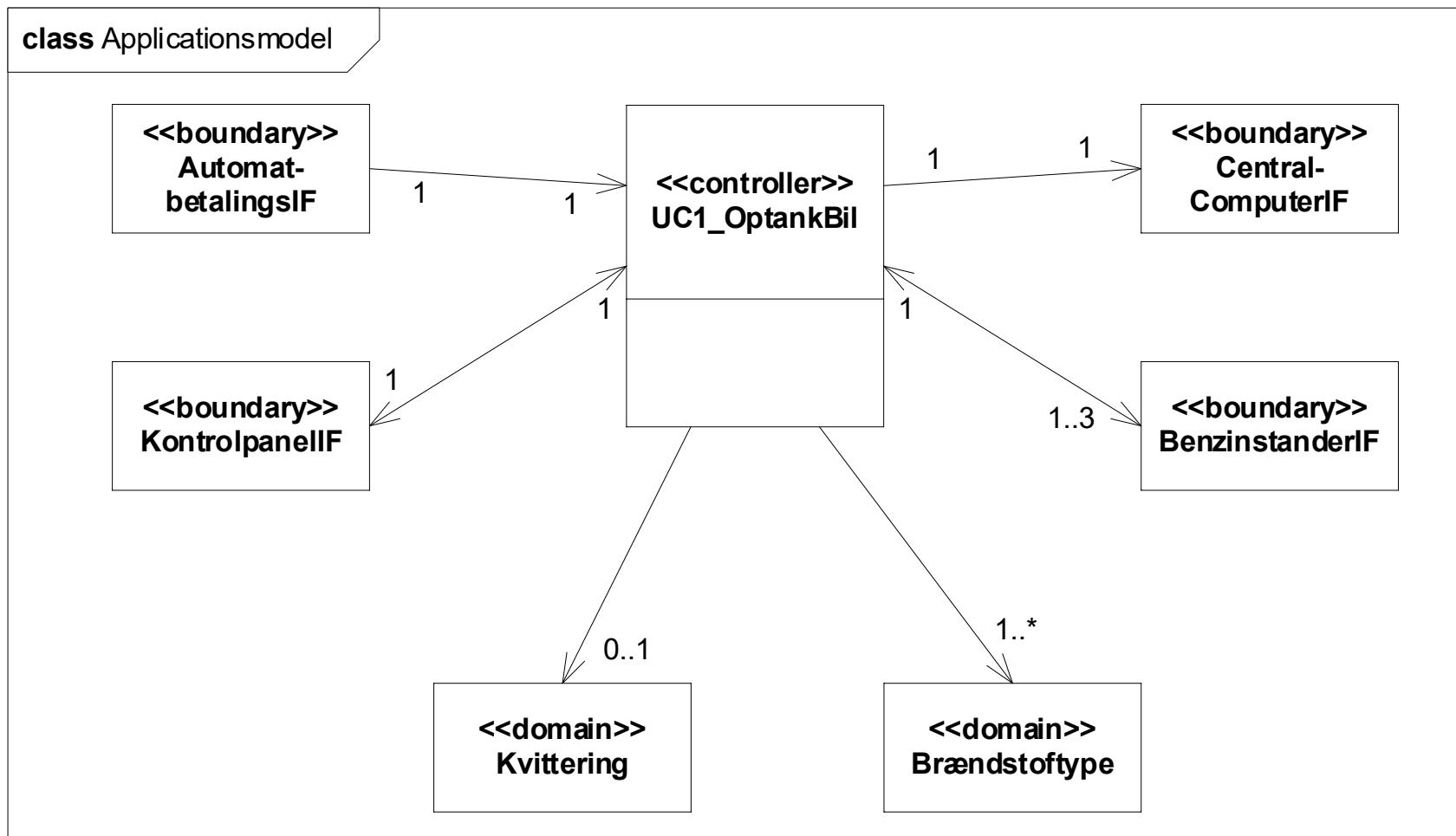
Domænemodellen for hele systemet



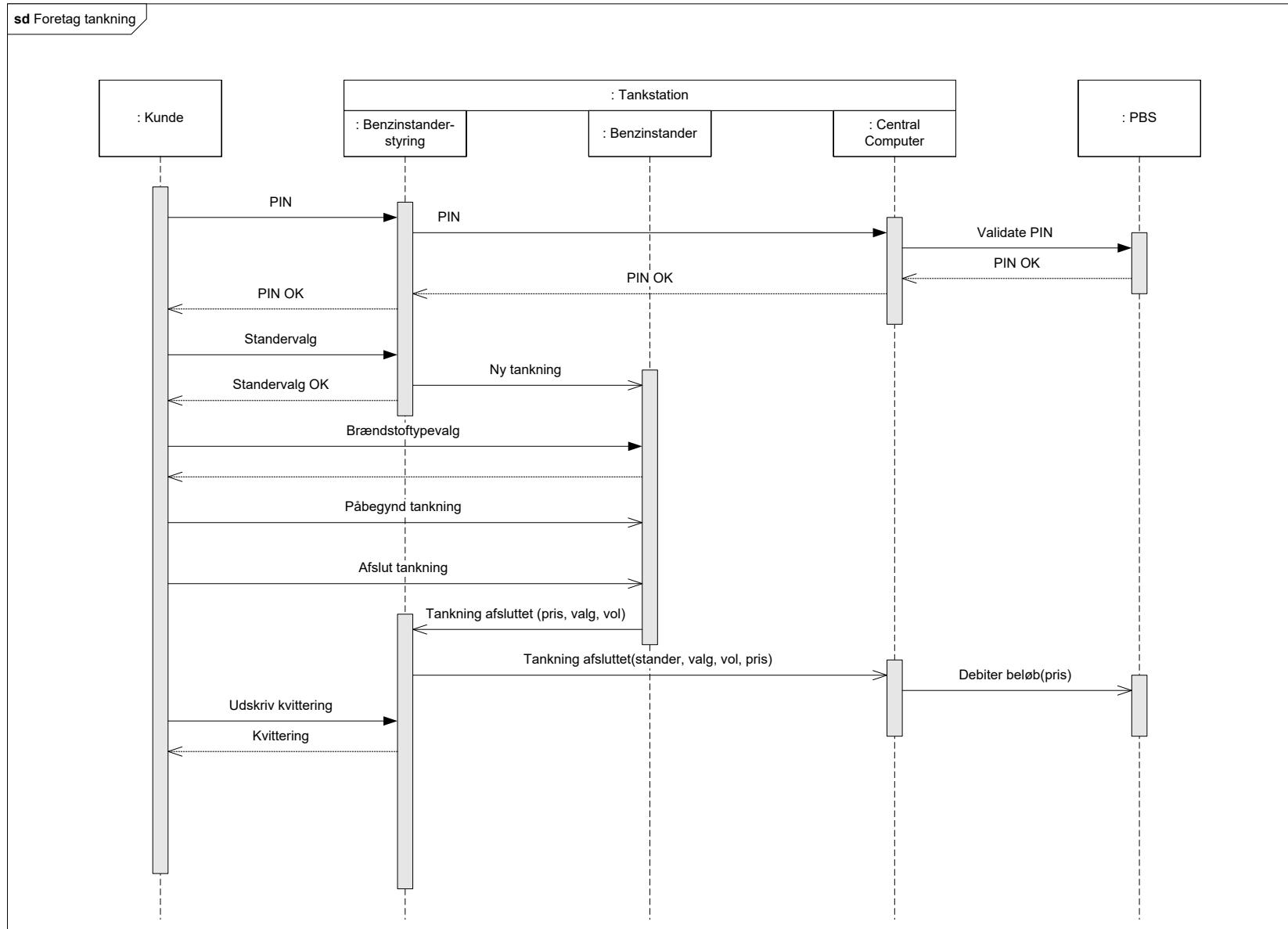
IBD for systemet



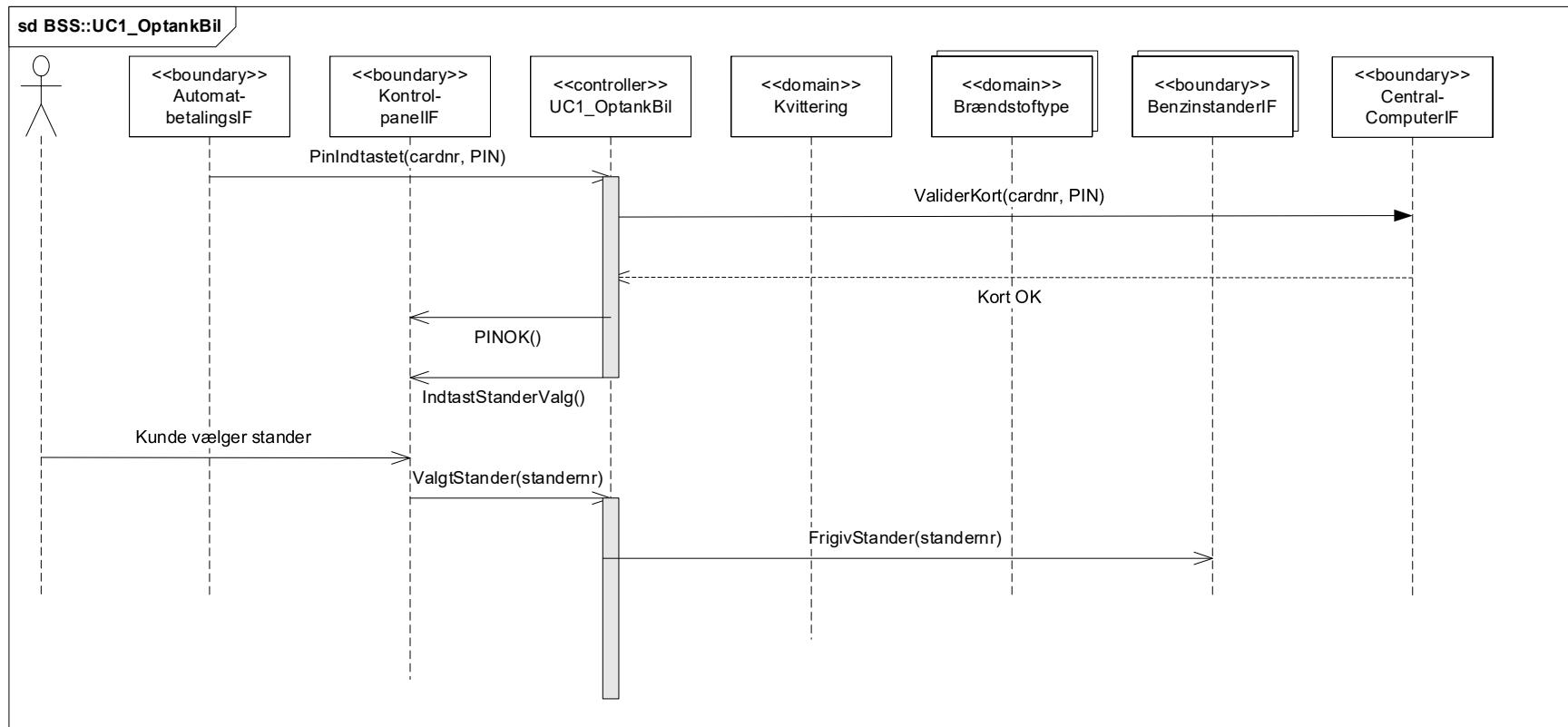
1. Version af klassediagram for Benzinstanderstyringen (BSS)



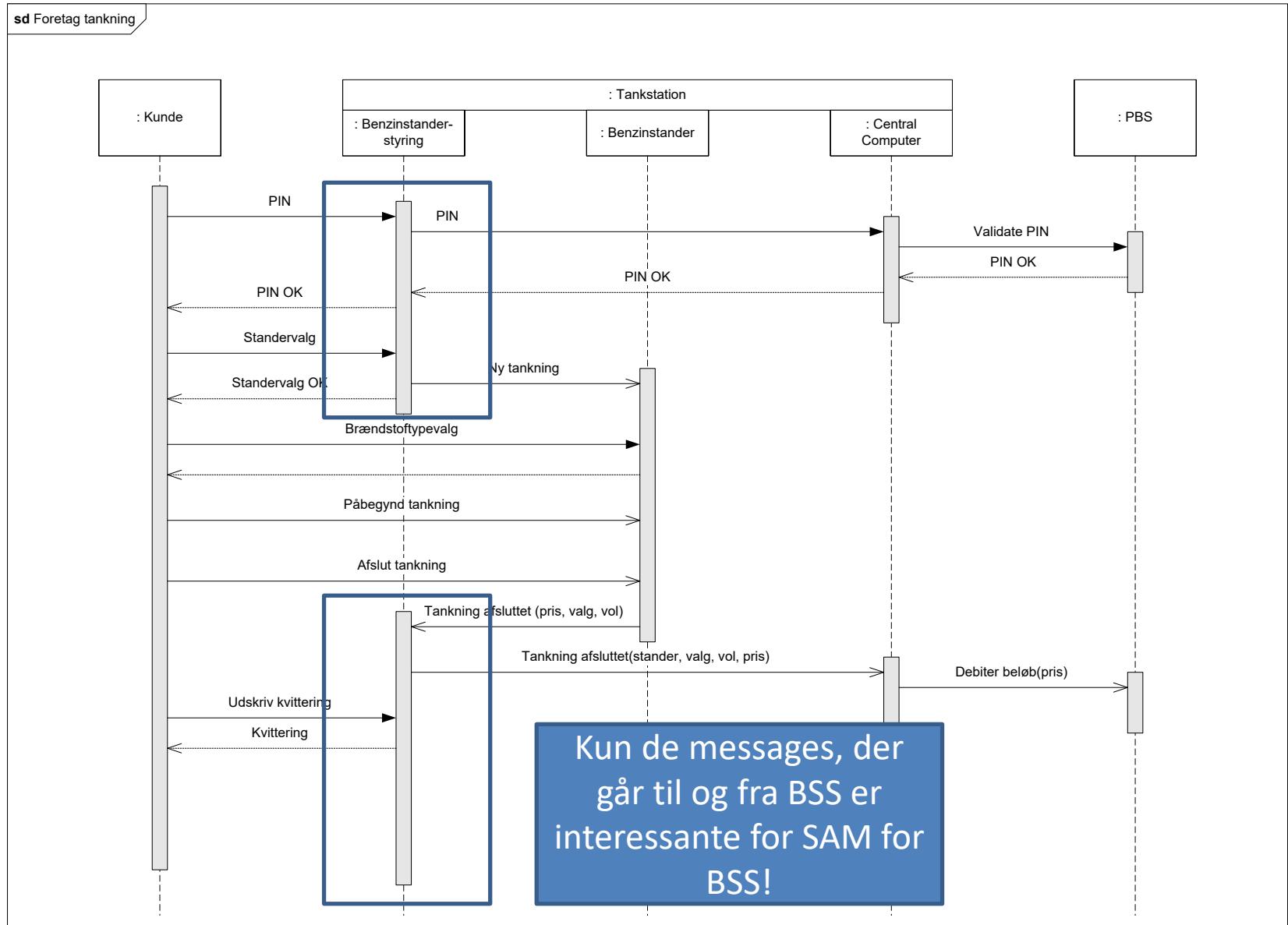
System SD for UC Optank Bil



Start på Applikationsmodellens SD for BSS for UC Optank Bil



System SD for UC Optank Bil



Your turn: System Application Model for Benzinstanderstyring – UC Optank Bil

- Førdiggør **Applikationsmodellen** for **subsystemet Benzinstanderstyringen** for "UC Optank Bil"
 - Brug som input
 - BDD og IBD
 - System sekvensdiagrammet
 - Domænemodellen
 - Den 1. version af klassediagrammet (se ovenfor)
1. Check at steps 1.1-1.4 er gennemført korrekt for den 1. version af klassediagrammet
 2. Gennemfør steps 2.1-2.5 ved at fortsætte med at arbejde med 1. version af klassediagrammet og lav det tilhørende sekvensdiagram
 3. Tænk og check om der mangler domæneklasser, hardware interface klasser, etc, eller om der nogen, der er overflødige **for dette subsystem!**

System Architectural Design

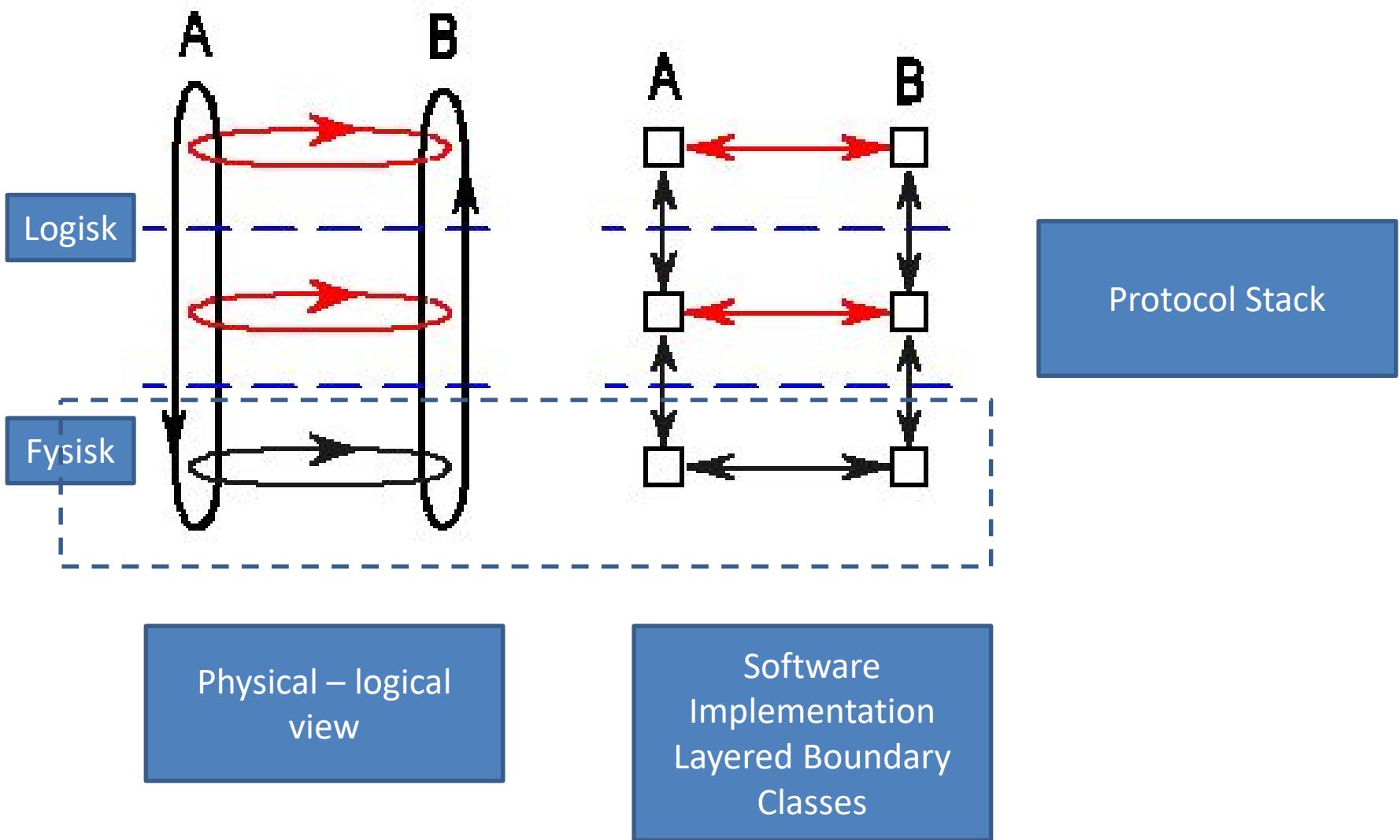
Protocols

I2ISE

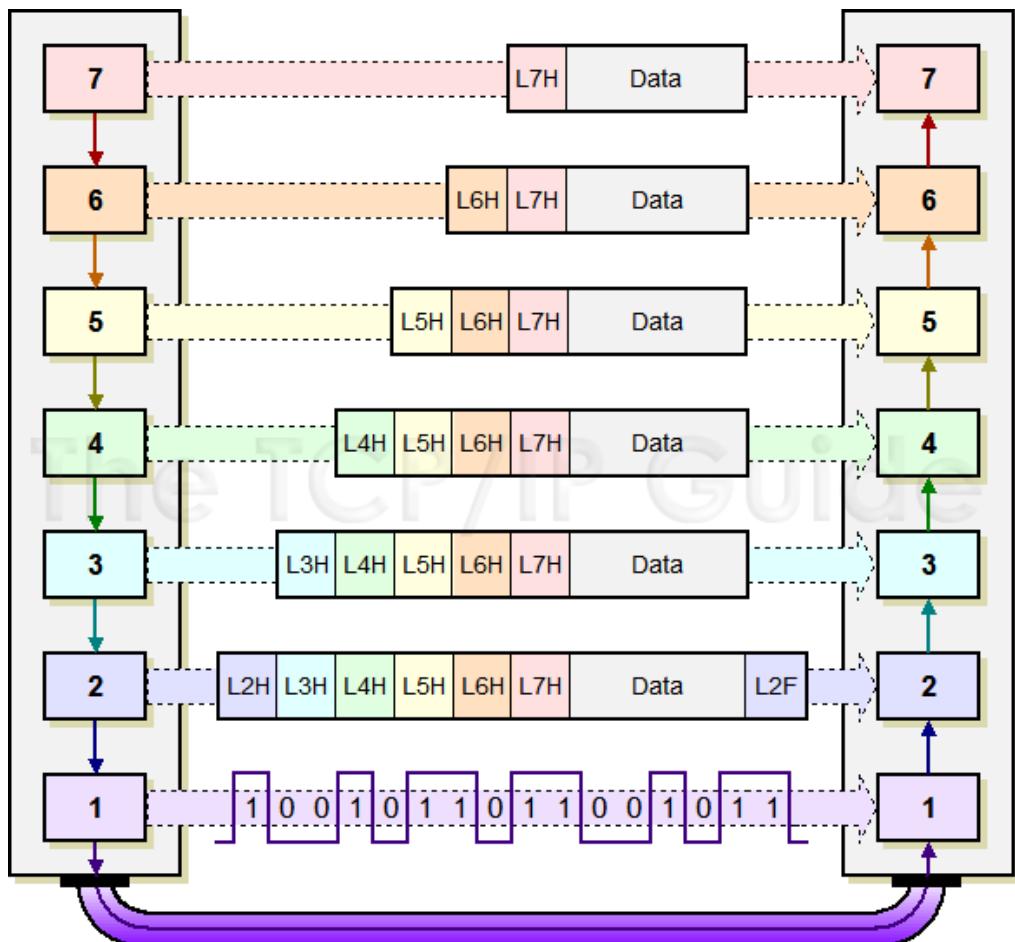
Protocols

- Protocols are one "step up" from the physical layer (signals, names, voltage levels etc.)
- Protocols define *how* the physical interface is used
 - E.g.: The *physical* interface is RS232 – 9 or 25 wires carrying data (Rx, Tx) and control (RTS, RTR, CTS, ...) signals
 - The protocol defines how data transmitted/received shall be interpreted.
- The protocol must specify the interface unambiguously

Physical – logical - software



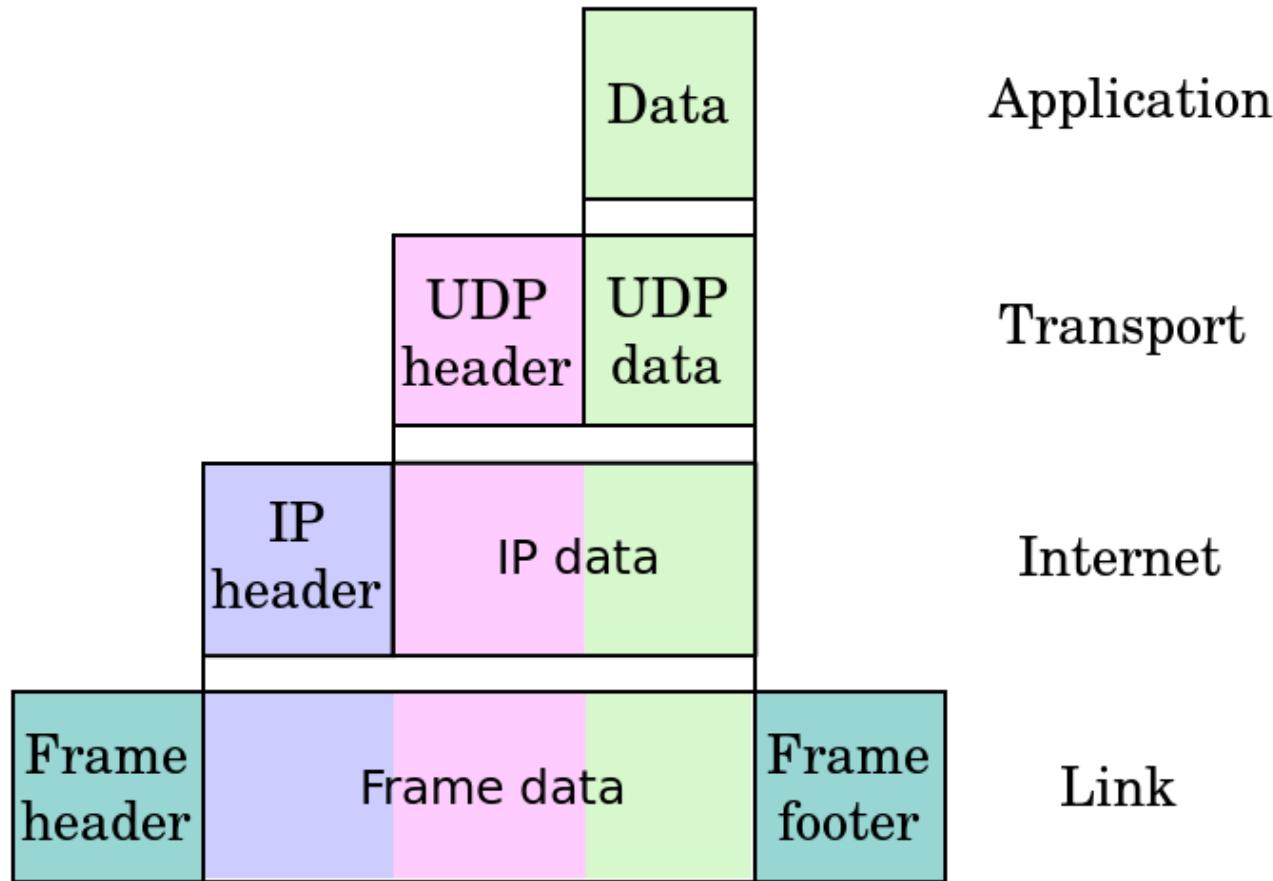
Example: Data encapsulation in the OSI 7-layer model (Open Systems Interconnection)



OSI layers

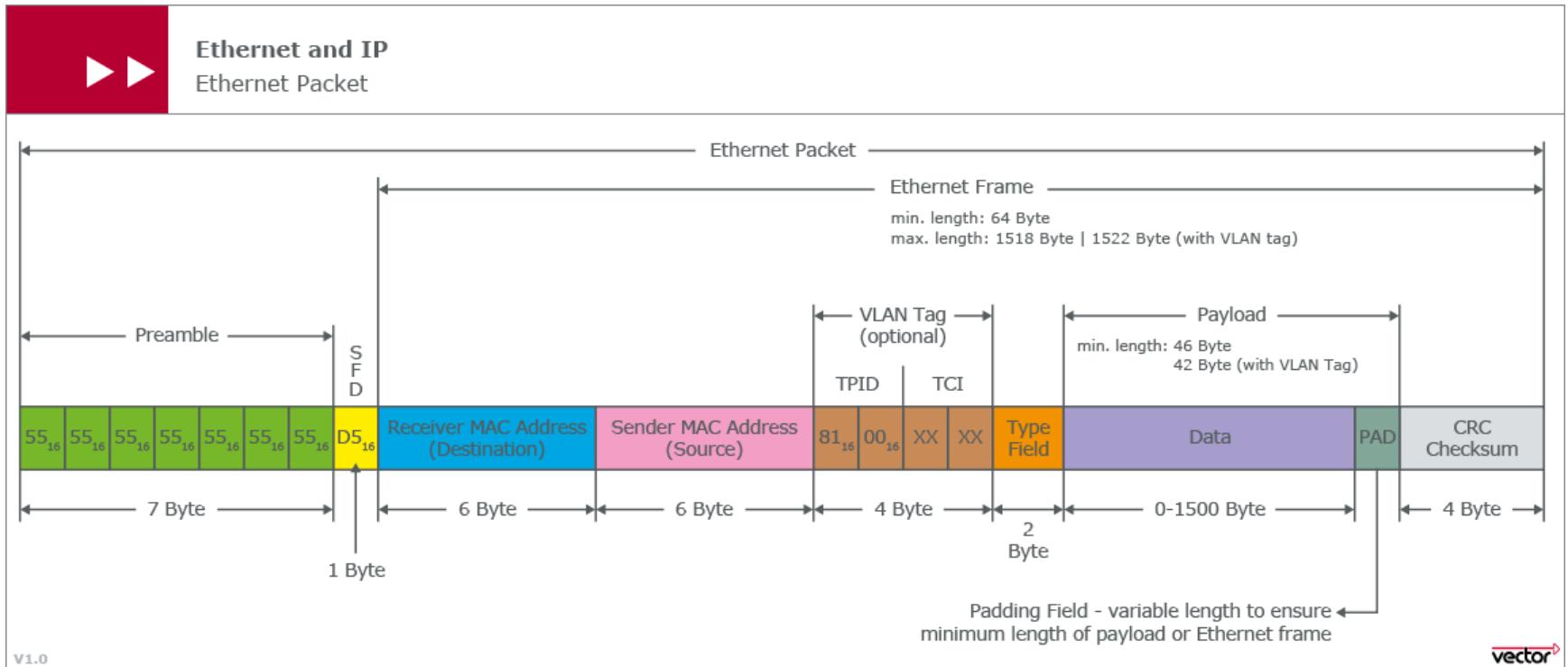
- Application Layer (7)
 - Communication between applications
 - Remote objects or functions
(Eg. HTTP and web browsers)
- Presentation Layer (6)
 - Conversion between data representation
- Session Layer (5)
 - Handles connections like a phone call, TCP connection
- Transport Layer (4)
 - Splitting of data in packages/frames (Segmentation/de-segmentation)
 - Ex. Internet
 - Transmission Control Protocol (TCP)
 - User Datagram Protocol (UDP)
- Network Layer(3)
 - Translates logical to physical addresses
 - Routing of messages through intermediate nodes
 - May deliver messages by split into several frames
- Data Link Layer (2)
 - Moves frames as a collection of bits
 - Acknowledgement from receiver
 - Ensure error free transmission
- Physical Layer (1)
 - Mechanical and electrical interface
 - Moves bits over a communication channel
 - Concerns how the connection is established

IP Layers for UDP/IP



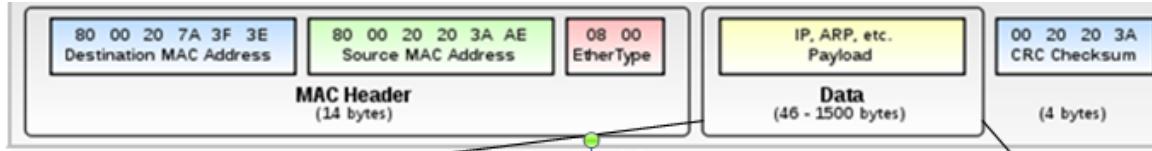
Unfortunately there is not an exact match to OSI!

Ethernet frame - logical level



UDP/IP embedded in Ethernet

Ethernet
Data



IP Data



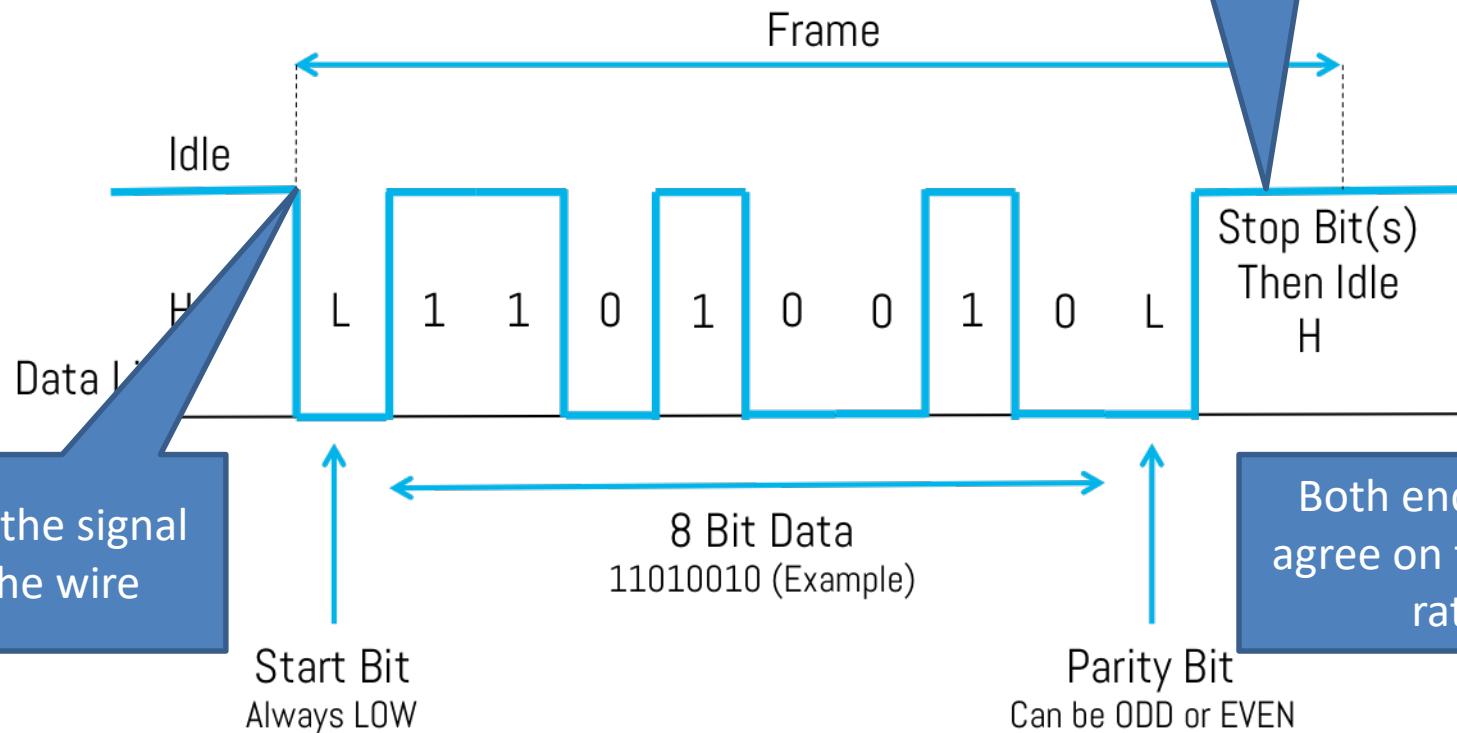
UDP Data



Physical layer examples

- RS232
- X10
- Ethernet

RS232 frame

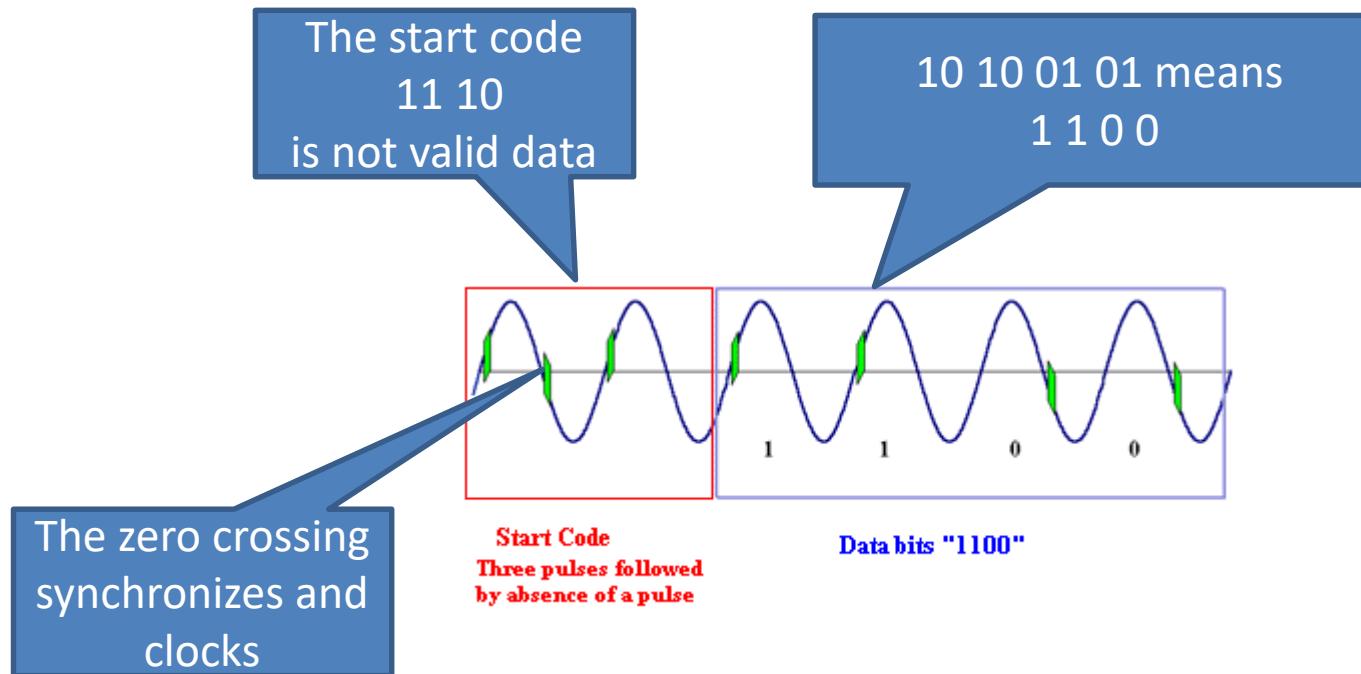


The AVR USART Frame Format
© maxEmbedded.com 2013

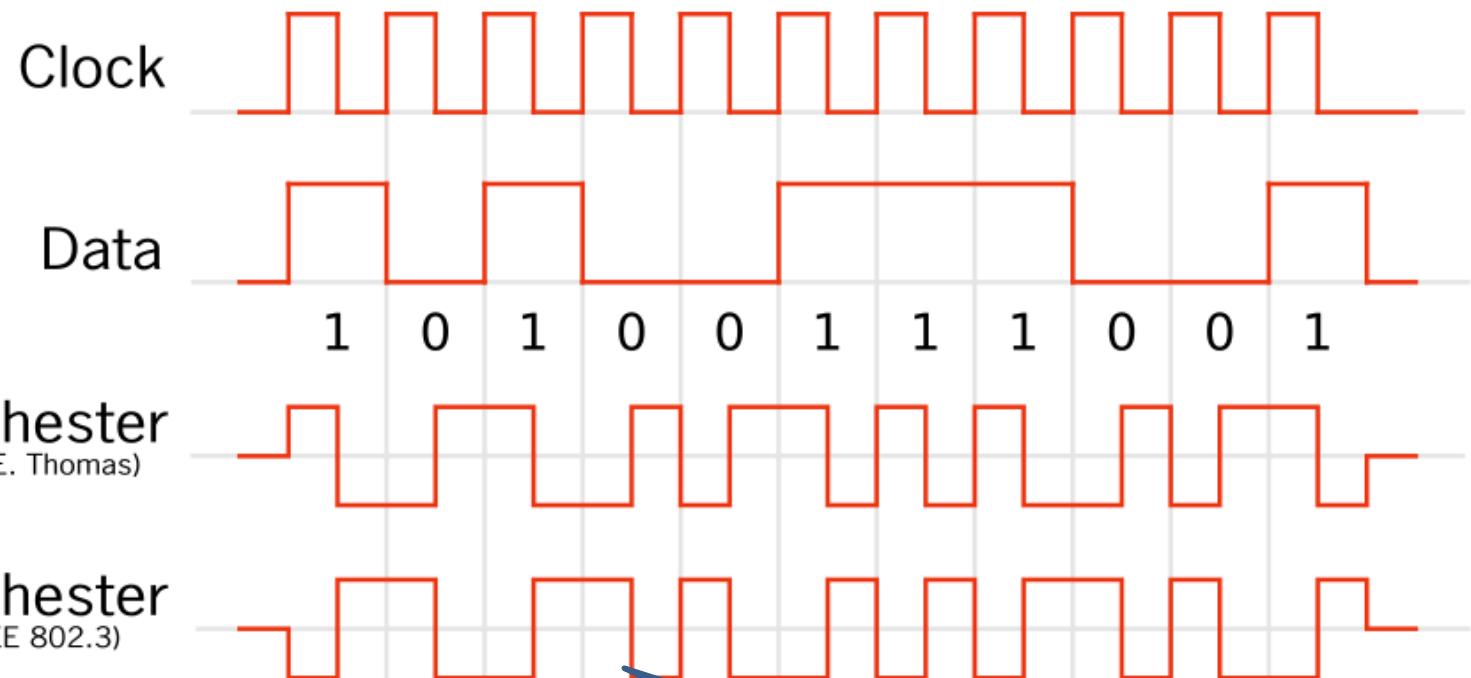


maxEmbedded.com
a guide to robotics and embedded systems

X10 – physical level



Ethernet – physical level



This code is self
clocking

This is the signal
on the wire

Handling of errors

- Error detection
 - Format: Header, Length, Payload (data), Footer
 - Checksums
 - Acknowledge
 - Timeouts
- Error correction/elimination
 - Retransmission
 - Resynchronisation
 - Error correcting codes
 - Robustness at the hardware level

Simple error correcting scheme

Triplet received	Interpreted as
000	0 (error free)
001	0
010	0
100	0
111	1 (error free)
110	1
101	1
011	1

Examples of error correcting codes

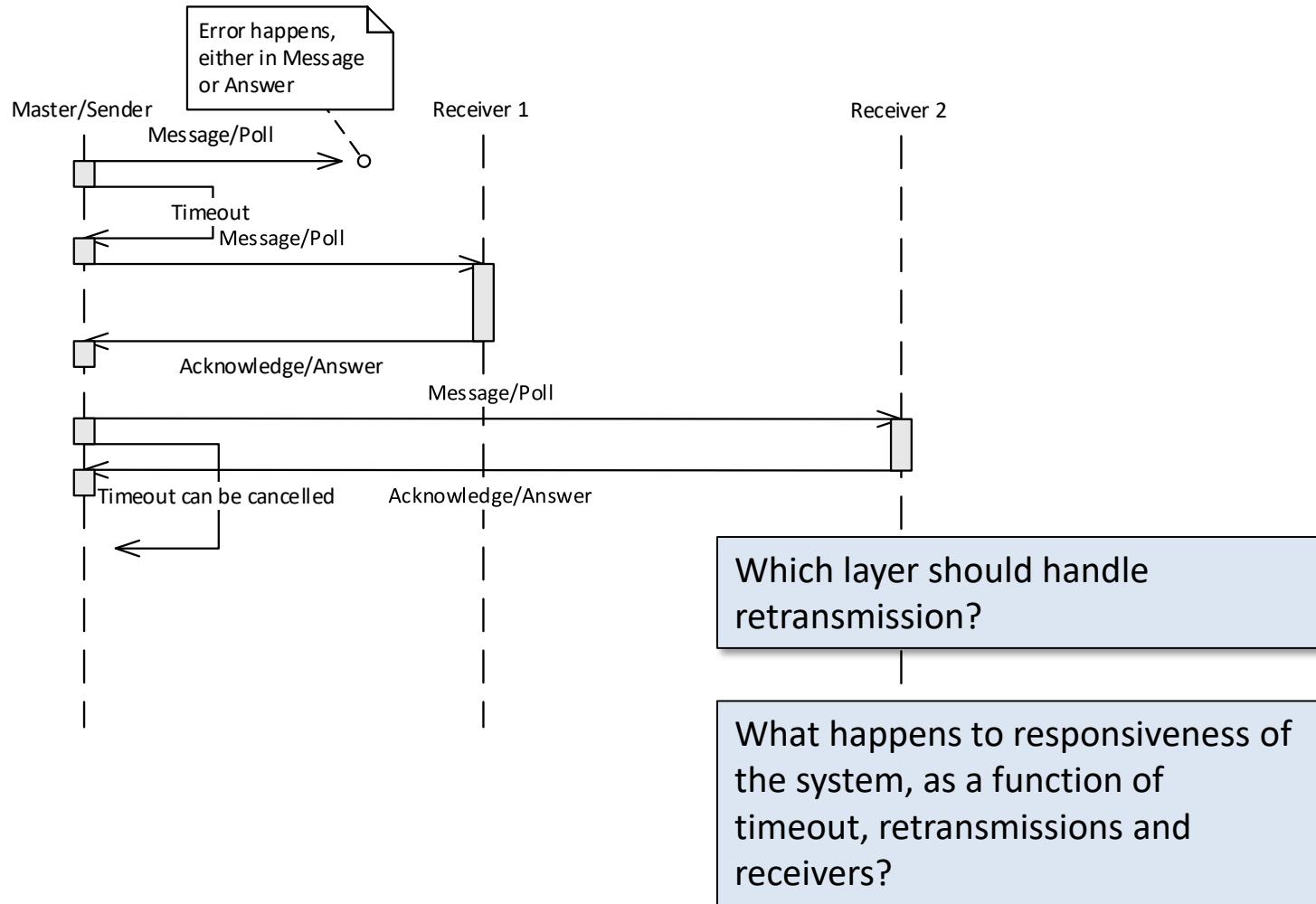
Detect	Correct	Method
1	0	Parity
1	1	Triple modular redundancy (see previous slide)
1	1	perfect Hamming such as Hamming(7,4)
2	1	Extended Hamming
3	3	perfect binary Golay code
4	3	extended binary Golay code

Used by NASA
for Deep Space
Missions

Can correct 3
errors by using
24 bits to send
12 data bits

Typical Master/Slave setup

Half Duplex



Retransmission is not an option!

- <https://voyager.jpl.nasa.gov/mission/status/>

Example: Proprietary protocol

Byte	0	1	2	3	4	n+3	n+4	
Contents	STX	Type	Len	B0	B1	...	Bn	ETX

Data request

Request for most recent data

Direction: Master > Slave

Type: '0' (ASCII)

Len: '00' (ASCII)

Data: -

Data response

Most recent data

Direction: Slave > Master

Type: '1' (ASCII)

Len: '04' (ASCII)

Data: B0: Sensor 1 LSB (binary)

B1: Sensor 1 MSB (binary)

B2: Sensor 2 LSB (binary)

B3: Sensor 2 MSB (binary)

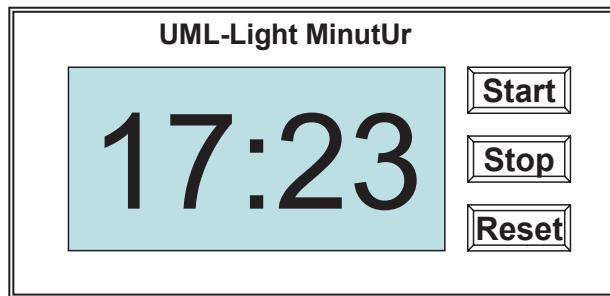
Your turn!

- Start the specification of a protocol between the PC program and the X.10 Controller in your semester project.
- Consider what information must flow (e.g. from System Sequence Diagrams)
(Consider how to test it using a terminal program by sending only ASCII characters.)
 - Header (e.g. STX = 'C', message type, ...)?
 - Data?
 - Footer (e.g. ETX = 0x13 <CR>)?
- Who is Master and Slave?
- Lessons learned in MSYS on 1st semester?
 - SendString, SendChar, SendInt
- Lessons learned from Application Models?
 - What Software modules could be feasible?

5 Eksempel 1: Minutur designet med UML-Light

5.1 Beskrivelse af eksemplet

Dette eksempel viser en model for et simpelt minutur. Minuturet kan startes, stoppes og resettes ved hjælp af tre knapper, der er anbragt på et Knappanel. Knappanelet er forbundet til en inputport på en microcontroller. Når minuturet er startet, optæller det tiden og viser den i minutter og sekunder på et display i formatet minutter:sekunder. Displayet er forbundet til en outputport på microcontrolleren. Til at styre tiden anvendes en i microcontrolleren indbygget timerkreds, der kan tælle i millisekunder. For at simplificere eksemplet mest muligt vil hovedprogrammet (main) her *polle* (periodisk aflæse) både knappanel og timer.



Figur 21. Skitse af brugergrænseflade for Minutur

5.2 UML-Light model

Som det ses af klassediagrammet på Figur 22, består programmet af fire objekter, der er instantieret ud fra klasserne Ur, Timer, Display og Knappanel. Hovedprogrammet er implementeret vha. funktionen *main()*.

Det er som tidligere nævnt vigtigt at supplere et klassediagram med en beskrivelse af hver klassens ansvar.

Klasserne på Figur 22 har følgende ansvar:

Hovedprogram:

Denne klasse af typen ”utility” repræsenterer selve hovedprogrammet, der er repræsenteret ved C-funktionen *main()*. Hovedprogrammet poller hhv. KnapPanel og Timer objektet for hændelserne *start*, *stop*, *reset* og *timeout*, der sendes til Ur objektet for behandling.

Ur:

Denne klasse har ansvaret for at implementere selve minuturfunktionaliteten. Klassen er beskrevet vha. en tilstandsmaskine, der er vist på tilstandsdiagrammet på Figur 23.

Timer:

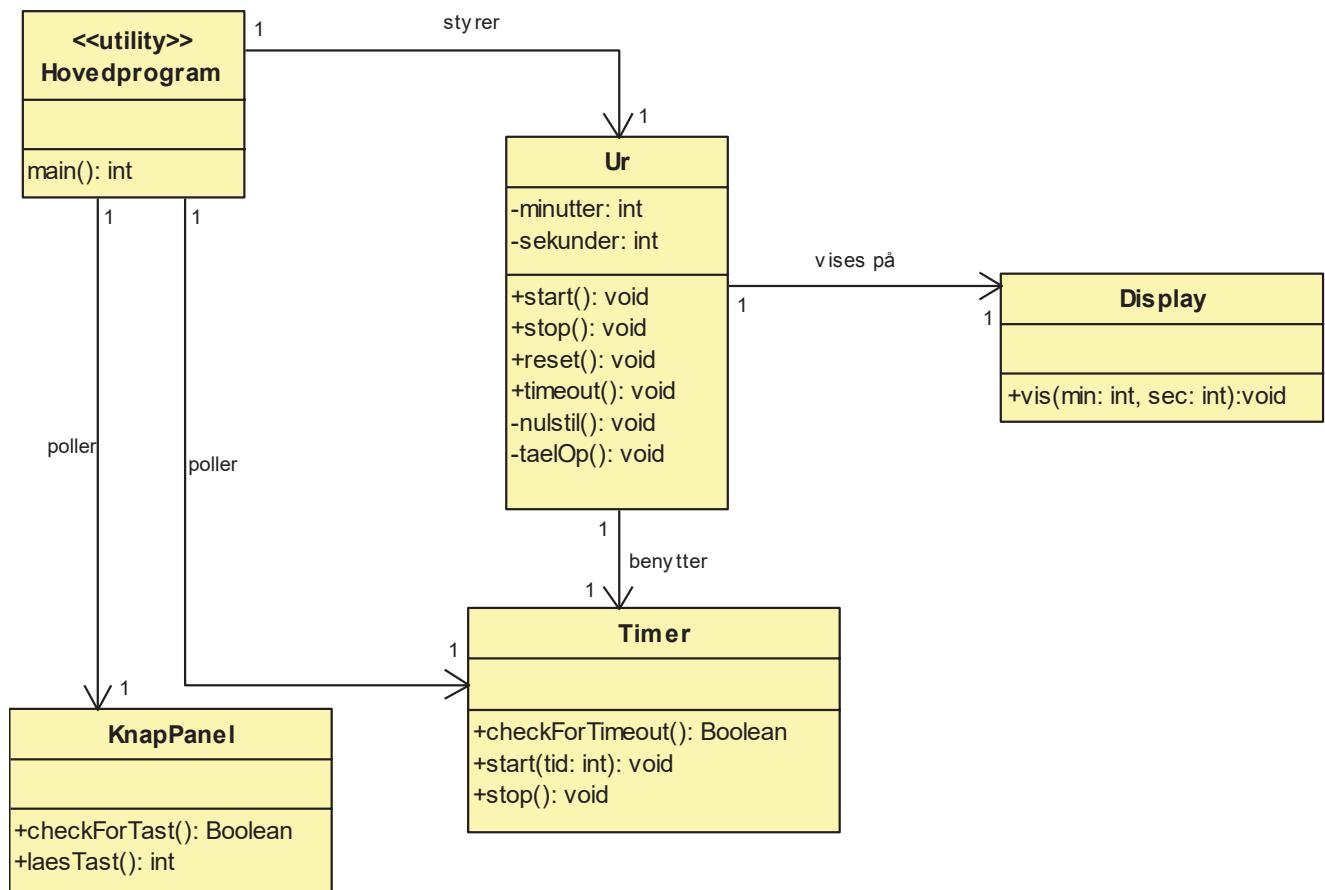
Denne klasse har ansvaret for at implementere SW grænsefladen til en hardwaretimer. Hardwaretimeren kan startes med et tælletal i milisekunder, hvorefter den tæller ned til 0.

KnapPanel:

Denne klasse har ansvaret for at teste, om en knap er aktiveret og i givet fald at kunne aflæse den aktuelle knaps værdi. Klassen håndterer et KnapPanel med knapperne *Start*, *Stop* og *Reset*.

Display:

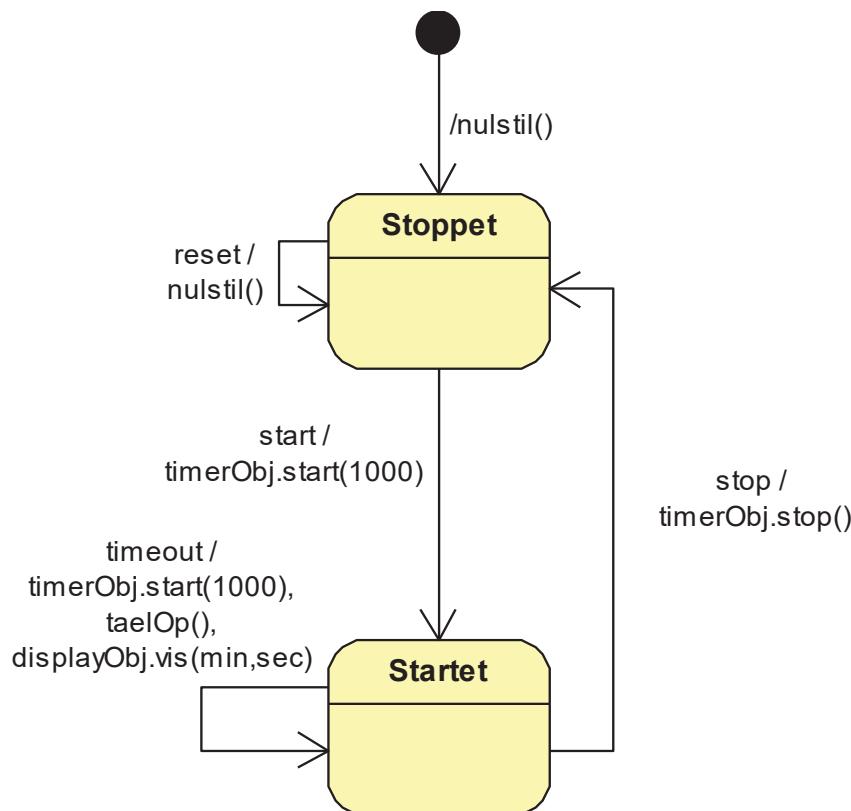
Denne klasse har ansvaret for at vise tiden i minutter og sekunder på et display med fire cifre og et kolon (format mm:ss).



Figur 22. Klassediagram for Minutur

Operationerne på klassediagrammet er fundet ved at udarbejde de to følgende diagrammer.

Klassen Ur er designet vha. en tilstandsmaskine, der er vist på tilstandsdigrammet på Figur 23.

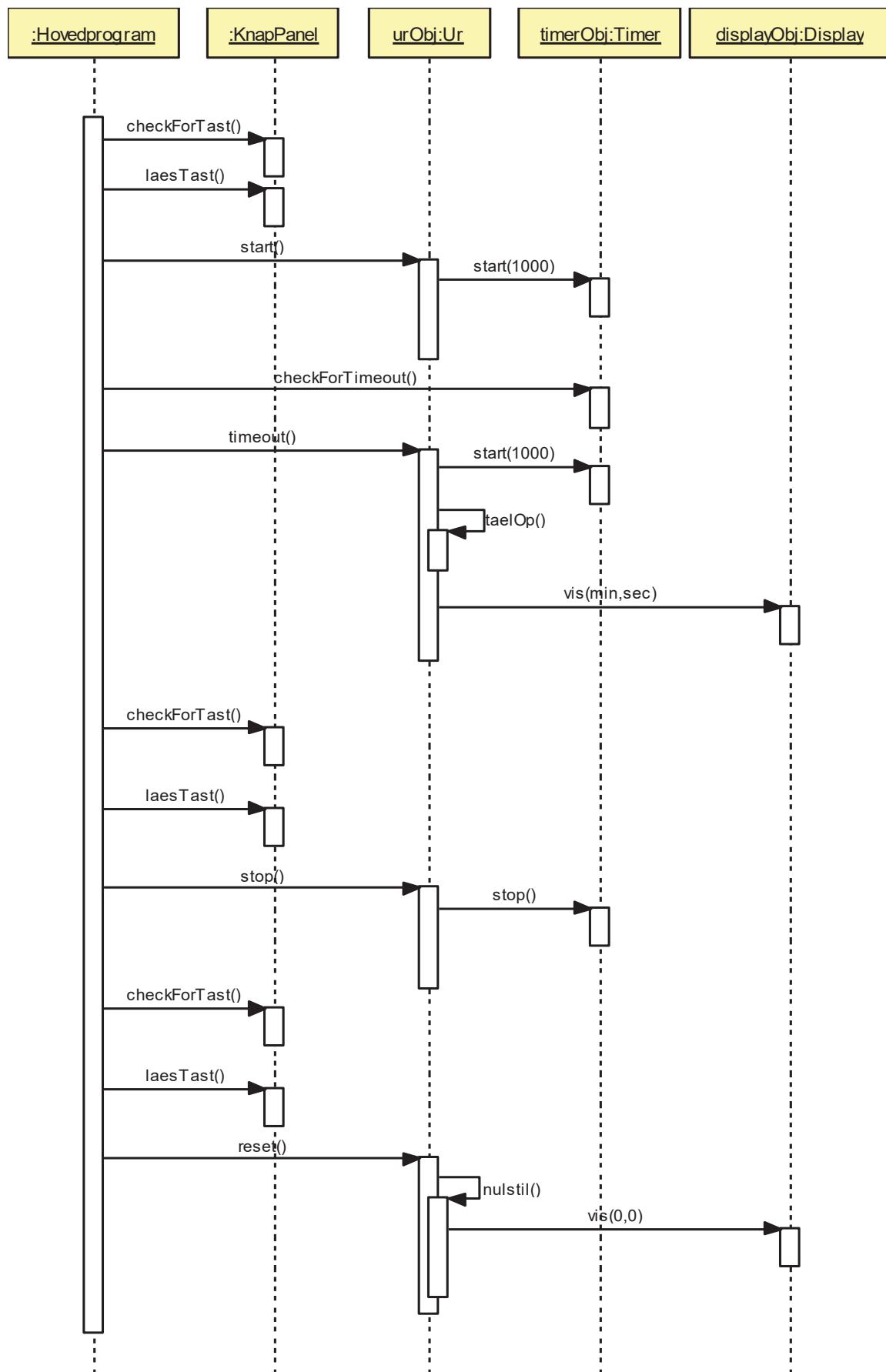


Figur 23. Tilstandsdiagram for Minuturet

Tilstandsdiagrammet har to tilstande *stoppet* og *startet*. I tilstanden *stoppet* har man mulighed for at nulstille minuturet vha. hændelsen ”*reset*”, der kommer, når man aktiverer *reset* tasten. Når brugeren aktiverer *start* tasten, vil der blive sendt en ”*start*” hændelse til tilstandsmaskinen, hvorefter der skiftes tilstand samtidig med, at der sendes en ”*start*” meddeelse til timerobjektet *timerObj*. Når tiden er gået (1000ms), modtages hændelsen ”*timeout*”, der bevirker at operationen *taelOp()* og operationen *vis(min,sec)* kaldes samtidig med, at der sendes en ny ”*start*” meddeelse til timerobjektet (*timerObj*).

Nu mangler vi kun at beskrive samspillet mellem hovedprogrammet og de fire objekter, der instantieres ud fra klassediagrammet. Dette samspil beskrives vha. UML-Light sekvensdiagrammet på Figur 24. Der er her kun vist et enkelt forløb af hhv. et *start*, et *timeout*, et *stop* og et *reset* scenario.

I de følgende to afsnit vises implementering af minuturet i henholdsvis C++ og i C.



Figur 24. Sekvensdiagram for Minutur

5.3 Minutur implementeret i C++

I dette afsnit vises, hvorledes de vigtigste af klasserne på klassediagrammet på Figur 22 kan implementeres vha. C++ kode.

Prøv at sammenligne koden med klassediagrammet og se den fine sammenhæng mellem design og kode.

Utilityklassen *Hovedprogram* implementeres i filen ***Hovedprogram.cpp***:

```
#include "Ur.h"
#include "KnapPanel.h"
#include "Timer.h"
#include "Display.h"

enum KnapHaendelse {START=1, STOP=2, RESET=3};

int main()
{
    KnapPanel knapPanelObj; // her oprettes de fire objekter som anvendes
    Timer timerObj;
    Display displayObj;
                                // ur objektet bliver her initialiseret med
                                // og display objekt pointere til timer
    Ur urObj(&timerObj,&displayObj);

    KnapHaendelse knapHaendelse;

    while ( true )           // Uendelig hovedløkke i programmet
    {
        // polling af knappanel
        if (knapPanelObj.checkForTast() == true)
        {
            knapHaendelse= (KnapHaendelse) knapPanelObj.laesTast();
            if (knapHaendelse == START)
                urObj.start();
            else if (knapHaendelse == STOP)
                urObj.stop();
            else if (knapHaendelse == RESET)
                urObj.reset();
        }
        // polling af timer for timeout
        if (timerObj.checkForTimeout() == true)
            urObj.timeout();
    }
    return 0;
} // end main
```

I C++ anvendes *enum* til at definere en såkaldt enumeration type, der definerer en datatype, der kun kan antage et bestemt antal værdier. I eksemplet definerer `enum KnapHaendelse {START=1, STOP=2, RESET=3};`, datatypen *KnapHaendelse*. En variabel af typen *KnapHaendelse* kan kun antage værdien START, STOP eller RESET, der ved defineringen også har fået tildelt de tilhørende konstantværdier. Anvendelsen af enumeration types gør programmet lettere at læse og anvendelsen sikrer samtidig også at konstanterne ikke overlapper i værdi.

Klassen *Ur* specificeres i filen ***Ur.h***:

```
#include "Timer.h"
#include "Display.h"

class Ur
{
public:
    Ur(Timer*, Display*);      // constructor operation
    void start();
    void stop();
    void reset();
    void timeout();
private:
    void nulstil();           // privat da den kaldes via tilstandsmaskinen
    void taelOp();             // privat da den kaldes via tilstandsmaskinen

    int minutter;
    int sekunder;
    Display* mitDisplay;
    Timer* minTimer;
    enum TILSTAND {STARTET, STOPPET} tilstand;
    const static int TIME1000MS= 1000;
};
```

Klassen *Ur* implementeres i filen ***Ur.cpp***, der implementerer tilstandsmaskinen. Her er tilstandsmaskinen direkte implementeret i koden på den simplest mulige måde, der er ok for dette enkle eksempel.

```
#include "Ur.h"
#include "Timer.h"
#include "Display.h"

Ur::Ur(Timer* timerPtr, Display* displayPtr)
{                                         // constructor operation
    minTimer= timerPtr;
    mitDisplay= displayPtr;
    tilstand = STOPPET;
    nulstil();
}

void Ur::start()                      // håndterer hændelsen: start
{
    if (tilstand == STOPPET)
    {
        tilstand= STARTET;
        minTimer->start(TIME1000MS);
    }
}

void Ur::stop()                       // håndterer hændelsen: stop
{
    if (tilstand == STARTET)
    {
        tilstand= STOPPET;
        minTimer->stop();
    }
}
```

```
void Ur::reset()          // håndterer hændelsen: reset
{
    if (tilstand == STOPPET)
    {
        nulstil();
    }
}

void Ur::timeout()         // håndterer hændelsen: timeout
{
    if (tilstand == STARTET)
    {
        minTimer->start(TIME1000MS);
        taelOp();
        mitDisplay->vis(minutter, sekunder);
    }
}

void Ur::nulstil()          // privat nulstil operation
{
    minutter=0;
    sekunder=0;
    mitDisplay->vis(minutter, sekunder);
}

void Ur::taelOp()           // privat taelOp operation
{
    sekunder++;
    if (sekunder >= 60)
    {
        sekunder=0;
        minutter++;
        if (minutter >= 60)
            minutter=0;
    }
}
```

Klasserne KnapPanel, Timer og Display er alle grænsefladeklasser til hardwaren. For hver af disse er der på tilsvarende måde en headerfil og en implementeringsfil. Disse klasser implementeres således, at deres *constructor* operation initialiserer hardwaren.

5.4 Minutur implementeret i C

Her skitseres, hvorledes det objektbaserede design af minuturet kan implementeres i programmesringssproget C.

5.4.1 C implementering af maksimum ét objekt pr. klasse

Først vises den simpleste og mest effektive implementering, der kan anvendes i de tilfælde, hvor der kun er ét objekt af hver klasse.

De private attributter og operationer implementeres her som *static* medlemmer, der i C betyder, at de kun kendes i den fil, hvori de er defineret. I modsætning til C++ så skal man i C definere disse private attributter og operationer vha. *static* i kodenfilen (.c) og ikke i headerfilen (.h).

Associationerne er her implementeret implicit ved, at en ”C-klasse” kun inkluderer de headerfiler, som den har associationer til på klassediagrammet. Der anvendes således ikke pointere til de andre objekter i denne simple men effektive implementering.

Mange C-compilere understøtter kun filnavne med 8 karakterer, hvorfor det kan være nødvendigt at anvende en forkortet version af Klassenavnene fra klassediagrammet til filnavnene.

Utilityklassen *Hovedprogram* implementeres i filen ***HovedPrg.c***:

```
#include "Define.h"
#include "Ur.h"
#include "KnapP.h"
#include "Timer.h"
#include "Display.h"

int main()
{
    enum KnapHaendelse knapHaendelse;
    // initiering af de fire "objekter"
    Display_init();
    Timer_init();
    KnapPanel_init();
    Ur_init();

    while ( true )    // Uendelig hovedløkke i programmet
    {
        // polling af knappanel
        if (KnapPanel_checkForTast() == true)
        {
            knapHaendelse= KnapPanel_laestAst();
            if (knapHaendelse == START)
                Ur_start();
            else if (knapHaendelse == STOP)
                Ur_stop();
            else if (knapHaendelse == RESET)
                Ur_reset();
        }
        // polling af timer for timeout
        if (Timer_checkForTimeout() == true)
            Ur_timeout();
    }
    return 0;
} // end main
```

I filen *Define.h* kan man f.eks. definere de globale konstanter og brugerdefinerede datatyper, der anvendes i projektet, hvorfor denne fil inkluderes som den første fil i alle øvrige filer. Her kan man f.eks. vha. *enum* definere en datatype *bool*, der kan antage værdien false eller true. (enum bool {false=0, true=1};).

Klassen *Ur* specificeres i filen *Ur.h*:

```
// class Ur
// private:
#define TIME1000MS 1000

// public:
    extern void Ur_init(); // Ur_init er constructor operationen
    extern void Ur_start();
    extern void Ur_stop();
    extern void Ur_reset();
    extern void Ur_timeout();
// end class Ur
```

Bemærk at alle public operationer navngives med klassenavnet efterfulgt af operationsnavnet.

Klassen *Ur* implementeres i filen *Ur.c*, der implementerer tilstandsmaskinen.

```
#include "Ur.h"
// inkludering af de "objekter" hvis operationer kaldes fra Ur
#include "Timer.h"
#include "Display.h"

//***** private (static) attributter *****
static enum TILSTAND {STARTET, STOPPET} tilstand;
static int minutter;
static int sekunder;

//***** private (static) operationer *****
static void nulstil() // privat nulstil operation
{
    minutter=0;
    sekunder=0;
    Display_vis(minutter,sekunder);
}

static void taelOp() // privat taelOp operation
{
    sekunder++;
    if (sekunder >=60)
    {
        sekunder=0;
        minutter++;
        if (minutter >=60)
            minutter=0;
    }
}

//***** public operationer *****
void Ur_init() // "constructor" operation
{
    tilstand = STOPPET;
    nulstil();
}
```

```
void Ur_start()           // håndterer hændelsen: start
{
    if (tilstand == STOPPET)
    {
        tilstand= STARTET;
        Timer_start(TIME1000MS);
    }
}

void Ur_stop()            // håndterer hændelsen: stop
{
    if (tilstand == STARTET)
    {
        tilstand= STOPPET;
        Timer_stop();
    }
}

void Ur_reset()           // håndterer hændelsen: reset
{
    if (tilstand == STOPPET)
    {
        nulstil();
    }
}

void Ur_timeout() // håndterer hændelsen: timeout
{
    if (tilstand == STARTET)
    {
        Timer_start(TIME1000MS);
        taelOp();
        Display_vis(minutter,sekunder);
    }
}

//*****
```

5.4.2 C implementering af flere objekter pr. klasse

For nogle klasser vil man også i et C-baseret projekt have brug for, at kunne oprette flere objekter af en given klasse. Princippet for dette vises her med udgangspunkt i klassen Ur.

Objektets datatype defineres vha. en *typedefinition* (*typedef*) og en *C struct* – denne nye type benævnes med samme navn som den tilhørende klasse på klassediagrammet.

```
typedef struct Ur_type
{
    int minutter;
    int sekunder;
    enum TILSTAND {STARTET, STOPPET} tilstand;
} Ur;
```

Herefter kan datatypen Ur anvendes til at skabe ”objekter” ud fra som det fremgår af følgende main() program.

Hovedprogrammet *main()* i filen **Main.c**, hvor der oprettes to Ur objekter.

```
int main()
{
    enum KnapHaendelse knapHaendelse;

    Ur mitUrNr1, mitUrNr2;      // her oprettes to Ur "objekter"

    Ur_init(&mitUrNr1);         // her initialiseres objekterne
    Ur_init(&mitUrNr2);
    Display_init();
    Timer_init();
    KnapPanel_init();

    Ur_start(&mitUrNr1);        // her kaldes operationen start()
    Ur_stop(&mitUrNr1);
    Ur_reset(&mitUrNr1);
    // etc.
    return 0;
}
```

Klassen *Ur* specificeres i filen **Ur.h**:

```
// class Ur
// private:
#define TIME1000MS 1000

typedef struct Ur_type
{
    int minutter;
    int sekunder;
    enum TILSTAND {STARTET, STOPPET} tilstand;
} Ur;

// public:
extern void Ur_init(Ur* const this); // Ur_init er constructoren
extern void Ur_start(Ur* const this);
extern void Ur_stop(Ur* const this);
extern void Ur_reset(Ur* const this);
extern void Ur_timeout(Ur* const this);
// end class Ur
```

Her vises et par eksempler på implementering af operationerne i filen **Ur.c**:

```
/****** static operationer *****

static void nulstil(Ur* const this) // privat nulstil operation
{
    this->minutter=0;
    this->sekunder=0;
    Display_vis(this->minutter, this->sekunder);
}
```

```
static void taelOp(Ur* const this) // privat taelOp operation
{
    this->sekunder++;
    if (this->sekunder >=60)
    {
        this->sekunder=0;
        this->minutter++;
        if (this->minutter >=60)
            this->minutter=0;
    }
}

//***** public operationer *****
void Ur_init(Ur* const this)           // "constructor" operation
{
    this->tilstand = STOPPET;
    nulstil(this);
}

void Ur_timeout(Ur* const this) // håndterer hændelsen: timeout
{
    if (this->tilstand == STARTET)
    {
        Timer_start(TIME1000MS);
        taelOp(this);
        Display_vis(minutter,sekunder);
    }
}
```

Af disse eksempler ses det, at hver operation i klassen Ur, som første parameter har en pointer, der identificerer det objekt, som operationerne skal udføres på. Dette gælder dog ikke for Display og Timer klasserne, hvor der her kun kan være ét objekt af hver. ”Objektpointeren” kaldes her for *this* som den hedder i C++. Som det ses af f.eks. operationen *taelOp()* så refereres objekts variable vha. denne *this* pointer (*this->sekunder++;*)

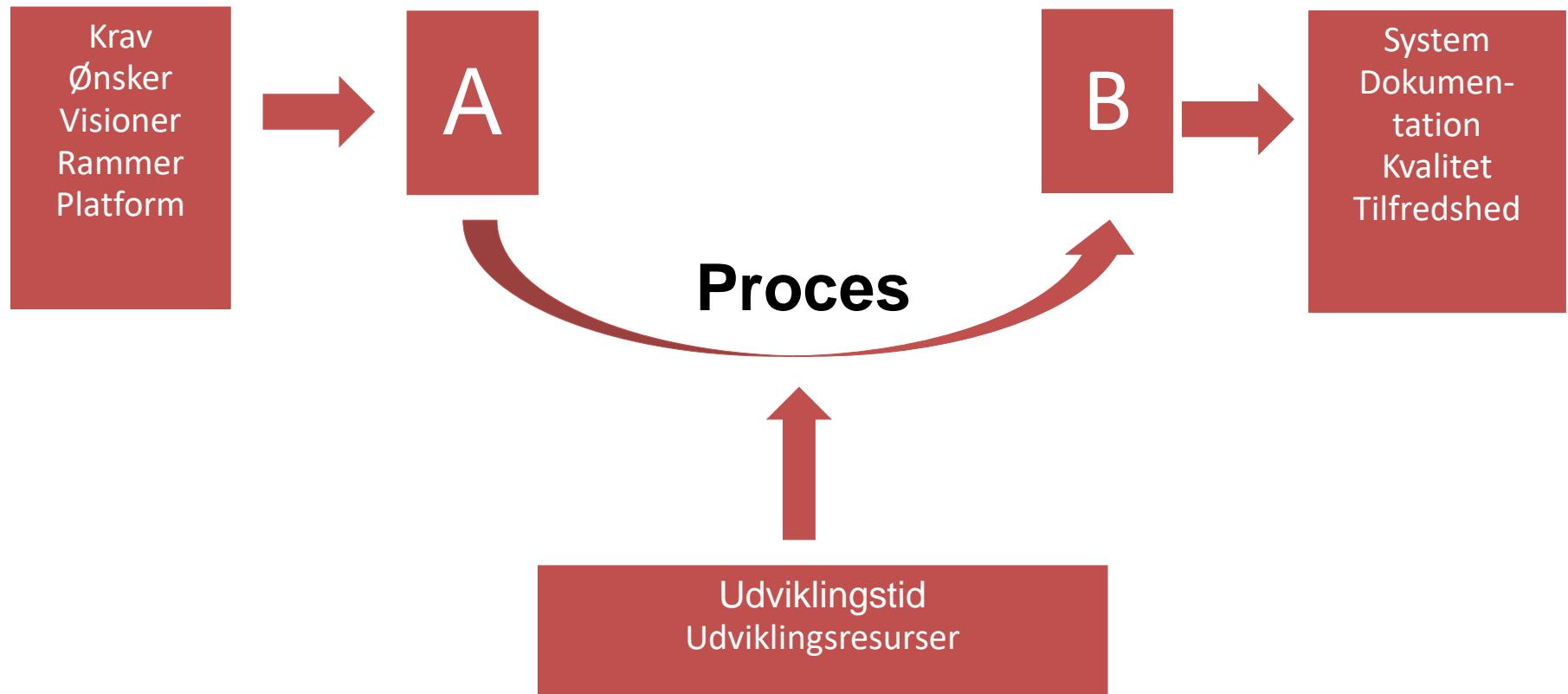
Development Processes

Introduction to Systems Engineering
I2ISE

Introduction

- What is a (development) process?
- Why do we need a development process?
- Some examples
 - Traditional, iterative, agile and the ASE processes

Systems engineering – skematisk set



What is a development process?

- A *process* is the action of taking something through a defined set of steps to transform something into something else
 - Milk → cheese, metal → car, thoughts → products, etc.
- A *development process* is a process defined to support development (of HW, SW, ...)
- A development process may define...
 - How to arrive at a product
 - What input is needed at what times
 - What (secondary) output there should be
 - ...

Why use a development process?

- Using a development process may seem to incur an overhead
 - E.g., you may not actually "produce" anything before "late" in the process
- So...why do we use it?
- Because we are engineers, so we are *concerned*
 - ...that we are producing the right thing
 - ...with the right capabilities
 - ...at the right time
 - ...at the right cost
 - ...

Why use a development process?



How the customer explained it



How the Project Leader understood it



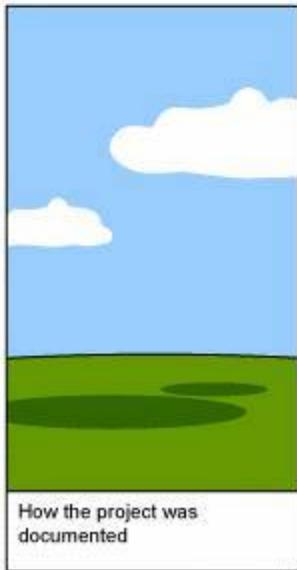
How the Analyst designed it



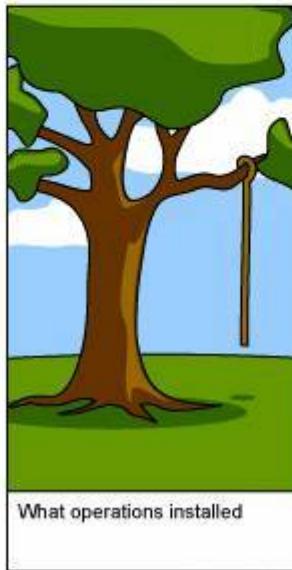
How the Programmer wrote it



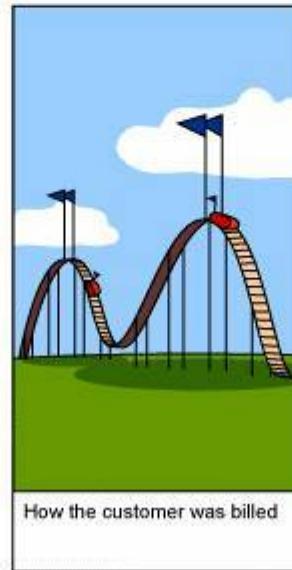
How the Business Consultant described it



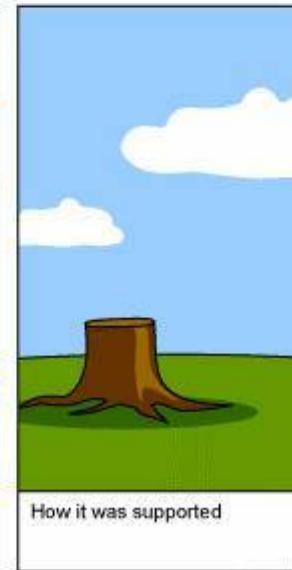
How the project was documented



What operations installed



How the customer was billed



How it was supported



What the customer really needed

Why use a development process?

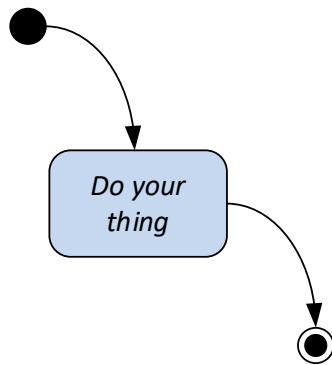
- Development processes answers some important questions:
 - What are you going to *produce*?
 - When will you be *done*?
 - What will it *cost*?
 - How will you handle *changes*?
- Answers to these questions are important to the customer
- Are the answers important to you? To your business? Why?

Examples:

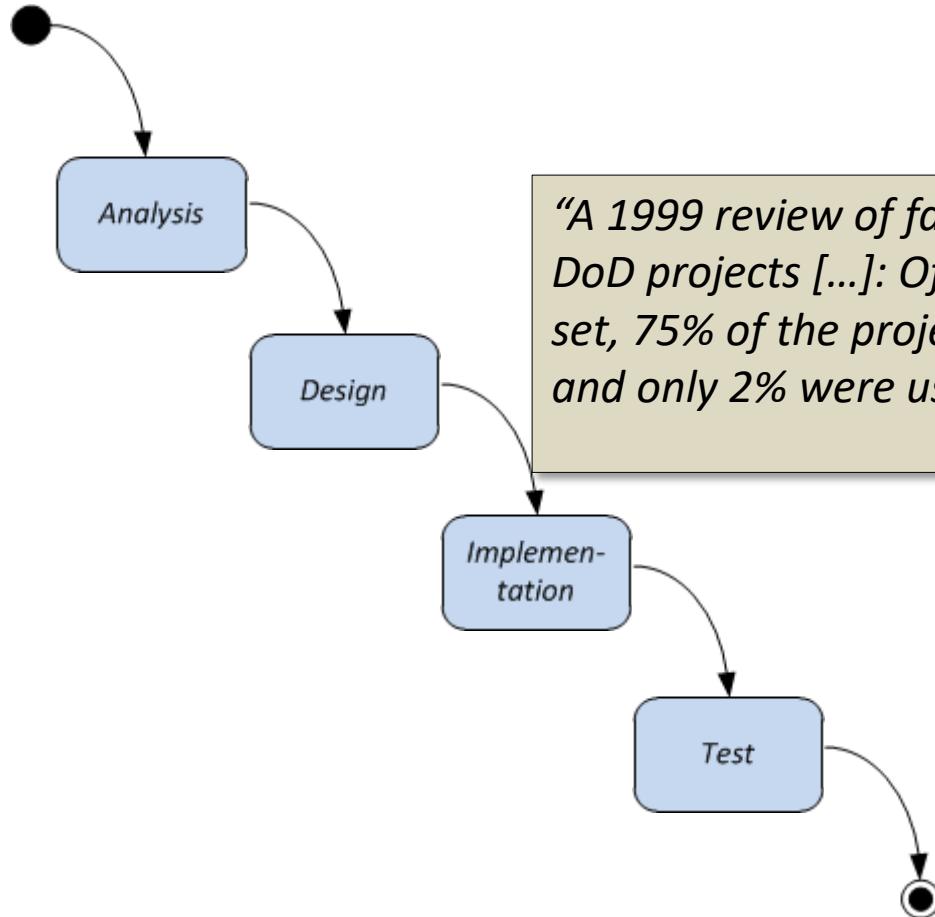
Traditional development processes

- The “null” process
- The waterfall process
- The V-model

The "null" process



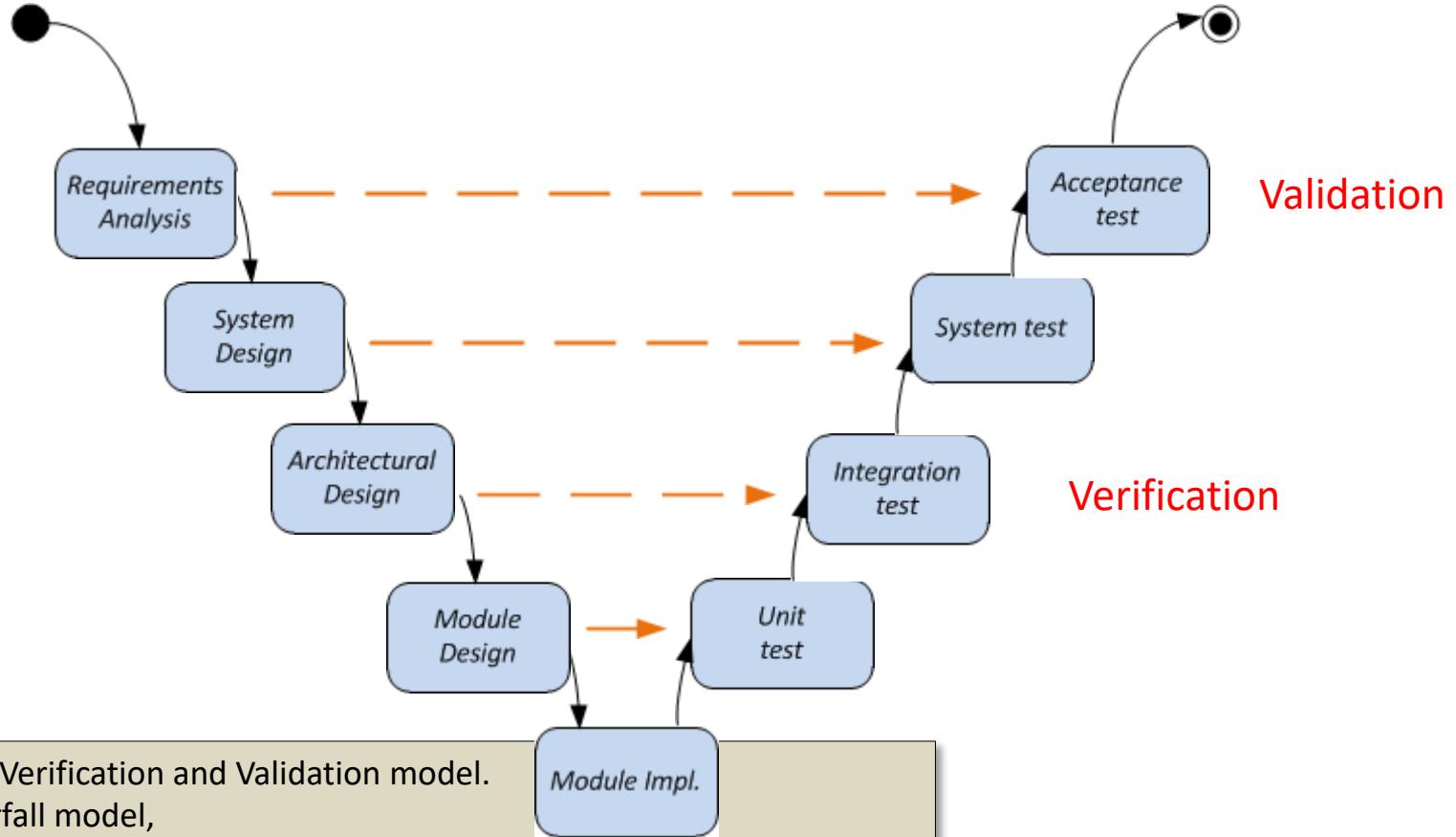
The waterfall process



"...a flawed, non-working model"
-Winston R. Royce, 1970

"A 1999 review of failure rates in a sample of earlier DoD projects [...]: Of a total \$37 billion for the sample set, 75% of the projects failed or were never used, and only 2% were used without extensive modification."
- S. Jarzombek, 1999

The V-model



"V- model means Verification and Validation model.
Just like the waterfall model,
the V-Shaped life cycle is a sequential path of execution of processes.
Each phase must be completed before the next phase begins.
Testing of the product is planned in parallel
with a corresponding phase of development" .. Try to improve quality

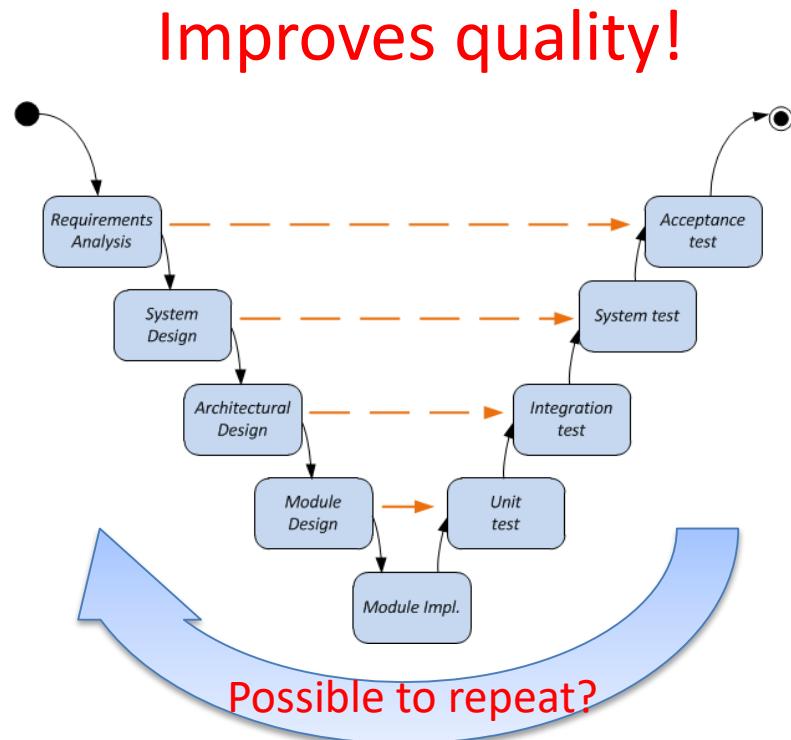
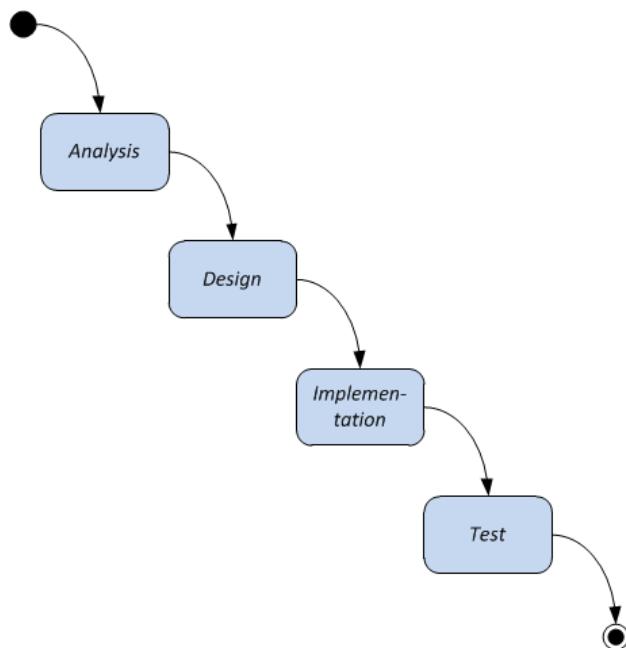
- ISTQB Certification

When to use the V-model?

- The V-shaped model should be used for **small to medium sized projects** where requirements are clearly defined and fixed.
- The V-Shaped model should be chosen when ample **technical resources are available** with needed technical expertise.
- **High confidence of customer** is required for choosing the V-Shaped model approach. Since, no prototypes are produced, there is a very high risk involved in **meeting customer expectations**.

Discussion

- What is the difference?

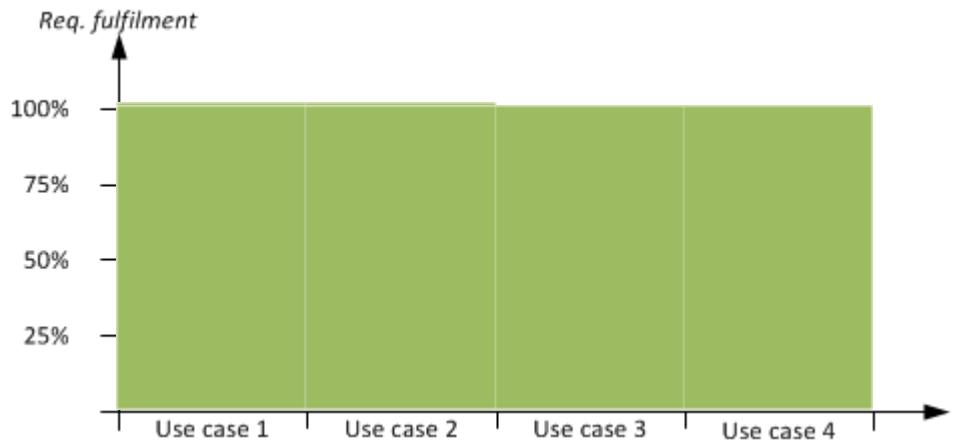


Iterative and incremental development processes

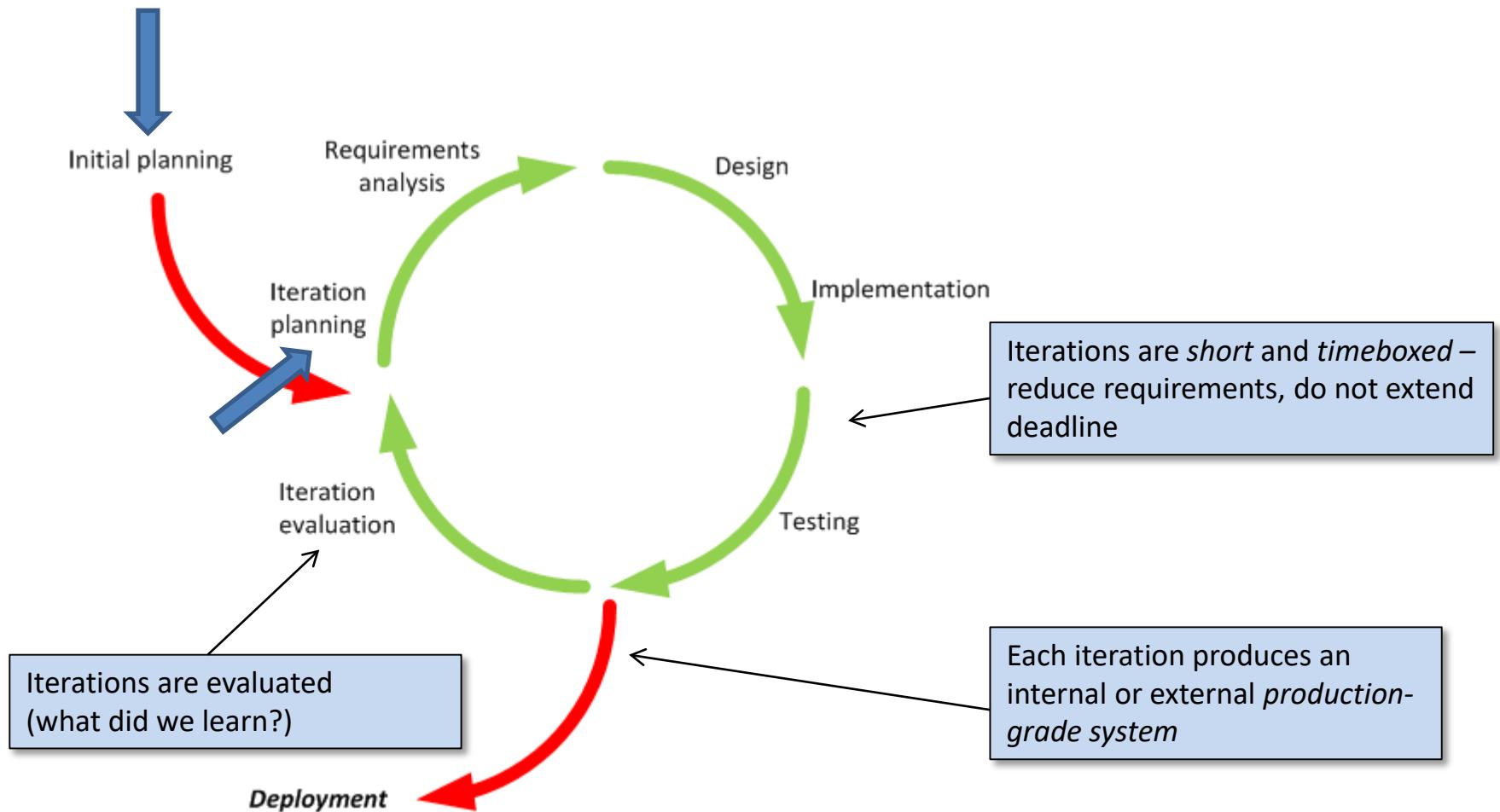
- *Iterative* refers to the repetitive nature of the process
 - An *iteration* is a single repetition of the same sub-process.
 - The sub-process result is a partial working system of *production-quality*
- *Incremental* refers to the *continued expansion* of system capabilities.

Iterative vs. incremental

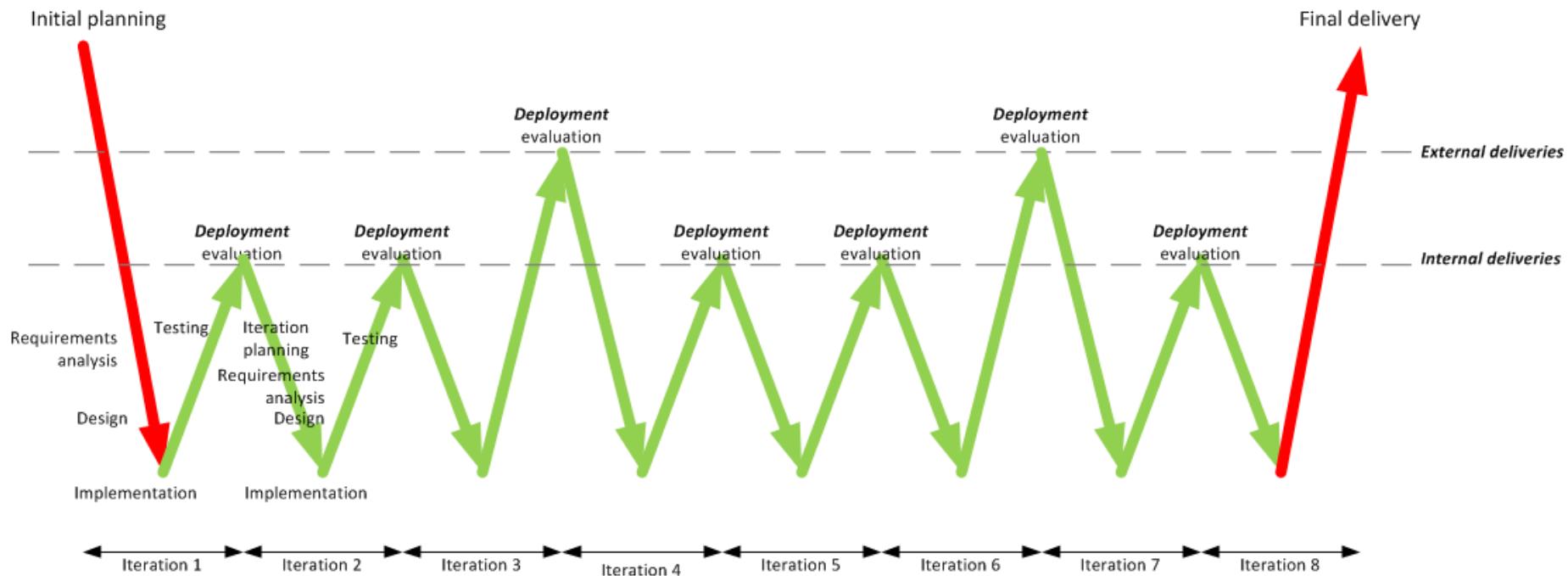
- Iterative *and* incremental



Iterations

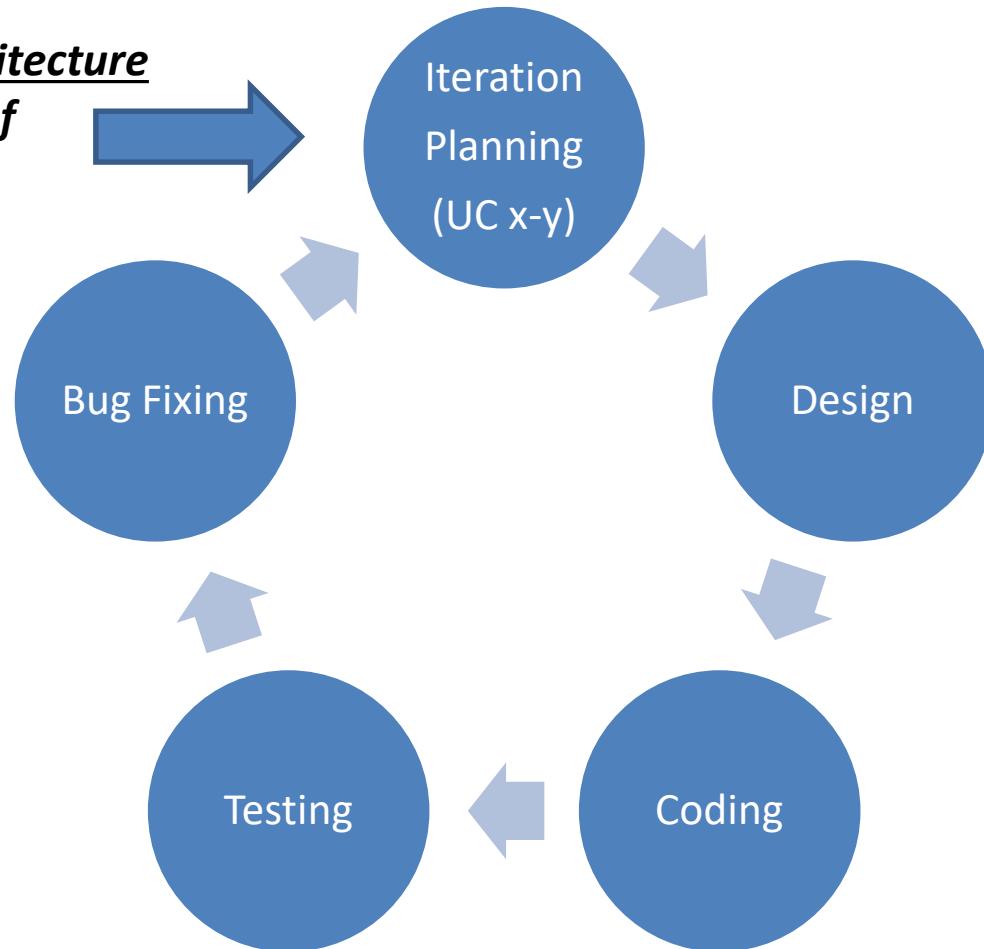


Iterations – another view



Typical SW Iterations

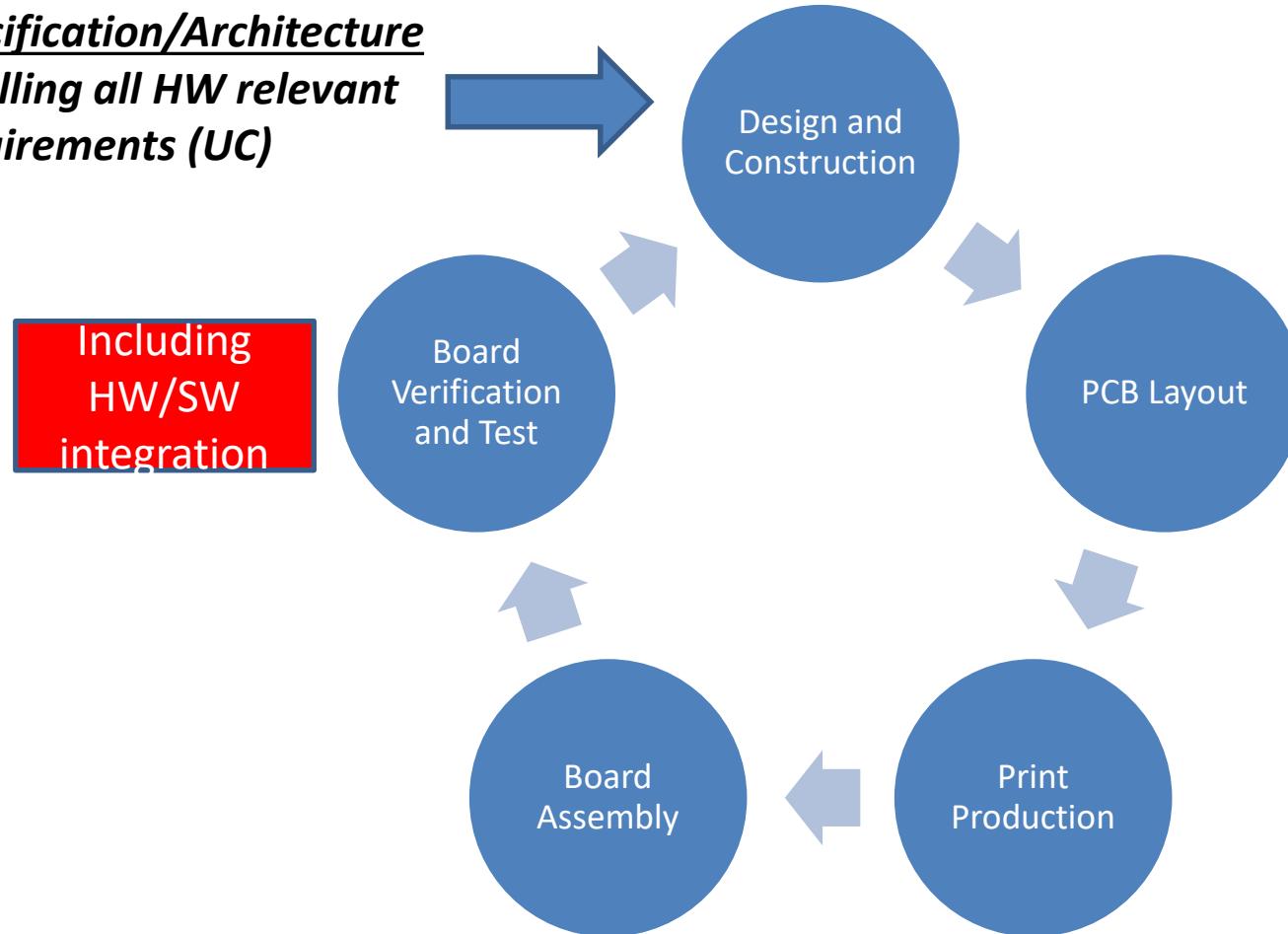
Specification/Architecture
**Selected number of
Use Cases (UC)**



Typical HW Board Spins (Iterations)

Specification/Architecture

Fulfilling all HW relevant requirements (UC)

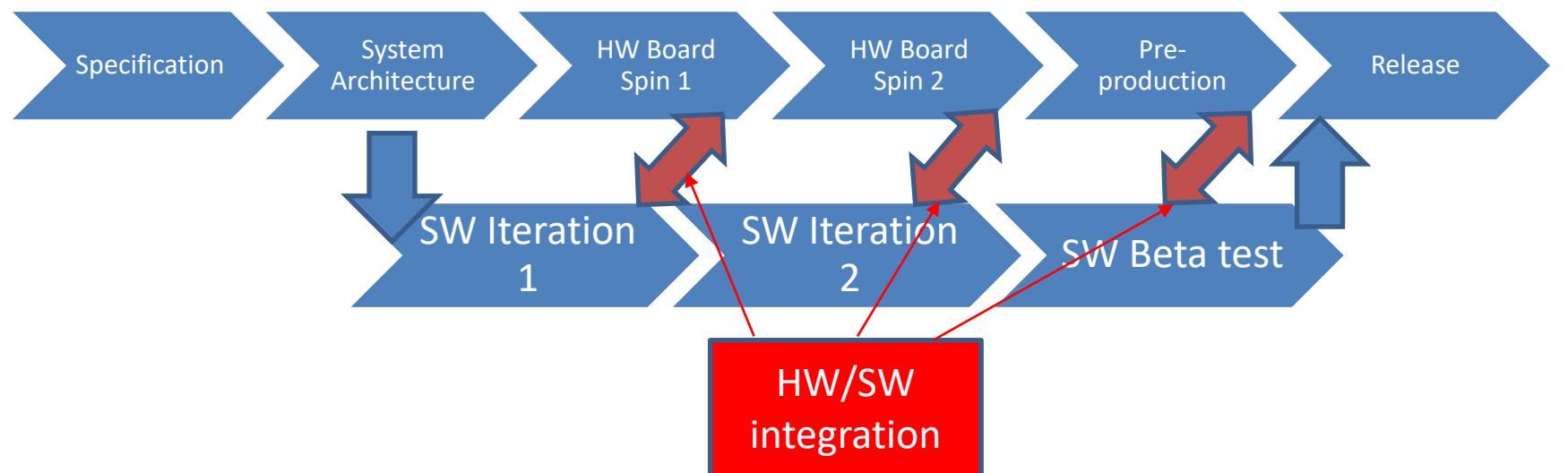


Embedded (HW/SW) Development

Overall Project Plan

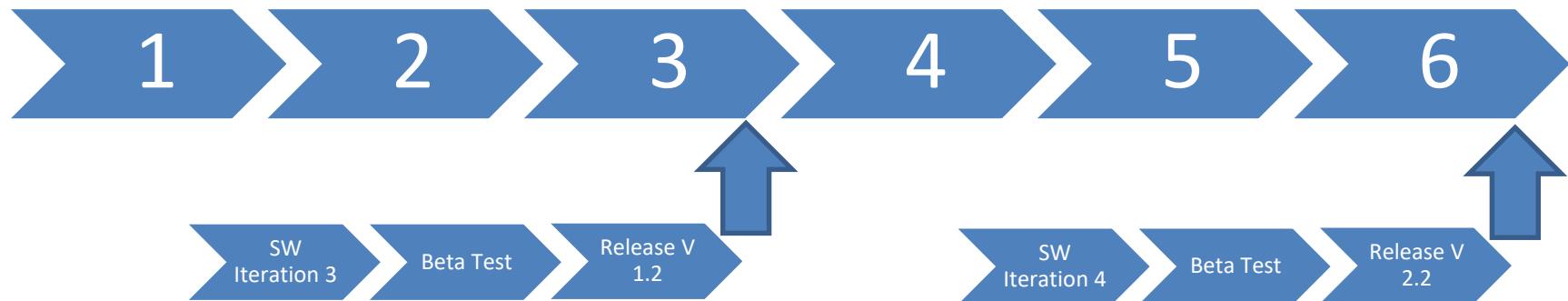
Project start

Project end



Embedded (SW) Development Product Life-time

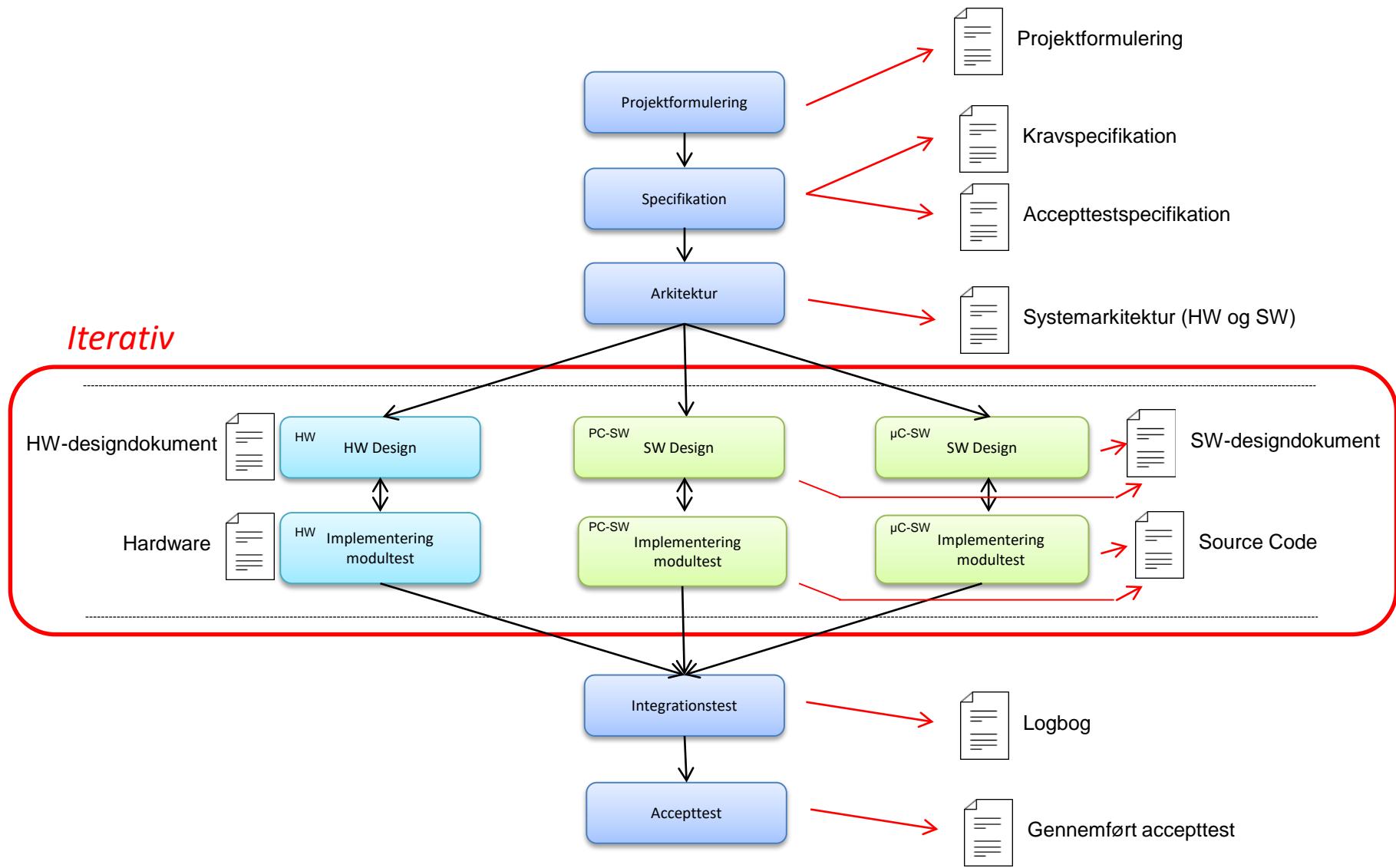
Product life (months)



Semesterprojektmodellen

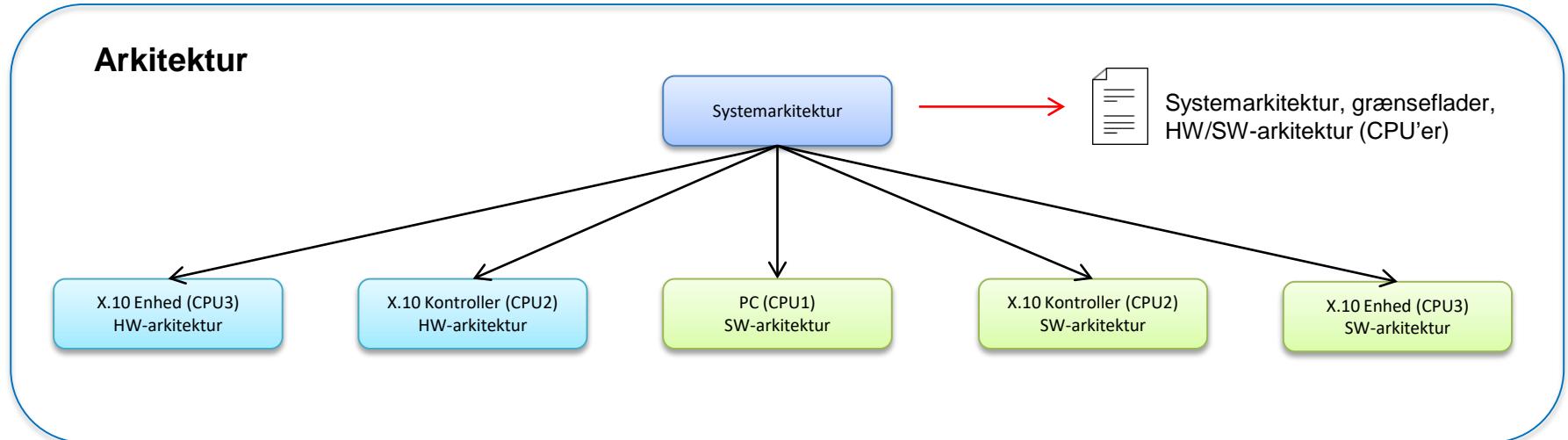
- This is the process you are going to use in your semester project
- A *use case*-driven, "middleweight" *semi-iterative* development process
- Accounts for both *hardware* and *software* development
 - Here the essential architecture needs to be fixed early in the project

Semesterprojektmodellen



Analyse og Design

- Fastlæggelse af systemarkitekturen



Refleksion

- Læs s. 4 – 10 i “Vejledning til udviklingsprocessen for projekt 2” og især arkitektur s. 9 - 10
- Sammenlign semesterprojektmodellen og vandfaldsmodel?
- Hvilke elementer fra V-modellen er også med?
- Hvordan er der indbygget kvalitet i semesterprojektet modellen?
- Hvad målet med arkitekturfasen?
- Hvad skal beskrivelsen af systemarkitekturen indeholde?

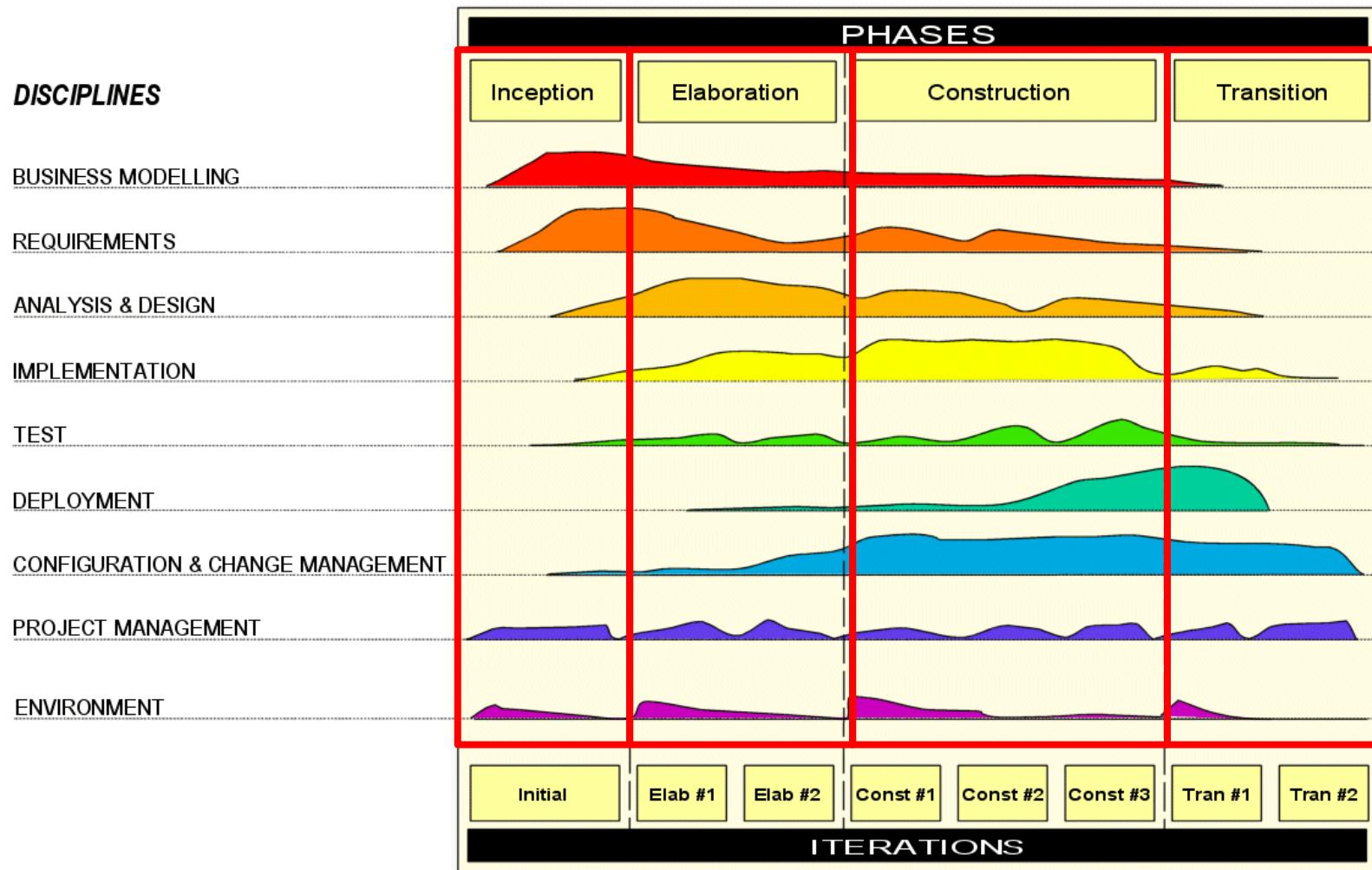
Example: Rational Unified Process

- Rational Unified Process (RUP)
 - Developed by *Rational Software* (now IBM)
 - Developed from the *Unified Process*
Jacobson, Booch, Rumbaugh
- Actually a process *framework* from which processes can be *instantiated*

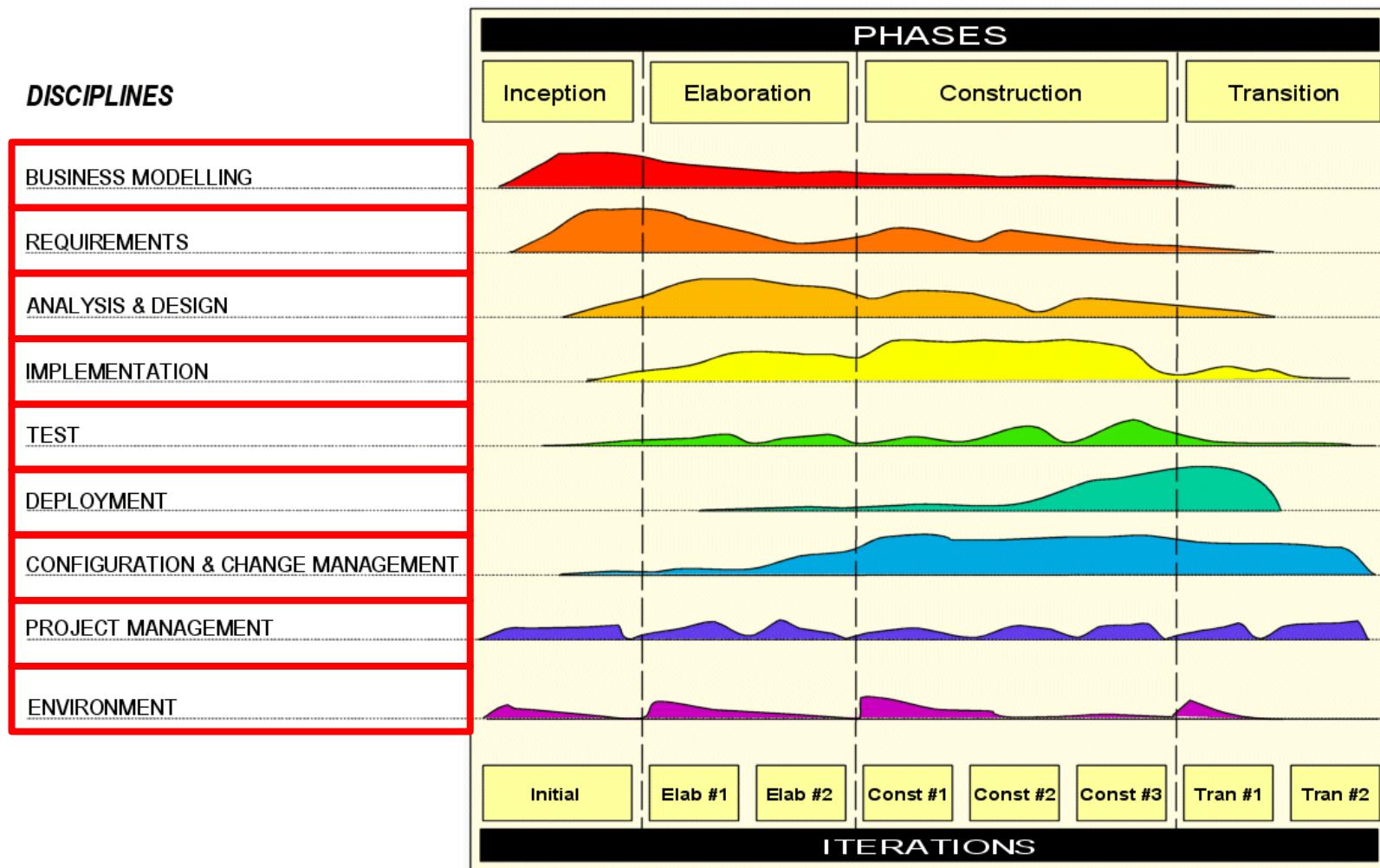
RUP: Phases and disciplines

- RUP defines 4 (sequential) *phases*
 - Inception Understand *what* to build
 - Elaboration Understand *how* to build it
 - Construction Build it
 - Transition Use/sell/ship it
- RUP defines 9 (concurrent) *disciplines*

RUP: Phases

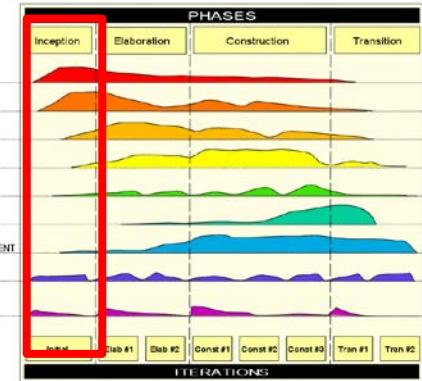


RUP: Disciplines



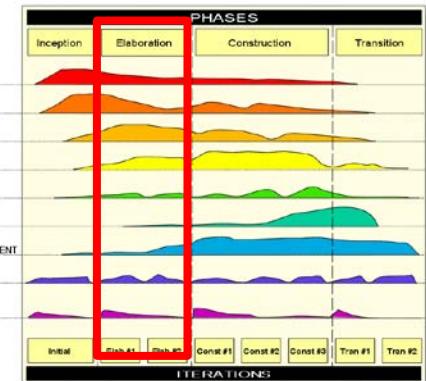
RUP: Inception phase

- Life-cycle objectives of the project are stated, so that the needs of every stakeholder are considered.
- Scope and boundary conditions, acceptance criteria and some requirements are established.
- Activities:
 - Problem description
 - Product limitations
 - Requirements definition (use cases)
 - Acceptance test plan
 - Risk analysis
 - High-level architectural considerations



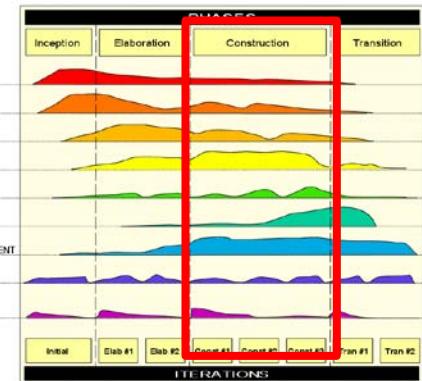
RUP: Elaboration phase

- Determine risks, stability of vision of what the product is to become
- Determine stability of architecture and expenditure of resources
- Activities:
 - Requirements elaboration, prioritization and allocation to Construction iterations
 - Risk mitigation
 - Domain analysis and design
 - HW/SW architectural considerations
 - Interface specifications



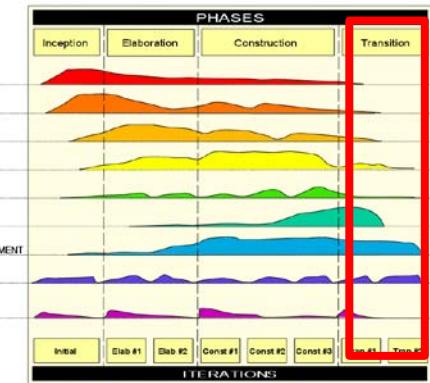
RUP: Construction phase

- Manufacture produce
- Manage risk, resources, etc. to optimize cost, schedule and quality
- Detailed iteration planning and tracking
- Activities:
 - Construction, unit/integration/system tests
 - Per-iteration working system prototype
 - Continuous focus on risk mitigation, planning etc.



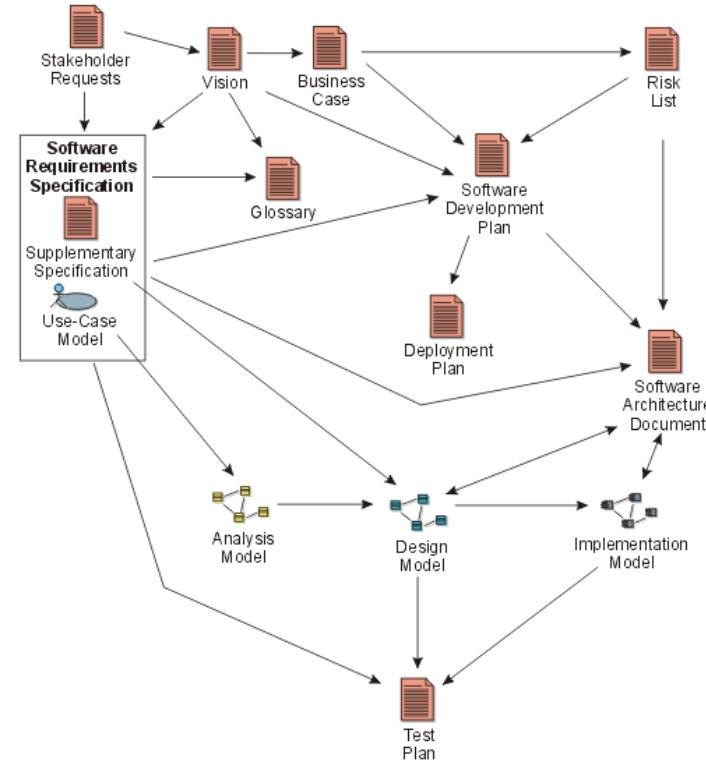
RUP: Transition phase

- Marketing, packaging, installing, configuring
- Supporting user community, making corrections, updates, etc.
- Activities:
 - Acceptance test (alpha/beta test if planned)
 - Corrections, configuration control
 - User education
 - Production tests and documentation
 - Marketing
 - Market implementation



RUP: Artifacts

- RUP defines a lot of *artifacts* associated to the disciplines
 - *Documents*
 - *Models* (or *model elements*) with associated *reports*
- Is RUP a “light” or “heavy” process?



What's the problem?

Disciplined execution
kills innovation

Innovation requires
"no discipline"



*Disciplined
execution*

Plans, deadlines,
documents



*Continuous
innovation*

Open environment,
no "management"

Agile methods

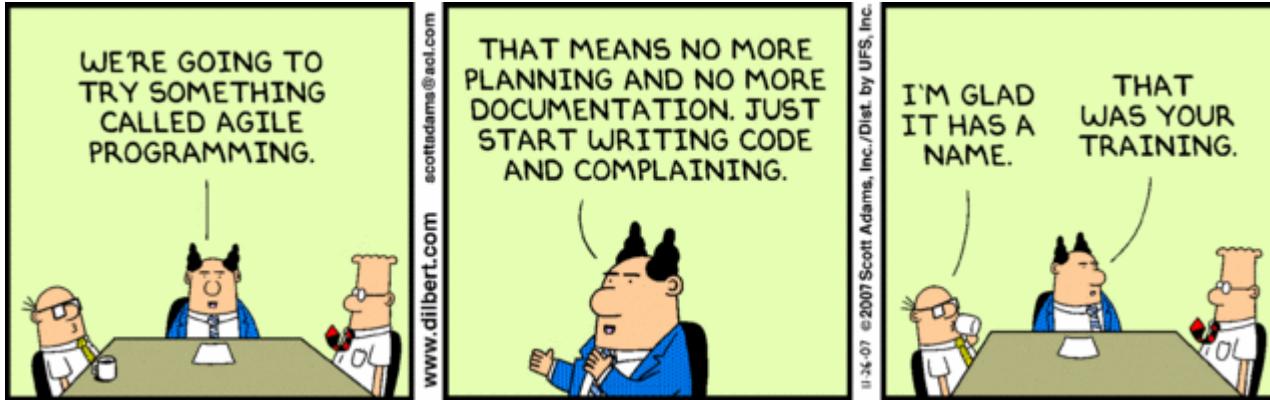
- Agile: *adræt, rapfodet, fleksibel, agil*
- Claiming that the traditional processes are fundamentally flawed and that many iterative processes are heavy, *agile* processes emerged in the 1990's.
- Defined in the *agile manifesto* in 2001 by 17 signatories.
- The agile "bottom line": Faith in *people* rather than *paper*
- ***Mostly used for pure software development***

Agile methods: The agile manifesto

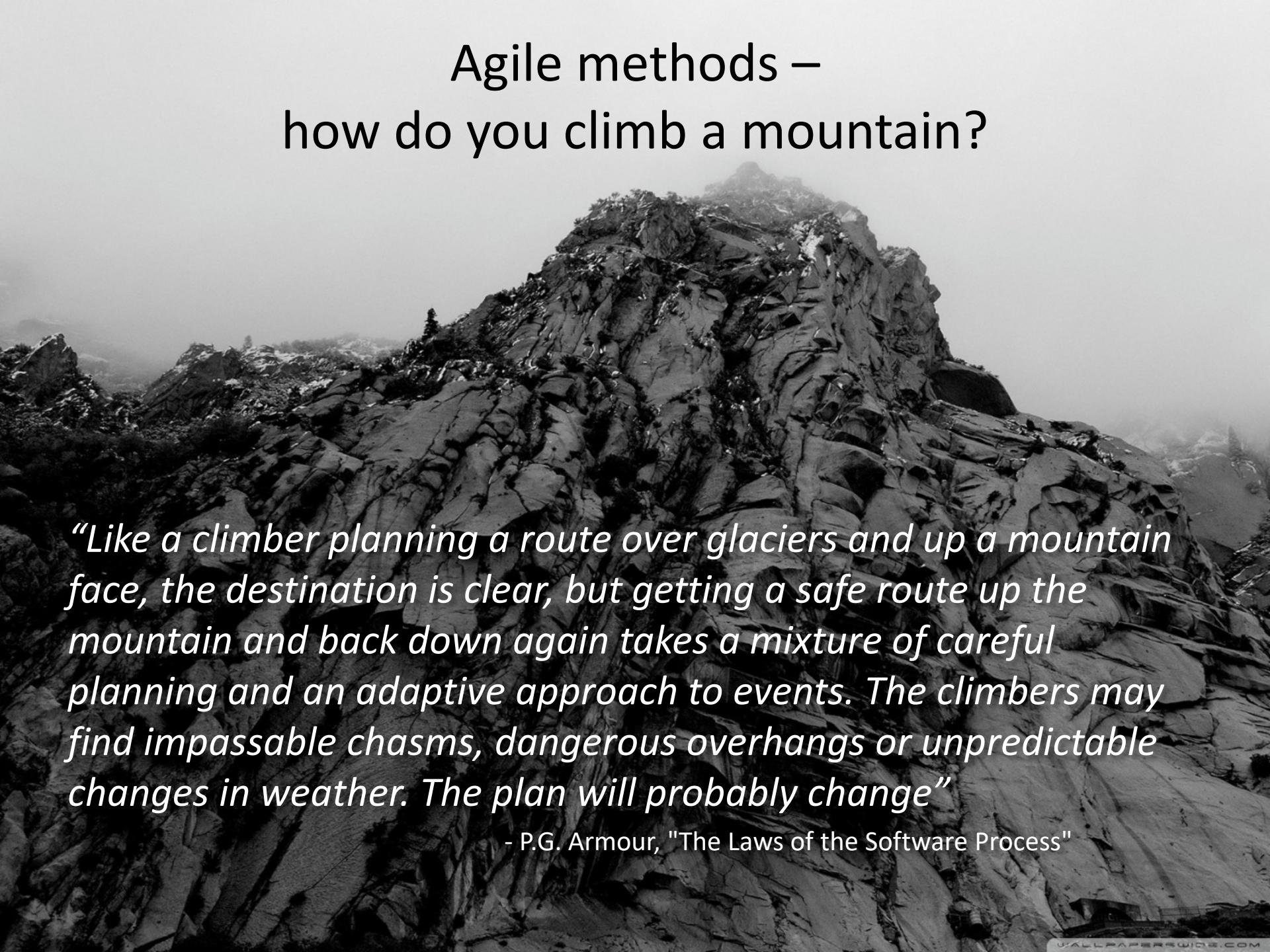
- **Individuals and interactions** over **processes and tools**
- **Working software** over **comprehensive documentation**
- **Customer collaboration** over **contract negotiation**
- **Responding to change** over **following a plan**

That is, while there is value in the items on the right, we value the items on the left more.

The danger of Agile methods



Agile methods – how do you climb a mountain?



"Like a climber planning a route over glaciers and up a mountain face, the destination is clear, but getting a safe route up the mountain and back down again takes a mixture of careful planning and an adaptive approach to events. The climbers may find impassable chasms, dangerous overhangs or unpredictable changes in weather. The plan will probably change"

- P.G. Armour, "The Laws of the Software Process"

Agile methods: Some of the 12 agile principles

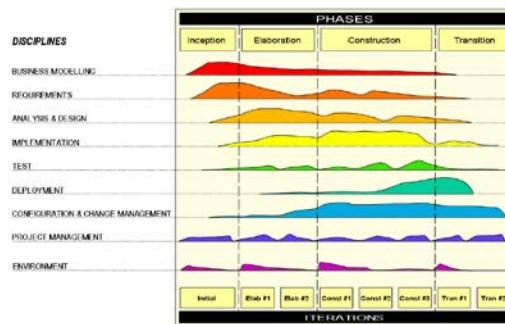
- Satisfy the customer through early and continuous delivery of valuable software
- Welcome changing requirements, even late in development
- Business people and developers must work together daily throughout the project
- Working software is the primary measure of progress
- Simplicity – the art of maximizing the amount of work not done – is essential

Example: eXtreme Programming (XP) (Software)

- Developed by Beck, Cunningham and others
 - First coined in 1996
- Some characteristics:
 - Focus on *customer satisfaction*
 - Permanent on-site *customer presence*
 - Short development cycles
 - Incremental planning
 - Continuous feedback
 - Evolutionary design
 - Pair programming

Discussion

- Imagine you are the *developer* in a team. What would make you feel more comfortable – RUP or an agile process? Why?
- Now imagine you are the *customer*. What would make you feel more comfortable – RUP or an agile process? Why?
- Do you think it is *easier* to work in an agile project than in a RUP project?



The state of things

- The days of the standard process are over...
- The most commonly seen process today is tailored to meet the business needs
- Usually the process will be highly iterative with selected agile elements (team, iterations, customer involvement, etc.)
- Usually, it will be managed by Scrum (which we'll learn about later)

Scrum

Introduction to Systems Engineering
I2ISE



Agile udvikling - Scrum



Starter kl. 14:15

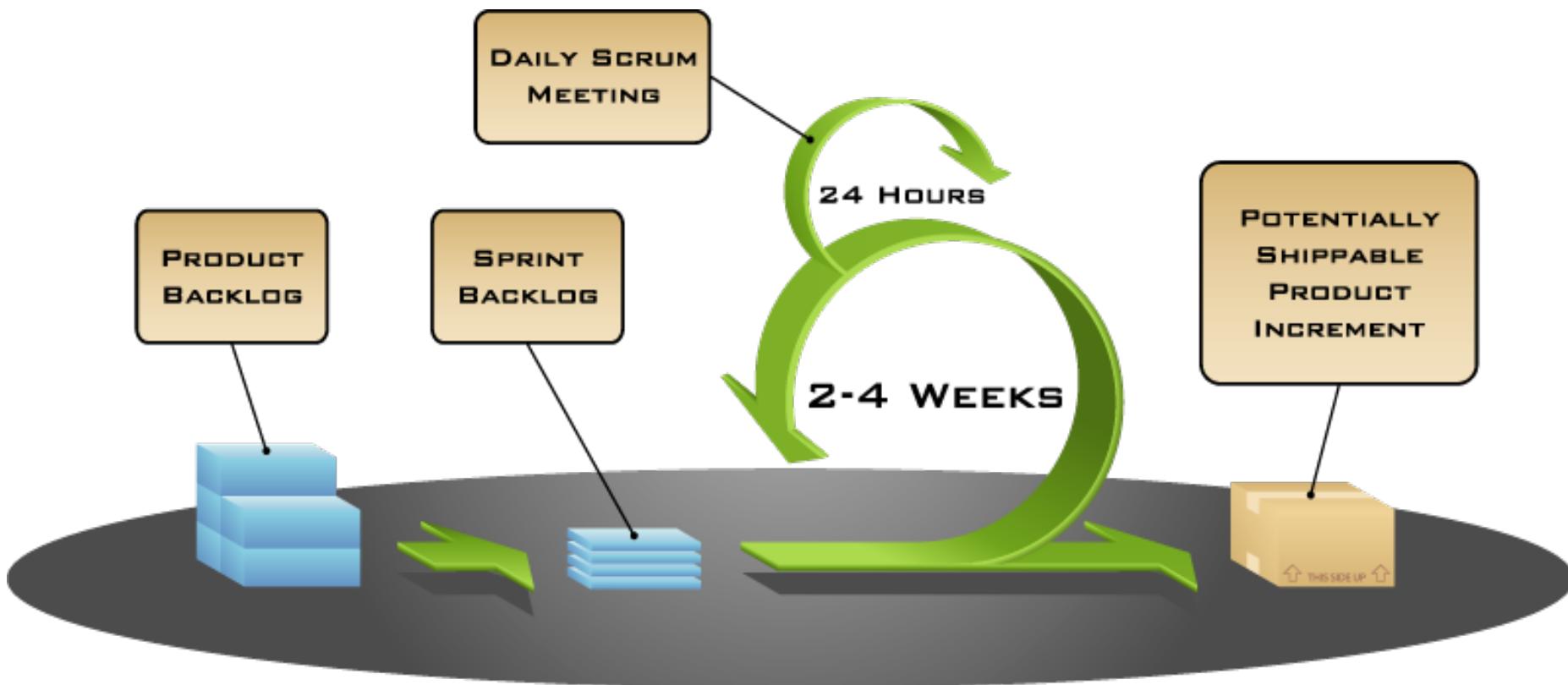
Project Management - Scrum

- Scrum is an iterative, incremental framework for project management
- Involve the team in planning, insulate them from changes during *sprints*
- Following slides from
www.mountaingoatsoftware.com

Scrum characteristics

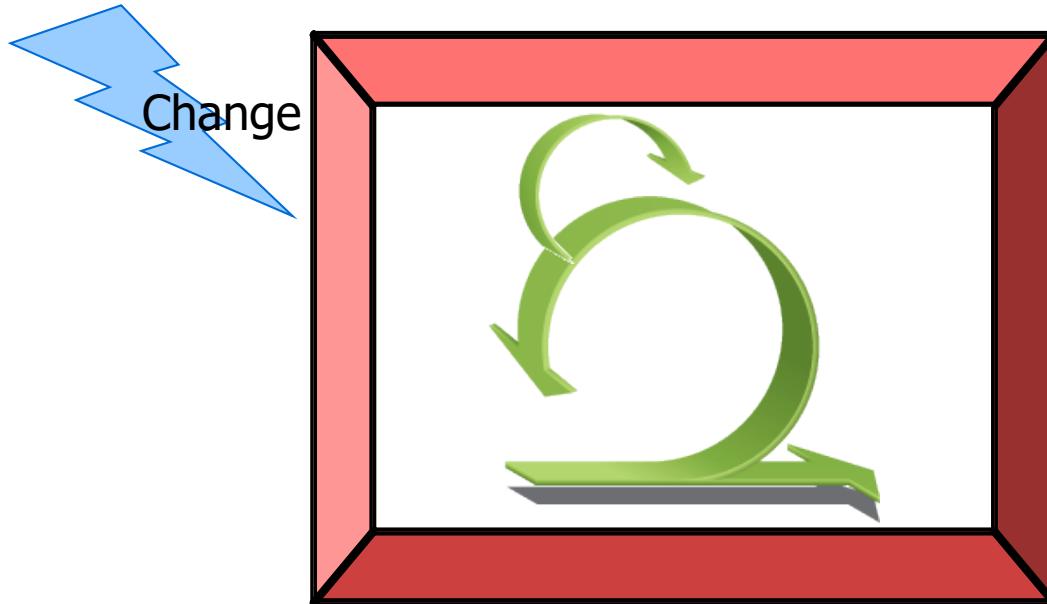
- Self-organizing teams
- Product progresses in a series of month-long “sprints”
- Requirements captured in a “product backlog”
- No specific engineering practices prescribed
- Needs an agile environment for delivering projects, don’t forget that (not like Dilbert’s boss)!
- One of the “agile processes”

Putting it all together



COPYRIGHT © 2005, MOUNTAIN GOAT SOFTWARE

No changes during a sprint



- Sprint durations are planned around how long you can commit to keeping change out of the sprint!

Scrum framework

Roles

- Product owner
- ScrumMaster
- Team

Ceremonies

- Sprint planning
- Sprint review
- Sprint retrospective
- Daily scrum meeting

Artifacts

- Product backlog
- Sprint backlog
- Burndown charts

Scrum framework

Roles

- Product owner
- ScrumMaster
- Team

Ceremonies

- Sprint planning
- Sprint review
- Sprint retrospective
- Daily scrum meeting

Artifacts

- Product backlog
- Sprint backlog
- Burndown charts

Roles - Product owner



- Defines the features of the product
- Decides on release dates and content
- Responsible for the profitability of the product
- Prioritizes features according to market value
- Adjusts features and priority every iteration, as needed
- Accepts or rejects work results

Roles – Scrum Master



- Represents management to the project
- Responsible for enacting Scrum values and practices
- Removes impediments
- Ensures that the team is fully functional and productive
- Enables close cooperation across all roles and functions
- Shields the team from external interferences

Roles - The team



- Typically 5-9 people
- Cross-functional:
 - Programmers, testers, user experience designers, etc.
- Members are (ideally) full-time
- Teams are (ideally) self-organizing
- Membership should (ideally) change only *between* sprints

Scrum framework

Roles

- Product owner
- ScrumMaster
- Team

Ceremonies

- Sprint planning
- Sprint review
- Sprint retrospective
- Daily scrum meeting

Artifacts

- Product backlog
- Sprint backlog
- Burndown charts

Artifacts - product backlog

- The requirements
- A list of all desired work on the project
- Ideally expressed such that each item has value to the users or customers of the product
- Prioritized by the product owner
- Reprioritized at the start of each sprint



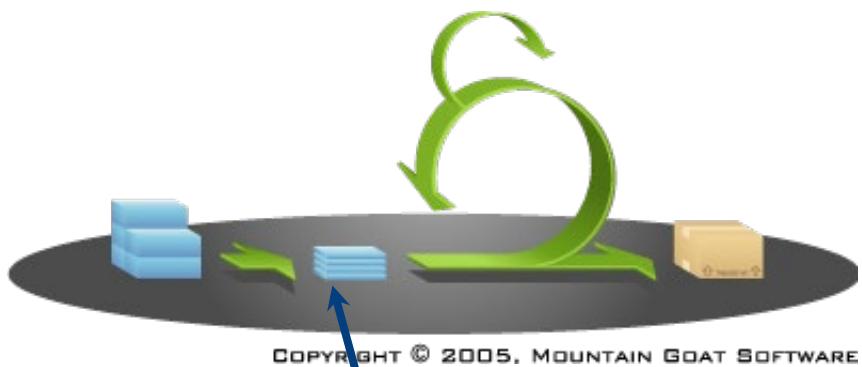
This is the
product backlog

Artifacts - a sample product backlog

Backlog item	Estimate
Allow a guest to make a reservation	3
As a guest, I want to cancel a reservation.	5
As a guest, I want to change the dates of a reservation.	3
As a hotel employee, I can run RevPAR reports (revenue-per-available-room)	8
Improve exception handling	8
...	30
...	50

Artifacts - sprint backlog

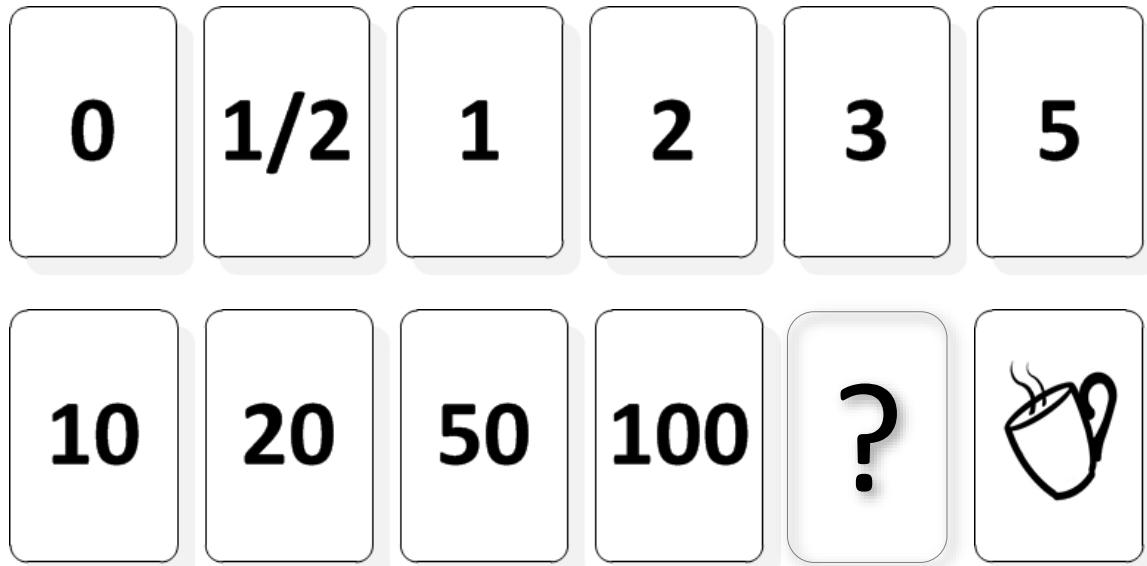
- The requirements for this sprint, related to *sprint goal*
- Individuals sign up for work of their own choosing
 - Work is never *assigned*
- Estimated work remaining is updated daily
 - Drives sprint *burndown chart*



This is the
sprint backlog

Planning – planning poker

- Another estimation technique: *Planning poker!* You need:
 - A deck of cards per team member
 - hours, days, or ideal days
 - Question mark (*cannot estimate – defer*)
 - Coffee cup (*I need a break!*)
 - An egg timer to structure discussion



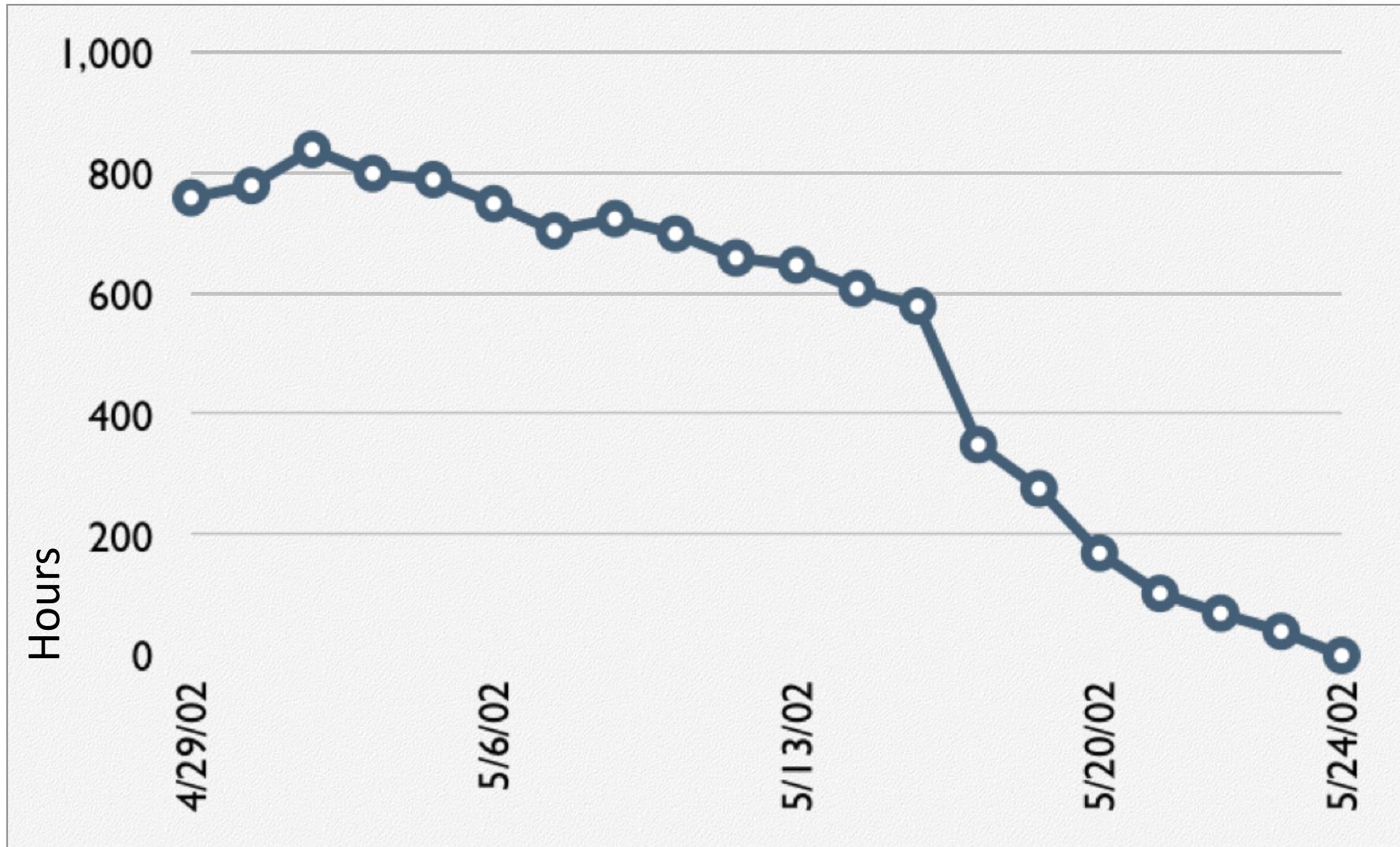
Planning – planning poker

- *Product owner* (e.g. PM) gives short introduction to tasks contents. *Team members* discuss task – **no numbers!**
- Each *team member* lays a planning card face down representing his/her estimate of the task
- Everybody calls at the same time
- Team members with low/high estimates offered a *soapbox* to explain estimates.
- Discussion continues
- *Moderator* or *Product Owner* may at any time set the egg timer
- Egg timer rings → discussion stops, new estimate
- Estimation process repeated until consensus is “reached”

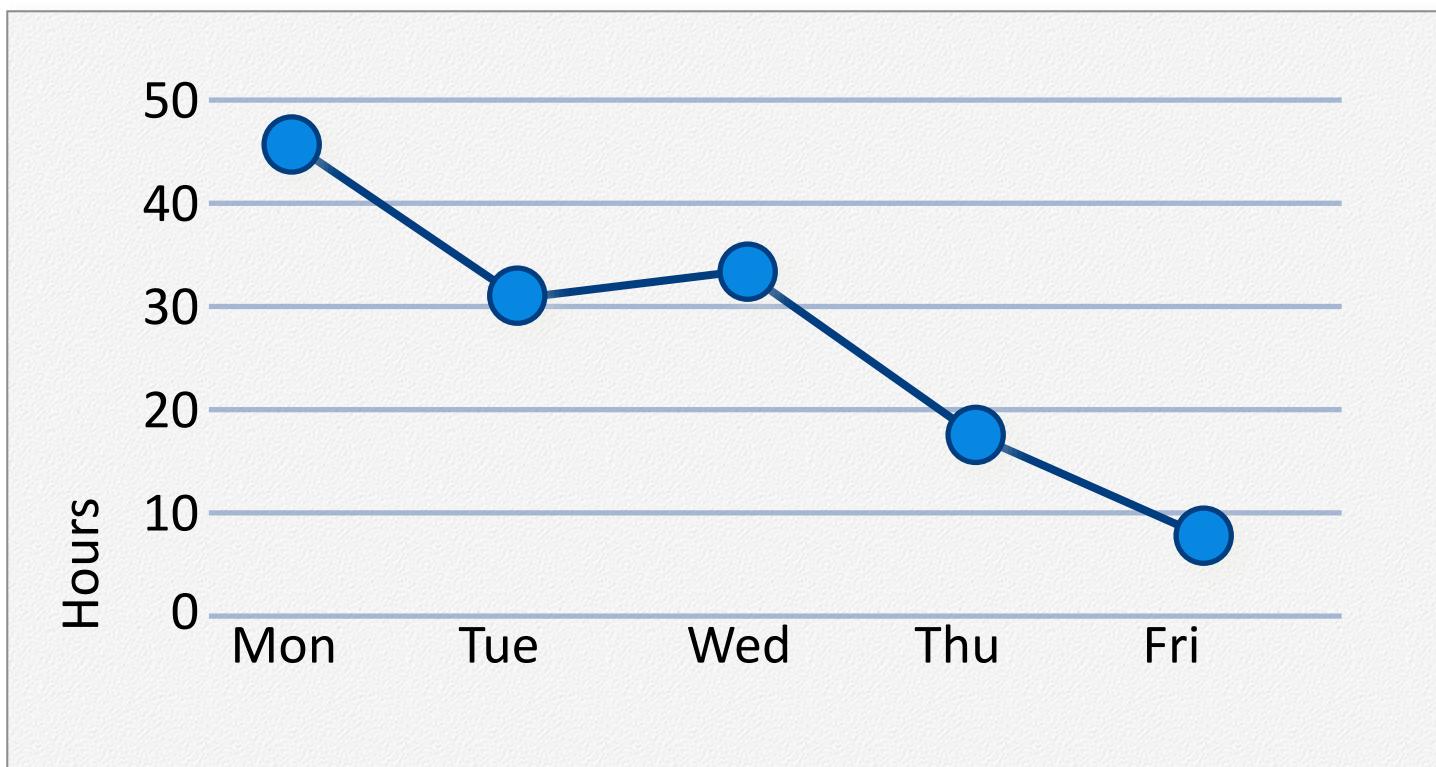
An example sprint backlog

Tasks	Mon	Tues	Wed	Thur	Fri
Code the user interface	8	4	8		
Code the middle tier	16	12	10	4	
Test the middle tier	8	16	16	11	8
Write online help	12				
Write the foo class	8	8	8	8	8
Add error logging			8	4	

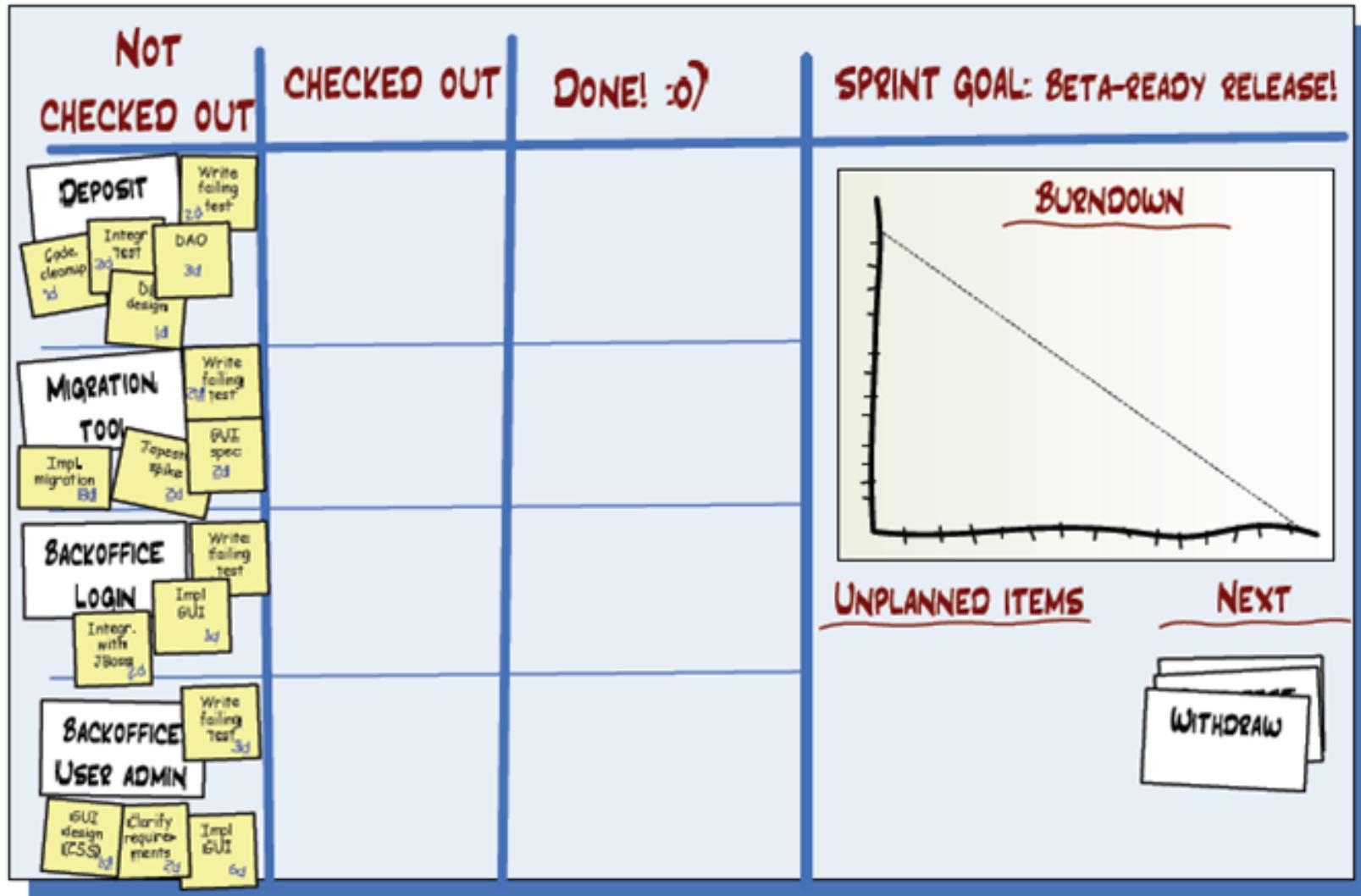
A sprint burndown chart



Tasks	Mon	Tues	Wed	Thur	Fri
Code the user interface	8	4	8		
Code the middle tier	16	12	10	7	
Test the middle tier	8	16	16	11	8
Write online help	12				



Artifacts – The Scrum Board



Scrum framework

Roles

- Product owner
- ScrumMaster
- Team

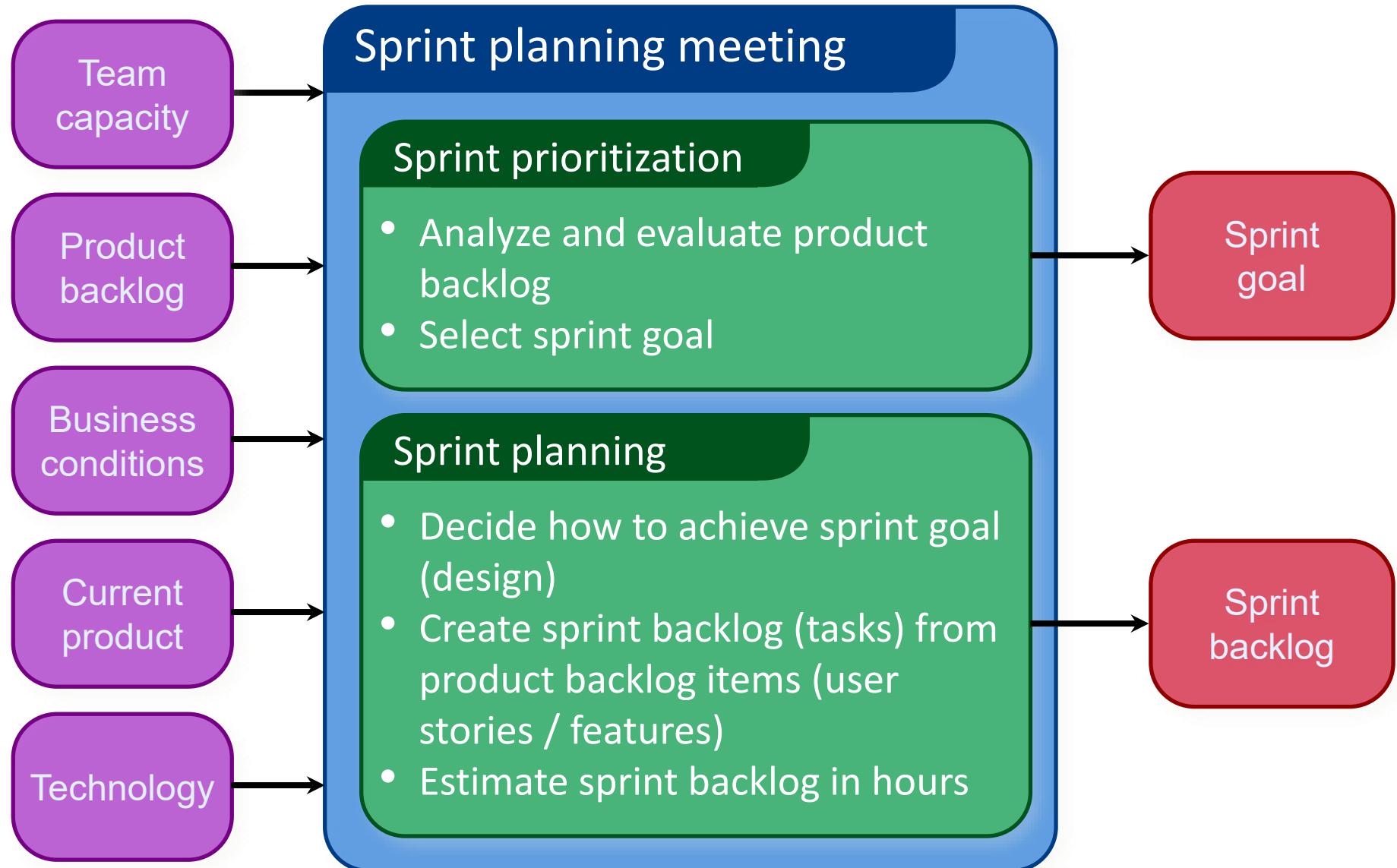
Ceremonies

- Sprint planning
- Sprint review
- Sprint retrospective
- Daily scrum meeting

Artifacts

- Product backlog
- Sprint backlog
- Burndown charts

Ceremonies - Sprint planning



Sprint planning - The sprint goal

- A short statement of what the work will be focused on during the sprint

Database Application

Make the application run on SQL Server in addition to Oracle.

Life Sciences

Support features necessary for population genetics studies.

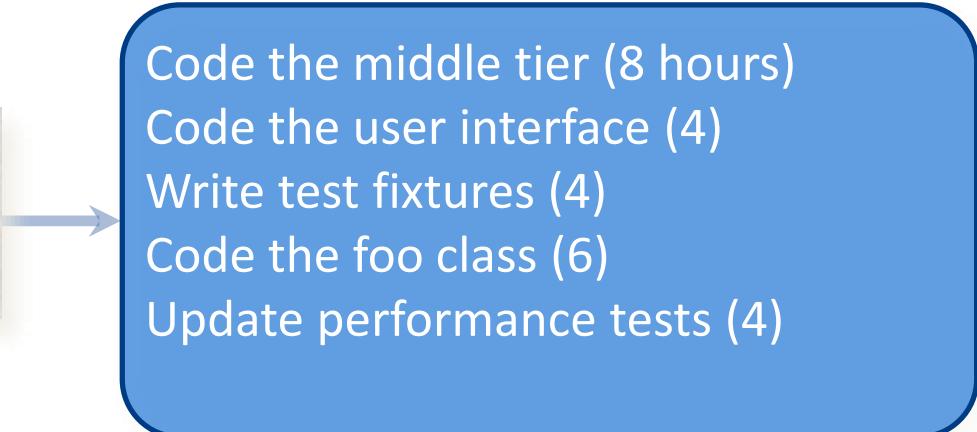
Financial services

Support more technical indicators than company ABC with real-time, streaming data.

Ceremonies - Sprint planning

- Team selects items from the product backlog they can commit to completing in the upcoming sprint
- Sprint backlog is created
 - Tasks are identified and each is estimated
 - Collaboratively, not done alone by the ScrumMaster
- High-level design is considered

As a vacation planner, I want to see photos of the hotels.



Ceremonies - daily scrum

- Parameters
 - Daily
 - 15-minutes
 - Stand-up
- *Not* for problem solving
 - Whole world is invited
 - Only team members, ScrumMaster, product owner, can talk
- Helps avoid other unnecessary meetings



Daily scrum – answer 3 questions

1

What did you do yesterday?

2

What will you do today?

3

Is anything in your way?

- These are **not** status for the ScrumMaster
 - They are commitments in front of peers

Ceremonies - The sprint review

- Team presents what it accomplished during the sprint
- Typically takes the form of a demo of new features or underlying architecture
- Informal
 - 2-hour prep time rule
 - No slides
- Whole team participates
- Invite the world



Ceremonies– Sprint retrospective

- Periodically take a look at what is and is not working
- Typically 15–30 minutes
- Done after every sprint
- Whole team participates
 - Scrum Master
 - Product owner
 - Team
 - Possibly customers and others

Sprint retrospective - Start / Stop / Continue

- Whole team gathers and discusses what they'd like to:

Start doing

Stop doing

This is just one
of many ways to
do a sprint
retrospective.

Continue doing

Scrum Øvelse

Opret en produkt backlog og planlæg et sprint for jeres semesterprojekt

- Opret en product backlog (15 min)
- Planlæg et sprint (15 min)

