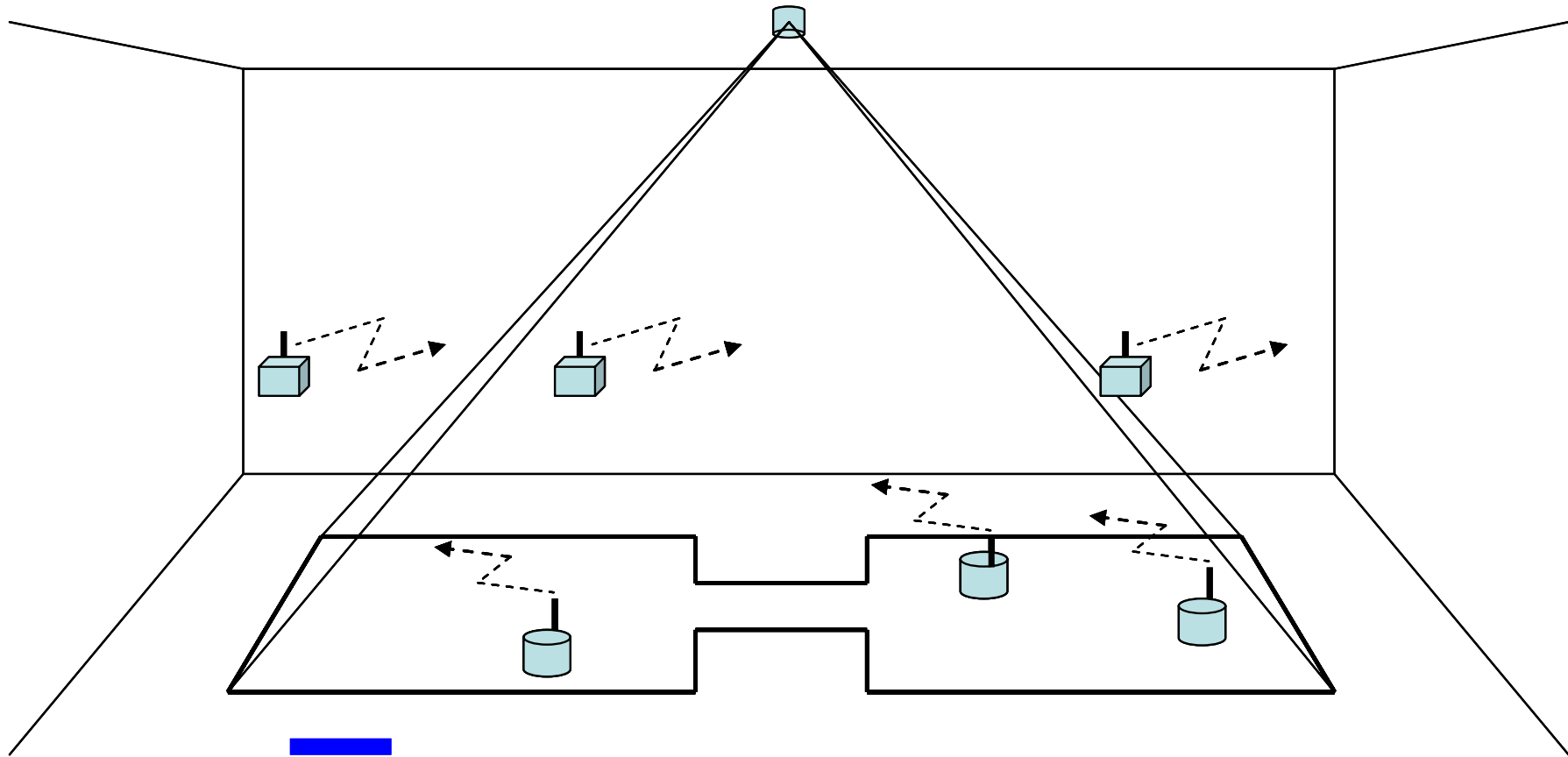


# Design of Embedded and Intelligent Systems (15 Credits)

- Course Responsible:
  - Tony Larsson
- Examiner: Tony Larsson
- Project Support:
  - Tommy Salomonsson
- Other Faculty Involved in Lectures and Labs
  - Nicholas W, Ulf H, Björn Å, Antanas V, Urban B, Jens L, Anita S, Joseph B, ...
- Course Web Page

# Project

## Autonomous vehicles for transports in limited spaces

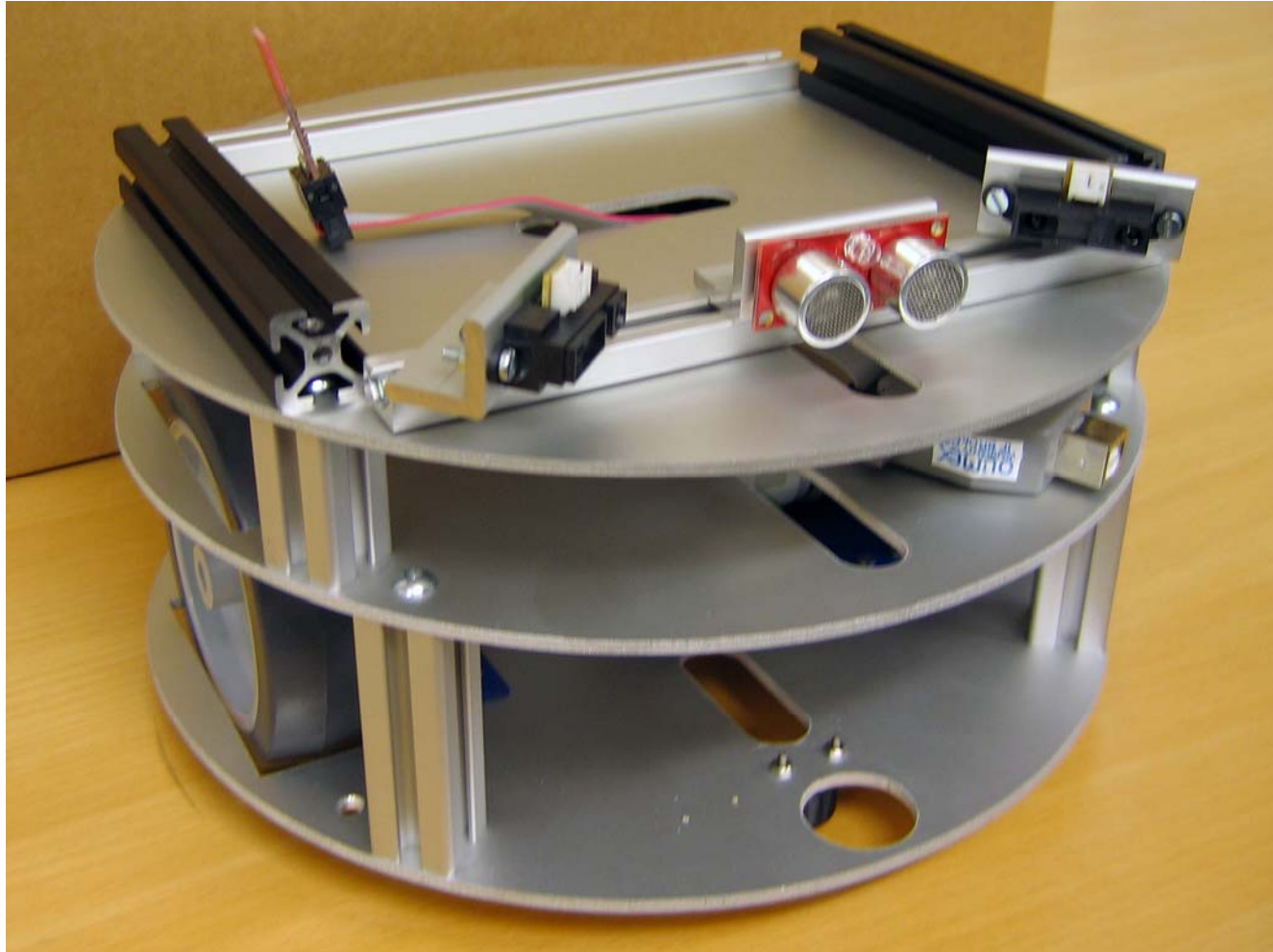


**Project playground consisting of two rooms with a passage in between; where the room borders, the vehicles, the base stations and a shared camera placed in the roof of the lab room are illustrated.**

# Praxis in your Project Task

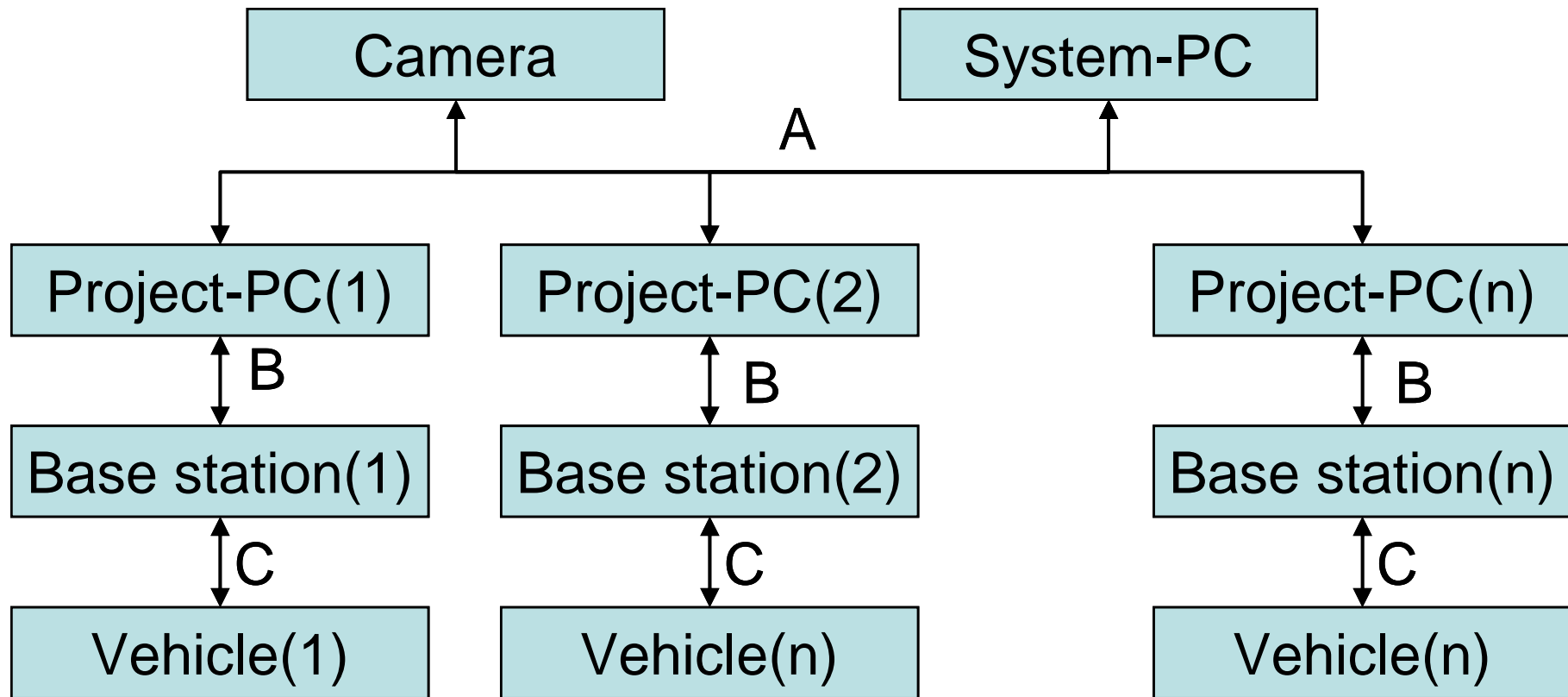
- What does this mean for you, for example:
  - Get image
  - Analyse image
  - Identify position
  - Make a plan
  - Set direction
  - Set speed
  - Get feedback
  - Communicate control commands and feedback
  - Think about effect of delay

# The PIE





# System Block Structure

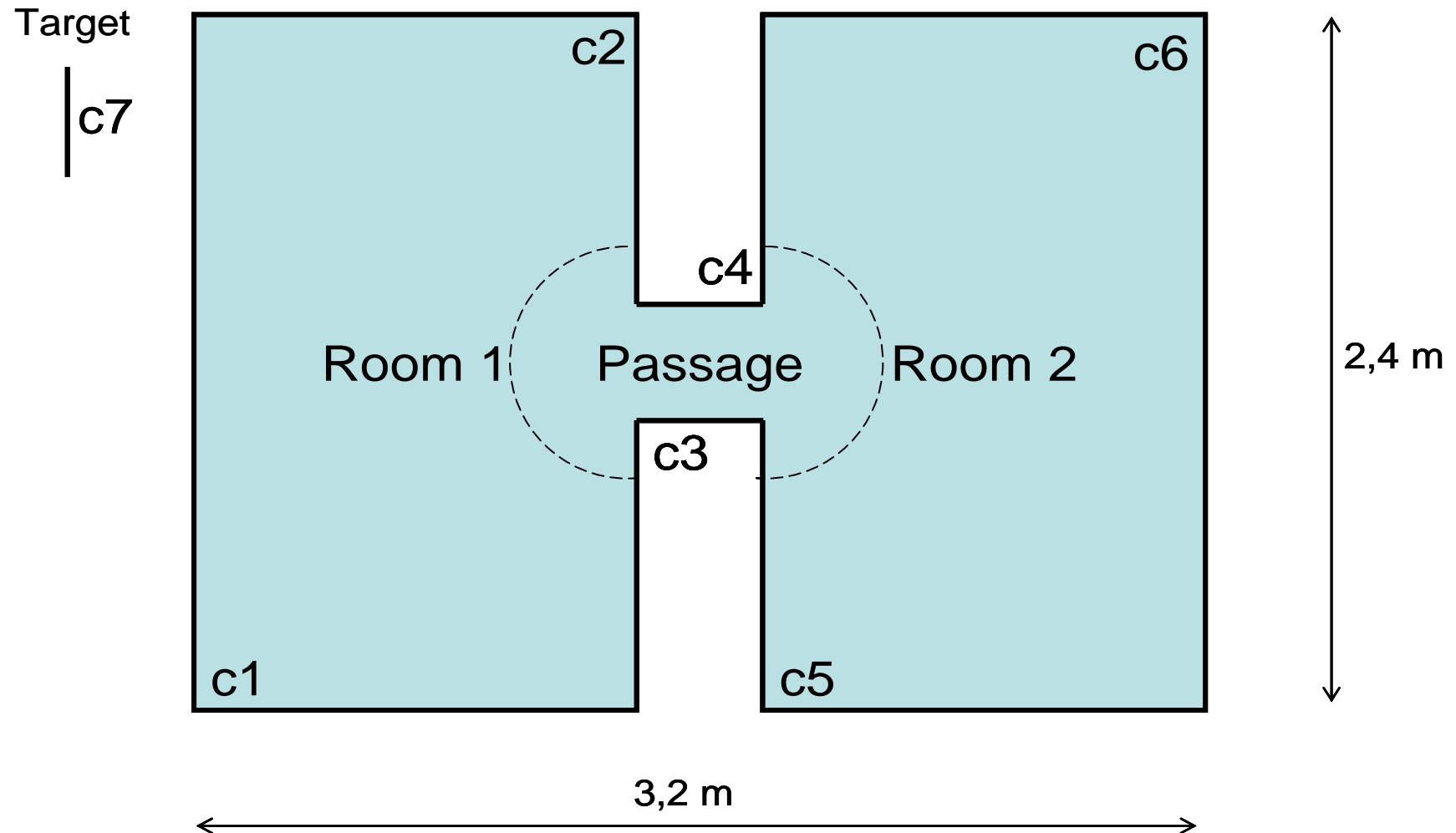


A = Ethernet

B = RS 232

C = 2,4 GHz Radio link

# Playground



Size of the playground (measured in pixels): c1 = <0, 0>, c2 = <700, 1200>, c3 = <700, 500>, c4 = <1100, 700>, c5 = <1100, 0>, c6 = <1600, 1200>  
c7 = defined at examination

# Project Tollgate Deliveries

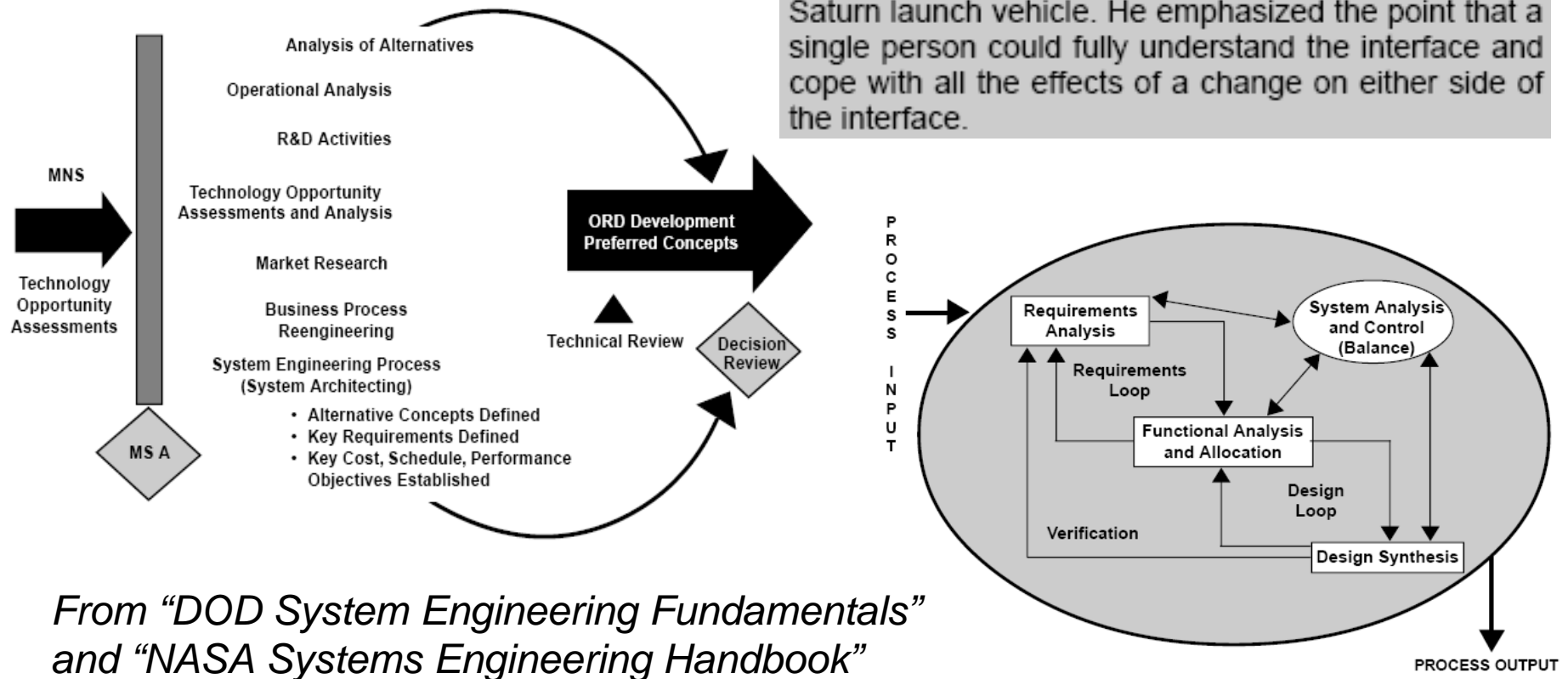
**Oral presentations, written documentations and demonstrations are part of the grade given.**

- **EX-1 (w38):** Refined definition of the problem and important limitations documented in a system requirement specification defining the systems functional and characteristics requirements.
- **EX-2 (w42):** System architecture description including component and task partitioning with requirements on each component/task. Common precedence and collision avoidance rules defined and agreed upon by all groups and documented in the description.
- **EX-3 (w46):** Proposed, implemented and tested component and task level solutions as well as arguments for different design trade-offs documented in written report.
- **EX-4 (w50):** System integration, system test report and preliminary demonstration of the system.
  - a) A single vehicle can be driven between several positions.
  - b) The vehicle can avoid static hindes placed on the play ground by the examiner.
- **EX-5 (w02):** Final test demonstration and examination including all groups and their vehicles, followed up by discussions and reflections about achieved experiences and results. The vehicle can be driven between several positions on the play ground together with the other vehicles driving simultaneously and avoid collisions with these in an intelligent way.

# System Engineering

## Simple Interfaces are Preferred

According to Morris, NASA's former Acting Administrator George Low, in a 1971 paper titled "What Made *Apollo* a Success," noted that only 100 wires were needed to link the *Apollo* spacecraft to the Saturn launch vehicle. He emphasized the point that a single person could fully understand the interface and cope with all the effects of a change on either side of the interface.

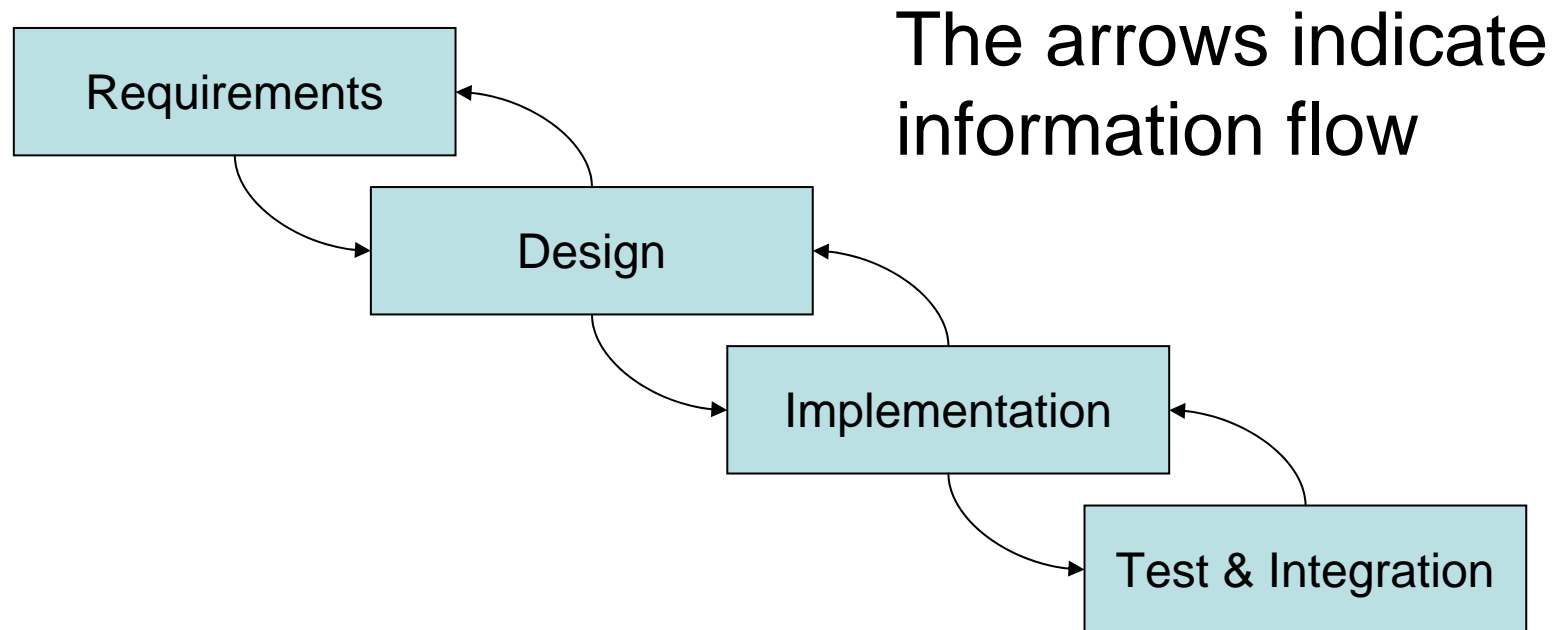


*From "DOD System Engineering Fundamentals" and "NASA Systems Engineering Handbook"*

# System Development Process

- The development of a system can be seen as a process.
- This process can be divided in sub-processes, also called phases or steps.
- Such a step is typically related to one specific work activity in the process.
- Each step needs input and produces some output as result.
- Most of the input and output is information, for example documents, drawings or code.

# The “Waterfall” model



Waterfall model, with some concurrency or overlap between the development steps/phases

# System Development Phases

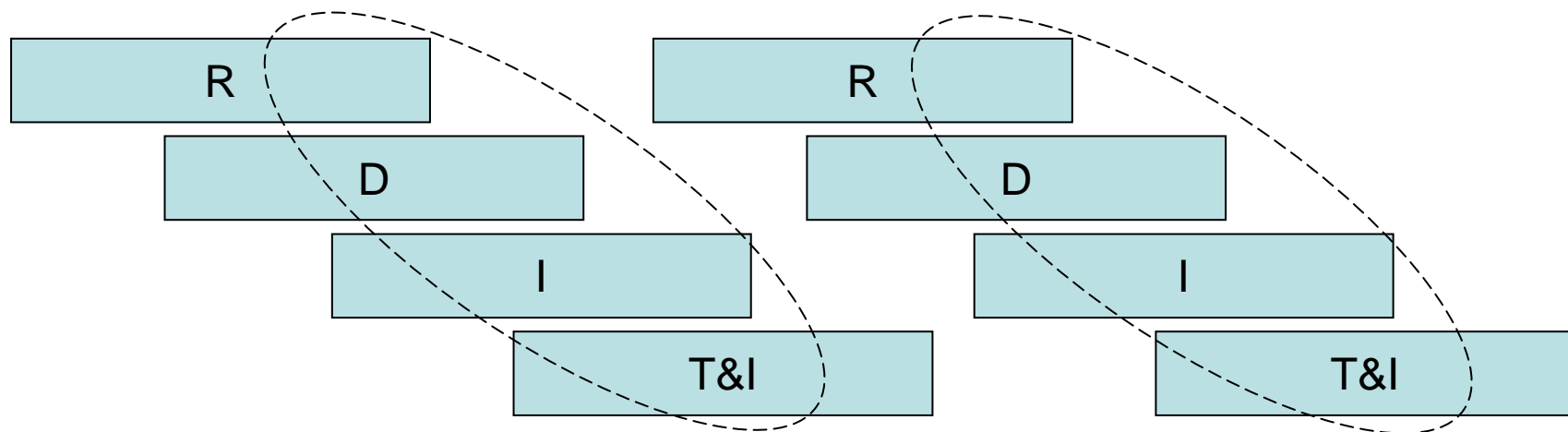
- Requirements (goals, needs and limitations)
- Design (concepts, functions and properties)
- Implementation (modules and connections)
- Test & Integration
- Maintenance (typically more than one phase)

# Design Mappings

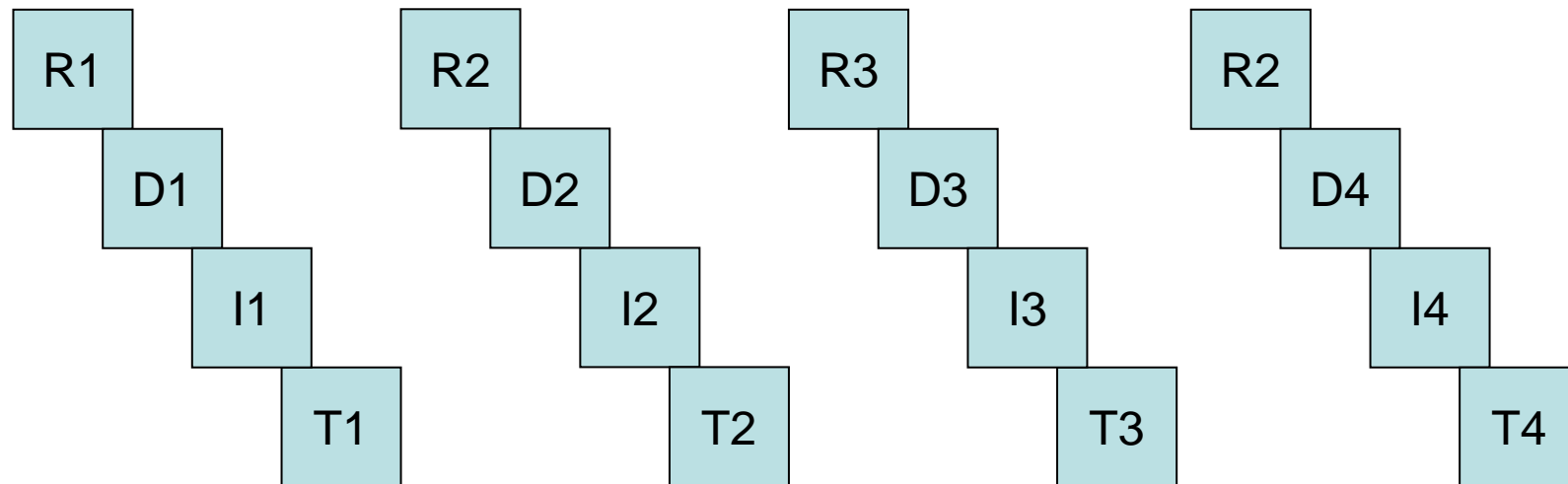
- From requirements (on system) to its functions (services or tasks)
- From requirements to properties
- From functions and properties to subsystems (modules) encapsulating solutions
  
- Repeat the above by identifying, analysing and refining requirements and functions to be provided by subsystems/modules

# Concurrent Development

- In a concurrent development process some work must be made on speculation
- Work may later be “undone or replaced” by other concurrently developed results
- Concurrent work processes can save calendar time but is usually more costly

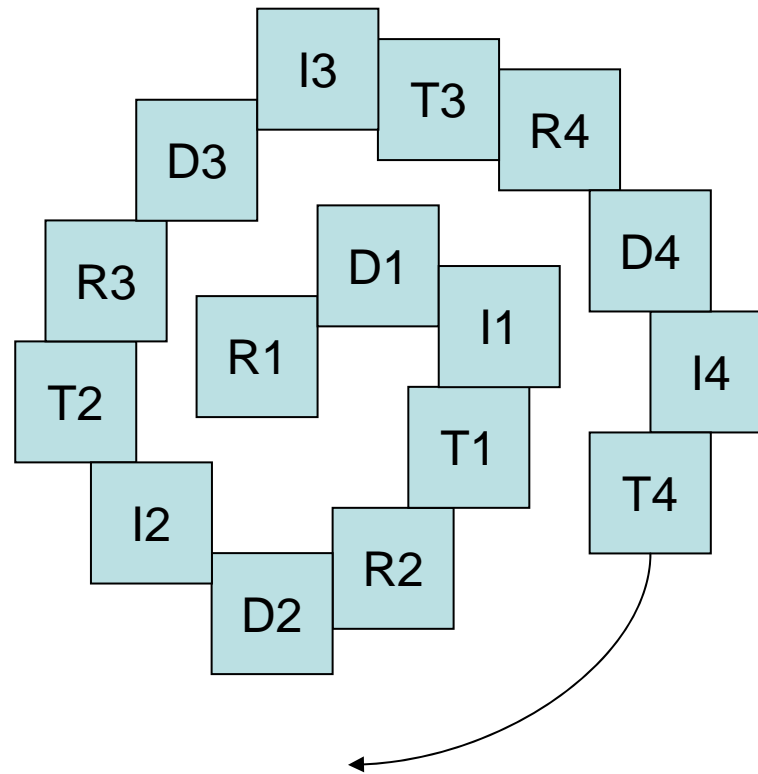


# The “Incremental” Model



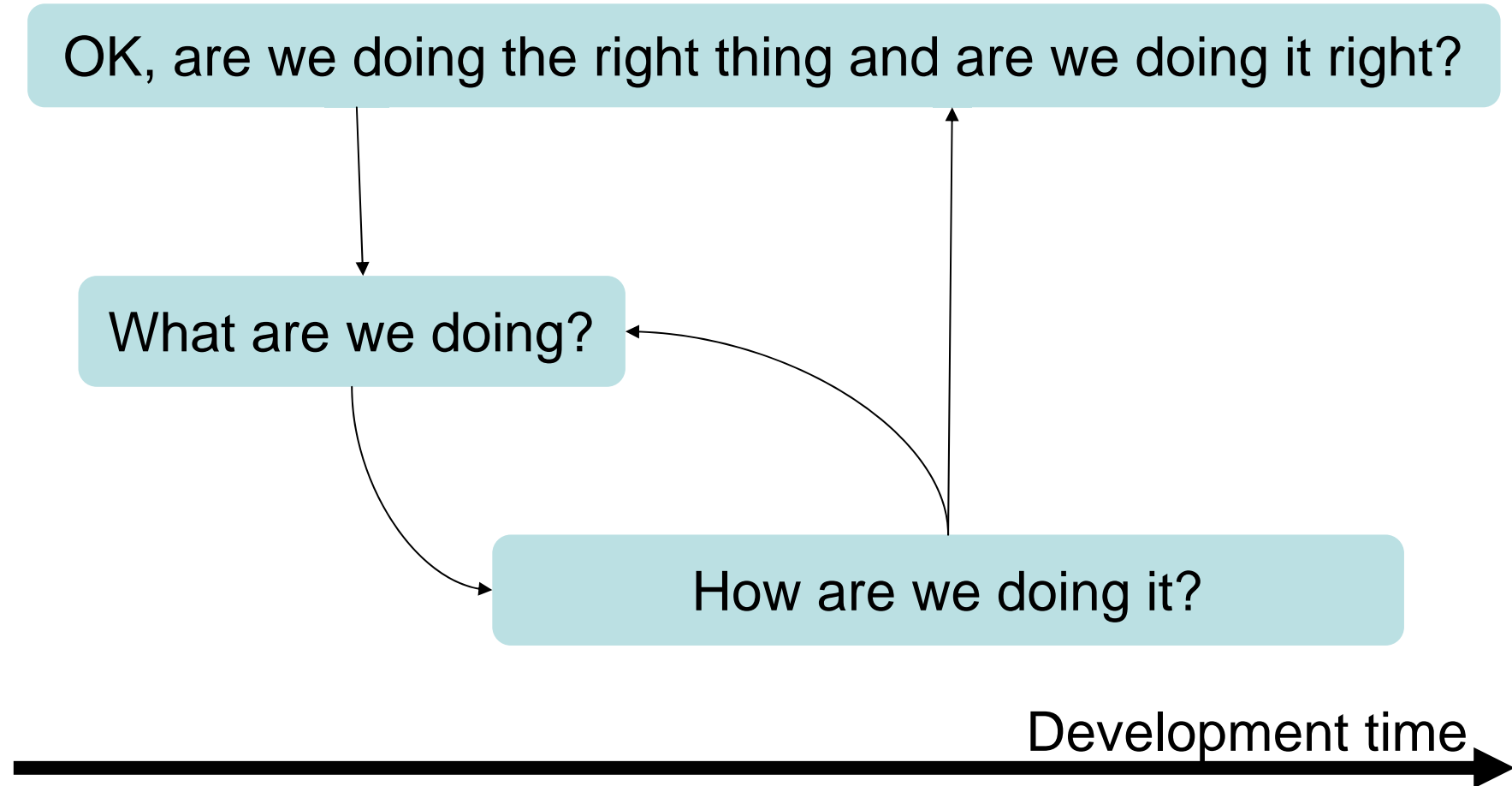
Development in increments or versions, where each new version is an extension or refinement of its predecessor

# The “Spiral” Model



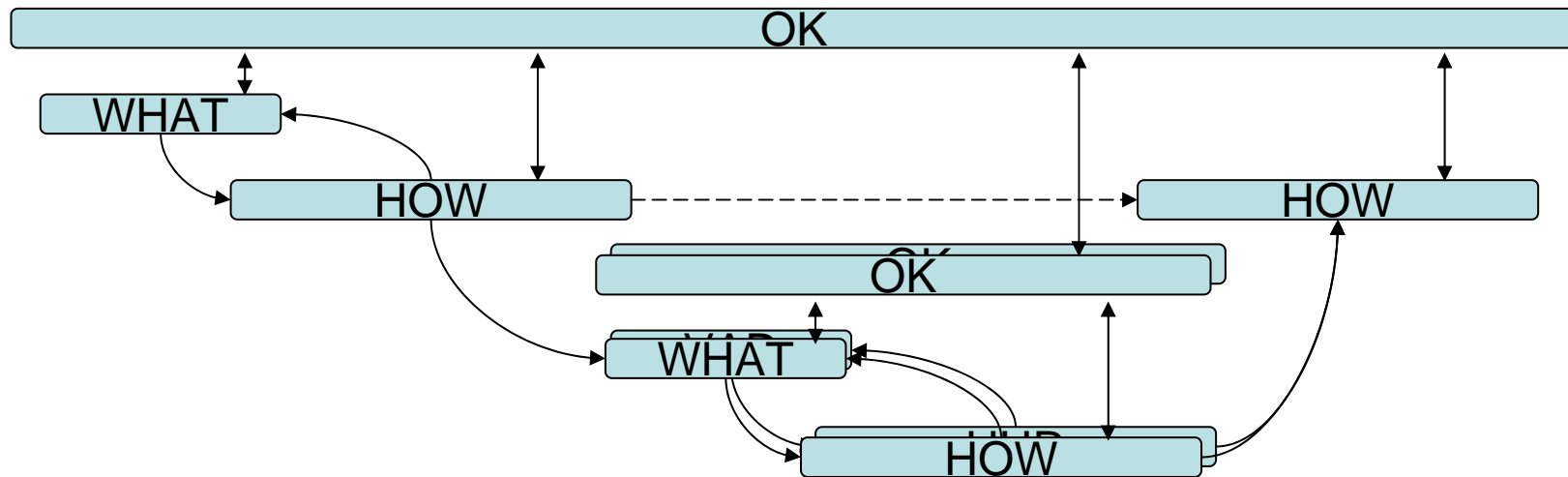
Development process viewed as a spiral

# What – How – Is it OK?



# Recursive Development

A system (and its software modules) can be described as a set of components that are connected to each other, recursively. We can also partition and execute the development process in the same way.



# Methods and Tools

## Tools for:

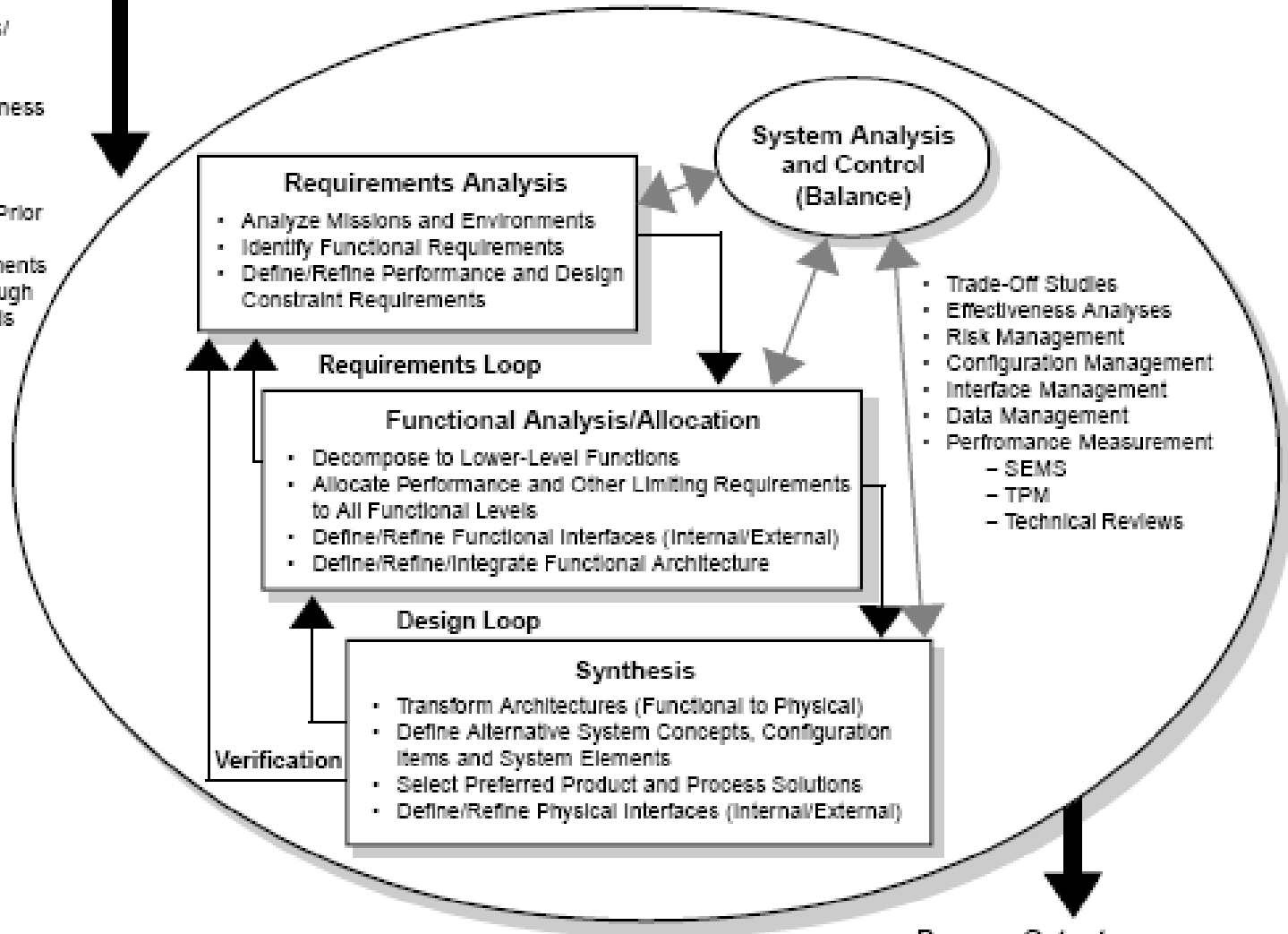
- requirements handling;
- modelling and design;
- implementation, test, debugging and integration.

## Methods for:

- Analysis, synthesis and use of tools;
- coordination of work;
- project control;
- team work, human interaction etc.

**Process Input**

- Customer Needs/Objectives/ Requirements
  - Missions
  - Measures of Effectiveness
  - Environments
  - Constraints
- Technology Base
- Output Requirements from Prior Development Effort
- Program Decision Requirements
- Requirements Applied Through Specifications and Standards



**Process Output**

- Development Level Dependent
  - Decision Database
  - System/Configuration Item Architecture
  - Specifications and Baselines

**Related Terms:**

- Customer - Organizations responsible for Primary Functions
- Primary Functions - Development, Production/Construction, Verification, Deployment, Operations, Support, Training, Disposal
- Systems Elements - Hardware, Software, Personnel, Facilities, Data, Material, Services, Techniques

# DODs Systems Engineering Process

# Systems Engineering

- **Recognize Need/Opportunity**
- **Identify and Quantify Goals**
- **Create Alternative Design Concepts**
- **Do Trade-off Studies**
- **Select Best Suited Concept**
- **Increase the Resolution of the Design  
(one step down in the design hierarchy)**
- **Repeat the previous process steps**

# The System Engineer's Dilemma

- To reduce cost at constant risk, performance must be reduced
- To reduce risk at constant cost, performance must be reduced
- To reduce cost at constant performance, higher risks must be accepted
- To reduce risk at constant performance, higher costs must be accepted
- In this context, development time is also a critical resource, and behaves like a cost

*Modified from "NASA Systems Engineering Handbook"*

# Increasing Demands on System Products

- Systems becomes larger and larger and more and more software intense
- Users demands and quality requirements increases
- Acceptance levels and expectations rise
- Products must be error free from start
- Each new product release must be an improvement of its predecessor

# Concept Evaluation

- Define purpose and market of the product
- Identify potential customers, customer groups and their requirements
- Identify conflicting requirements
- Estimate cost and time needed to develop, manufacture and support the product
- Decide if it is worth the effort and risk to start develop the new product

# Risks and uncertainties

- It takes time to develop – a lot can happen
- Risks are e.g. technical, economical and human
- Uncertain market and market share
- Uncertain competitors and product substitutes
- Customer demands change
- Competitors become better and/or cheaper
- A product is depending on other products
- A product is depending on competent developers

# Opportunities

- Market window – the window of opportunity
- Market value – how much are customers ready to pay for your product
- Product niche not addressed by competitors
- Compliance to new (and old) standards
- Sub-suppliers may provide new techniques

# Requirements Gathering

- Get to know the application/problem area
- Analyse product concept description/specification
- Get to know users and market
- Avoid implementation oriented terms
- Avoid leading questions
- Customers think in terms of previous and current experiences and solutions
- Analyse what the customer need and benefit from
- Capture both function and characteristic needs
- Consider the cost of each requirement

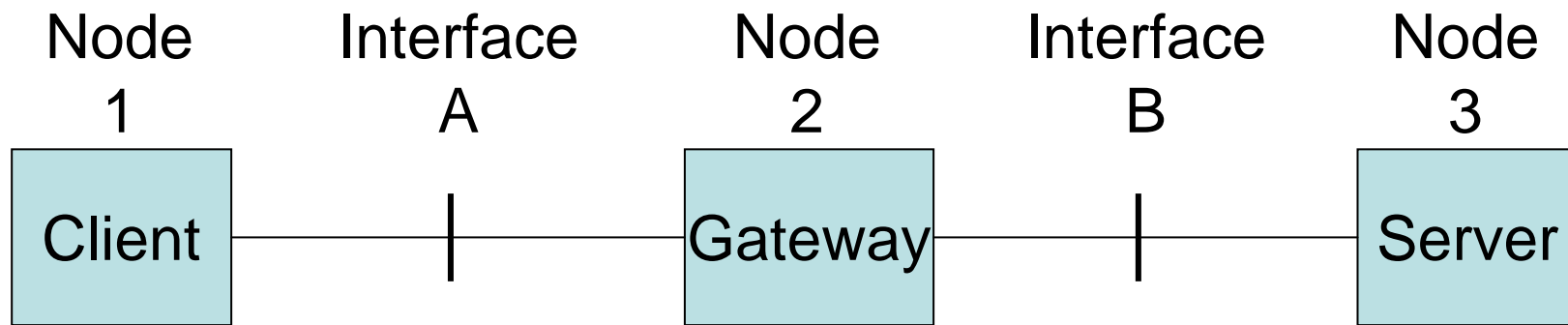
# Requirement Gathering Methods

- Interview (free form)
- Questionnaire (controlled or planned interview)
- Observation in real world situation
- Protocol analysis
- Document analysis
- Workshop (brain storming, yellow labels, etc.)
- Prototyping, demonstration and evaluation

# Prototypes and demonstrators

- Prototypes enables users to get a glimpse of and imagine new products and related experiences
- Prototypes can be used to create a context that the analyst and user can investigate together
- Useful when requirements are unclear or difficult to understand or when the situation or system is very new and/or unknown
- Areas such as the design of user interfaces are difficult just to talk or write about (**real hands-on experiences are needed**)
- A prototype makes the analysis easier since you can see how things look and how interaction sequences feel

# Physical Reference Model



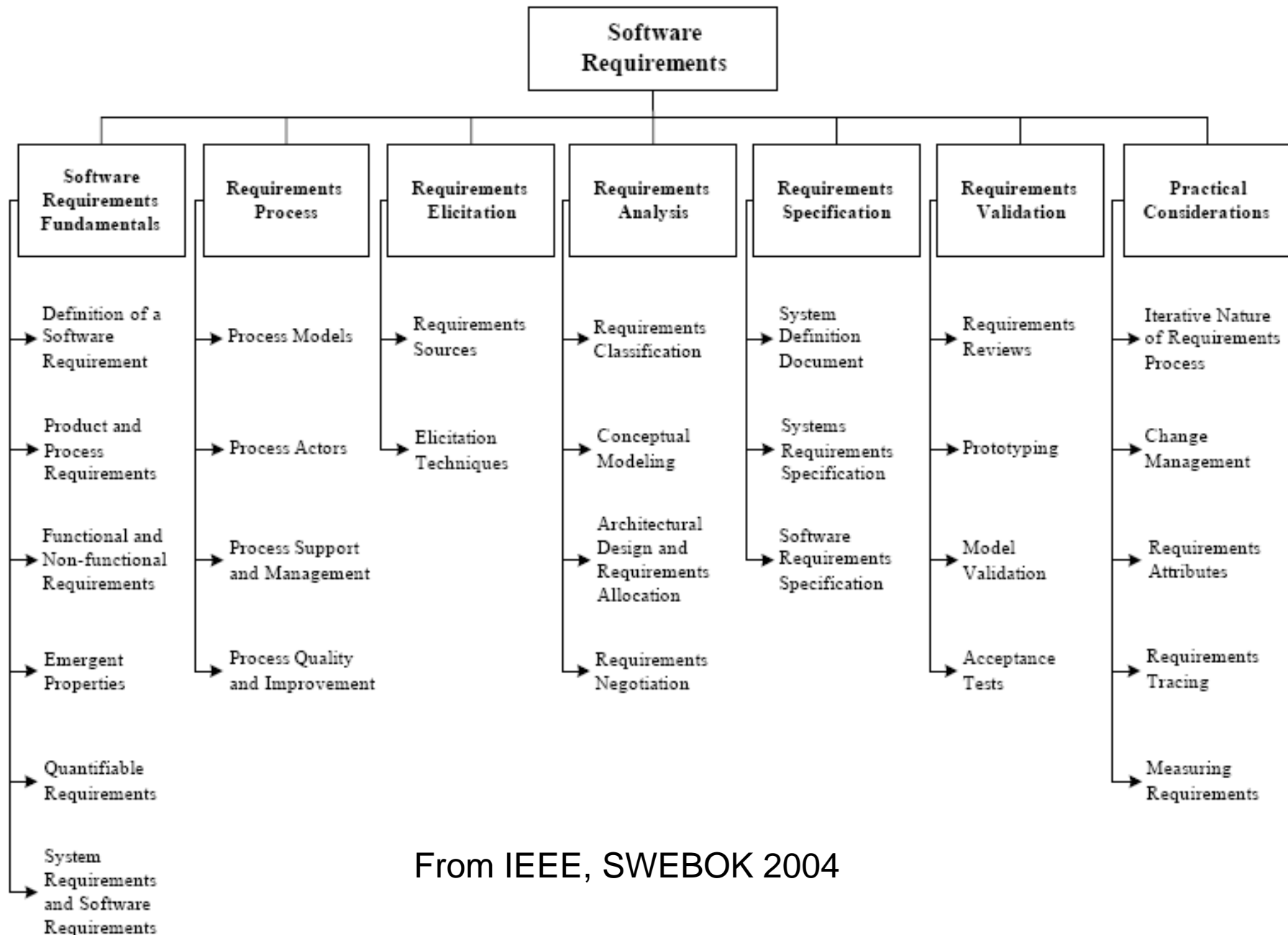
A static view of physical nodes and the interfaces between these

# Use of Use Case Scenarios

- Capture functional requirements by analysis of examples, called use cases (scenarios)
- A use case describes one normal scenario and a few extensions or exceptions
- A use case is a **sequence of statements telling how a user and a system will interact, step by step, from a given state**
- A user (e.g., a man or a machine) in a specific role is called an actor

# Modelling and Prototyping

- The certainty of project plans depends on the problems and risks involved
- Some problems and risks can be anticipated
- Risk and uncertainties with new technology, methods and ideas must be evaluated
- Models and prototypes can help to analyse and to verify that requirements are realistic and achievable



From IEEE, SWEBOK 2004

# Operational Requirements

## Basic Questions (from DOD)

- **Operational distribution or deployment**
  - Where will the system be used?
- **Mission profile or scenario**
  - How will the system accomplish its mission objective?
- **Performance and related parameters**
  - What are the critical system parameters to accomplish the mission?
- **Utilization environments**
  - How are the various system components to be used?
- **Effectiveness requirements**
  - How effective or efficient must the system be in performing its mission?
- **Operational life cycle**
  - How long will the system be in use by the user?
- **Environment**
  - In what environments will the system be expected to operate?

# Requirement Attributes

- Achievable (can be reached)
- Unambiguous (no fuzziness)
- Verifiable (can be put to test)
- Consistent (no contradictions)
- Complete (nothing that shall be there is missing)
- Need oriented (not just what or how)
- Appropriate (not too detailed or abstract)

# Requirements Analysis

- Reasons and motivations for new system/product?
- Customer/user expectations?
  - Who are the users and how will they use the product?
  - What do the users expect of or require from the product?
  - What is their level of expertise or previous experience?
- Environment and operational requirements?
- Existing and planned interfaces?
- Functions that the system should/must provide?
- Constraints (hardware, software, economic, procedural) to which the system must comply?

# Gather Business Knowledge

- Get understanding of the application area or business where the system is to be used
- Identify shortcomings in the current system
- Identify technical context and constraints
- Learn about detailed procedures and data that are used in current operations
- Identify the requirements on the system beyond current solution or state of the art

# Planning of Interview Process

- Identify who shall be interviewed (roles in an organization and related individuals)
- Start from the top:
  - inform about the purpose of the interview
  - get an overview of the organization
  - get necessary permissions and support
- Identify how to get access to the information that you need, for example where shall the interviews take place

# Planning of Interview, cont.

- Decide if you need to see documentation, environment and/or equipment, etc.
- Plan your objectives, questions and the agenda before the interview
- Control the interview think about the ordering and the type of questions

# Ordering of Questions

1. Get a picture of context and background
2. Ask about the system and its: operations, processes, documents/forms, volumes, events, results, ..., i.e. get details and facts
3. Interrogate about problems and limitations in the system or its ... (details as above)
4. Ask about needs and requirements motivating a new system or new functions

# Types of Interview Questions

## **Closed questions**

yes/no or quantitative answers (good if someone like to talk too much)

## **Open questions**

invite for expansion, elaboration and explanation of topic (good if someone not like to talk, may get such persons more engaged)

## **Probing questions**

follow up open questions, for example asking:

*"Oh, that seems interesting; tell me more ..."*

# Types of Interview Questions, cont.

## **Verifying questions**

check if you have got it right:

*"Do I understand you to say ..."*

## **Sequencing questions**

narrow down:

*"Tell me step by step how it works ..."*

## **Anecdotal questions**

understand and highlight specific scenarios or problems, e.g. by asking:

*"What do you do if this does not work ..."*

# Interview Form (Questionnaire)

Project .....

Interviewer .....

Place, date and time .....

Time (interval) .....

Participants .....

Comments .....

Question/Answer

What do the company do? ...

What is the market for the products? ...

What are the products that you company offers? ...

How do you exchange information with ....?

What tools do you use to ...?

.....

.....

.....

.....

# Interview Guidelines

- Avoid leading questions
- Keep questions short and precise
- Ask only one question at a time
- Check that the question is understood if possible before the answer is given
- Use language and terminology that the interviewed person (manager, engineer, farmer, ...) recognise and understand well

# Protocol Analysis

- Combination of interview and observation
- The interviewed person performs a task in front of the interviewer (analyst) and describes each part of the task in detail
- The task is performed using the existing system or a prototype of the new system
- The goal is to capture implicit and "taken for granted" elements in the system, procedure or service under investigation

# Requirement Form

Req - ID	.....
Name	.....
Source	.....
Receiver	.....
Priority	.....
Function/Service	.....
	.....
	.....
	.....
Characteristic/Attribute	.....
	.....
	.....
Motivation	.....
Related Requirements	.....
Solution Proposal	.....
Change History	.....

# Validation and Verification

- Are the requirements valid and relevant, no less no more?
- Are facts and statements supported by real evidence?
- Are each requirement correct, complete and consistent?
- Is the whole set of requirements without contradiction?
- Are there backwards compatibility requirements
- Document review – to find missed requirements
- Logical (formal) analysis
- Use case analysis
- Prototype evaluation
- Functional test plan
- Is it possible to write a users manual?
- Will the market afford the cost for this?

# Categorise Requirements

- Function (Service)
- Performance (Quality of Service)
- Priority (Relative importance)
- Composition level
  - System requirements
  - Component requirements
  - Service requirements (cross system/component)
- Technical
- Human

# Structuring of Requirements

## System

Function 1

Function 2

--

Function N

Characteristic 1

Characteristic 2

--

Characteristic M

## Subsystem 1

Function 1.1

Function 1.2

--

--

# Priority Valuation

- A standardized scale for valuation of requirements enables comparison and prioritization
- Requirements of different categories are hard to compare; a two level approach can help:
  - Define importance of each category
  - Define importance of each requirement in each category
- When giving value and selecting requirements we are setting the design/implementation space
- In case of requirement conflicts – a compromise or modification may solve the dilemma

# Valuation Method 1

- In a small set of requirements each requirement can be compared to all other, pair-wise
- By just having to compare pairs of requirements the judgment process becomes more reliable
- If a person give “consistent” valuations the result has high belief value
- If different persons comes to similar “correlating” conclusions the result has even higher belief value
- Subsystems and requirement areas can also be compared to each other
- We can then make a total value ranking of the requirements within and between the different sets

# Valuation Method 2

- Give a sum of marks to each member of the requirement valuation team
- Optionally, limit the number of marks that a member can give to a single requirement
- Estimate average, median and variation of marks (value) given to each requirement
- Less variation gives more belief value
- To find a total ranking, give weight to the different requirement areas

# Understand the Problem

As a means to talk about, understand and capture the requirements you must identify:

- goal (system, product or service to develop),
- actors (technical and human),
- use cases (work or action sequences),
- stimuli or data to handle by the system,
- physical reference model

Learn about the problem in the real world.

If the system is to be used in the jungle; go to the jungle and learn.

# Actors and Use Cases

- What actors are interacting with the system?
- What do the different actors do?
- Who will operate and maintain the system?
- For an IT system: who should create, read, write, update or remove data; needed to operate or manage the system?
- Is the system connected to external actors (databases, sensors, actuators or other systems etc.)?
- To what extent is the system and the application SW influenced by HW interrupts and other similar things?
- Is also other SW components, the SW platform or the OS behaving as actors in use cases?
- Should an OS that generates clock and timer events also be seen as a kind of actor?

# Use Cases

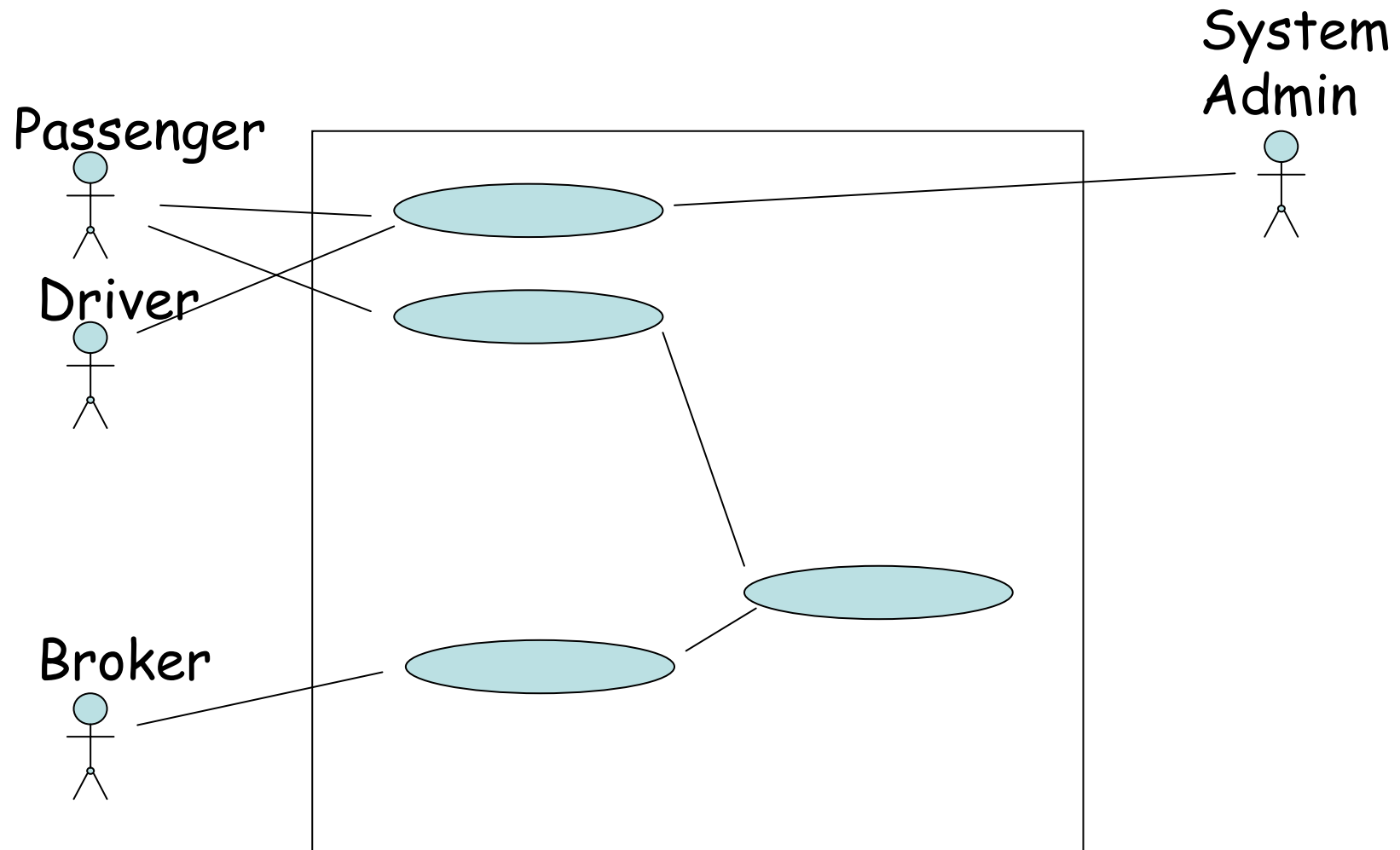
- What functional needs are there?
- What shall the system do to fulfil its purpose?
- What does the different actors need to do?
- What tasks and roles does an actor stand for?
- Specify each identified actor and its roll?
- What does an actor expect from the system?
- Does the system store or handle data?
- What information does the data represent?
- How about state changes, are they observable?
- What external events must the system handle?
- Legacy, component or platform dependencies?

# Use Case Diagrams

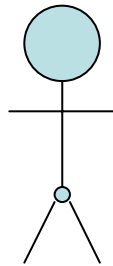
Capture and show in graphical form;

- the relationships between actors and use cases (in the system analyzed)
- how use cases are related to (optional) extensions
- how use cases are related to other use cases

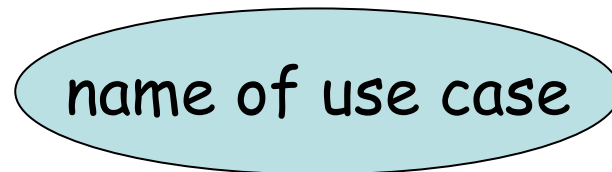
# Use Case Diagram



# Use case diagram notation



Actor



Use Case



System boarder

# Example, use case (request)

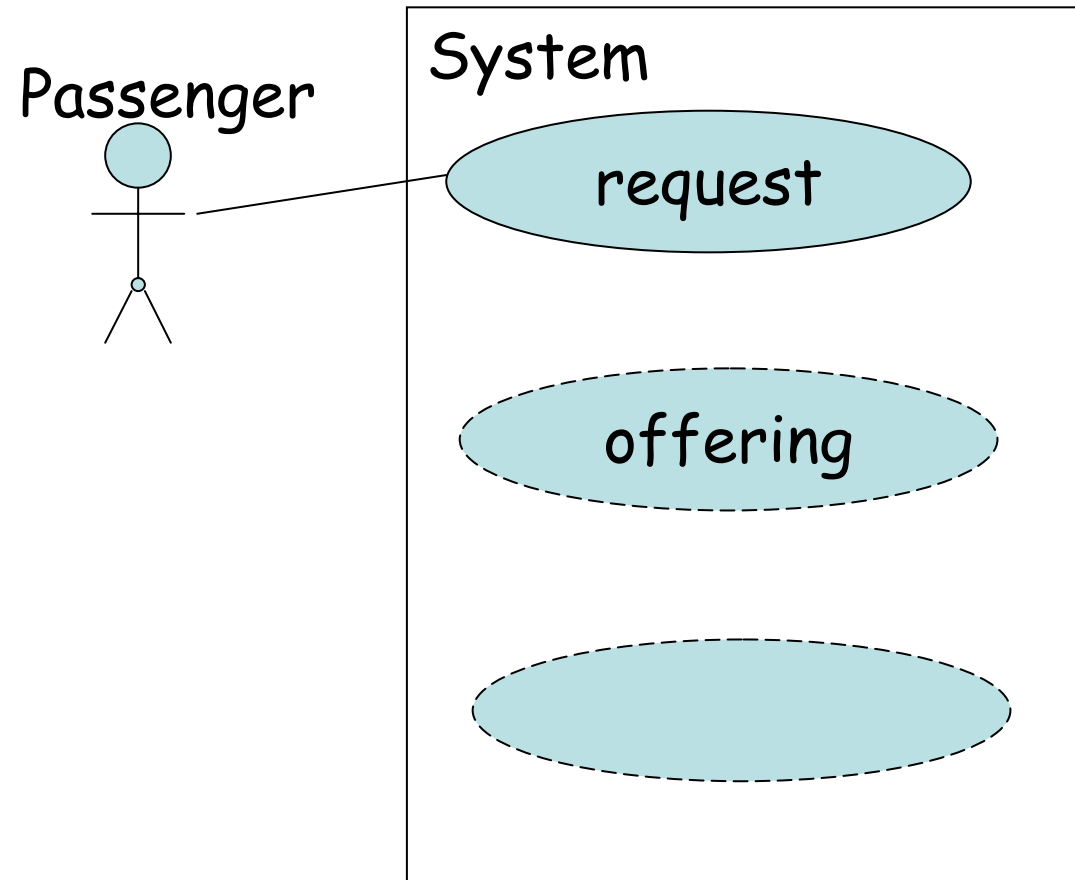
## Normal (success) scenario:

- Passenger tells from where to start (time point and location)
- Passenger tells where to travel
- Passenger tells when to departure (an interval)
- Passenger tells when to arrive (an interval)
- Passenger gets transport proposals, sorted
- Passenger selects a proposal

## Extensions (alternatives or exceptions):

- There is no transport proposal available
- The customer rejects all transport proposals
- Customer do not like driver or vehicle

# Example, Passenger Use Case



# Example, use case (offering)

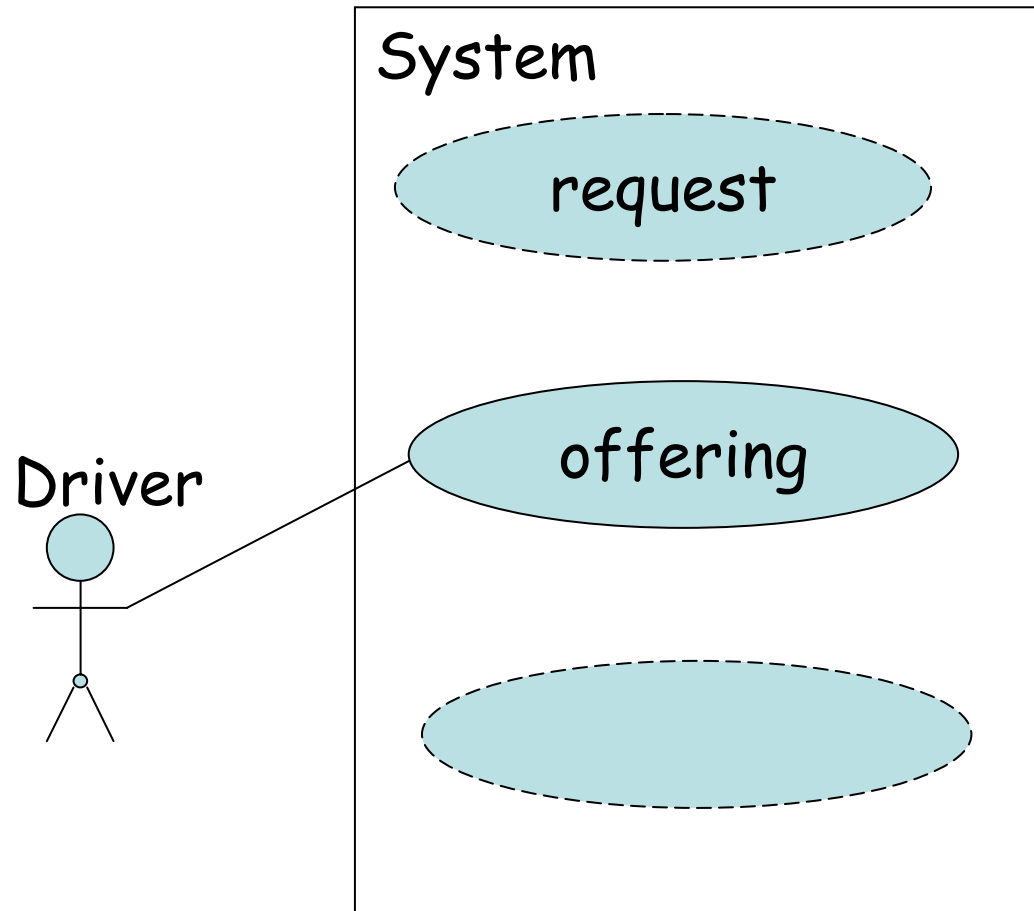
## Normal scenario:

- Driver tells from where to start
- Driver tells where to travel
- Driver tells when to departure (an interval)
- Driver tells when to arrive (an interval)
- Driver tells acceptable access detour
- Driver waits on acceptance response

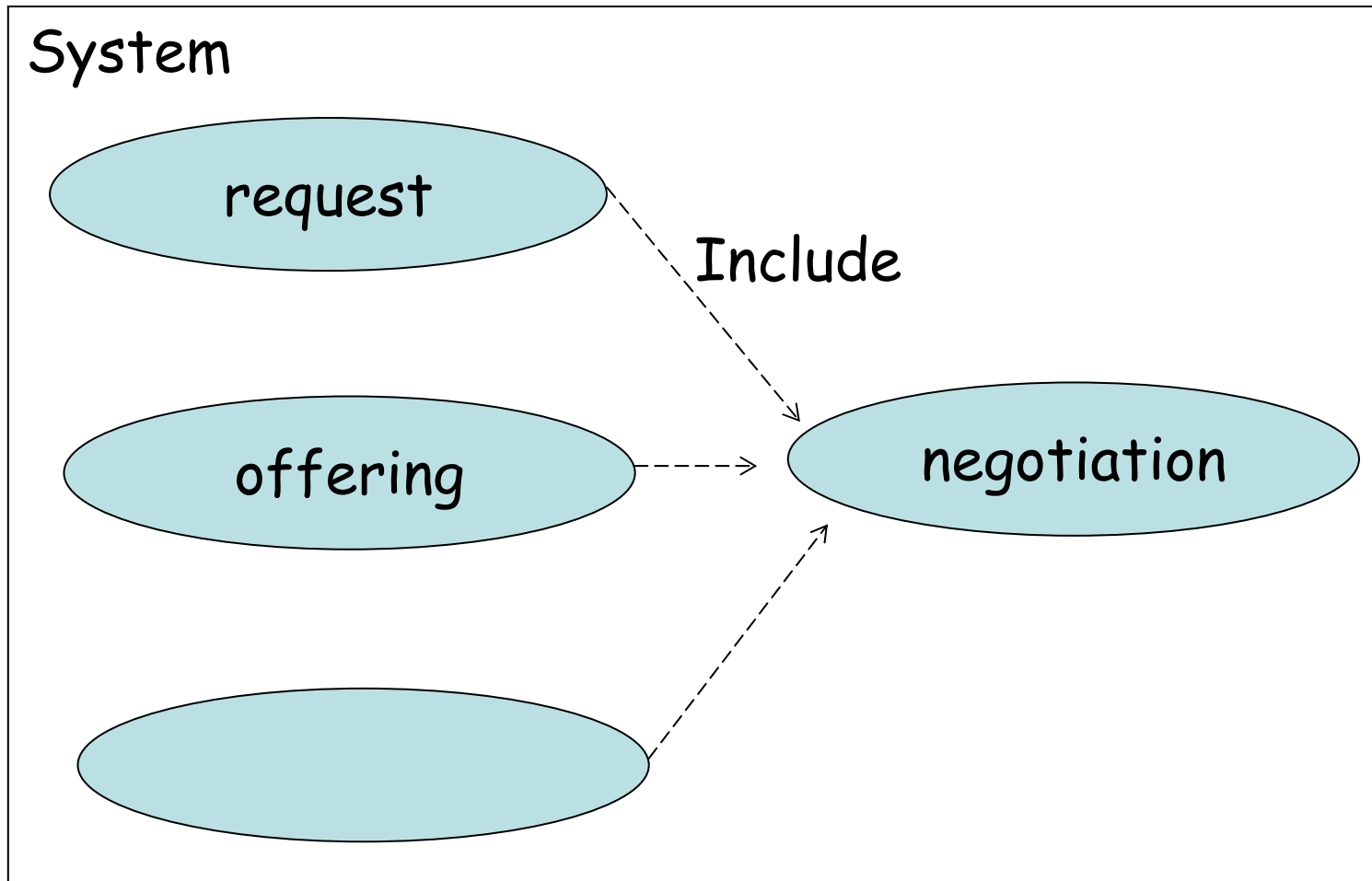
## Extensions:

- No one accepts the offering
- More than one accepts the offering

# Example, Driver Use Case

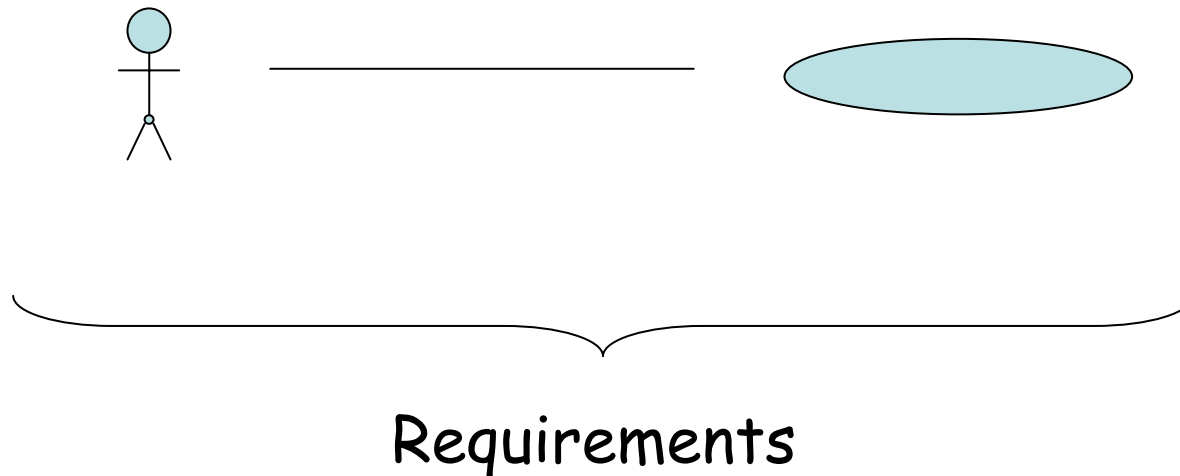


# Use Case, cont.

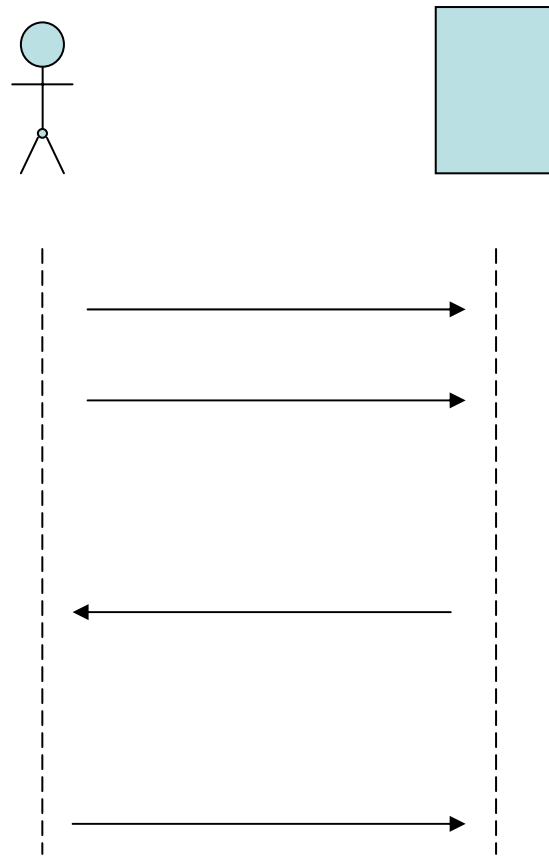


# Use case and actor perspectives

Requirements should be captured and analysed for each use case seen from each actors perspective



# Use Case - Sequence Diagram



Sequence diagram, system level

# Use Case Terminology

Actor	Anyone or anything that exhibits behaviour that affects the system
Primary actor	Actor initiating an interaction with the system, to archive a goal
Goal	What the use case is intended to do
Scope	The system scope that the use case is part of
Level	Abstraction level of the goal for the primary actor

# Use Case Terminology

Stake holder	Some (man or machine) with an interest in the behaviour of the system.
Precondition	The state of the system (or what must be true) before the use case is initiated.
Trigger/event	The real world event that activates the use case.
Success criteria	Post condition that must be true after that the use case is performed.

# Use Case Terminology

Main scenario	Step by step description of a case in which everything works as normal.
Extensions	What can happen in less normal cases, also described step by step.
Identification	Different steps in the main success scenario is indicated by numbers. Extensions are indicated by adding letters to the numbers (4a, 4b, ...).
Cross references	Other uses cases referred to are <u>underlined</u> .

# Use Case - Form

Use Case: .....

Primary actor: .....

Goal: .....

Scope: .....

Level: .....

Stakeholders: .....

Preconditions: .....

Trigger/event: .....

Success criteria: .....

Main scenario: .....

.....

.....

.....

Extensions: .....

.....

Cross References .....

# Example Use Case

## **Normal (success) scenario:**

- Passenger tells from where to start
- Passenger tells where to travel
- Passenger tells when to departure (an interval)
- Passenger tells when to arrive (an interval)
- Passenger gets transport proposals, sorted
- Passenger selects a proposal

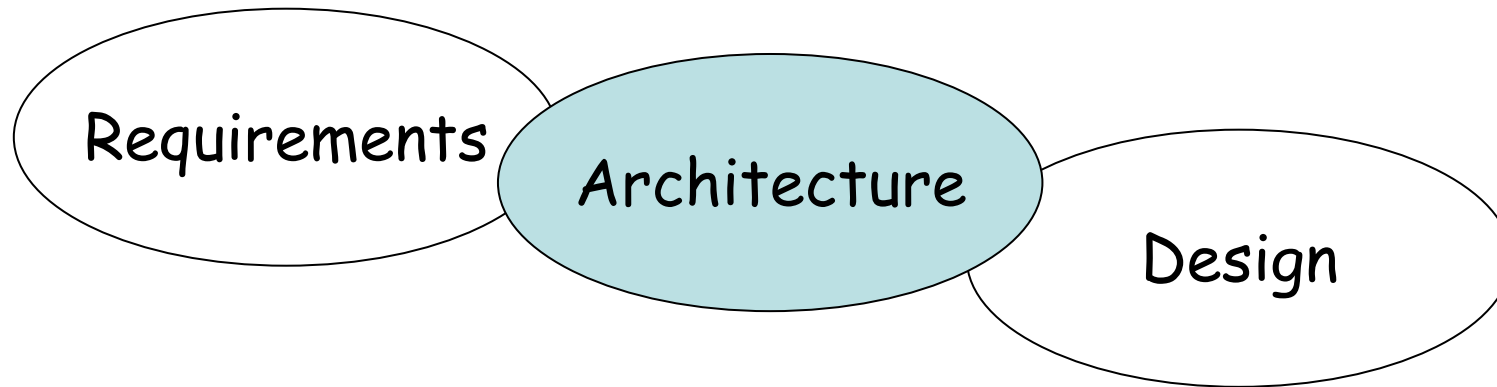
## **Extensions (alternatives or exceptions):**

- There is no transport proposal available
- The customer rejects all transport proposals
- Customer do not like driver or vehicle

# What and How

- The most important perspectives of design:
  - First, make efforts to find out WHAT to do
  - Second, find out HOW this can be done
- The activities needed to do this can, if specified, yield a more detailed design process description

# Architecture



Architecture provides structure and solution strategies by considering stakeholders requirements and views to act as guideline for the more detailed design and implementation

# Why, Where and When

- The design process should include issues about why to do a design (motivation)
- Where (in what environment) it is to be used
- It is also important to know when a product is needed on the market
- These issues influence the design choices and the time plan and resources needed for the development

# Behaviour

- The what and when perspectives describe the behaviour of a system
- By a systems behaviour we mean:
  - what actions/operations to do, within the systems scope
  - what interactions, with other systems
  - when in time, actions and interactions are performed (absolute or relative to time reference)

# State and time related behaviour

- Find out time and state related behaviour, describing when services shall be given
- State machines/matrixes can be used to model relations between states and state transition triggering events
- A state chart is a hierarchical mean to model complex state machine behaviour

# Structure

- The how perspective describe the implementation structure of a solution or system, i.e. how it can be realised
- By structure we mean:
  - what parts that constitutes a system and;
  - how these parts are connected to enable the actions and interactions that will implement the systems behaviour

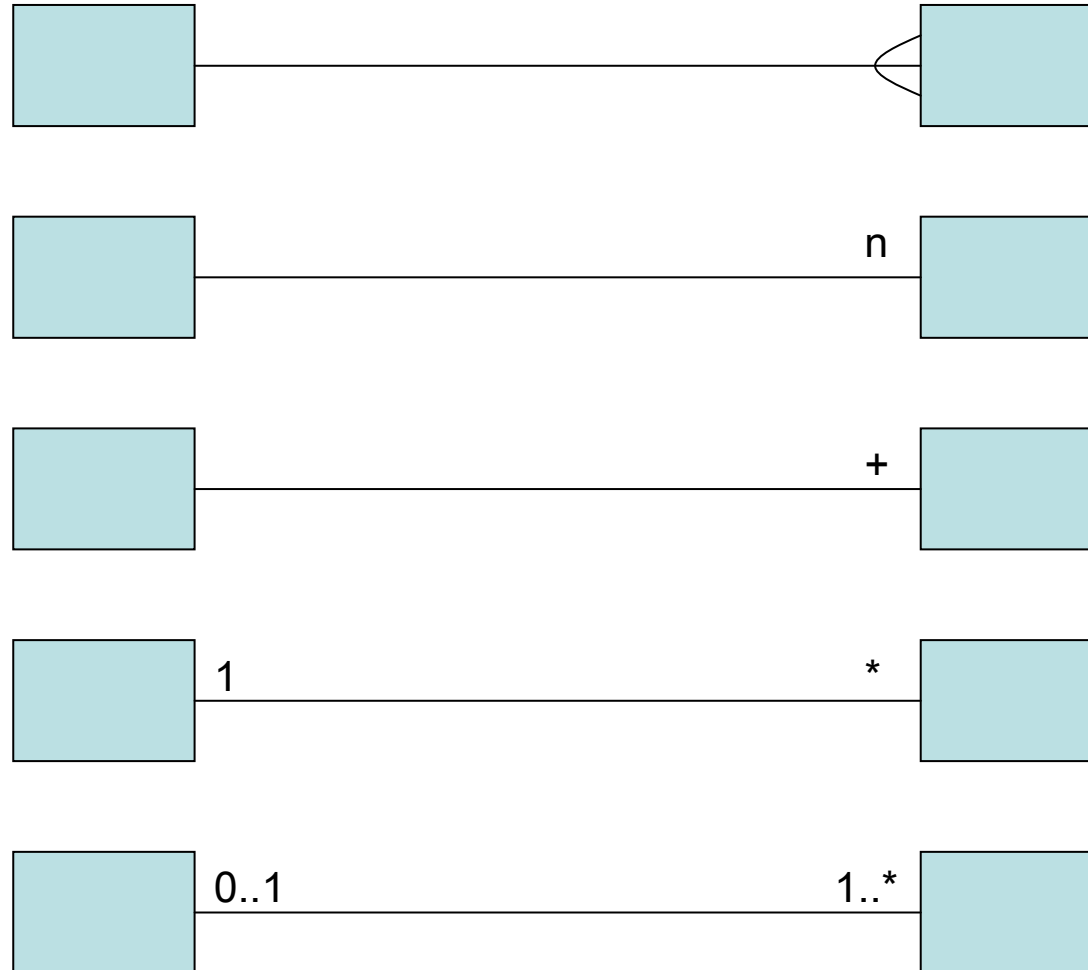
# Data modelling (concept level)

1. What, do we need to handle information about?
2. How are the information related (EAR modelling).
3. Identify object classes in the domain to mirror the information content of the application.
4. Find out how the classes are related.
5. Focus on the needs do not model the entire world.
6. Only model what is needed for the system or application.
7. Identify how and how many objects will be instantiated, i.e. their cardinality.

# Objects and Classes

1. Identification of objects (entities) and their relations to other objects is an important part of system analysis work (compare ER and EAR modelling).
2. When we talk about objects we do not always distinguish between their class or type and the actual objects or instances of the class.
3. For example, when we say that there is a one to many relation between objects we typically mean between one instance of a class and a set of instances of the same or another class.
4. Modelling languages helps us to make this clear.

# Syntax (or notation) differences



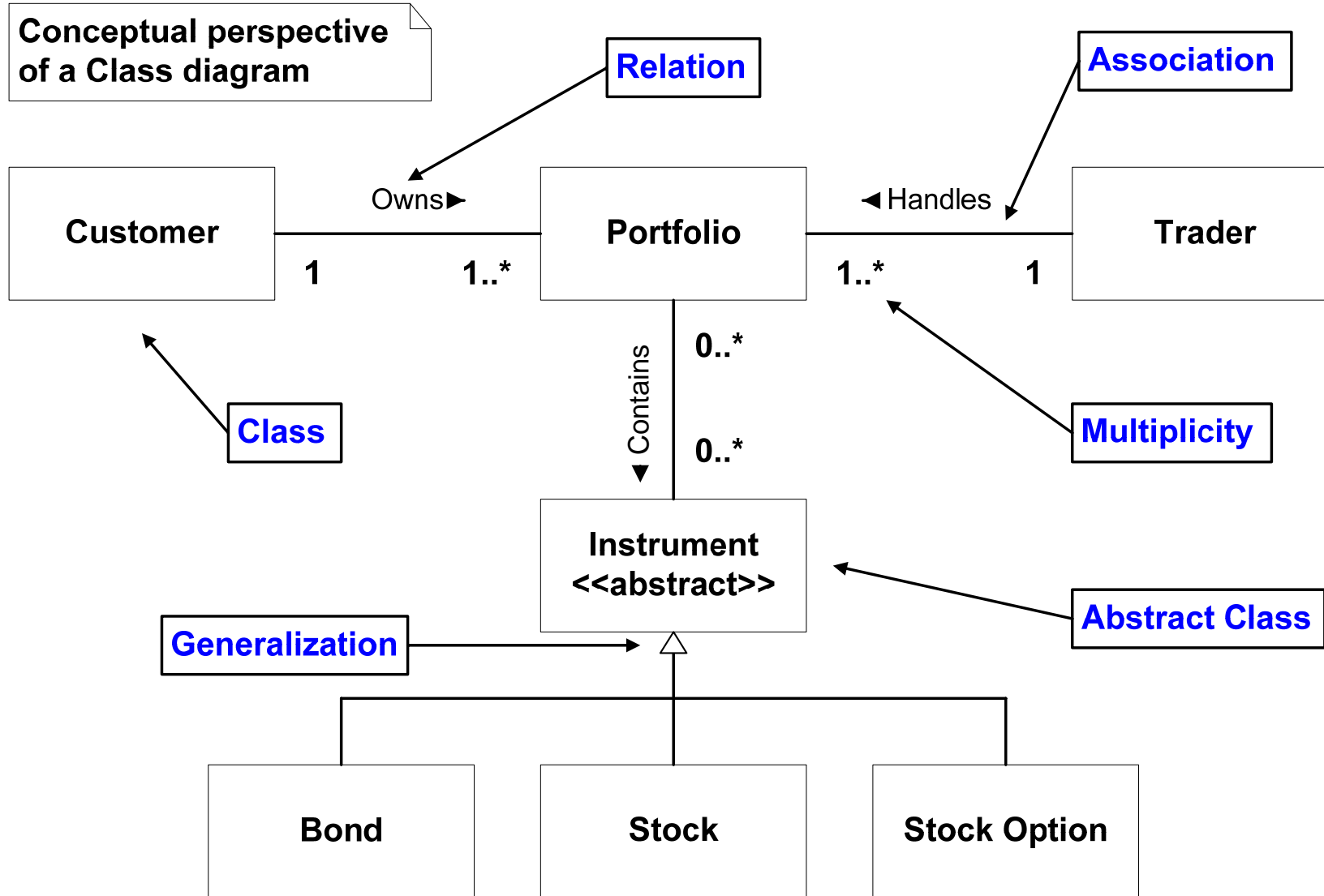
# Abstraction in early design phase

- Avoid to be influenced by implementation and programming language terminology
- Think of objects as containers of data to be handled by operations invoked by messages
- These messages might latter be implemented by procedure/method calls

# Objects, operations and messages

- For communication between objects within a process or protected memory area messages can be replaced by method (or procedure) calls in the implementation
- For distributed systems with objects communicating over process and/or processor borders, messages are sent via interface (proxy) objects as asynchronous messages or remote procedure calls

# Class diagram



# Conceptual classes and objects

1. Initially we are mainly interested in the object classes and what number of instances that we are dealing with.
2. The purpose is not to define individual attributes of the classes and we are even less interested in what methods that a class may contain.
3. What we aim for is to identify (and name) object classes (record types) that represent important information that the system is going to handle.
4. We also want to know if these classes will occur as one, a few or in many object instances and if these are grouped (or partitioned) in some way.

# Class Design

- To design good classes requires you identify and consider possible pros and cons
- One important factor is how to assign responsibilities to the different classes
- Design patterns are models for how to assign responsibilities in certain kinds of situations

# Responsibility of a Class

- Class members are responsible for what they:
  - Know about (their data)
  - Can perform (by use of their methods)
- This related to their roles or obligations in the overall behaviour of the system
- Partition the overall task by thinking about which classes that should be responsible for different parts of the data and methods

# Responsibilities

A class member (object) can **KNOW**:

- about its own private encapsulated data
- about other related classes/objects
- what it can do, directly or indirectly,  
i.e. what services it can provide

A class member (object) can **DO**:

- something on data handled by the class/object
- initiate actions/methods in other classes/objects
- coordinate activities in other classes/objects

# CRC Cards

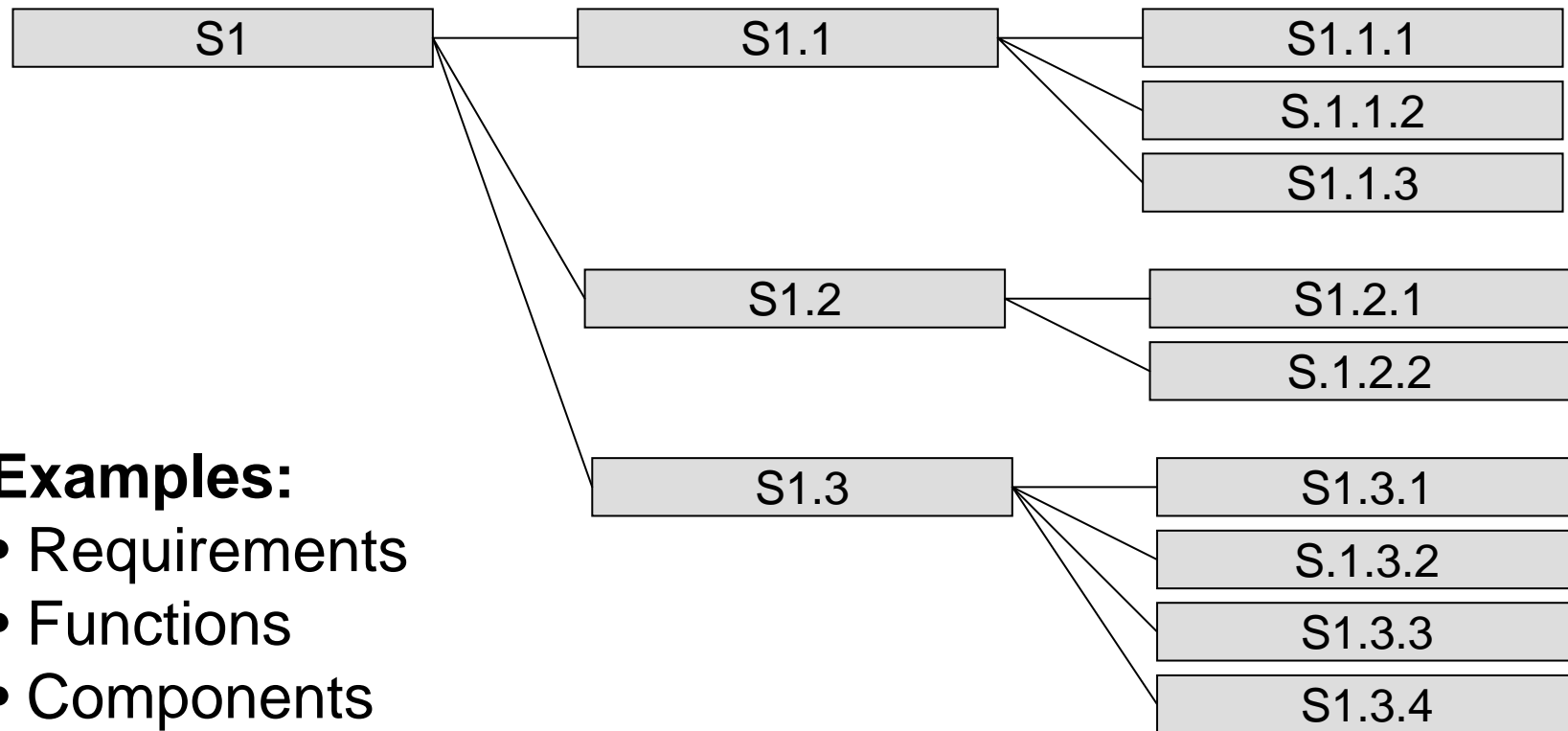
Class-Responsibility-Collaborator (CRC) cards

Class name	
Responsibility 1 Responsibility 2 . . . . Responsibility n	Collaborator 1 Collaborator 2 . . Collaborator m

# Inheritance and other relationships

- Inherit from another class or use composition and cooperation between class instances, check:
  - are there other important relationships or associations between the classes or objects?
  - will the classes found be able to represent the information that the system shall handle?
- Analysis and definition of external interfaces is a separate and later issue
- Technology matters (such as whether to use proxies or not) is of no concern at this stage

# Decomposition Structure

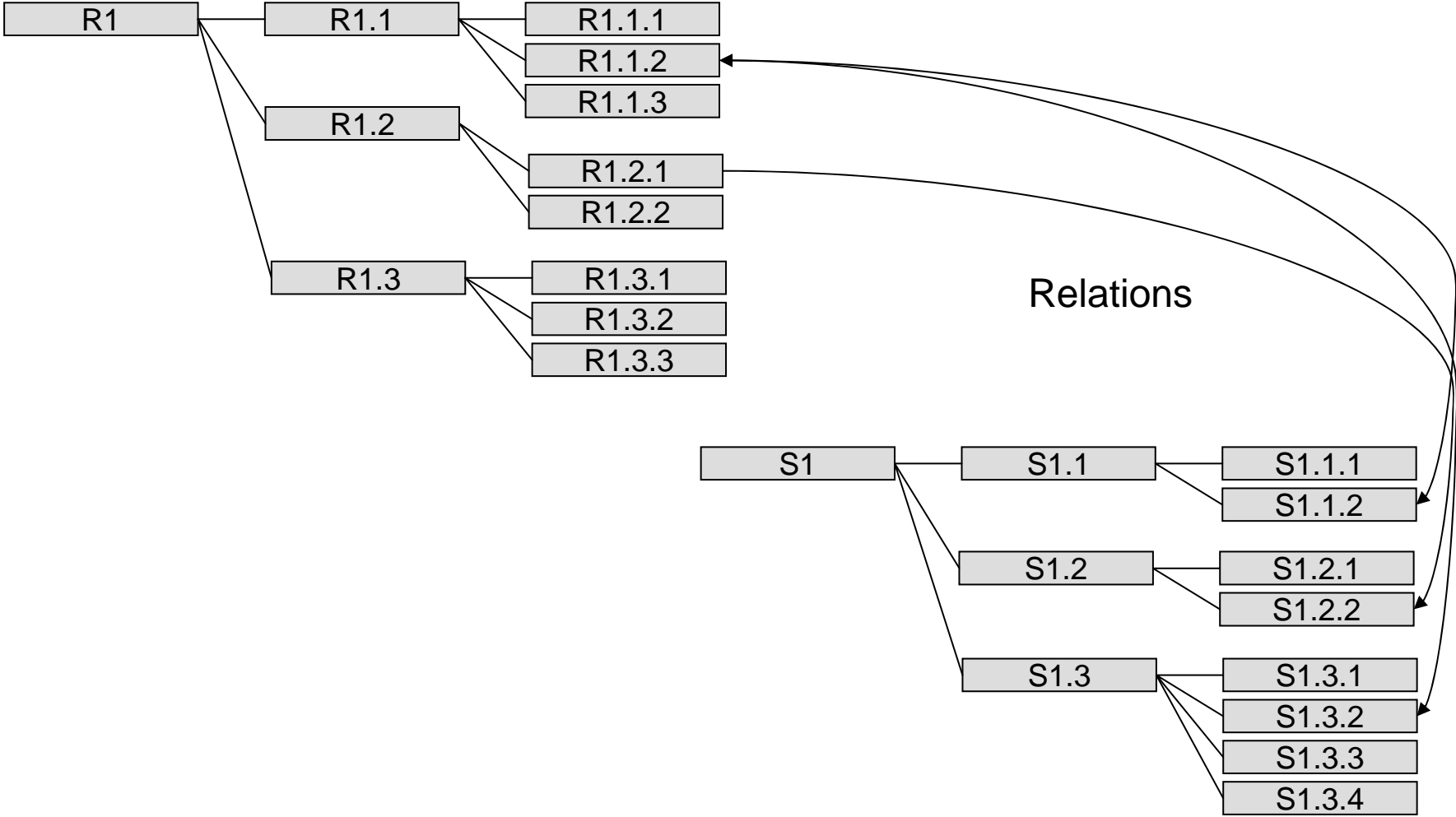


## Examples:

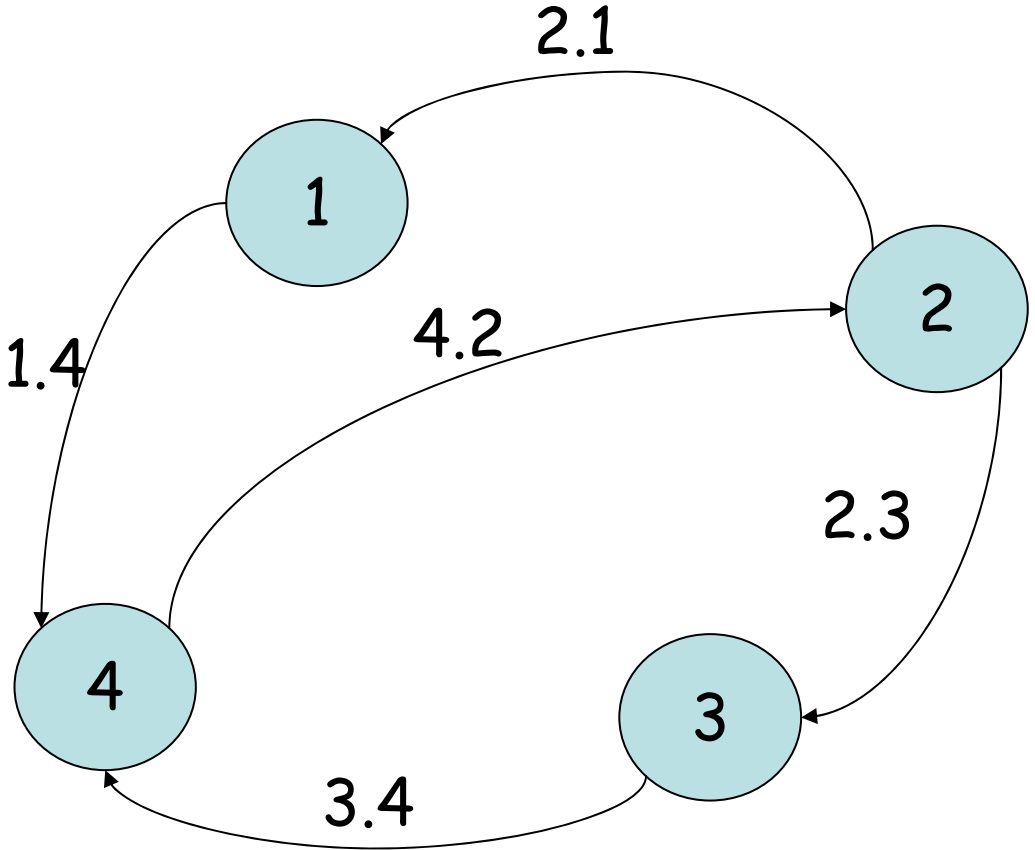
- Requirements
- Functions
- Components
- HW
- SW
- etc.

**“Divide and conquer”**

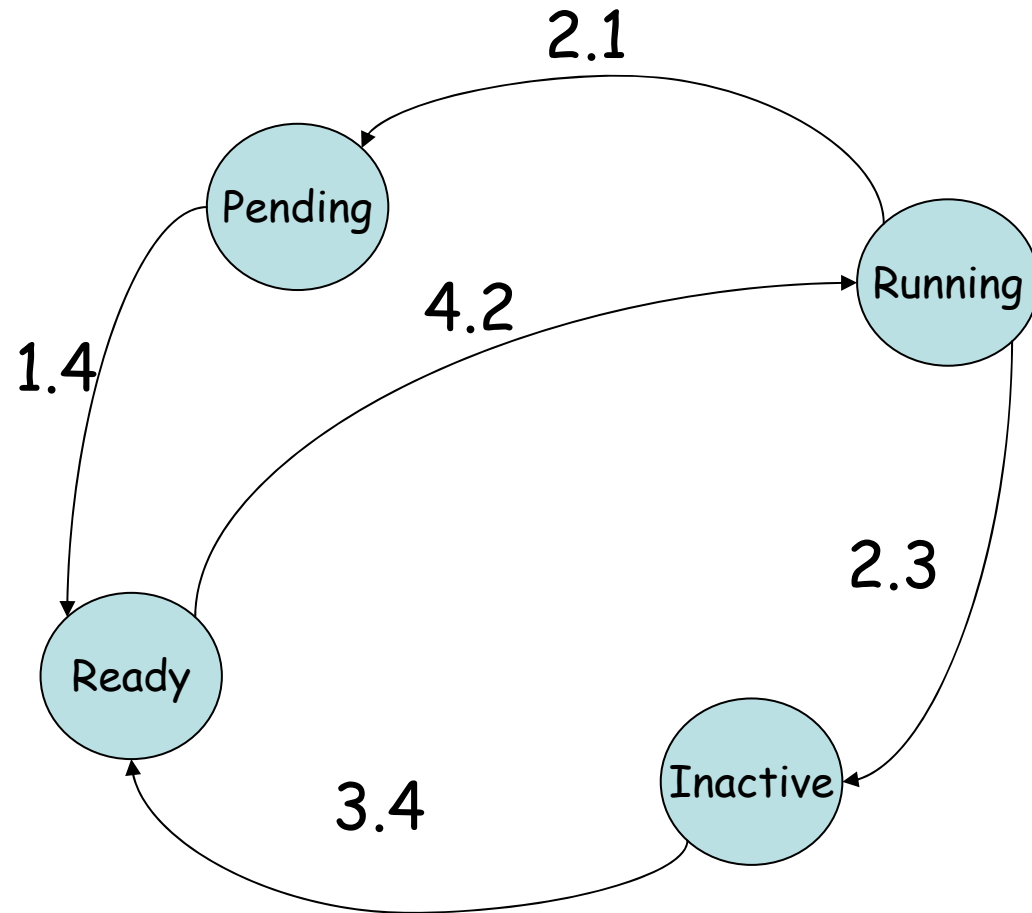
# Traceability



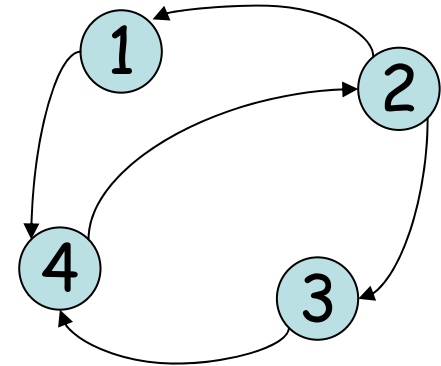
# State Transition Models



# State Transition Models



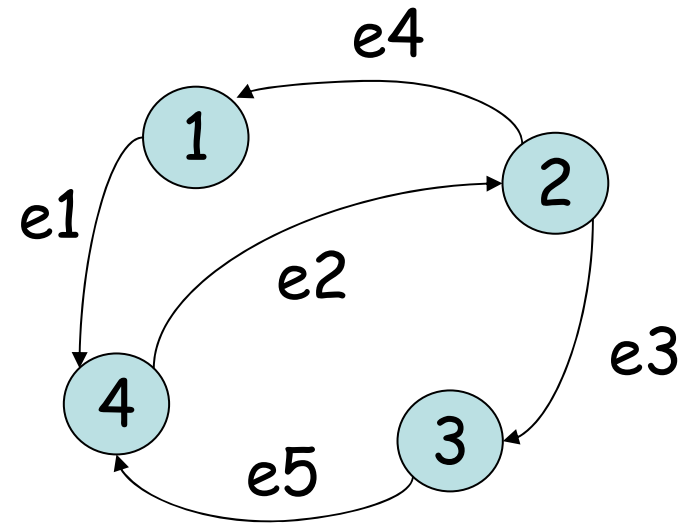
# State Transition Matrix



	e1	e2	e3	e4	e5	
s1	s4 a1.4					
s2			s3 a.2.3	s1 a2.1		
s3					s4 a3.4	
s4		s2 a4.2				

# State Transition Matrix

	e1	e2	e3	e4	e5
s1	s4				
s2			s3	s1	
s3					s4
s4		s2			



# State Transition Matrix

	e1	e2	e3	e4	e5	e6	e7	e8
s1	s4	s2						
s2					s3			
s3			s4					s1
s4		s2		s6				
s5	s1					s1		
s6	s1						s5	

# Architecture

- A good architecture should provide guideline and framework for the remaining design work
- But first we must propose, analyse and chose between alternative architectures
- The choice of architecture can:
  - Have a large effect on characteristics such as the systems reliability, capacity and scalability
  - Make it easier to solve some requirements
  - May have negative impact on the implementation of other requirements

# Architecture, cont.

- A strategy for organizing the system
- System components has to follow architecture rules for the system
- Components shall be possible to evolve and be maintainable over the system life cycle
- It is too easy to optimise an individual component, function or characteristics at the cost of others
- The architecture should:
  - give structure and balance
    - and may for example simplify component replacement, testing and fault handling etc.

# Architecture, cont.

- An architecture can have many purposes
- One may be to enable flexibility and upgradeability
- However, a flexible architecture is not necessarily a good architecture, for example if cost or performance is important
- An architecture that supports platform independence and portability is neither necessarily a good architecture
- The fact that an architecture have a positive connotation does not mean that it is good

# Balance

- Is not flexibility and portability good properties?
- Yes they are, but more important is to provide the characteristics that are required in your case
- If this means that code size and performance is key, for example for a high volume product, other characteristics are of less or no importance
- Especially if otherwise "good" characteristics have a negative impact on other important, highly valued, characteristics of a product

# Architecture choices

- Client-Server
- Client-Agent-Server
- Central or distributed data, with or without general database management support
- Function, object or component oriented
- Data, event or time driven
- Push or Pull
- Publish-Subscribe

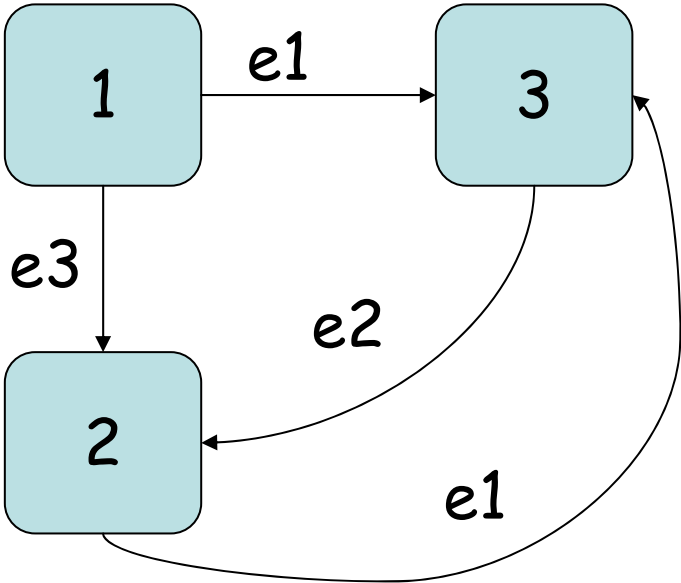
# Architecture choices, cont.

- Pre-emptive or non preemptive scheduling
- Fixed priority rate monotonic or dynamic priority  
early deadline first scheduling
- State-full or state-less (thick or thin)
- Asynchronous or synchronous communication
- Autonomous or fully user controlled
- Value passing or reference passing
- Process, thread or state object

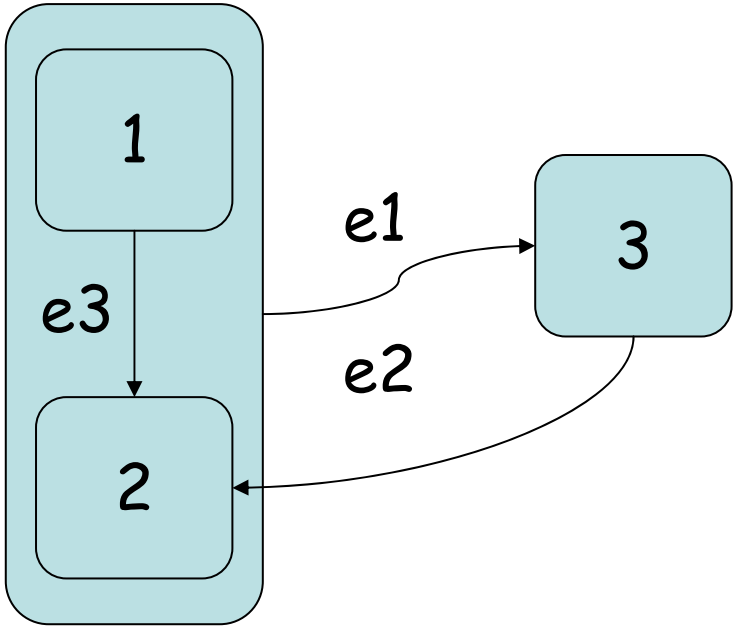
# Client architecture, examples

- Client contain general GUI support configured by parameters and data from a server
- Client contains application specific GUI components
- Client contains little or no application state, i.e. application state is stored in server DB
- Client contains most or all application state

# State Charts

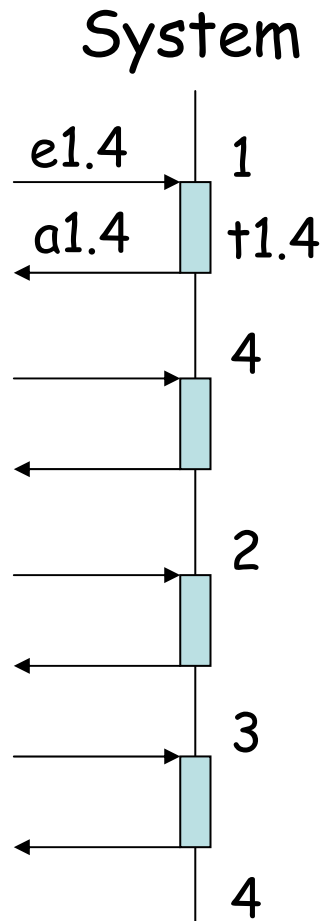


Flat or expanded



Hierarchical

# Sequence Diagram



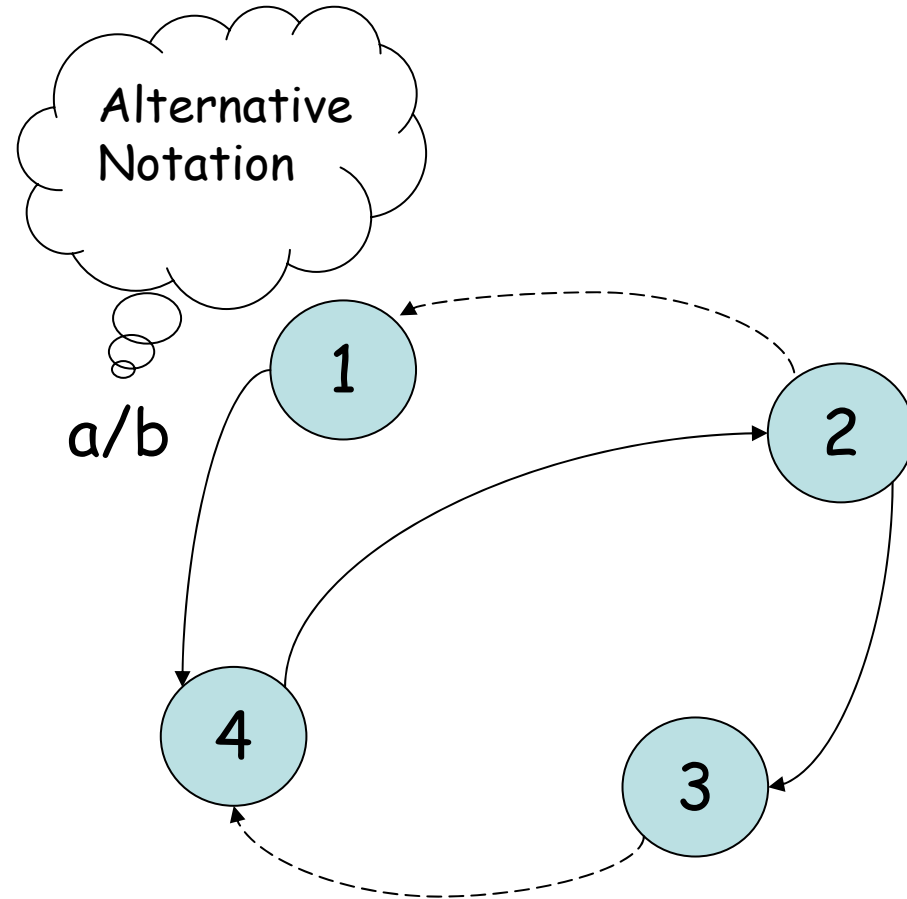
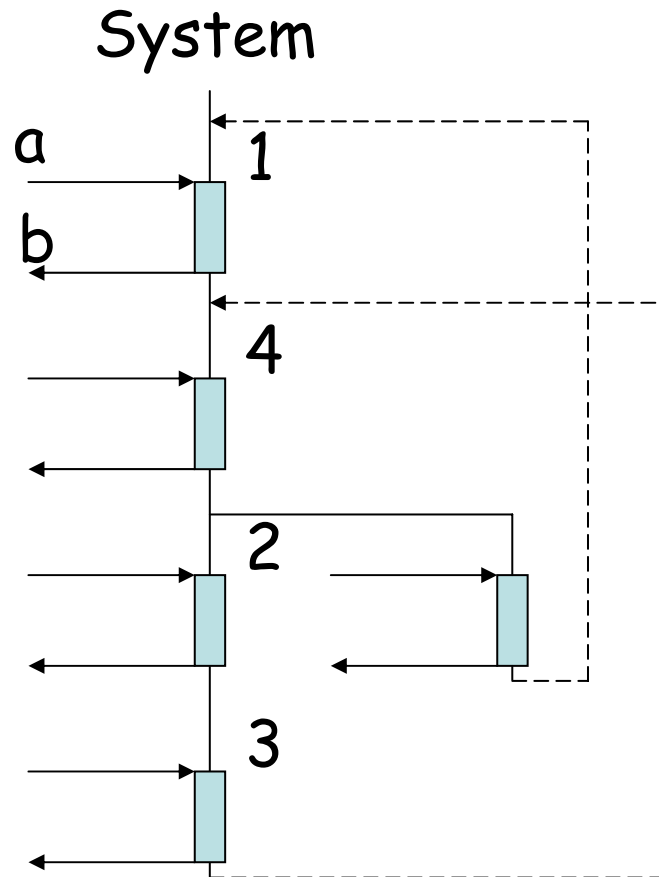
The transition from state 1 to state 4 (called transition 1.4) is triggered by event 1.4 and result in action 1.4.

t = transition

e = event

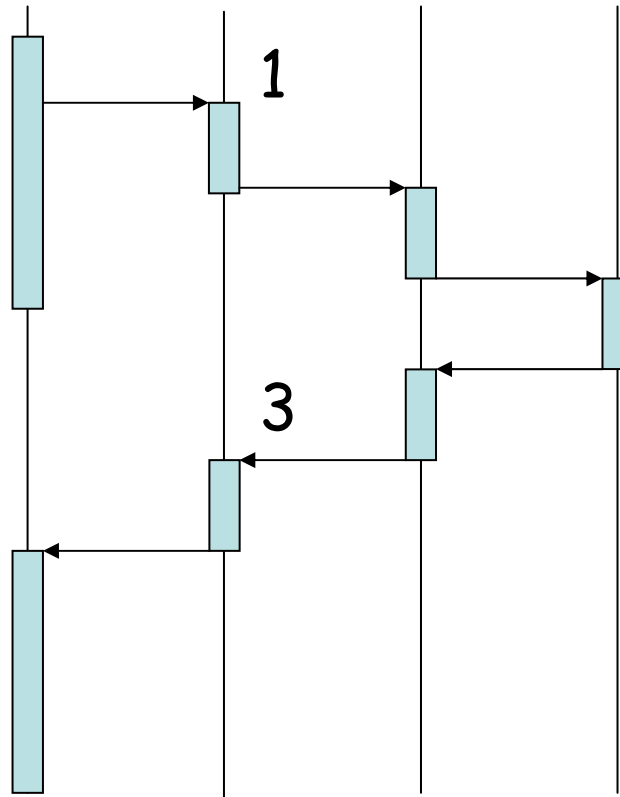
a = action

# Sequence of State Transitions

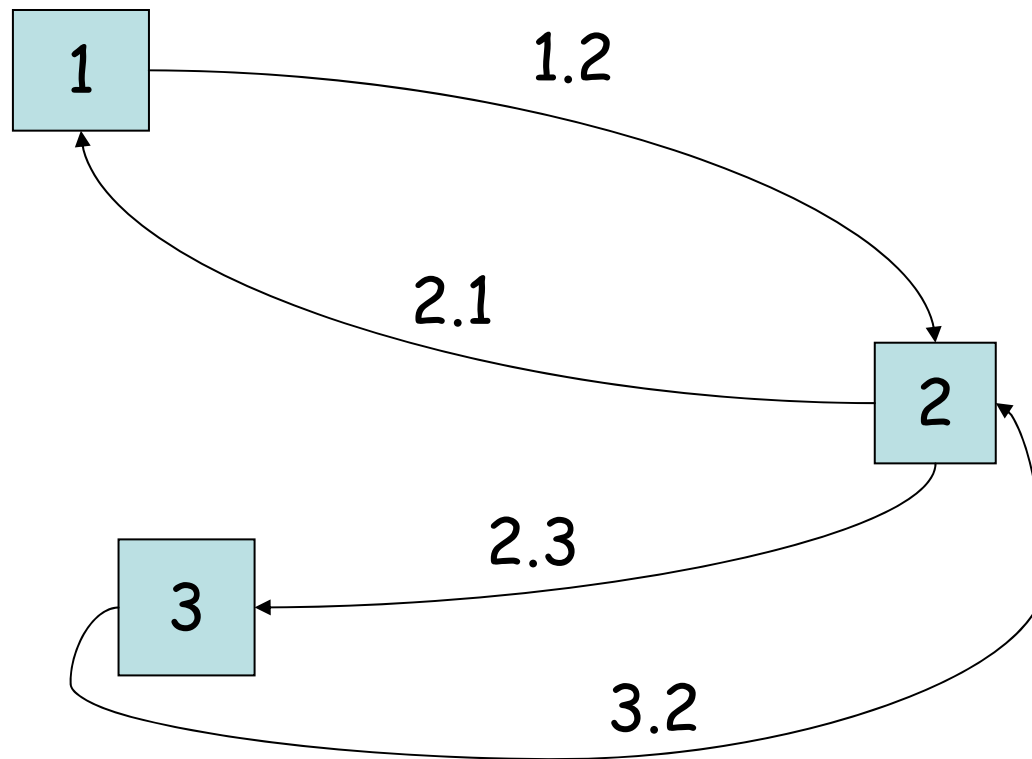


# Modelling cooperating state machines

User Client Agent Server

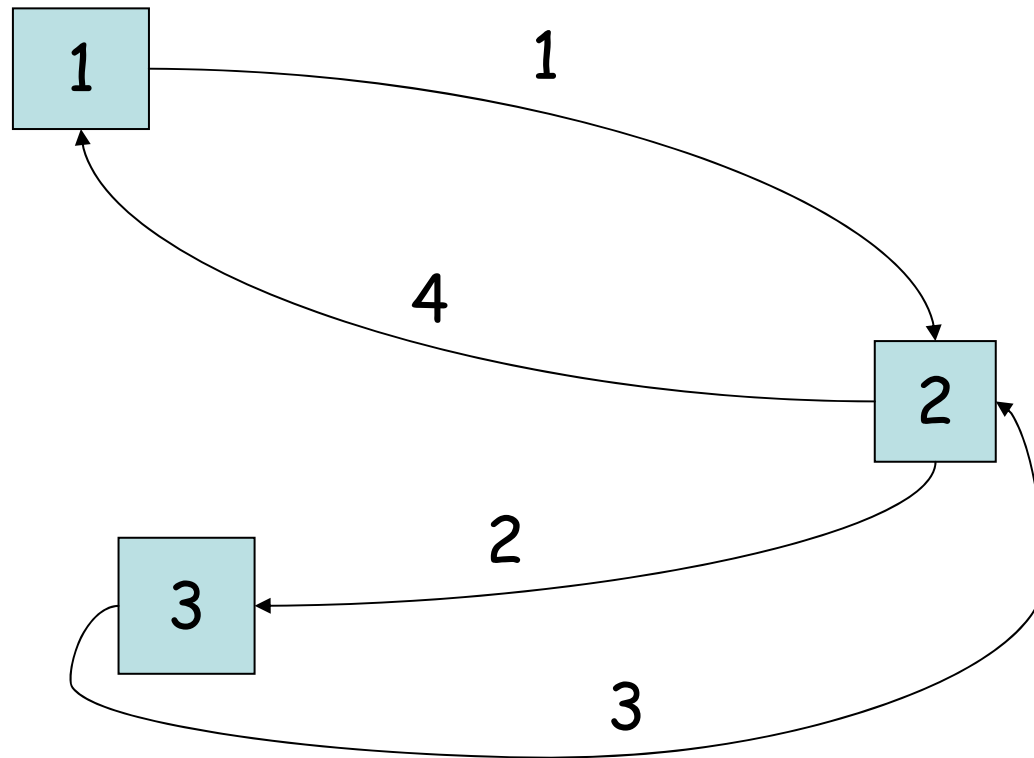


# Composition



Numbering/naming of nodes and links/connections

# Communication Diagram



Interaction sequence order numbering

# Decomposition into Parts

Assuming that we have a picture of the functions and the data to handle and an idea about the overall architecture principles:

- Make a decomposition of the system into parts (subsystems or components)
- Decide a how these parts should interwork

# Allocation of Functions and Data

- Whether a specific function or data should belong to one part or to another is difficult to decide and the preferred choice may change when more information has been gathered
- You can't design a competitive and needed product without knowledge and there is a lot that you must get knowledge about

# Data and Function Cohesion

- By analysis of the relationships between data instances we can often identify clusters of closely related data
- Functions associated with and applicable to such data clusters can then be identified

# Data and Function Decoupling

- Look for borders where the coupling between the clusters are small
- A decoupled cluster of related data and functions (use cases) is a good candidate or base for a component
- Such components with low coupling can be distributed without too negative effects on the communication costs

# Component Interfaces

- Having identified potential components we should now think about the interfaces needed between these
- An interface should decouple components from each other
- The interface of a component should hide its internals

# Interfaces

- An interface must only give access to the services, functions or data that the component want to provide
- Interfaces should be designed with the future in mind and for backwards compatibility etc.
- Suitable interfaces is key to long term success and usefulness of a component

# Component

1. Part that is replaceable and useful in several system contexts
2. Have (or can access) knowledge about other component types and services
3. Lack knowledge about component instances and connections at design time
4. Provides a service interface and uses such interfaces
5. Is adapted to its environment by passing parameters via interface or ports at compile, linking, load or run time
6. Communicate its data via output ports (or is directed via name servers) to other components input ports

# HW and SW dependencies

- Analyse and get to know the physical world where the system is intended to work
- Get to know the HW technology base
- Identify relationships to the OS and other SW components (to be used)
- Some applications work at higher, less HW dependent levels
- Other applications work at lower more HW specific and HW dependent levels
- HW and SW components can be regarded as implementation level actors related by use cases at their specific level

# HW and SW dependencies, cont.

- Lower level SW handle:
  - interrupts;
  - memory management and addressing issues;
  - cooperation with HW units via drivers and communication buffers
- In early phases of a development project we should identify the dependencies between SW and HW (but save the solution to later phases)
- Embedded SW components mainly interact with:
  - surrounding HW represented by drivers
  - other SW components
  - supporting SW platform

# Development effort estimation

- Make an estimate of the size of the project required to finish a first version of the product
- Partition the system into reasonably sized parts or work packages
- For a large system do the partitioning in several steps until the system complexity is conquered
- Then, based on previous experiences, estimate the development cost for each small part
- Estimate also other uncertainties and risks

# Design Patterns

- **Documented problem-solution pairs** aimed to be reused in different contexts where similar (matching) problems exist
- **Guidelines** aimed to simplify the assignment of responsibilities to objects
- **Describe architecture concepts** such as separation of concerns by modularization and/or layering

# Regularity and repetition

- Design patterns make it easier to see and talk about commonly occurring problems and their solutions
- By using patterns repeatedly, simple regular structures can be designed
- Such regular structures are easy to maintain

# Examples of design patterns

- Client-Server
- Presentation - Application Logic – Data Store (3-tier, client-server-DB)
- Module (cohesive loosely coupled)
- Layer (above layer)
- Command handler
- Resource handler

# Examples of design patterns, cont.

- Information handler
- Low coupling
- High cohesion
- Creators and factories
- Controllers and coordinators
- Interface, adapter, facade and proxy

# Cross cutting concern

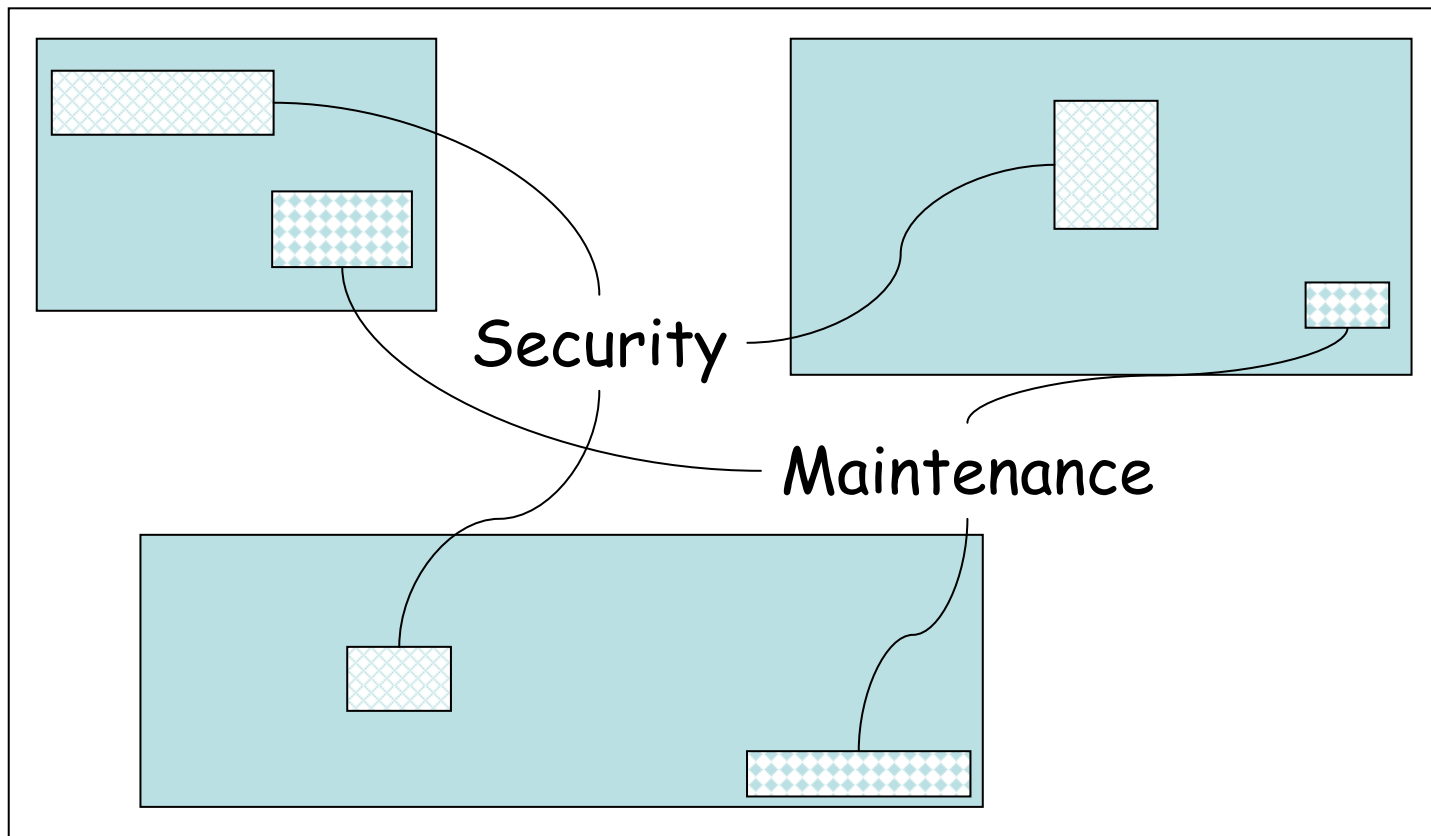
- There are several problems that influence a system as a whole
- Such problems are called cross cutting concerns or aspects

# Cross cutting security concerns

- Security is a matter that may cut across application module boundaries
- Security related code must be scattered into other application code
- Changing the security code then becomes hard, because it has to be changed in many places
- This spreading of the security code also compromise maintainability and reuse

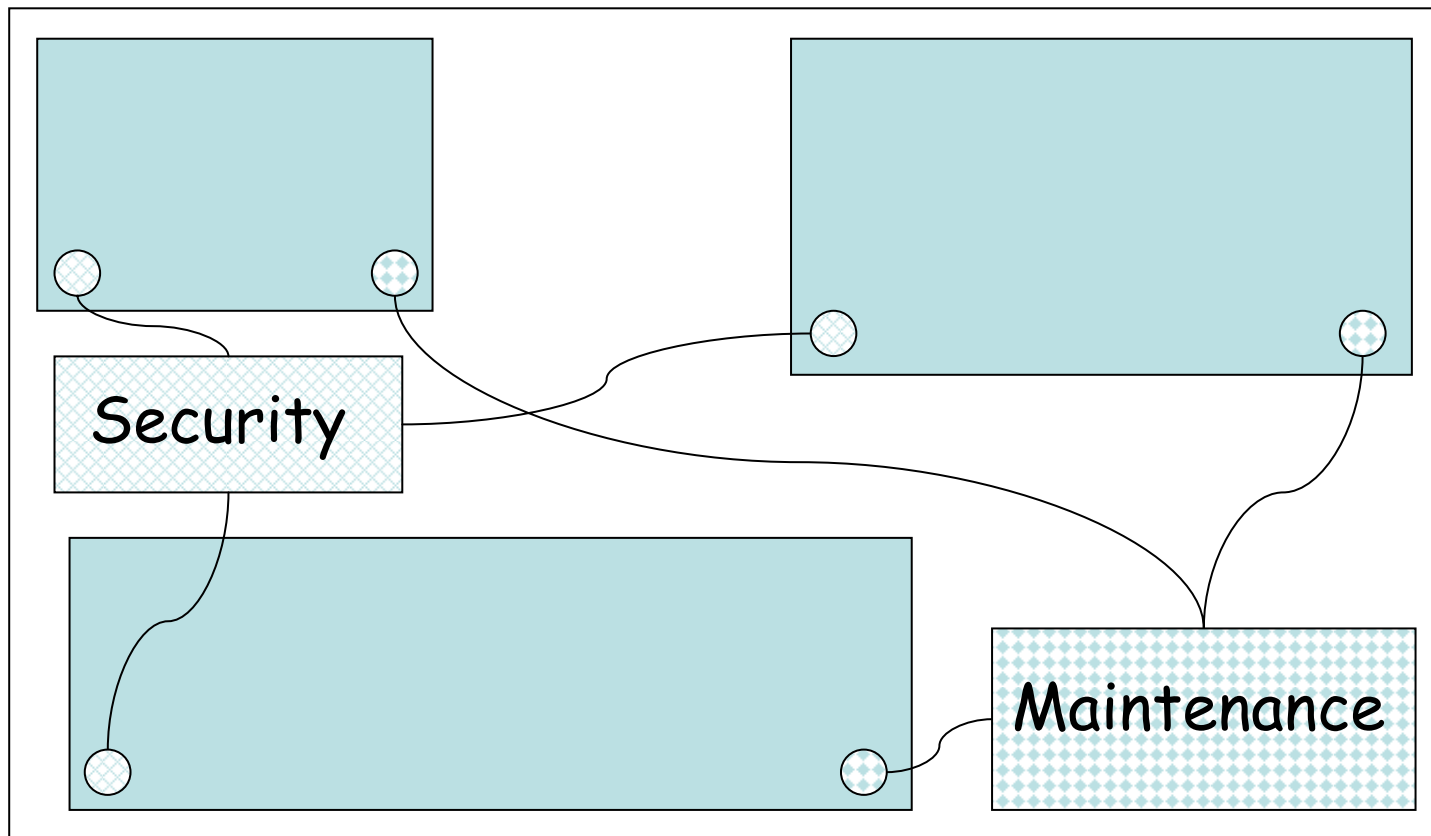
# Major and Minor Concerns

Code for some aspects or concerns has a tendency to be spread across the system over many of its components



# Modularized Concerns

Ideally we want the concern specific code to be gathered in a single and dedicated module



# Aspect Oriented Programming

- AOP develops cross cutting concerns, or aspects, as isolated issues
- AOP combines solutions to these issues with application modules at build or run time using a technique called “weaving”

# Weaving

Traditional code linking can resolve and connect method and variable names

“Weaving” is linking that adds the ability to:

- replace method bodies with new implementations,
- insert code before and after method calls,
- instrument variable reads and writes, and
- modify class structures by adding new methods, fields, parent classes and interfaces

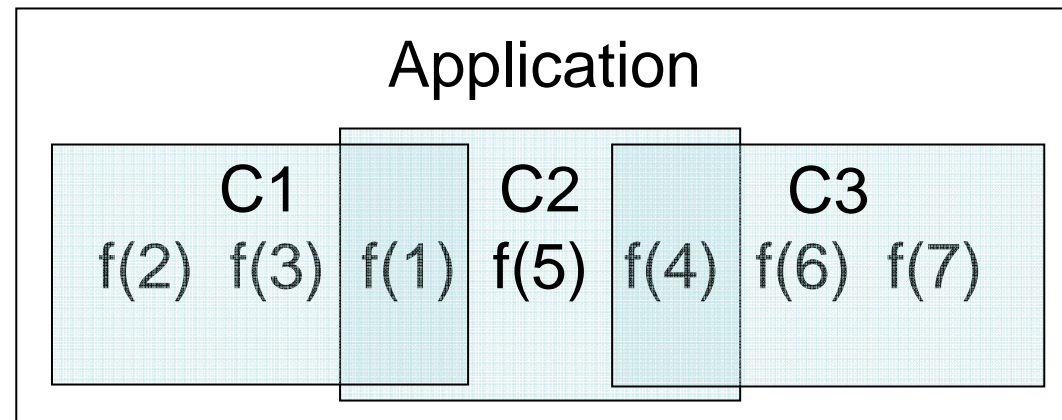
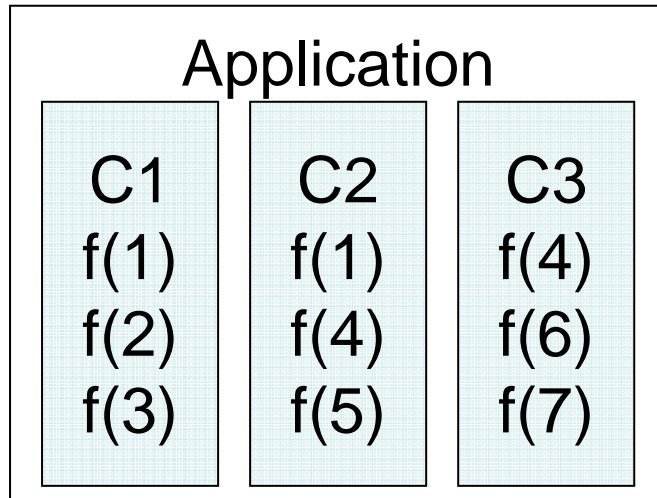
# SW product sharing

- SW component products must be possible to add and remove as needed
- Many SW techniques requires that we package (bundle) and load all code related to a component
- However, a smart loader will only load code not already installed, enforcing code sharing between components
- To support updates of such SW components the loader must be able to distinguish between different versions of the code

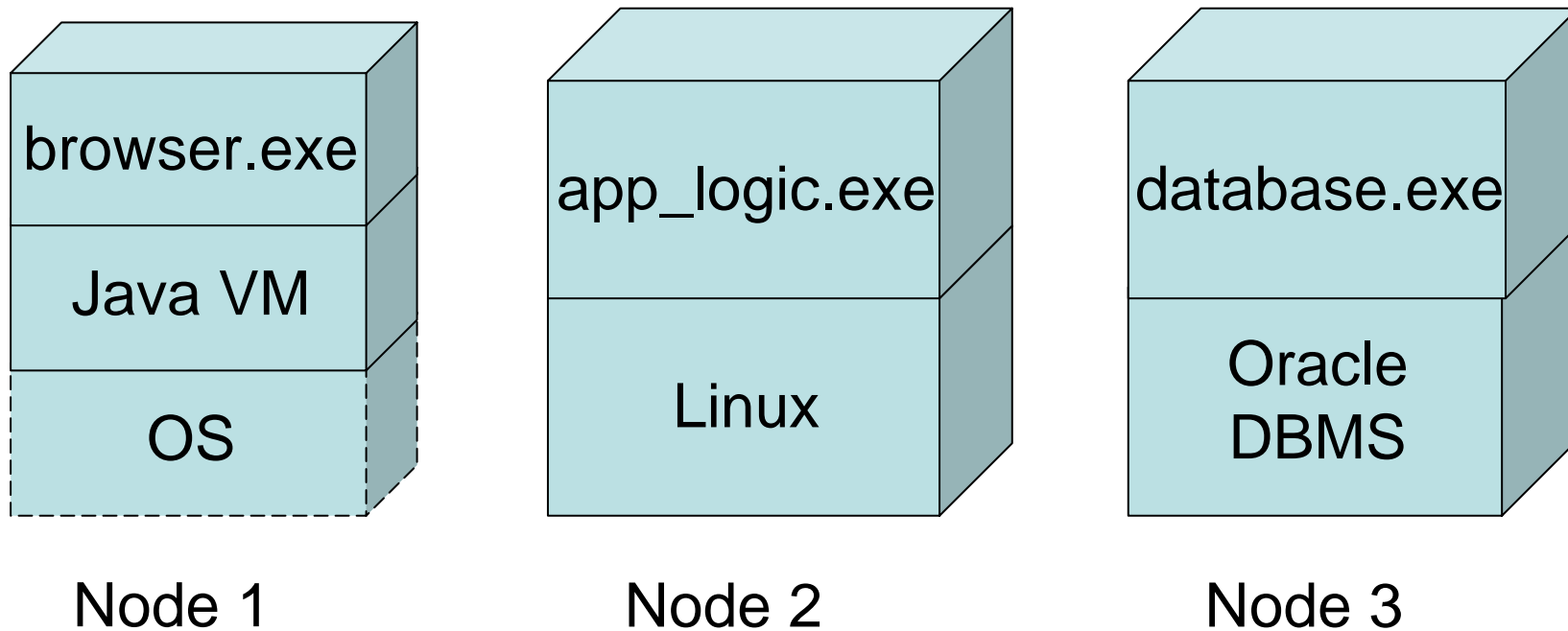
# Packaging

- To package software as products or components in a proper way we must know what the programming language or platform technique supports
- We don't want to change the name of a function (method) just because we have to make an update or error correction
- It is practical if the package loader can distinguish between versions of the same function (for example by use of identities hidden to the programmer)

# Code sharing between packets



# Deployment Diagram



Static view of the run-time configuration of physical (processing) nodes and allocated SW components

# UML 2.0 – 13 diagram types

## **Structural**

- Capturing the static properties of classes, objects, interfaces and physical components and their relationships and dependencies

## **Behavioural**

- Capturing the dynamic message interaction and state changes within and between modelled items

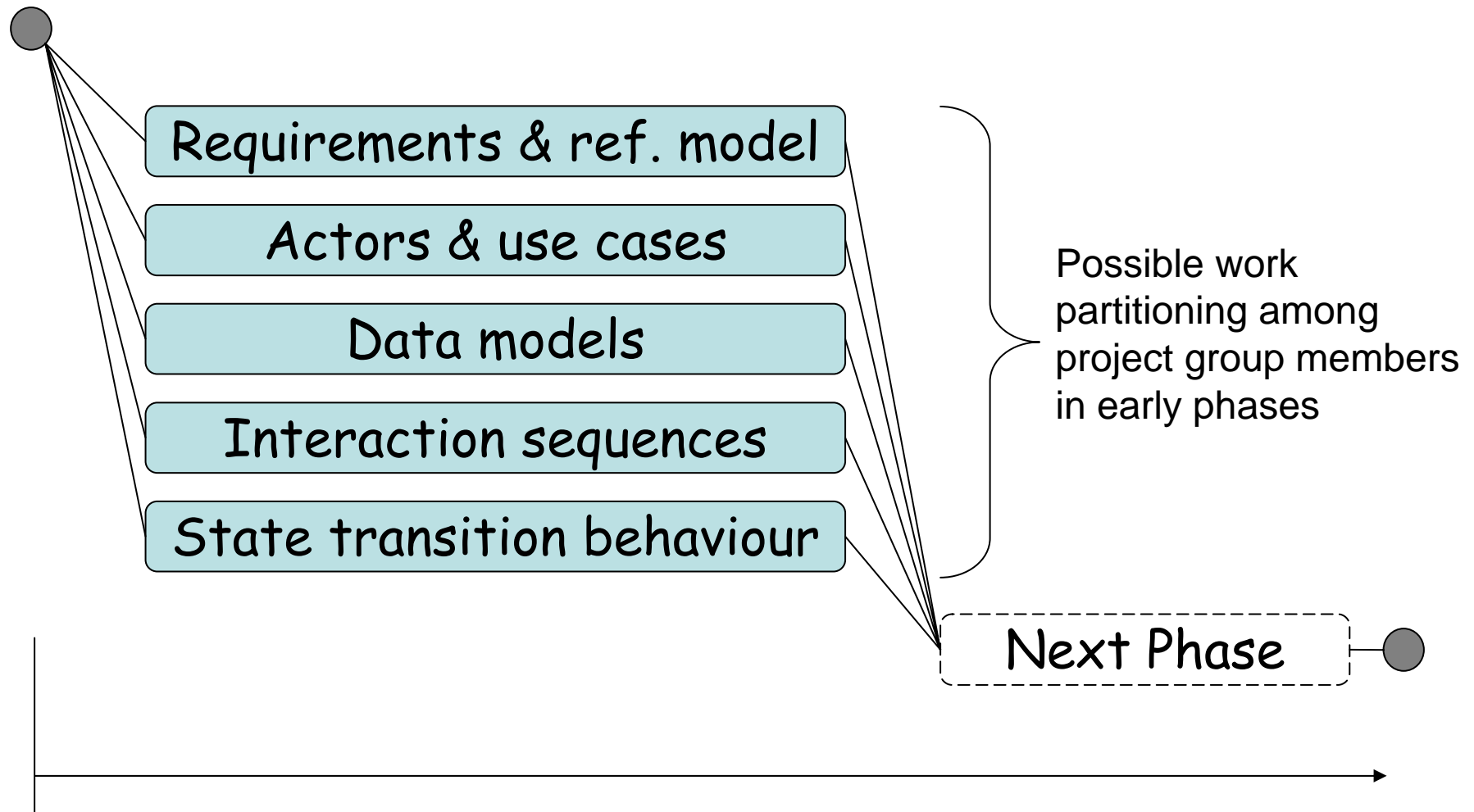
# Structural Modelling (6 Diagrams)

1. Package
2. Class
3. Object
4. Composite Structure
5. Component
6. Deployment

# Behavioural Modelling (7 Diagrams)

1. Use Case
2. Activity
3. State Machine
4. Communication
5. Sequence
6. Timing
7. Overview

# Concurrent Engineering



# Interface Design

- An interface defines a boarder line to an item (system or component) or between items
- An interface provides and external view of the encapsulated internals of and item
- An interface hide or abstract internal details, not to be known outside the system
- Items handling different concerns shall only interchange the information needed to solve their task or tasks, across their interfaces

# Protocols

- Many different protocols are used when interchanging information between systems and parts of systems, such as software tasks
- The protocols needed depend on the kind of systems or parts that needs to communicate as well as the distances and medias separating the communicating parties

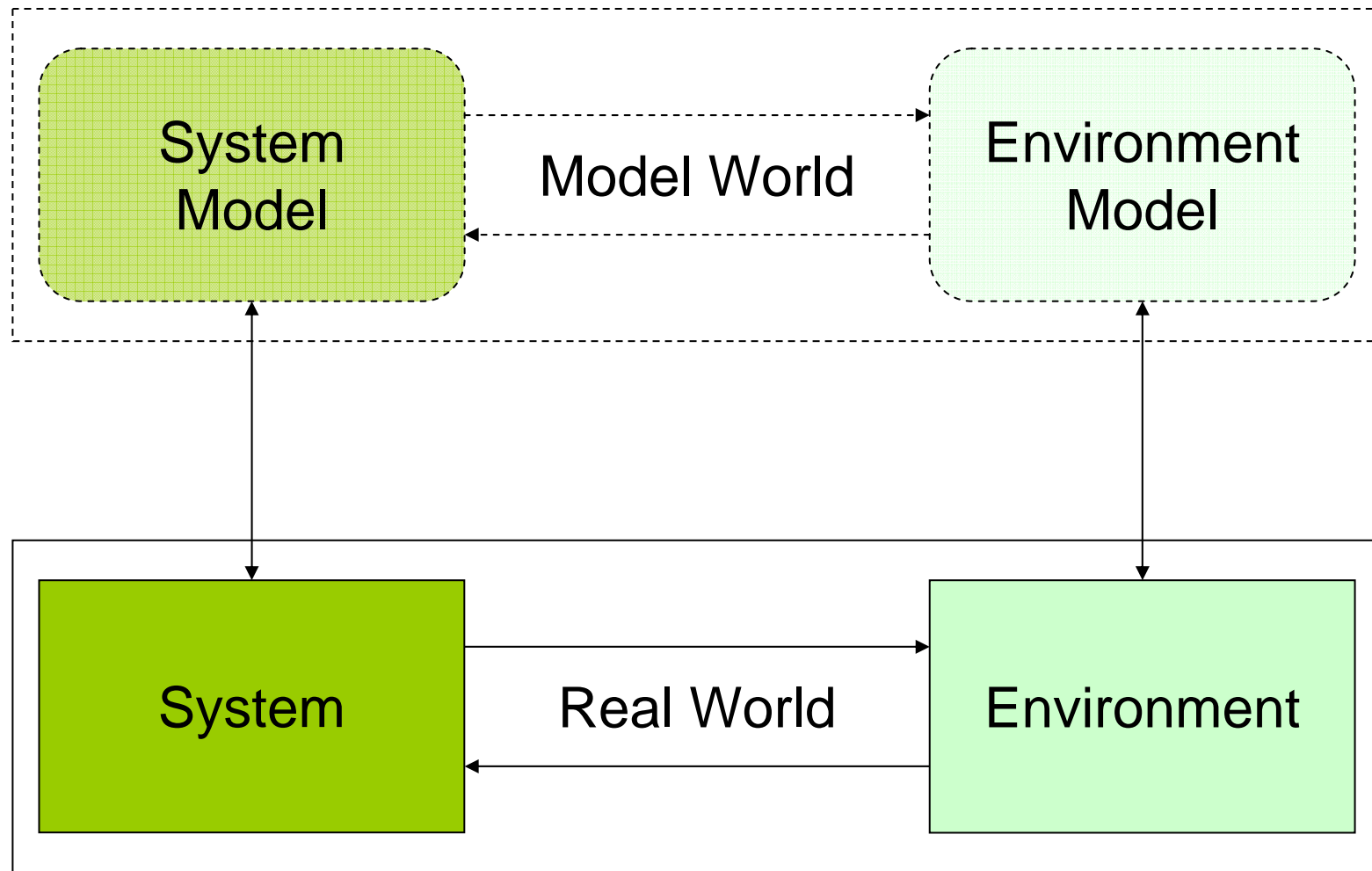
# Protocols, cont.

- Protocols are made to help exchange information between two (and sometimes several) communicating parties
- In general a protocol is defined by its information content, its formatting and the ordering of the information exchange
- The OSI model divides protocols in 7 layers
- Protocols can be nested such that a lower layered protocol carries a higher layer

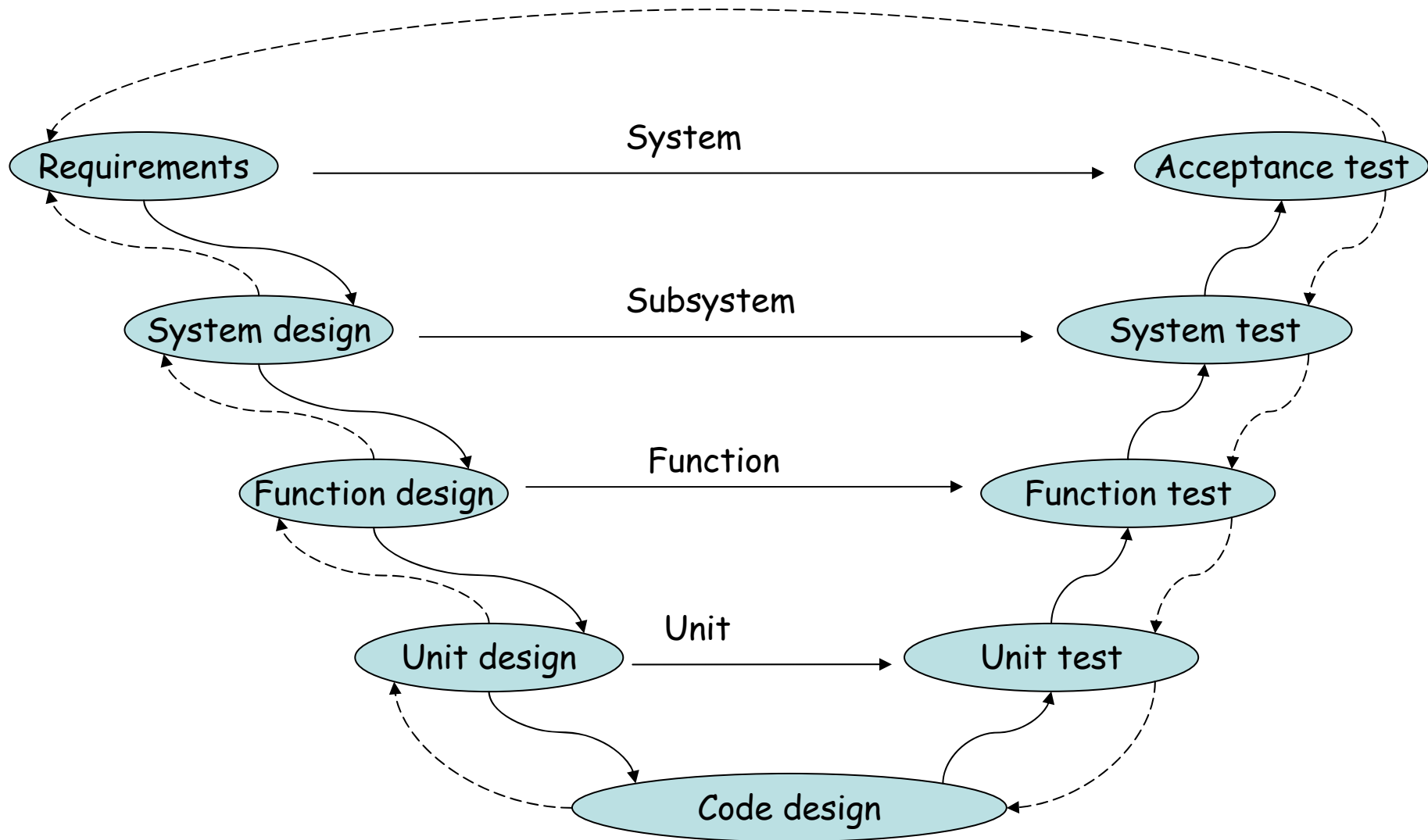
# Protocol, cont.

- All protocols include formatting issues
- A session level protocol such as HTTP includes formatting rules for its own messages but is also used to carry HTML formatted presentation information, to be visualized in proper temporal order

# Real world and model world



# The V model



# Test Plan

- To achieve desired quality, the test plan must cover all discriminating cases that can occur in real life
- The test plan divide the system under test in different function areas and characteristics
- Cover each area by suitable test cases
- To be economical, perform the tests in a suitable order
- Catch problems early to simplify debugging

# Test Cases

Test (x)

State(i) and Pre\_Cond(m) and Event(x)

-----

State(j) and Post\_Cond(n) and Event(y)

## Coverage:

- Not realistic to cover all data value combinations
- But cover all major branch conditions, states, state transitions and related state controlling events

# Test Cases

**Test(x)**

State(a)

Event(\_)

Pre\_Cond(\_) → State(\_), Post\_Cond(\_),

Event(\_)

Pre\_Cond(\_) → State(\_), Post\_Cond(\_), State(b)

.

.

.

Event(\_)

Pre\_Cond(\_) → State(\_), Post\_Cond(\_),

**Test(y)**

# Test Case: Tree Structure

