



## Lecture 2

# REQUIREMENTS SPECIFICATION

PB007 Software Engineering I  
Faculty of Informatics, Masaryk University  
Fall 2020

# Requirements engineering (RE)



- ✧ **Requirements** are descriptions of system **services** and **constraints** under which the system operates and is developed.
  - It may range from a **high-level abstract statement** of a service to a detailed **mathematical functional specification**.
- ✧ **Requirements engineering** is the **process** of establishing requirements.

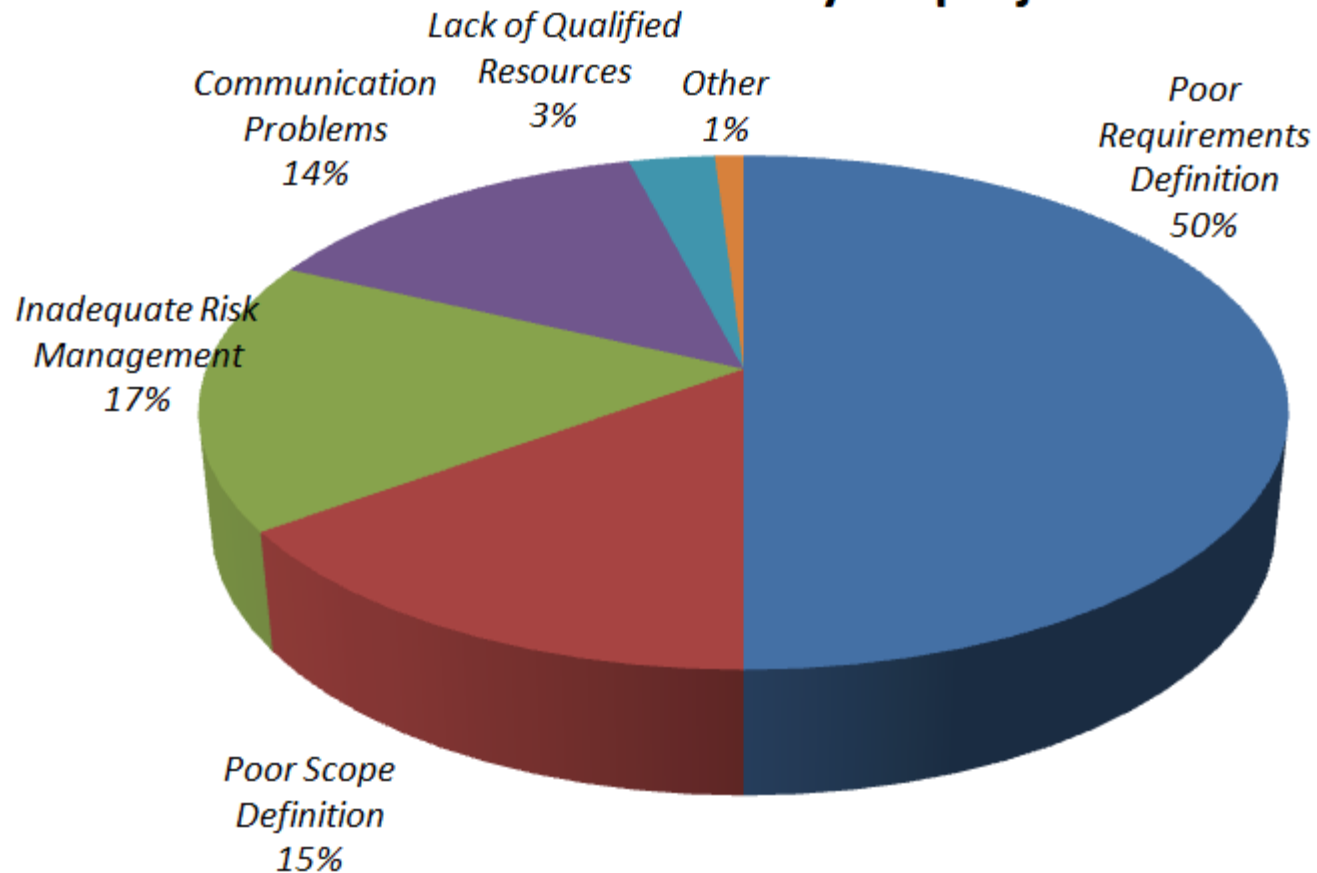
We know the UML Use Case diagram and related process by now.  
Why is that not enough?

Do we really need to be that serious about it? And what about agile?

# Motivation



## Why do projects fail?



Source: ESI International survey of 2000 business professionals, 2005

# Outline

---



- ✧ Requirements and their types
- ✧ Requirements engineering process
- ✧ Non-functional requirements
  
- ✧ UML Activity diagram



---

# Requirements and their Types

## Lecture 2/Part 1

# Functional and non-functional requirements



## ✧ Functional requirements

- Statements of **services the system provides**, how the system should react to particular inputs and how the system should **behave** in particular situations.
- E.g. *A user **shall be able to** search the appointments lists for all clinics.*

## ✧ Non-functional requirements

- Properties and **constraints on the services** offered by the system such as timing, reliability and security constraints, constraints on the development process, platform, standards, etc.
- E.g. *The system **shall be available** on Mon–Fri, 8 am – 5 pm, with downtime not exceeding five seconds in any one day.*

Can you think of more examples of the two types?

# Requirements quality criteria



## ✧ Complete

- Are **all functions** required by the customer **included**?

## ✧ Consistent

- Are there any requirements **conflicts** or **contradictions**?

## ✧ Precise

- Is there **one and only interpretation** in the system context?

## ✧ Verifiable

- Can the requirements be **checked**?

## ✧ Realistic

- Can the req. be **implemented** with the available resources, such as budget, time and technology?



---

# Requirements Engineering Process

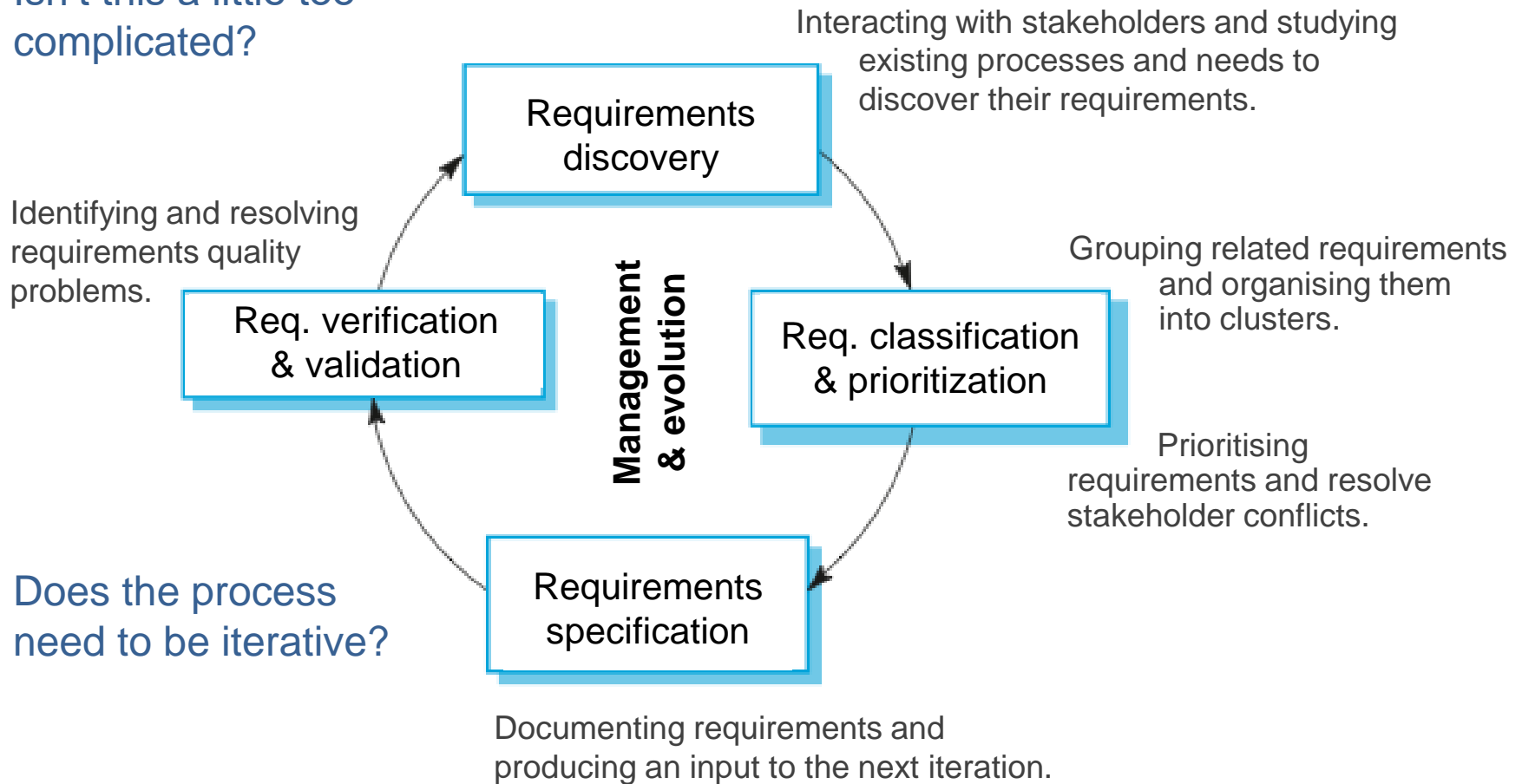
## Lecture 2/Part 2



# The requirements engineering process



Isn't this a little too complicated?



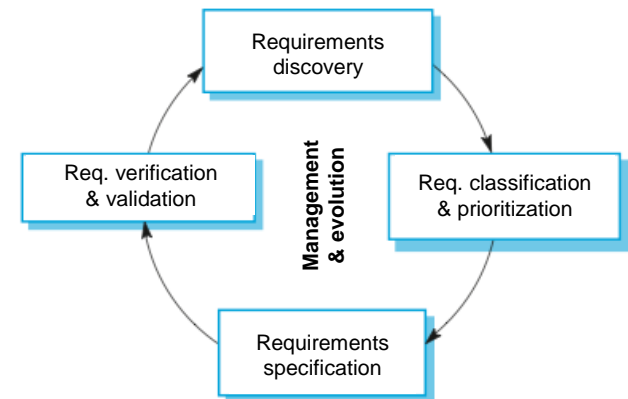
Does the process need to be iterative?

# Requirements discovery



- ❖ Software engineers work with system **stakeholders**:
  - end-users, managers, maintenance engineers, domain experts, trade unions, etc.

- ❖ To find out about:
  - the application domain,
  - the services to provide,
  - the required system performance,
  - hardware constraints,
  - other systems, etc.



- ❖ As far as possible, it should set of **WHAT** the system **should do** rather than **HOW** it will do (implement) it.

# Requirements discovery techniques



## ✧ Questionnaires

## ✧ Interviews

- Small number of software engineers and stakeholders
- **Open interviews** where various issues are explored
- **Closed interviews** based on pre-determined list of questions

## ✧ Workshops

- Free brainstorming of all involved stakeholders

## ✧ Ethnography

- Observe and analyse existing processes, i.e. **how people actually work**, under what social and organizational factors.

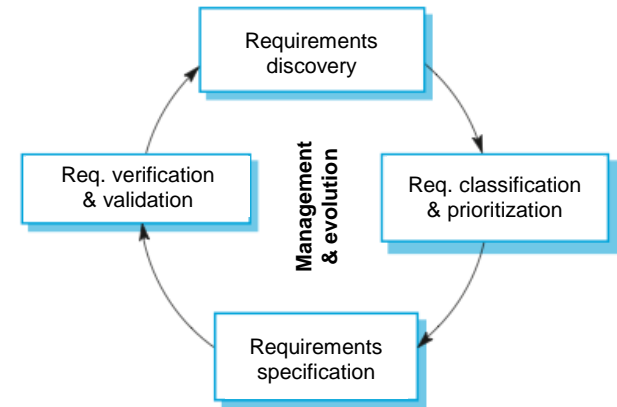
Is there a recommended order if the techniques are combined?

# Requirements classification & prioritization



## ✧ MoSCoW criteria

- **Must have** – mandatory, fundamental
- **Should have** – important, may be omitted
- **Could have** – truly optional
- **Want to have** – can wait for later releases



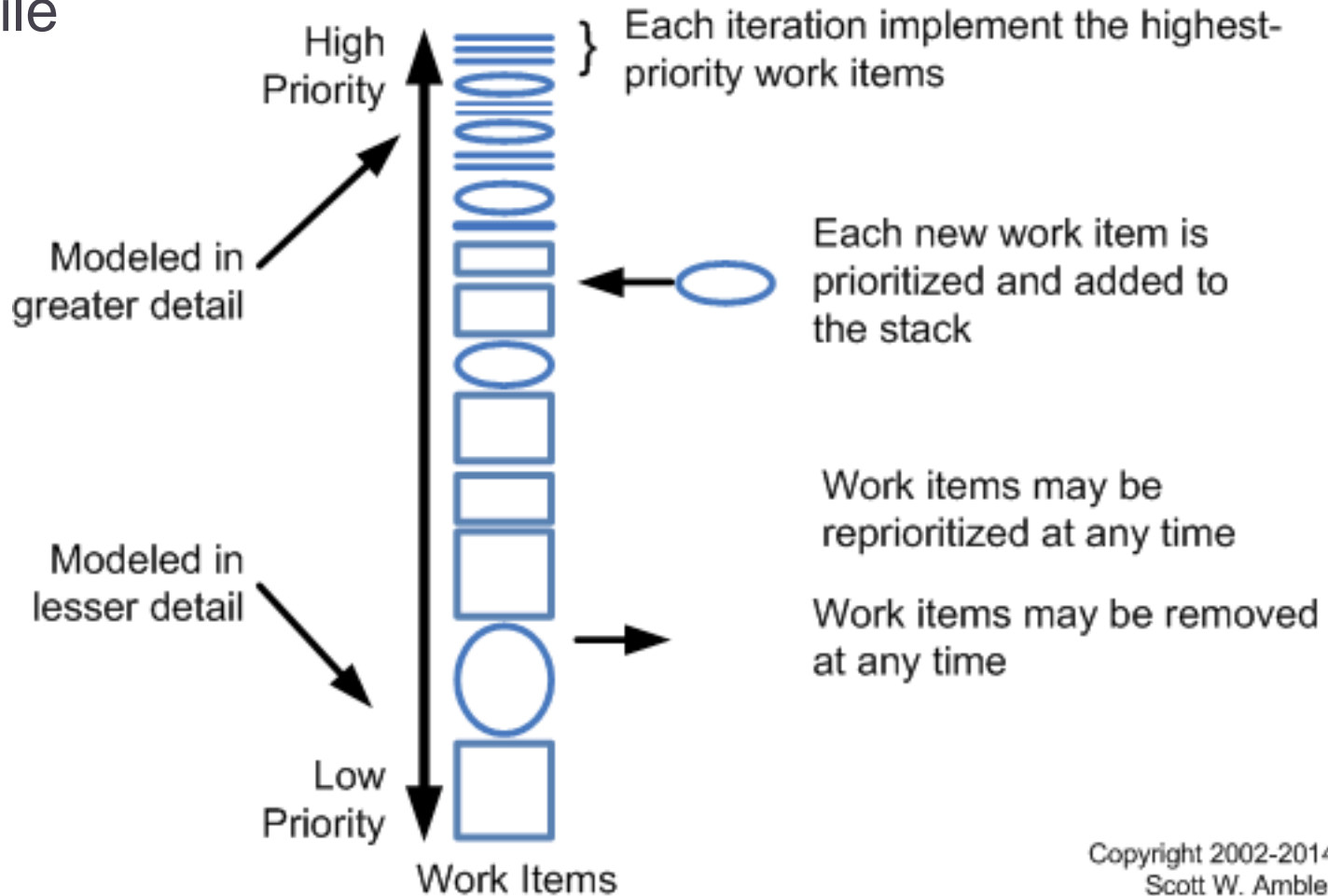
## ✧ Rational Unified Process (RUP) attributes

- **Status** – Proposed/Approved/Rejected/Incorporated
- **Benefit** – Critical/Important/Useful
- **Effort** – number of person days/functional points/etc.
- **Risk** – High/Medium/Low
- **Stability** – High/Medium/Low
- **Target Release** – future product version

# Requirements classification & prioritization



## ✧ Agile

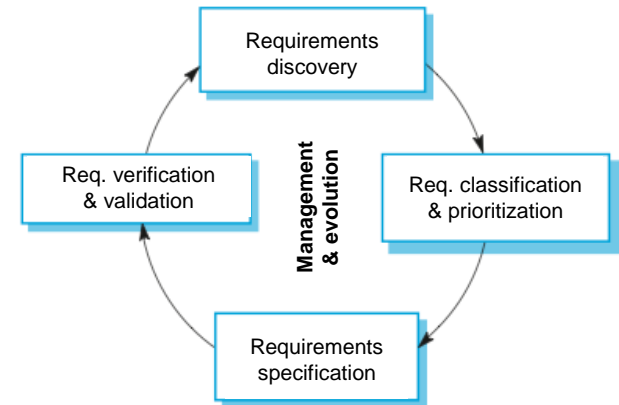


Copyright 2002-2014  
Scott W. Ambler

# Requirements specification



- ✧ Natural language
  - E.g. project assignment
- ✧ Structured language
  - E.g. textual spec. of UML use cases
- ✧ Graphical notation
  - E.g. UML Use Case or Activity diagram
- ✧ Mathematical specification
  - E.g. finite state machines



Is there any use for the mathematical specification?

# Requirements verification & validation



## ✧ Requirements verification

- Concerned with checking requirements **completeness, consistency, preciseness, verifiability** and **realism**.

## ✧ Requirements validation

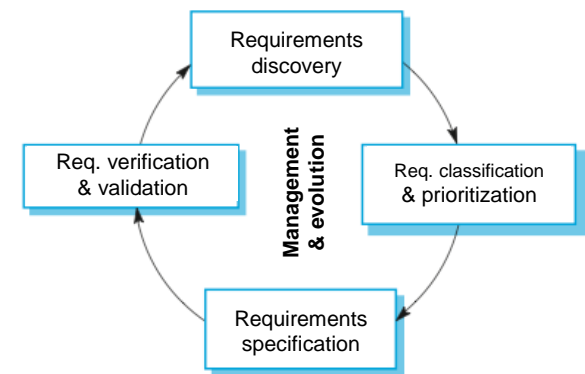
- Concerned with checking that the requirements define the system **that the customer really wants/needs**.

## ✧ Techniques

- Requirements reviews
- Prototyping, A/B testing

Who should be involved?

Why A/B testing?



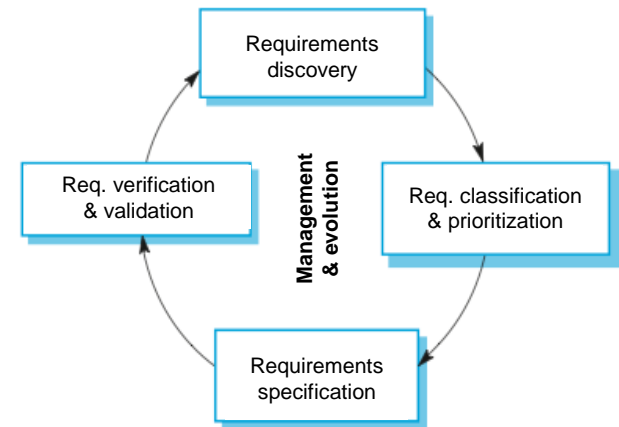
Remember that fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

# Requirements management and evolution



## ✧ Requirements management

- Process of **managing changing requirements** during the requirements engineering process and system development.
- Each requirements **change should be analysed** before deciding whether to accept it.



## ✧ New requirements emerge due to

- **Business, organizational and technical changes.**

## ✧ Traceability

- Maintenance of **links between dependent requirements** is important to assess the impact of requirements changes.





# Non-functional Requirements

## Lecture 2/Part 3

# Functional and non-functional requirements



## ✧ Functional requirements

- Statements of **services the system should provide**, how the system should react to particular inputs and how the system should **behave** in particular situations.

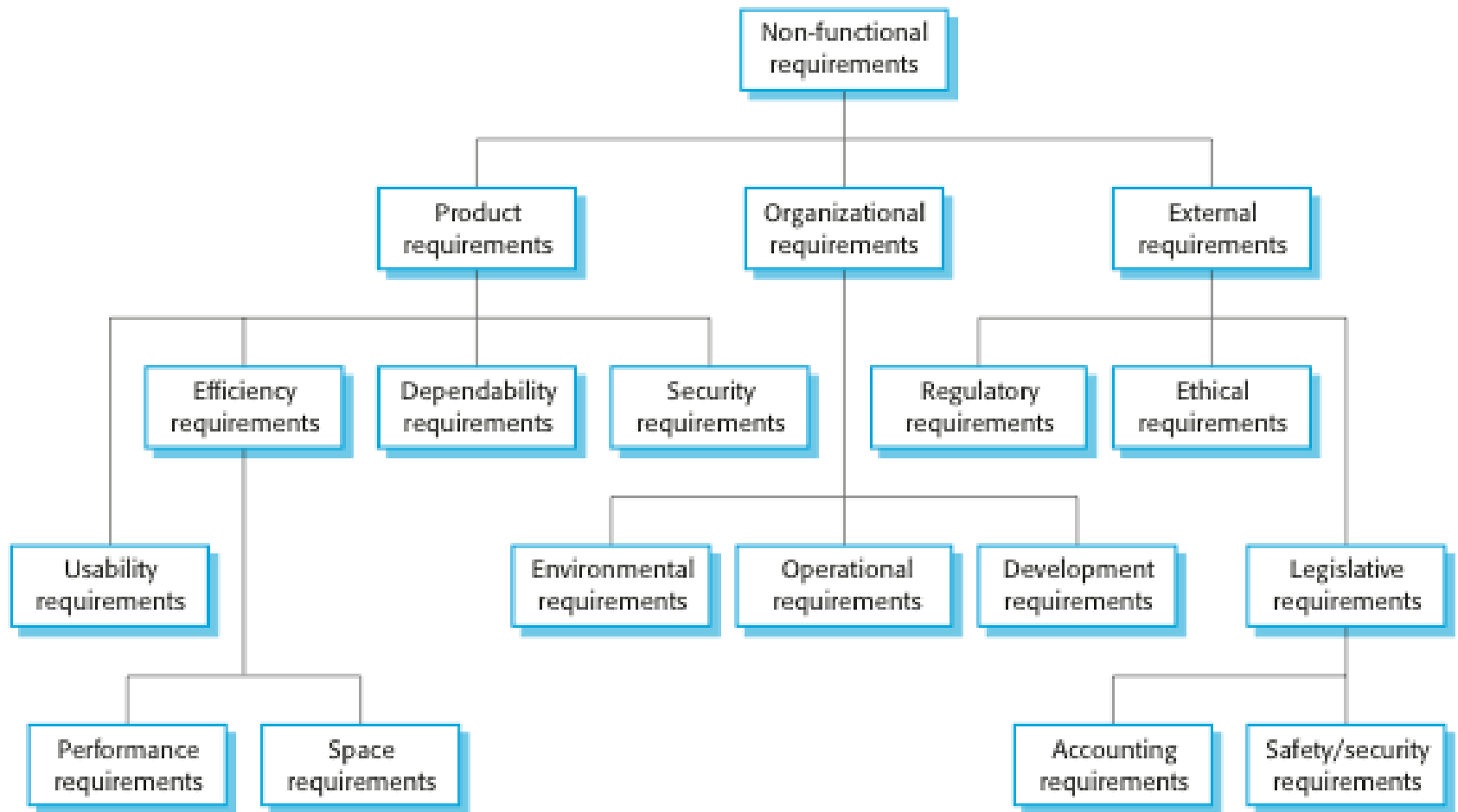
## ✧ Non-functional requirements Why are they so important?

- Properties and **constraints on the services** offered by the system such as timing, reliability and security constraints, constraints on the development process, platform, standards, etc.

## ✧ **Non-functional requirements** help us to define

- **Quality** of the software product
- **Conformance** to its context (organization and legislation)

# Non-functional requirements classification



# Non-functional requirements classification



## ✧ Product requirements

- Requirements which specify that the delivered product must behave with a certain **quality** e.g. execution speed, reliability, etc.

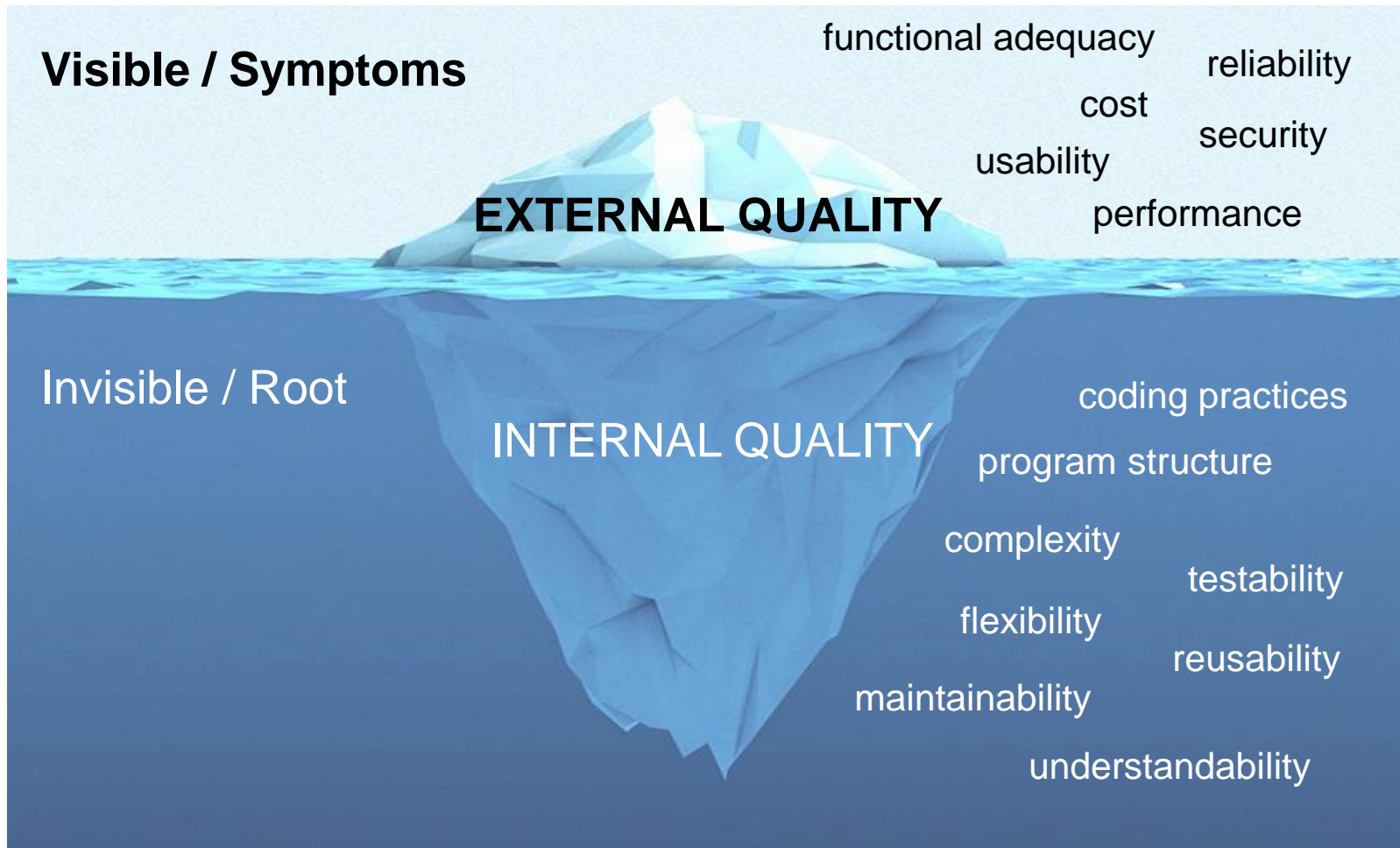
## ✧ Organisational requirements

- Requirements which are a consequence of **organisational policies and procedures** e.g. process standards used, implementation requirements, etc.

## ✧ External requirements

- Requirements which arise from **factors which are external** to the system and its development process e.g. various **legislative requirements**.

# Product requirements and SW Quality



# Product requirements



## ✧ Dependability

- Availability
- Reliability
- Safety
- Security

## ✧ Efficiency

- Performance
- Space/resource utilization

## ✧ Usability

## ✧ Modifiability

## ✧ Resilience, Robustness

## ✧ Portability

## ✧ Adaptability

## ✧ Complexity

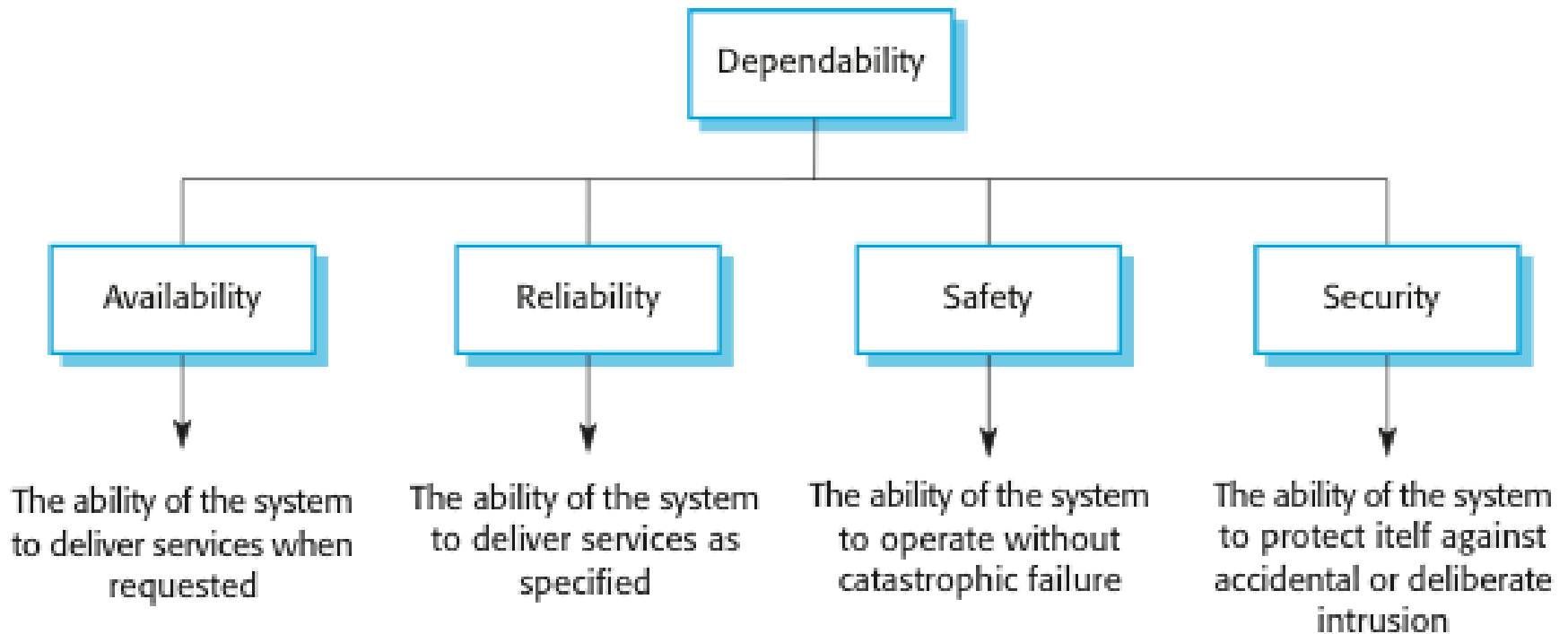
## ✧ Modularity, Reusability

## ✧ Understandability

## ✧ Learnability

## ✧ Testability

# Principal dependability attributes



# Availability



- ✧ The probability that a system, at a point in time, will be operational and able to deliver the requested services
  
- ✧ Concerned with
  - How long the system should be operating without a failure.
  - How long a system is allowed to be out of operation.
  
- ✧ Can be expressed quantitatively
  - Using mean time to failure (MTTF) and repair (MTTR) as  $MTTF / (MTTF + MTTR)$ .
  - I.e. availability of 0.999 means that the system is up and running for 99.9% of the time. **Can you explain of what time?**
  - **Can MTTF and MTTR be derived from the defined availability?**



# Reliability



- ✧ The probability of failure-free system operation over a specified time in a given environment for a given purpose
  
- ✧ Concerned with
  - How system fault/error/failure is detected.
  - How frequently system fault/error/failure may occur.
  - What happens when a fault/error/failure occurs.
  
- ✧ Can be expressed quantitatively
  - Using the probability of failure on demand (POFOD) within a single service or usage scenario execution, as  $1 - \text{POFOD}$ .
  - Could MTTF and MTTR be also used here?

# Reliability terminology



Term	Description
Human error or mistake	Human behavior that results in the introduction of faults into a system. <i>E.g., in the wilderness weather system, a programmer might decide that the way to compute the time for the next transmission is to add 1 hour to the current time. This works except when the transmission time is between 23.00 and midnight .</i>
System fault	A characteristic of a software system that can lead to a system error. <i>E.g., the inclusion of the code to add 1 hour to the time of the last transmission, without a check if the time is greater than or equal to 23.00.</i>
System error	An erroneous system state that can lead to system behavior that is unexpected by system users. <i>E.g., the value of transmission time is set incorrectly (to 24.XX rather than 00.XX) when the faulty code is executed.</i>
System failure	An event that occurs at some point in time when the system does not deliver a service as expected by its users. <i>E.g., no weather data is transmitted because the time is invalid.</i>

# Safety



- ✧ Safety is a property of a system that reflects the system's ability to operate, normally or abnormally, without danger of causing **human injury or death** and without **damage to the system's environment**.
- ✧ It is important to consider software safety as most devices whose failure is critical now incorporate software-based **control systems**.
- ✧ Safety requirements are often exclusive requirements i.e. they **exclude undesirable situations** rather than specify required system services. These generate functional safety requirements.

# Safety terminology



Term	Definition
Accident (or mishap)	An unplanned event or sequence of events which results in human death or injury, damage to property, or to the environment. <i>E.g. an overdose of insulin.</i>
Hazard	A condition with the potential for causing or contributing to an accident. <i>E.g. a failure of the sensor that measures blood glucose.</i>
Damage	A measure of the loss resulting from a mishap. Damage can range from many people being killed as a result of an accident to minor injury or property damage. <i>E.g., damage resulting from an overdose of insulin could be serious injury or the death of the user of the insulin pump.</i>
Hazard severity	An assessment of the worst possible damage that could result from a particular hazard. <i>E.g. when an individual death is a possibility, a reasonable assessment of hazard severity is 'very high'.</i>
Hazard probability	The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from 'probable' (say 1/100 chance) to 'implausible'. <i>E.g. the probability of a sensor failure in the insulin pump that results in an overdose is probably low.</i>
Risk	This is a measure of the probability that the system will cause an accident. The risk is assessed by considering the hazard probability, the hazard severity, and the probability that the hazard will lead to an accident. <i>E.g. the risk of an insulin overdose is probably medium to low.</i>

# Security



- ✧ A system property that reflects the system's ability to **protect itself** from accidental or deliberate external attack.
- ✧ Defends the system against:
  - Threats to the **confidentiality** of the system and its data
    - Can disclose information to people or programs that do not have authorization to access that information.
  - Threats to the **integrity** of the system and its data
    - Can damage or corrupt the software or its data.
  - Threats to the **availability** of the system and its data
    - Can restrict access to the system and data for authorized users.

Security is an essential pre-requisite for availability, reliability and safety.

# Security terminology



Term	Definition
Asset	Something of value which has to be protected (e.g. <i>patients records in a health-care system</i> ). The asset may be the system itself or data used by that system.
Exposure	Possible loss or harm to a computing system (e.g. <i>financial loss from patients' legal action or loss of reputation</i> ). This can be loss or damage to data, or can be a loss of time and effort if recovery is necessary after a security breach.
Vulnerability	A weakness in a computer-based system that may be exploited to cause loss or harm (e.g. <i>weak password</i> ).
Attack	An exploitation of a system's vulnerability. Generally, this is from outside the system and is a deliberate attempt to cause some damage.
Threats	Circumstances that have potential to cause loss or harm. You can think of these as a system vulnerability that is subjected to an attack (e.g. <i>guessing the weak password</i> ).
Control	A protective measure that reduces a system's vulnerability. E.g. <i>encryption is an example of a control that reduces a vulnerability of a weak access control system, or a password checking system in our example</i> .

# Dependability attribute dependencies



## ✧ Questions

- Can a reliable system be unavailable? And vice versa?
- Can you give an example of an **unreliable & safe** system?
- Can you give an example of an **reliable & unsafe** system?

## ✧ Some facts about dependencies

- Safe system operation depends on the system being available and operating reliably, but not only on it.
- A system may be unreliable because its data has been corrupted by an external attack.
- Service attacks on a system are intended to make it unavailable.
- If a system is infected with a virus, you cannot be confident in its reliability or safety.

# Performance



- ✧ Performance is about **timing** – response time to events (interrupts, messages, requests from users, or the passage of time).
  - For a web-based financial system, the response might be the **number of transactions that can be processed in a minute**,
  - or the **expected duration of a single transaction** (given as a random variable).
- ✧ Highly **sensitive to concurrency** effects (number of users, shared resources), hardware, operating system implementation (e.g. scheduler strategy), etc.
- ✧ Often accompanied by characterization of **throughput** and **resource utilization**.



# Usability



- ✧ Usability is concerned with **how easy** it is for the user to **accomplish a desired task** and the kind of user support the system provides.
- ✧ It can be broken down into the following areas:
  - Learning system features.
  - Using a system efficiently.
  - Minimizing the impact of errors.
  - Adapting the system to user needs.
  - Increasing confidence and satisfaction.
- ✧ Always follow Human-Interface Guidelines (HIG) if available (Windows HIG, Mac OS HIG, and others)

# Modifiability



- ✧ Modifiability is about the **cost of change**.
- ✧ **What** can change (the artifact)?
  - The **functions** that the system computes, the **platform** the system exists on (the hardware, operating system, middleware, etc.), the **environment** within which the system operates, etc.
- ✧ **When** is the change made and **who** makes it (the environment)?
  - During **implementation** (by modifying the source code), compile (using compile-time switches), **build** (by choice of libraries), **configuration setup** (by a range of techniques, including parameter setting) or **execution** (by parameter setting).
  - By a **developer**, an end **user**, or a system **administrator**.

# Organisational requirements



## ✧ Development requirements

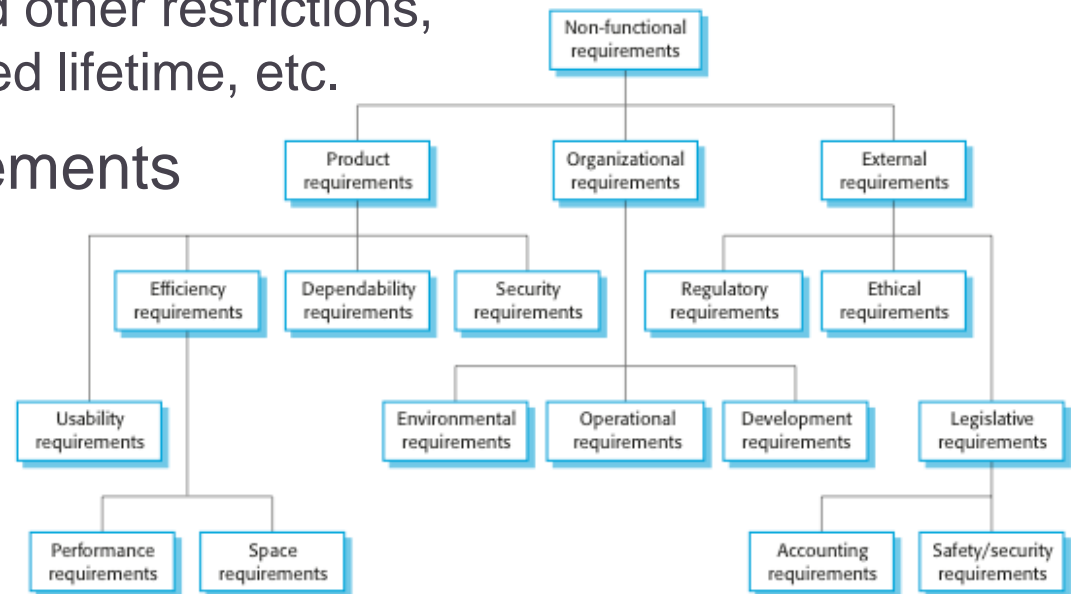
- Programming language, development environment, process standards, time to market, rollout schedule, costs, etc.

## ✧ Operational requirements

- Execution platform and other restrictions, system usage, projected lifetime, etc.

## ✧ Environmental requirements

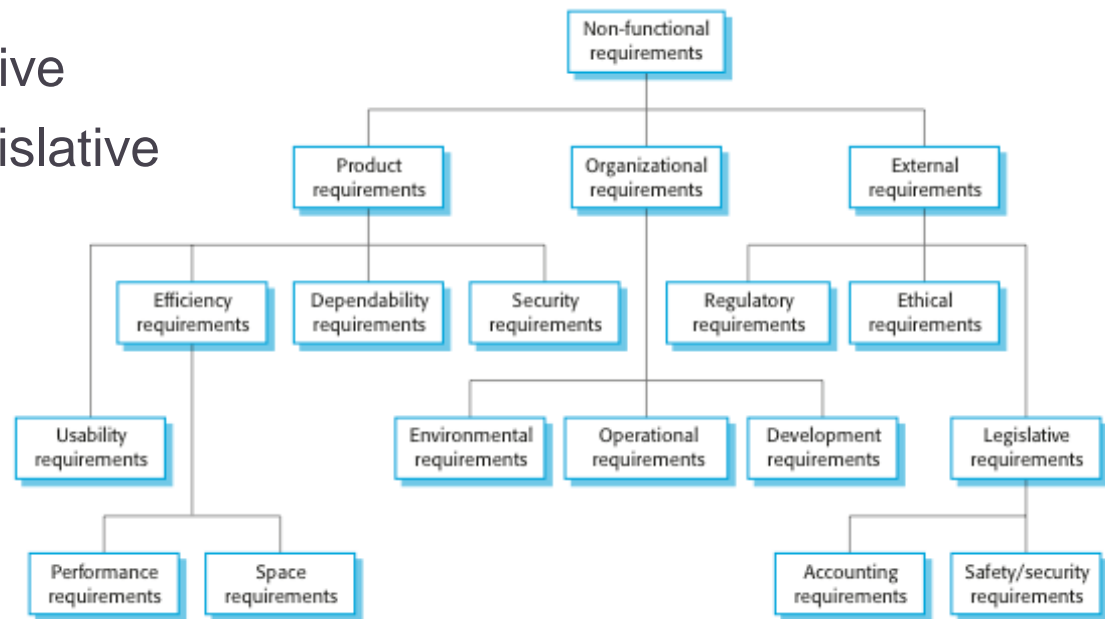
- Integration with legacy systems, targeted market, etc.



# External requirements



- ✧ Regulatory requirements
- ✧ Ethic requirements
- ✧ Legislative requirements
  - Accounting legislative
  - Safety/Security legislative



# Non-functional req. implementation



- ✧ Non-functional requirements may **affect the overall architecture** of a system rather than the individual components.
  - For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
- ✧ A single non-functional requirement, such as a security requirement, may **generate a number of related functional requirements** that define system services that are required.
  - It may also generate requirements that restrict existing requirements.

# Key points



- ✧ Requirements = services + constraints
- ✧ Requirements engineering is an **iterative activity**.
  - Requirements discovery, prioritization, specification, verification & validation, management & evolution.
- ✧ Non-functional quality attributes help us to define the **quality** of the software product.
  - Visible – e.g. availability, reliability, safety, security, performance.
  - Invisible – e.g. modifiability, testability.
- ✧ Consider non-functional requirements right from the **beginning** of your software project



---

# UML Activity Diagram

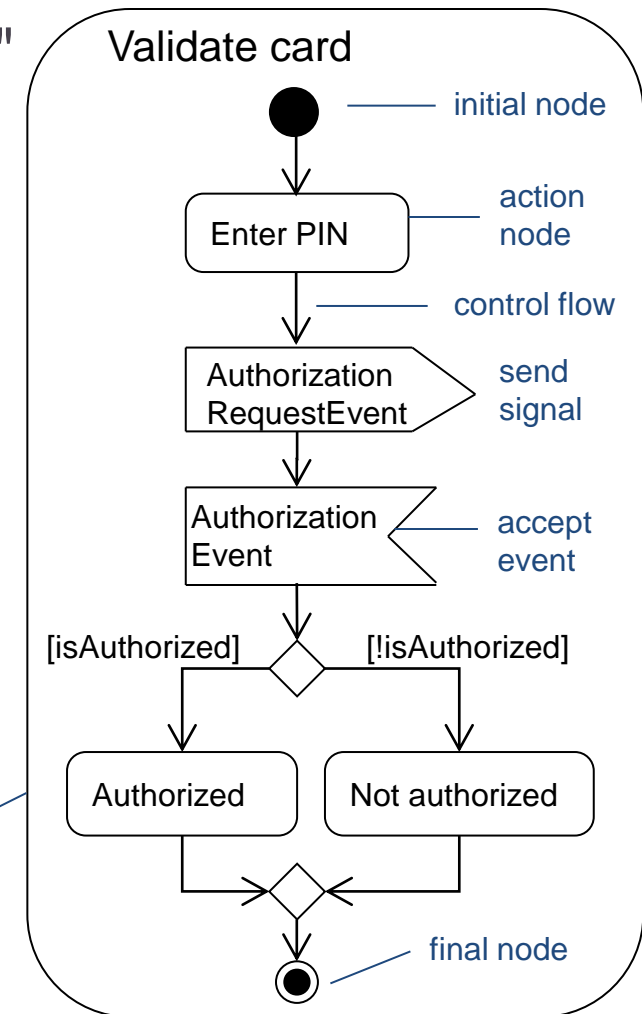
## Lecture 3/Part 2

# What are activity diagrams?



- ✧ Activity diagrams are "OO flowcharts"
- ✧ They allow us to model a process as a collection of nodes and edges between those nodes
- ✧ Use activity diagrams to model the behavior of:
  - use cases
  - collaborations (of classes, components, people, departments)
  - operations and methods
  - business processes

activity





# Activities

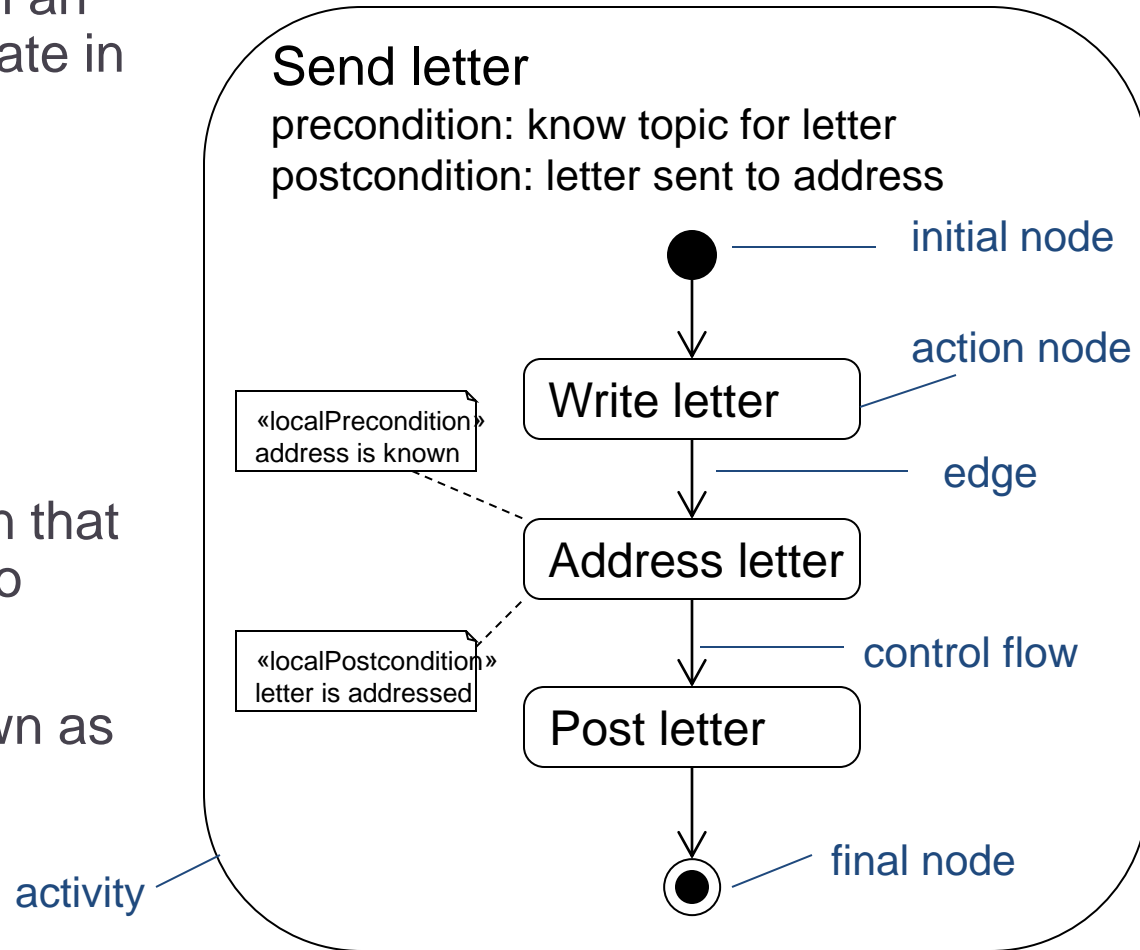


- ✧ Activities are networks of nodes connected by edges
- ✧ There are three categories of node:
  - **Action nodes** – represent atomic units of work within the activity
  - **Control nodes** - control the flow through the activity
  - **Object nodes** - represent the flow of objects around the activity
- ✧ Edges represent flow through the activity
- ✧ There are two categories of edge:
  - **Control flows** - represent the flow of control through the activity
  - **Object flows** - represent the flow of objects through the activity
- ✧ What is the difference between an action and activity?  
How can I recognize one from another in the diagram?

# Activity diagram syntax



- ✧ Activities usually start in an **initial node** and terminate in a **final node**
- ✧ Activities can have **preconditions** and **postconditions**
- ✧ When an action node finishes, it emits a token that may traverse an edge to trigger the next action
- ✧ This is sometimes known as a **transition**



# Activity diagram semantics



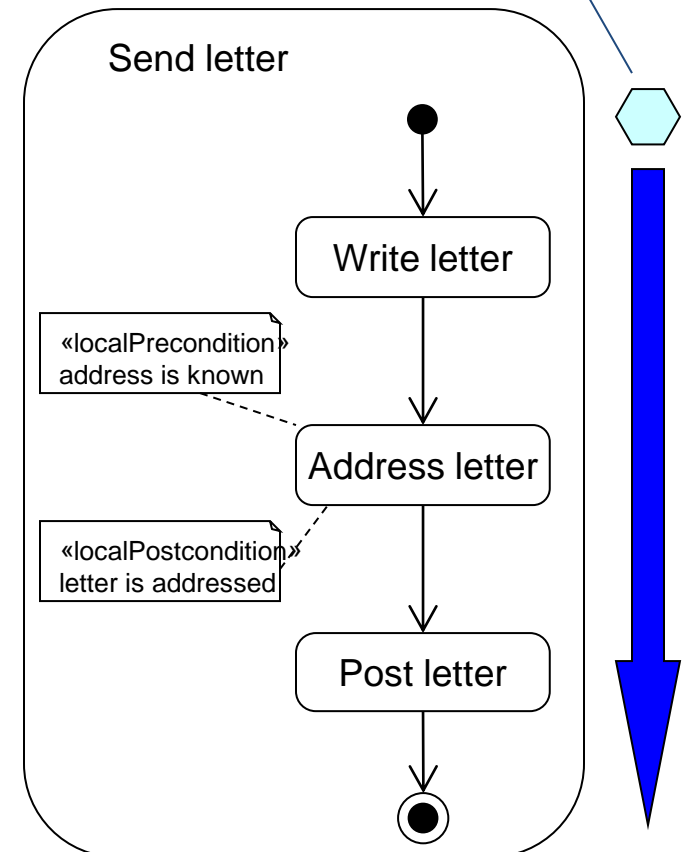
## ✧ The **token game**

- Token – an object, some data or a focus of control
- Imagine tokens flowing around the activity diagram

## ✧ Tokens traverse from a source node to a target node via an edge

- The source node, edge and target node may all have constraints controlling the movement of tokens
- All constraints **must** be satisfied before the token can make the traversal

imaginary flow of control token



# Action nodes

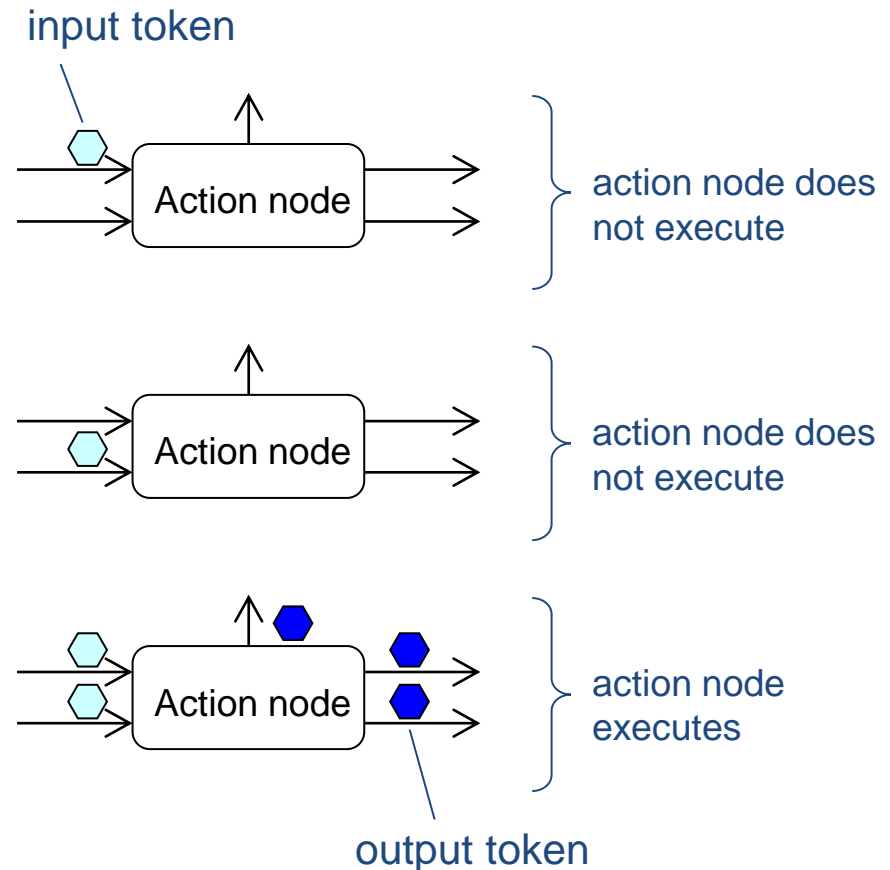


✧ Action nodes offer a token on **all** of their output edges when:

- There is a token **simultaneously** on each input edge
- The input tokens satisfy all preconditions specified by the node

✧ Action nodes:

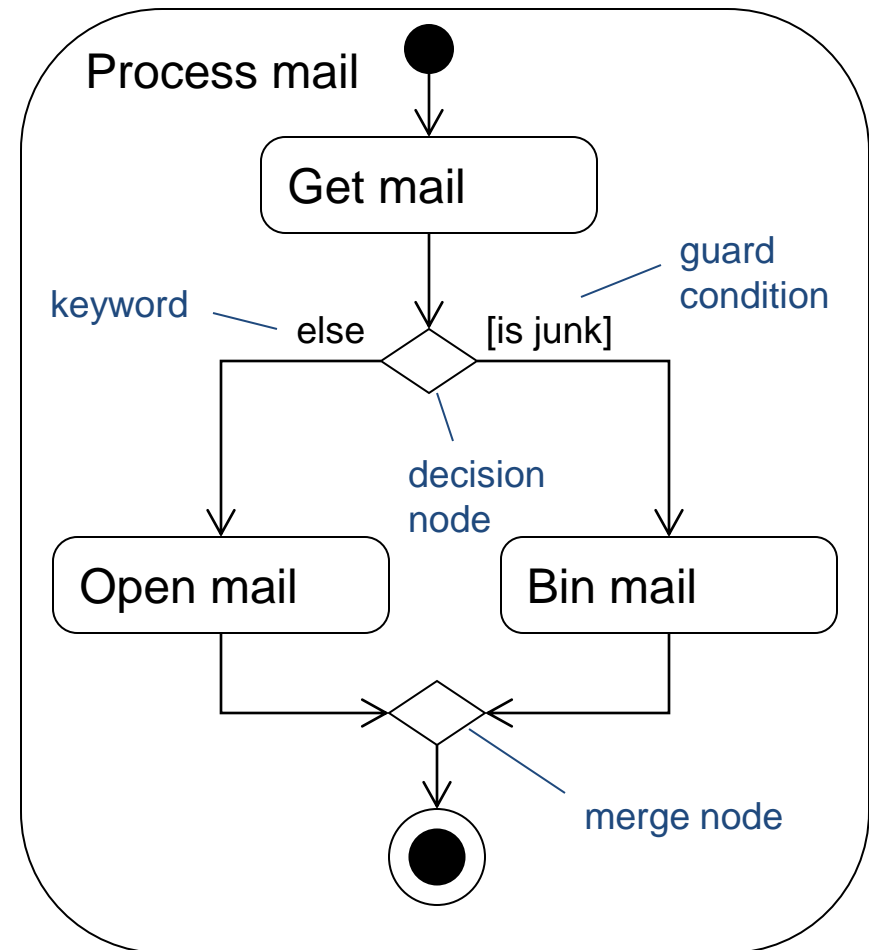
- Perform an implicit fork on their output edges when they have finished executing



# Decision and merge nodes



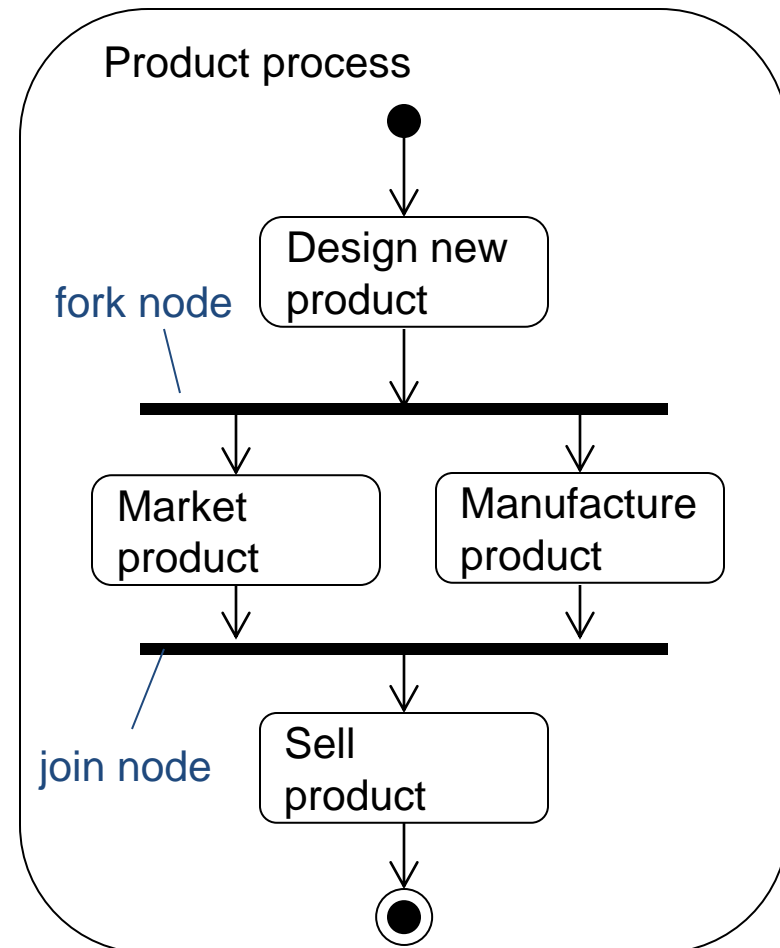
- ✧ A **decision node** is a control node that has one input edge and two or more alternate output edges
  - Each edge out of the decision is protected by a **guard condition**
  - guard conditions must be mutually exclusive
  - The edge can be taken if and only if the guard condition evaluates to true
  - The keyword **else** specifies the path that is taken if *none* of the guard conditions are true
- ✧ A **merge node** allows through any of several alternate flows
  - It has two or more input edges and exactly one output edge



# Fork and join nodes – concurrency


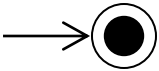
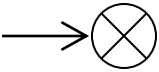
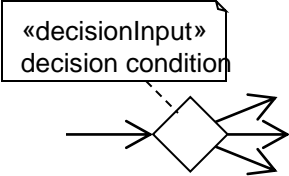
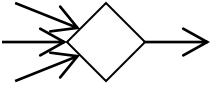
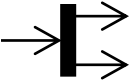
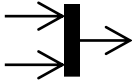


- ✧ **Forks nodes** model concurrent flows of work
  - Tokens on the single input edge are replicated at the multiple output edges
- ✧ **Join nodes** synchronize two or more concurrent flows
  - Joins have two or more incoming edges and exactly one outgoing edge
  - A token is offered on the outgoing edge when there are tokens on all the incoming edges i.e. when the concurrent flows of work have all finished




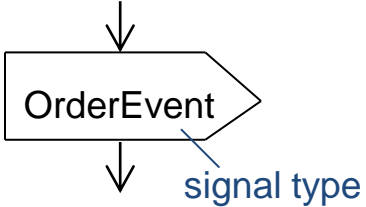
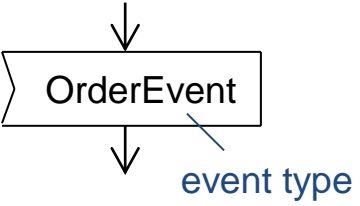

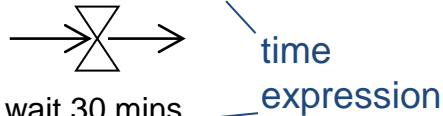
# Control nodes



control node syntax	control node semantics	
	Initial node – indicates where the flow starts when an activity is invoked	
	Activity final node – terminates an activity	Final nodes
	Flow final node – terminates a specific flow within an activity. The other flows are unaffected	
	Decision node– guard conditions on the output edges select one of them for traversal May optionally have inputs defined by a «decisionInput»	
	Merge node – allows through any of its input edges	
	Fork node – splits the flow into multiple concurrent flows	
<p>{join spec}</p> 	Join node – synchronizes multiple concurrent flows May optionally have a join specification to modify its semantics	

# Types of action node



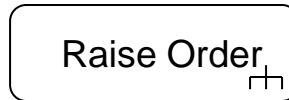
action node syntax	action node semantics
	<p>Call action - invokes an activity, a behavior or an operation. The most common type of action node.</p> <p>See next slide for details.</p>
	<p>Send signal action - sends a signal asynchronously. The sender <i>does not</i> wait for confirmation of signal receipt.</p> <p>It may accept input parameters to create the signal</p>
	<p>Accept event action - waits for events detected by its owning object and offers the event on its output edge. Is enabled when it gets a token on its input edge. If there is <i>no</i> input edge it starts when its containing activity starts and is <i>always</i> enabled.</p>
 	<p>Accept time event action - waits for a set amount of time. Generates time events according to it's time expression.</p>



# Call action node syntax



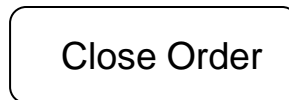
✧ The most common type of node



call an activity  
(note the rake icon)

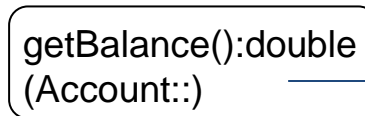
✧ Call action nodes may invoke:

- an activity
- a behavior
- an operation



call a behavior

✧ They may contain code fragments in a specific programming language

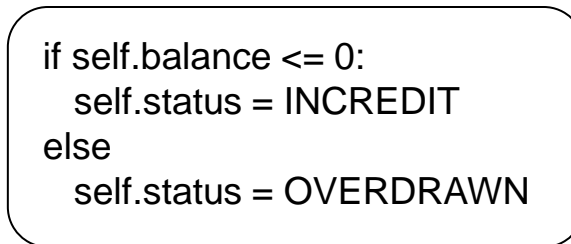


operation name  
class name  
(optional)

- The keyword 'self' refers to the context of the activity that owns the action



node name  
operation name  
(optional)



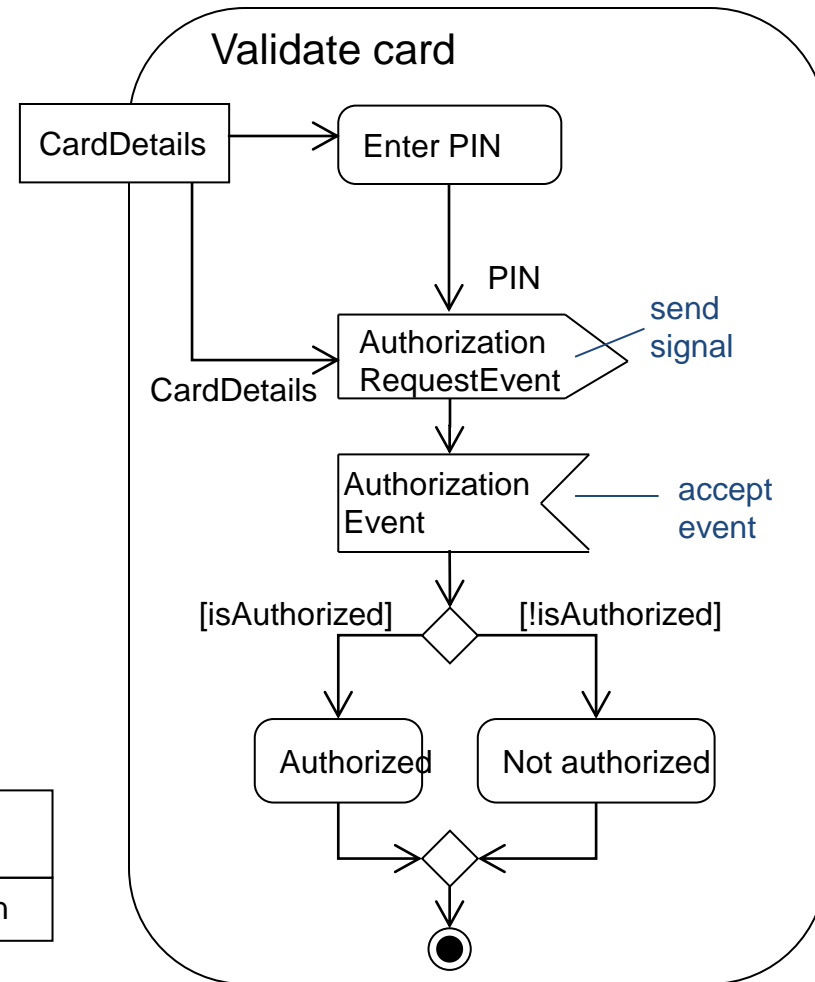
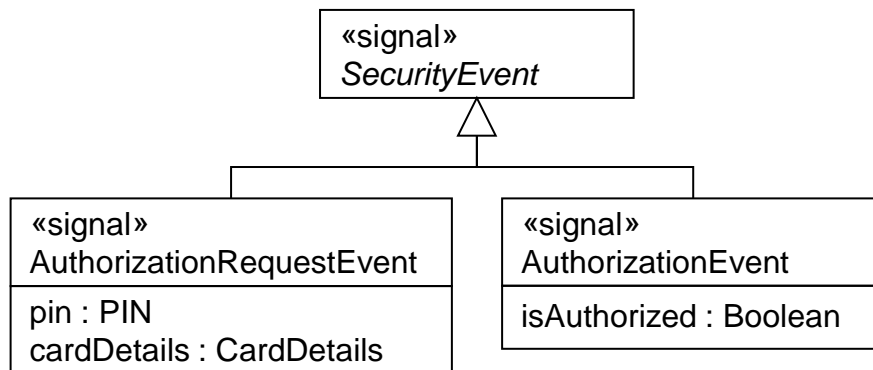
programming language  
(e.g. Python)

call an operation

# Sending signals and accepting events



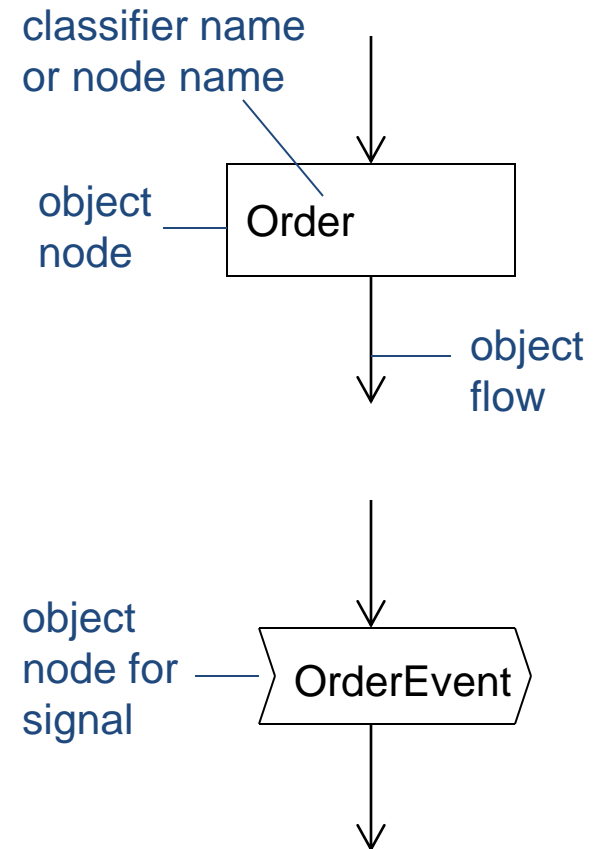
- ✧ Signals represent information passed asynchronously between objects
  - This information is modelled as attributes of a signal
  - A signal is a classifier stereotyped «signal»
- ✧ The accept event action asynchronously accepts event triggers which may be signals or other objects



# Object nodes



- ✧ Object nodes indicate that instances of a particular classifier may be available
  - If no classifier is specified, then the object node can hold any type of instance
- ✧ Multiple tokens can reside in an object node **at the same time**
  - The upper bound defines the maximum number of tokens (infinity is the default)
- ✧ Tokens are presented to the single output edge according to an ordering:
  - FIFO – first in, first out (the default)
  - LIFO – last in, first out
  - Modeler defined – a selection criterion is specified for the object node



# Object node syntax



✧ Object nodes have a flexible syntax. You may show:

- upper bounds
- ordering
- sets of objects
- selection criteria
- object in state

Order

order objects may be available

Order

zero to 12 Order objects may be available

{upperBound = 12}

Order

last Order object in is the first out (FIFO is the default)

{ordering = LIFO}

Set of Order

sets of Order objects may be available

«selection»  
monthRaised = "Dec"

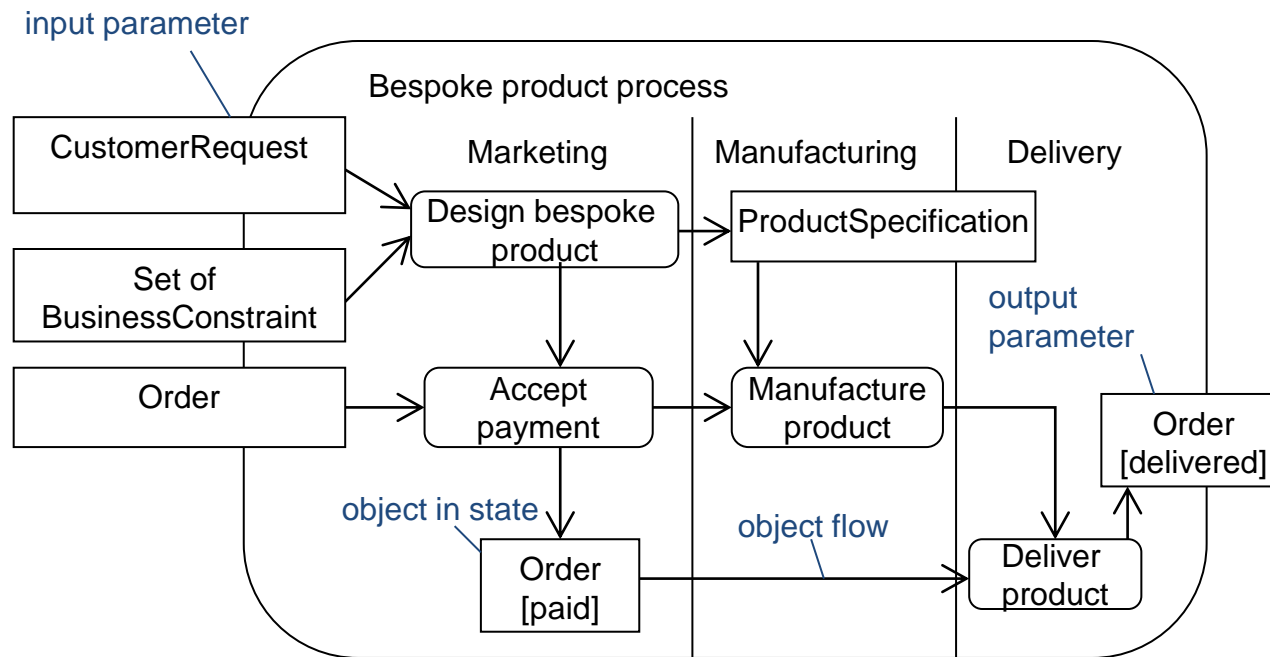
Order

Order objects raised in December may be available

Order  
[open]

select Order objects in the open state

# Activity parameters and partitioning

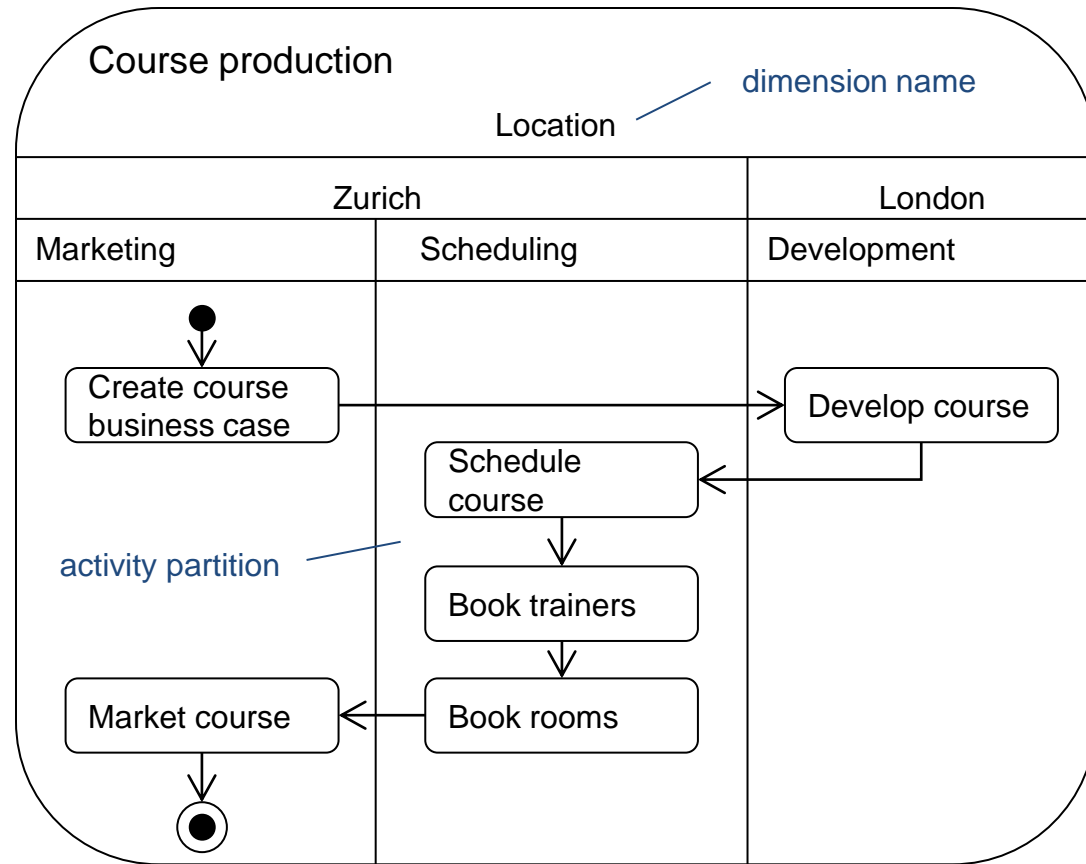


- ✧ Object nodes can provide input and output parameters to activities
  - Input parameters have one or more output object flows into the activity
  - Output parameters have one or more input object flows out of the activity
- ✧ Draw the object node overlapping the activity boundary

# Activity partitions



- Each activity partition represents a high-level grouping of a set of related actions
  - Partitions can be hierarchical
  - Partitions can be vertical, horizontal or both
- Partitions can refer to many different things e.g. business organisations, classes, components and so on
- If partitions can't be shown clearly using parallel lines, put their name in brackets directly above the name of the activities



(London::Marketing)  
Market product

(p1, p2)  
SomeAction

nested partitions

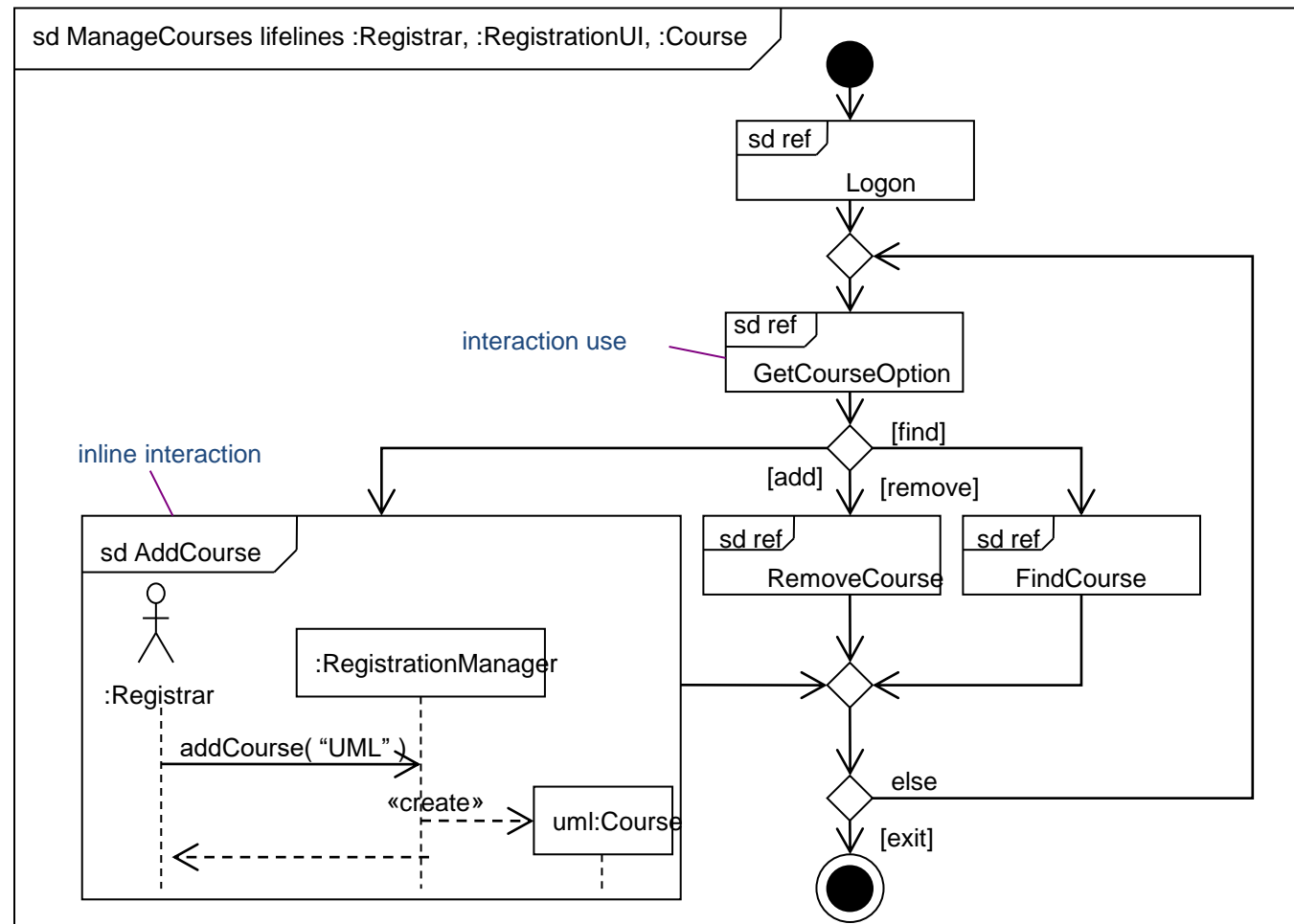
multiple partitions



# Interaction overview diagrams



- ✧ Model the high level flow of control between interactions
- ✧ Show interactions and interaction occurrences
- ✧ Have activity diagram syntax



# Key points



✧ Activity diagrams can model flows of activities using:

- Activities and connectors
- Activity partitions
- Action nodes
  - Call action node
  - Send signal/accept event action node
  - Accept time event action node
- Control nodes
  - Decision and merge
  - Fork and join
- Object nodes
  - Input and output parameters
  - Pins

✧ Interaction overview diagrams as their advanced feature

