Requirements Engineering

Lecture 2

Requirements engineering

- The **process** of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- The **requirements** themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process.
  - It may range from a **high-level abstract statement** of a service or of a system constraint to a detailed **mathematical functional specification**.

Outline

- Requirements and their types
- Requirements specification
- Requirements engineering process
  - Requirements elicitation and analysis
  - Requirements validation
  - Requirements management
- UML Use Case diagram

Requirements and their Types

Lecture 2/Part 1

Types of requirements

- User requirements
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.
- System requirements
  - A structured document setting out **detailed descriptions of the system's functions**, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

User and system requirements

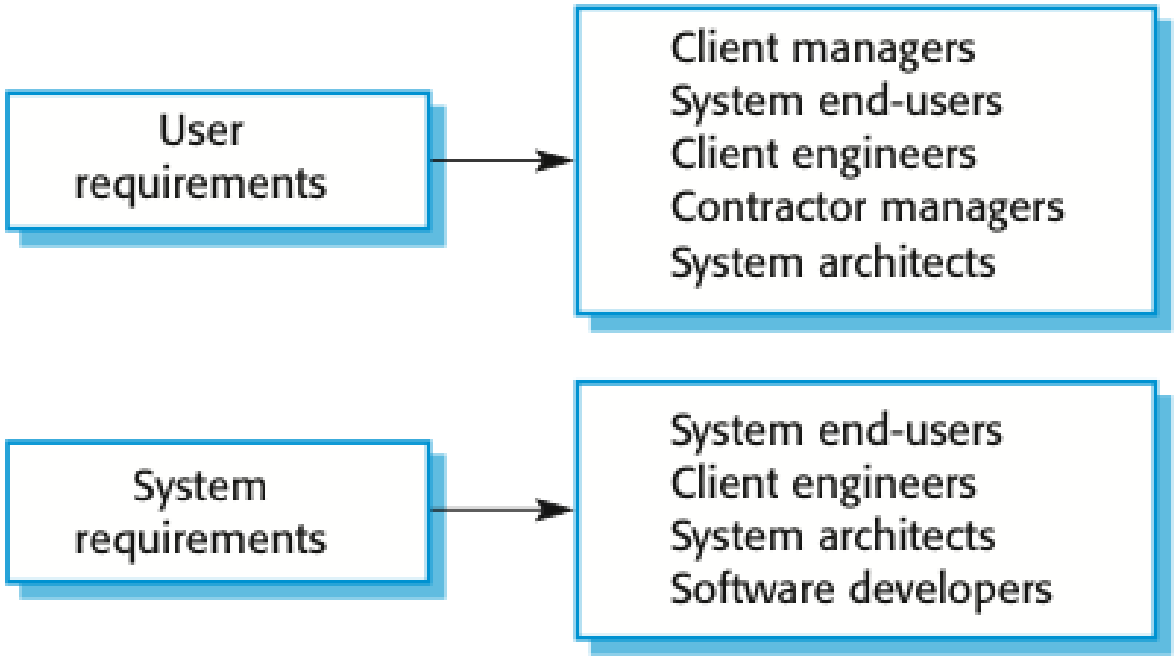
User requirement definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g. 10mg, 20 mg, etc.) separate reports shall be created for each dose unit.
1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

Readers of different types of requirements specification



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.
  - May state what the system should not do.
- 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.
  - Often apply to the **system as a whole** rather than individual features or services.

Functional requirements for the MHC-PMS

- A user shall be able to search the appointments lists for all clinics.
- The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
- Each staff member using the system shall be uniquely identified by his or her 8-digit employee number.

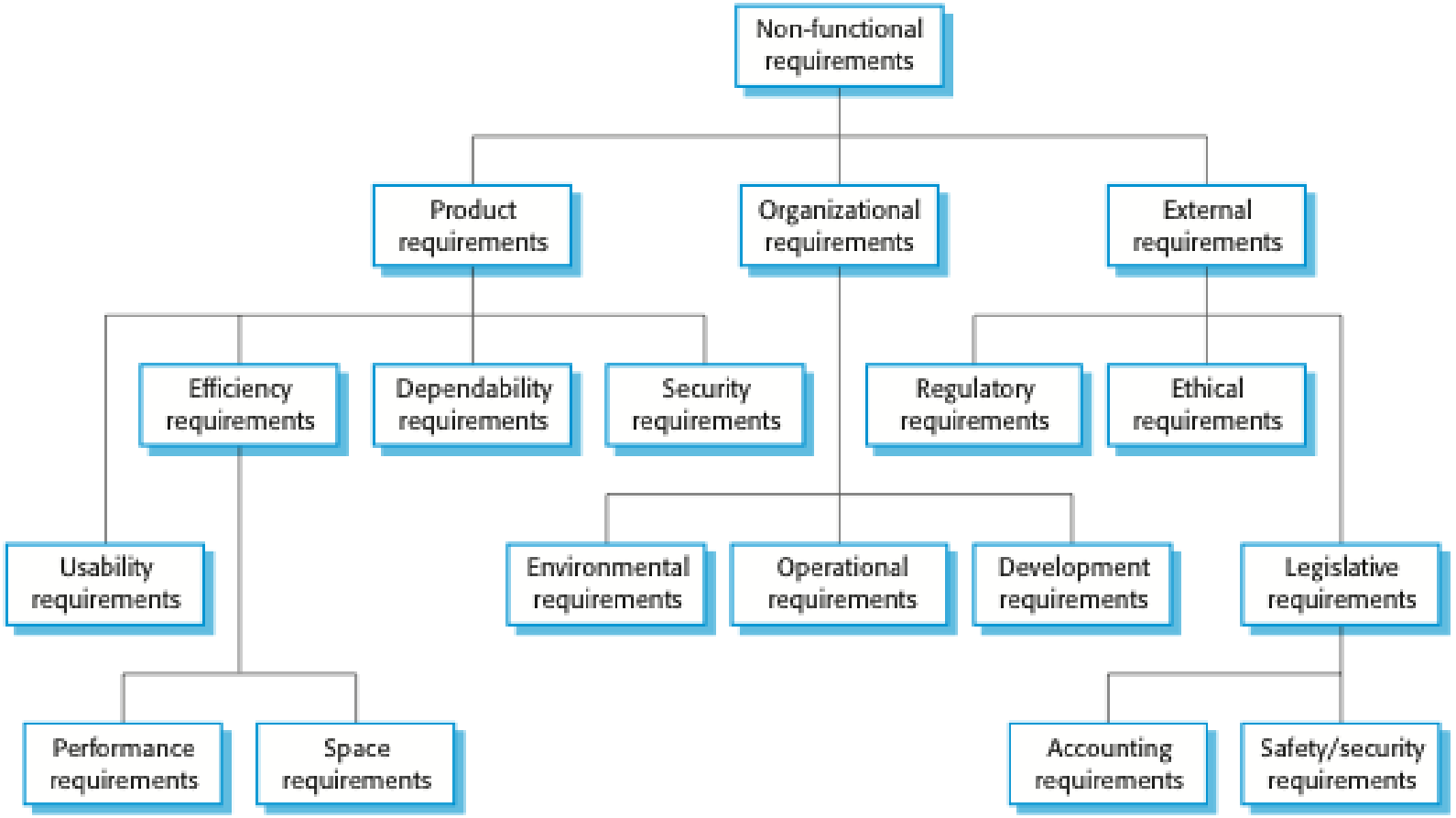
Requirements precision, completeness and consistency

- Precise
  - They should have just **one interpretation** in the system context, which is enforced by the following two properties.
- Complete
  - They should include descriptions of all facilities required.
- Consistent
  - There should be no conflicts or contradictions in the descriptions of the system facilities.
- In practice, it is very hard (sometimes impossible) to produce a complete and consistent requirements document.

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. interoperability requirements, legislative requirements, etc.

Types of non-functional requirements (excerpt)



Examples of non-functional requirements in the MHC-PMS

Product requirement
The MHC-PMS shall be available to all clinics during normal working hours (Mon–Fri, 08.30–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

Organizational requirement
Users of the MHC-PMS system shall authenticate themselves using their health authority identity card.

External requirement
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Non-functional requirements 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.

Verifiability of non-functional requirements

- Goals vs. verifiable requirements
- Usability requirement example
  - **Goal:** The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized.
  - **Verifiable non-functional requirement:** Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.

Metrics for specifying non-functional requirements

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Key points

- Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.
- Non-functional requirements often constrain the system being developed and the development process being used.
- They often relate to the emergent properties of the system and therefore apply to the system as a whole.

Requirements Specification

Lecture 2/Part 2

The software requirements document

- The software requirements document is the official statement of what is required of the system developers.
- Should include a definition of all the above mentioned **requirements types**, and may respect a number of standards (e.g. IEEE standard).
- It is NOT a design document. As far as possible, it should set of **WHAT the system should do** rather than HOW it should do it.
- Information in requirements document depends on **type of system and the approach to development** used (plan-driven vs. agile approach).

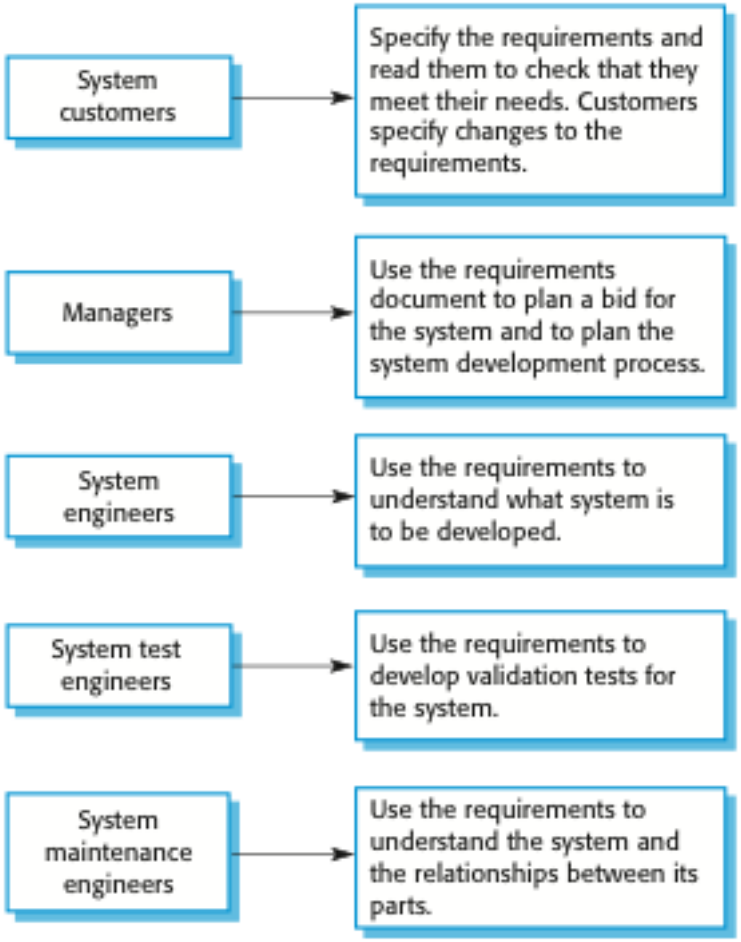
The structure of a requirements document

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.

The structure of a requirements document

Chapter	Description
System requirements specification	This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Users of a requirements document



Ways of writing a system requirements specification

Notation	Description
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract

Natural language specification

- Used for writing requirements because it is expressive, intuitive and universal. This means that the requirements can be understood by users and customers.
- Guidelines:
  - Invent a standard format and use it for all requirements.
  - Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
  - Use text highlighting to identify key parts of the requirement.
  - Avoid the use of computer jargon.
  - Include an explanation (rationale) of why a requirement is necessary.

Problems with natural language

- Lack of clarity
  - Precision is difficult without making the document difficult to read.
- Requirements confusion
  - Functional and non-functional requirements tend to be mixed-up.
- Requirements amalgamation
  - Several different requirements may be expressed together.

Example requirements for the insulin pump software system

3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. *(Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.)*

3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. *(A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.)*

Structured natural language specification

- Works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business system requirements.
- Example notations:
  - Form-based specification
  - Tabular specification

A form-based specification of a requirement for an insulin pump

Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level.

Description

Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

Inputs Current sugar reading (r2); the previous two readings (r0 and r1).

Source Current sugar reading from sensor. Other readings from memory.

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.

A structured specification of a requirement for an insulin pump

Action
CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.
Requirements
Two previous readings so that the rate of change of sugar level can be computed.
Pre-condition
The insulin reservoir contains at least the maximum allowed single dose of insulin.
Post-condition r0 is replaced by r1 then r1 is replaced by r2.
Side effects None.

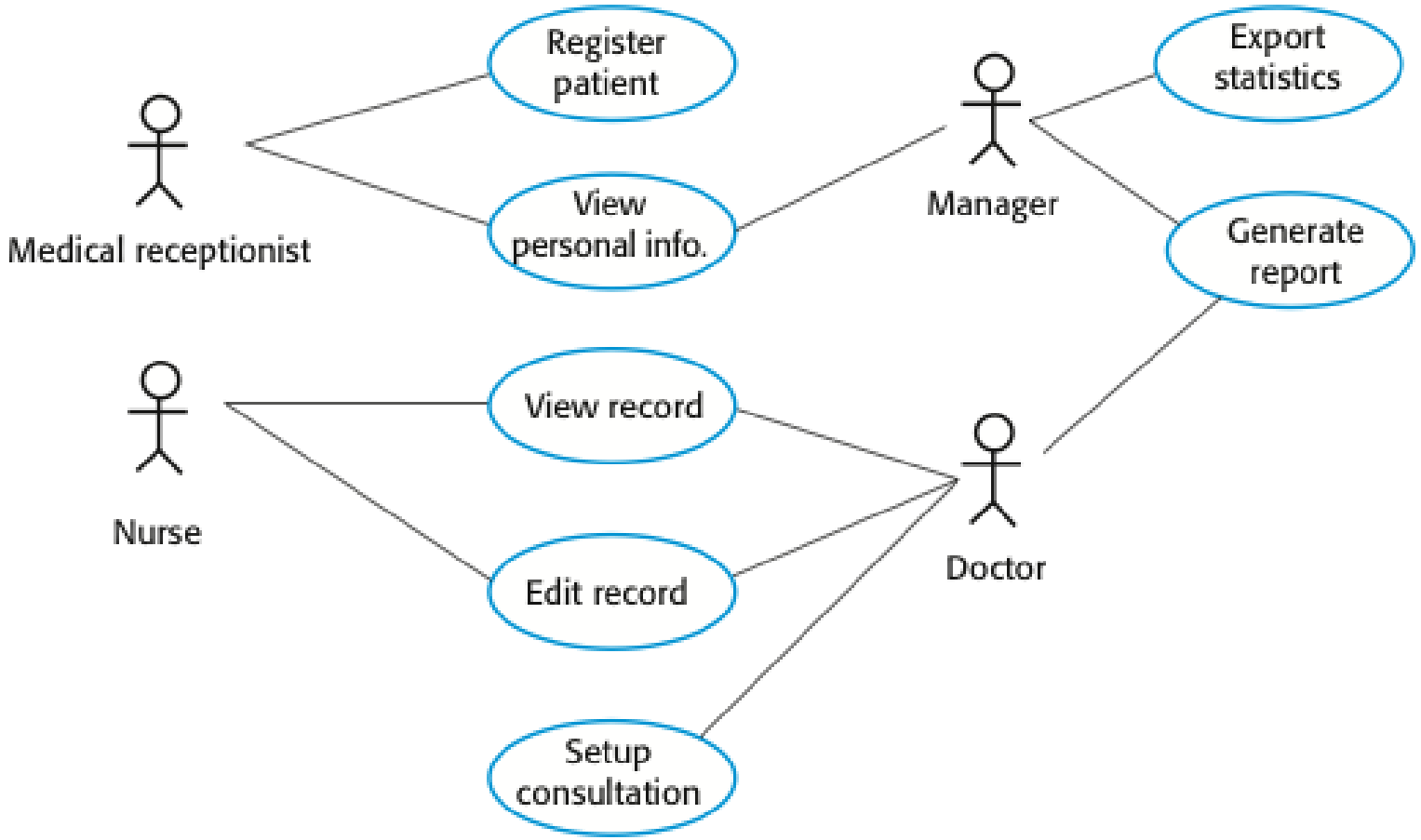
Tabular specification of computation for an insulin pump

Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 – r1) < (r1 – r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing ((r2 – r1) ≥ (r1 – r0))	CompDose = round ((r2 – r1)/4) If rounded result = 0 then CompDose = MinimumDose

Use cases

- UML Use-cases are a scenario based technique which identify the actors in an interaction and which describe the interaction itself.
- A set of use cases should describe all possible interactions with the system.
- High-level graphical model supplemented by more detailed tabular description (see Part 4 of this lecture).
- UML Activity diagrams and Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.

Use cases for the MHC-PMS



Requirements Engineering Process

Lecture 2/Part 3

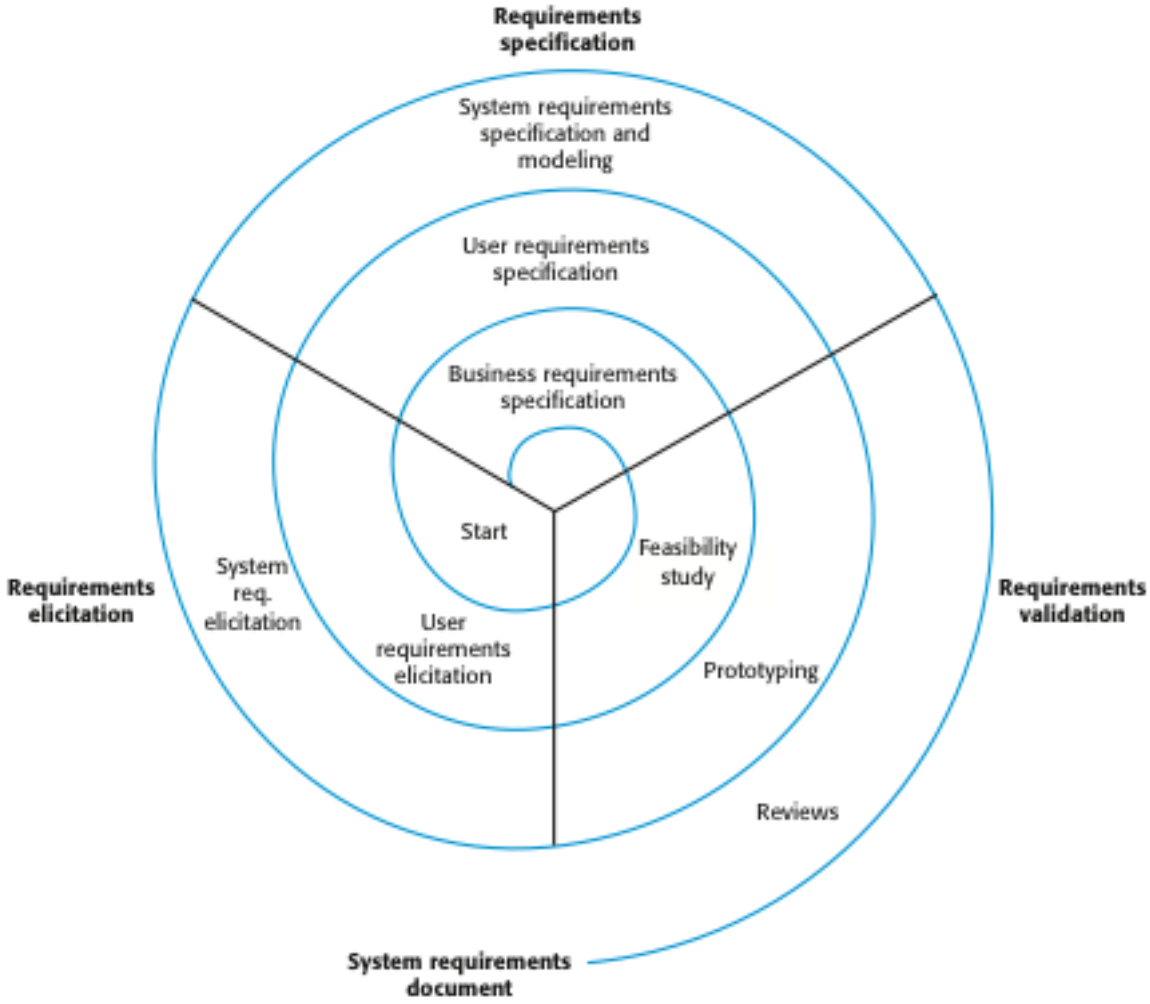
Outline

- Requirements elicitation and analysis
- Requirements validation
- Requirements management

Requirements engineering processes

- The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- However, there are a number of generic activities common to all processes
  - Requirements elicitation and analysis;
  - Requirements validation;
  - Requirements management.
- In practice, RE is an iterative activity in which these processes are interleaved.

A spiral view of the requirements engineering process



Requirements elicitation and analysis

- 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 that the system should provide,
  - the required system performance,
  - hardware constraints,
  - other systems, etc.

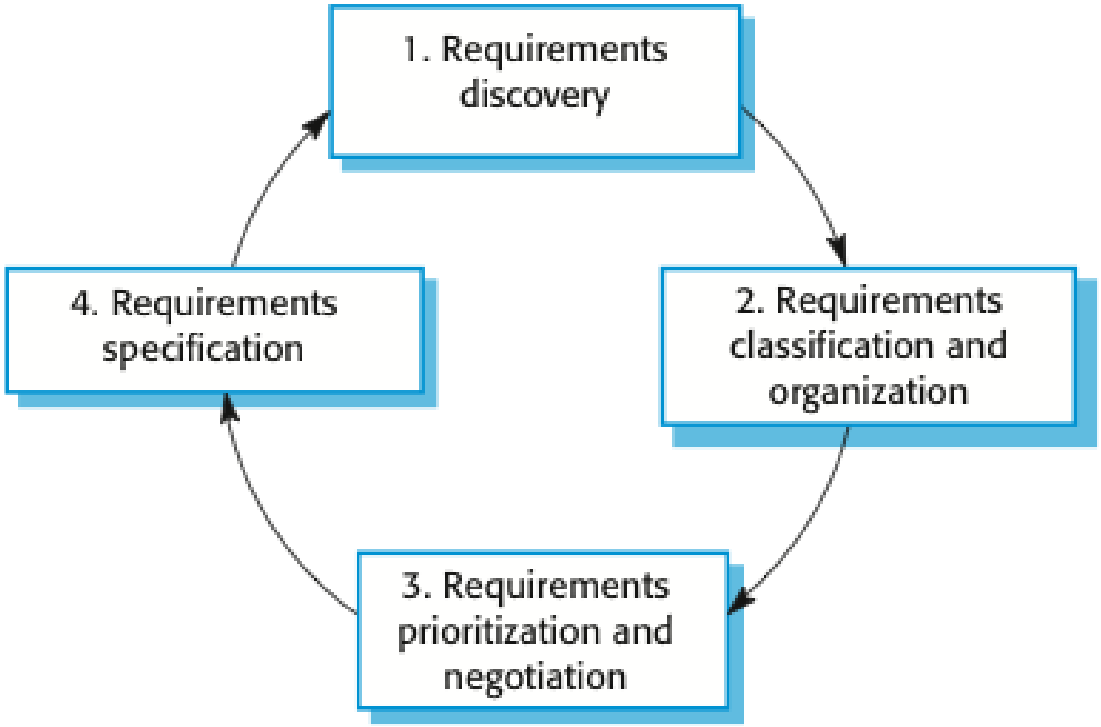
Stakeholders in the MHC-PMS

- Patients whose information is recorded in the system.
- Doctors who are responsible for assessing and treating patients.
- Nurses who coordinate the consultations with doctors and administer some treatments.
- Medical receptionists who manage patients' appointments.
- IT staff who are responsible for installing and maintaining the system.

Stakeholders in the MHC-PMS

- A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
- Health care managers who obtain management information from the system.
- Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

The requirements elicitation and analysis process



Process activities

- Requirements discovery
  - Interacting with stakeholders to discover their requirements.
- Requirements classification and organisation
  - Groups related requirements and organises them into coherent clusters.
- Prioritisation and negotiation
  - Prioritising requirements and resolving requirements conflicts.
- Requirements specification
  - Requirements are documented and input into the next round of the spiral.

Requirements prioritisation

- MoSCoW criteria
  - **M**ust have – mandatory requirement fundamental to the system
  - **S**hould have – important requirement that may be omitted
  - **C**ould have – truly optional requirement
  - **W**ant to have – requirement that can wait for later releases
- 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

Problems of requirements elicitation

- Stakeholders **don't know what they really want**.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have **conflicting requirements**.
- Organisational and political factors may influence the system requirements.
- The **requirements change** during the analysis process. New stakeholders may emerge and the business environment may change.

Interviewing

- Formal or informal interviews with stakeholders are part of most RE processes.
- Types of interview
  - **Closed interviews** based on pre-determined list of questions
  - **Open interviews** where various issues are explored with stakeholders.
- Effective interviewing
  - Be **open-minded**, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
  - Prompt the interviewee to get **discussions** going using a springboard question, a requirements proposal, or by working together on a prototype system.

Ethnography

- A social scientist spends a considerable time observing and analysing how people actually work.
- People do not have to explain or articulate their work.
- Social and organisational factors of importance may be observed.
- Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.

Requirements validation

- Concerned with demonstrating that the **requirements define the system that the customer really wants**.
- Requirements error costs are high so validation is very important
  - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements validation

- ✧ Consistency
  - Are there any requirements conflicts?
- ✧ Completeness
  - Are all functions required by the customer included?
- ✧ Realism
  - Can the requirements be implemented given available budget and technology
- ✧ Verifiability
  - Can the requirements be checked?

Requirements validation techniques

- Requirements reviews
  - Systematic manual analysis of the requirements.
- Prototyping
  - Using an executable model of the system to check requirements.
- Test-case generation
  - Developing tests for requirements to check testability.

Requirements reviews

- Regular reviews should be held while the requirements definition is being formulated.
- Both client and contractor staff should be involved in reviews.
- Reviews may be **formal** (with completed documents) or **informal**. Good communications between developers, customers and users can resolve problems at an early stage.

Review checks

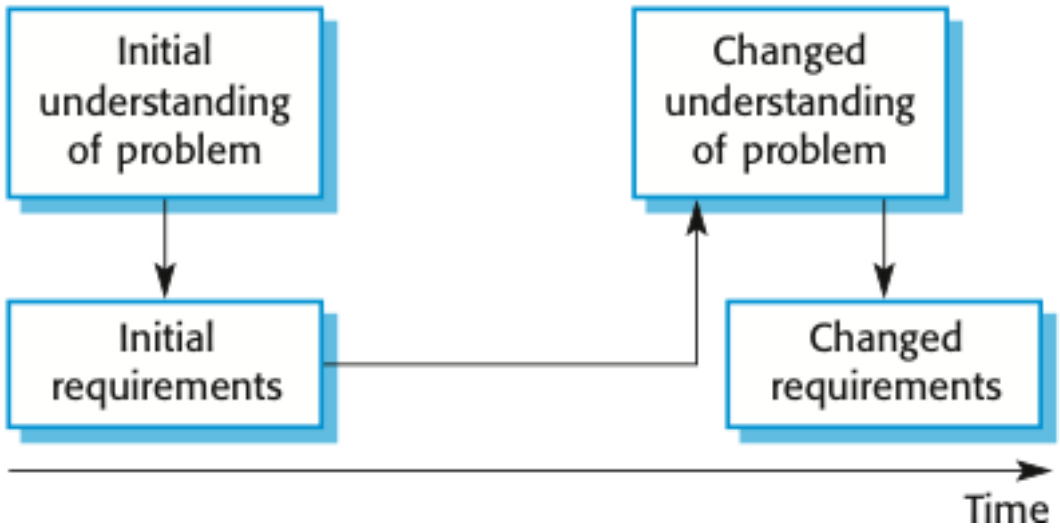
- Besides consistency, completeness, realism and verifiability, reviews check:

- Comprehensibility
  - Is the requirement properly understood?
- Traceability
  - Is the origin of the requirement clearly stated?
- Adaptability
  - Can the requirement be changed without a large impact on other requirements?

Requirements management

- Requirements management is the process of **managing changing requirements** during the requirements engineering process and system development.
- New requirements emerge as a system is being developed and after it has gone into use.
- You need to keep track of individual requirements and **maintain links between dependent requirements** so that you can assess the impact of requirements changes. You need to establish a formal process for making **change proposals** and linking these to system requirements.

Requirements evolution



Changing requirements

- The business and technical environment of the system always changes after installation.
  - New **hardware** may be introduced, it may be necessary to interface the system with **other systems**, **business priorities** may change (with consequent changes in the system support required), and new **legislation and regulations** may be introduced that the system must necessarily abide by.
- The people **who pay** for a system and the **users** of that system are rarely the same people.
- Large systems usually have a **diverse user community**, with many users having different requirements and priorities that may be **conflicting**.

Requirements management planning

- Establishes the level of requirements management detail that is required.
- Requirements management decisions:
  - ***Requirements identification*** Each requirement must be uniquely identified so that it can be cross-referenced with other requirements.
  - ***A change management process*** This is the set of activities that assess the impact and cost of changes.
  - ***Traceability policies*** These policies define the relationships between requirements, and between the requirements and the system design that should be recorded.
  - ***Tool support*** Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Requirements change management

✧ Deciding if a requirements change should be accepted

- ***Problem analysis and change specification***
  - During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
- ***Change analysis and costing***
  - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
- ***Change implementation***
  - The requirements document and the system design and implementation are modified. Ideally, the document should be organized so that changes can be easily implemented.

Key points

- You can use a range of techniques for **requirements elicitation** including interviews, use-cases discussion and ethnography.
- **Requirements validation** is the process of checking the requirements for validity, consistency, completeness, realism and verifiability.
- Business, organizational and technical changes inevitably lead to changes to the requirements for a software system. **Requirements management** is the process of managing and controlling these changes.

UML Use Case Diagram

Lecture 2/Part 4

Outline

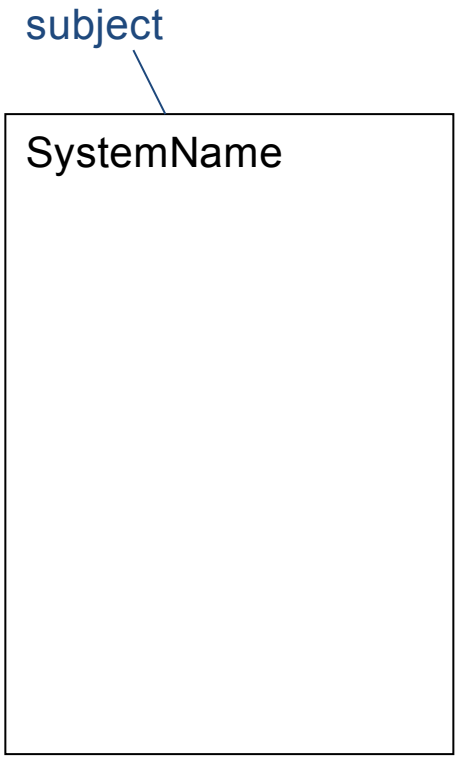
- Use Case modelling
  - System boundary – subject
  - Use cases
  - Actors
- Textual Use Case specification
- Advanced Use Case modelling
  - Actor generalisation
  - Use case generalisation
  - «include»
  - «extend»

Use case modelling

- Use case modelling is a form of requirements engineering
- Use case modelling proceeds as follows:
  - Find the system boundary
  - Find actors
  - Find use cases
    - Use case specification
    - Scenarios
- It lets us identify the system boundary, who or what uses the system, and what functions the system should offer

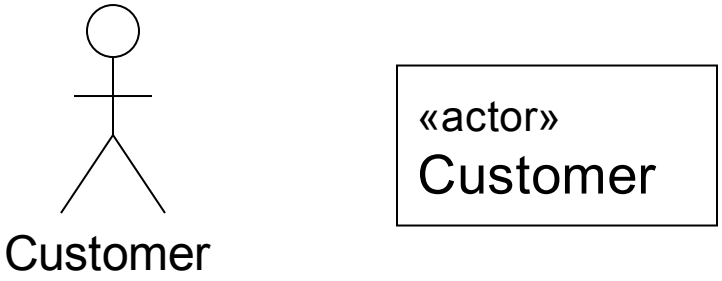
The subject

- Before we can build anything, we need to know:
  - Where the **boundary** of the system lies
  - **Who** or what uses the system
  - What **functions** the system should offer to its users
- We create a Use Case model containing:
  - **Subject** – the edge of the system
    - also known as the system boundary
  - **Actors** – who or what uses the system
  - **Use Cases** – things actors do with the system
  - **Relationships** – between actors and use cases



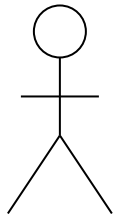
What are actors?

- An actor is anything that interacts *directly* with the system
  - Actors identify who or what uses the system and so indicate where the system boundary lies
- Actors are *external* to the system
- An Actor specifies a *role* that some external entity adopts when interacting with the system



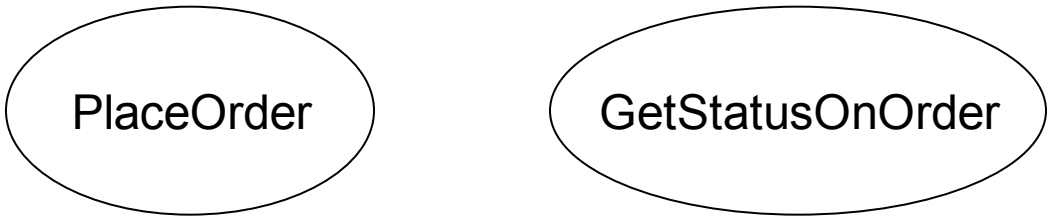
Identifying Actors

- When identifying actors ask:
  - Who or what uses the system?
  - What roles do they play in the interaction?
  - Who installs the system?
  - Who starts and shuts down the system?
  - Who maintains the system?
  - What other systems use this system?
  - Who gets and provides information to the system?
  - Does anything happen at a fixed time?



What are use cases?

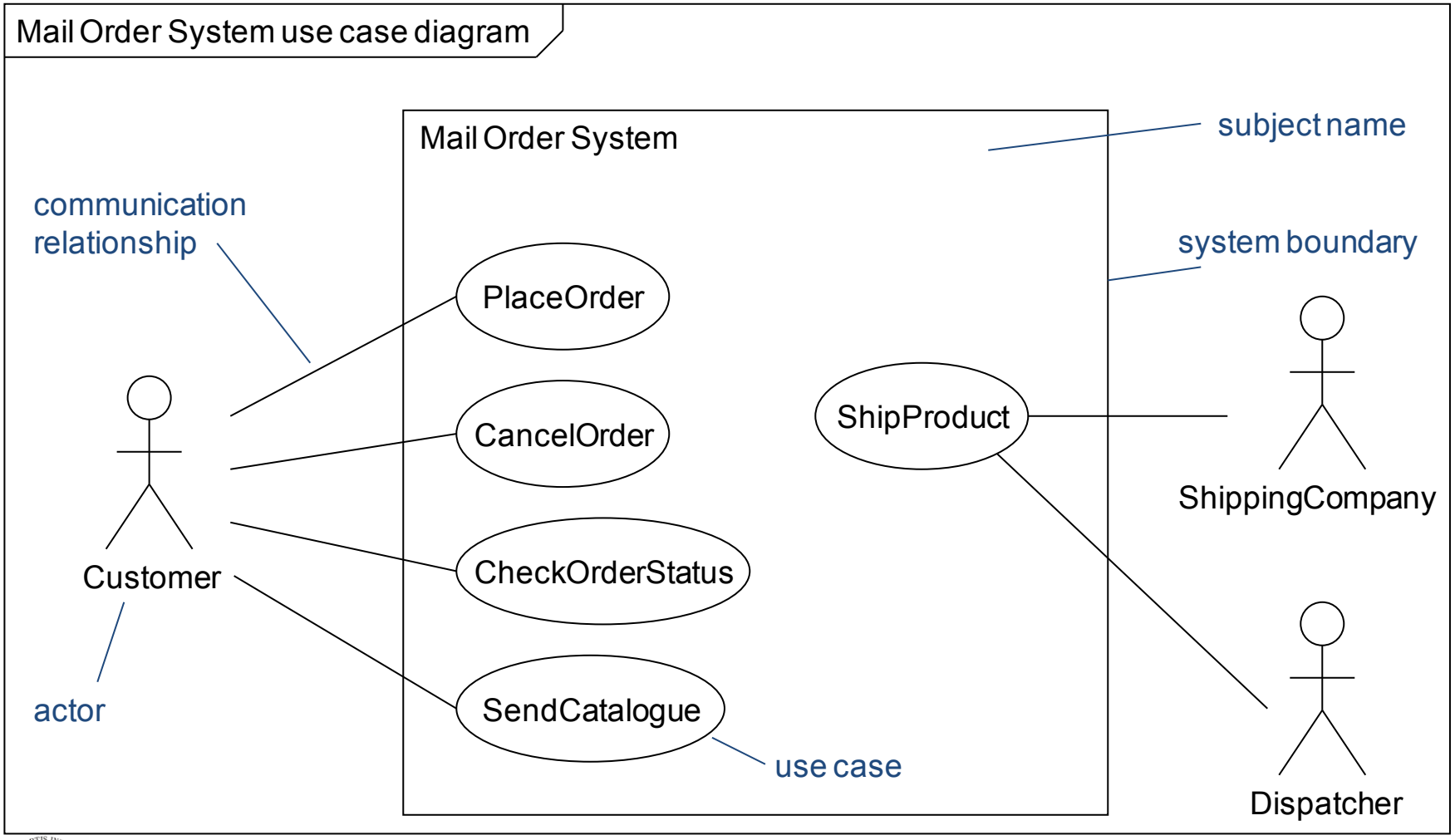
- A use case is something an actor needs the system to do. It is a "case of use" of the system by a specific actor
- Use cases are *always* started by an actor
  - The *primary actor* triggers the use case
  - Zero or more *secondary actors* interact with the use case in some way
- Use cases are *always* written from the point of view of the actors



Identifying use cases

- Start with the list of actors that interact with the system
- When identifying use cases ask:
  - What functions will a specific actor want from the system?
  - Does the system store and retrieve information? If so, which actors trigger this behaviour?
  - What happens when the system changes state (e.g. system start and stop)? Are any actors notified?
  - Are there any external events that affect the system? What notifies the system about those events?
  - Does the system interact with any external system?
  - Does the system generate any reports?

The use case diagram



Textual use case specification

use case name	Use case: PaySalesTax
use case identifier	ID: 1
brief description	Brief description: Pay Sales Tax to the Tax Authority at the end of the business quarter.
the actors involved in the use case	Primary actors: Time
	Secondary actors: TaxAuthority
the system state before the use case can begin	Preconditions: 1. It is the end of the business quarter.
the actual steps of the use case	Main flow: 1. The use case starts when it is the end of the business quarter. (implicit time actor) 2. The system determines the amount of Sales Tax owed to the Tax Authority. 3. The system sends an electronic payment to the Tax Authority.
the system state when the use case has finished	Postconditions: 1. The Tax Authority receives the correct amount of Sales Tax.
alternative flows	Alternative flows: None.

Naming use cases

- Use cases describe something that happens
- They are named using **verbs** or **verb phrases**
- Naming standard [1]: use cases are named using UpperCamelCase e.g. PaySalesTax

1 UML 2 does not specify *any* naming standards.
All naming standards here are based on industry best practice.

Pre and postconditions

- Preconditions and postconditions are *constraints*
- Preconditions constrain the state of the system *before* the use case can start
- Postconditions constrain the state of the system *after* the use case has executed
- If there are no preconditions or postconditions write "None" under the heading

Use case: PlaceOrder

Preconditions:
1. A valid user has logged on to the system

Postconditions:
1. The order has been marked confirmed and is saved by the system

Main flow

<number> The <something> <some action>

- The flow of events lists the steps in a use case
- It *always* begins by an actor doing something
  - A good way to start a flow of events is:
    1) The use case starts when an <actor> <function>
- The flow of events should be a sequence of short steps that are:
  - Declarative
  - Numbered,
  - Time ordered
- The main flow is always the *happy day* or *perfect world* scenario
  - Everything goes as expected and desired, and there are no errors, deviations and interrupts
  - Alternatives can be shown by branching or by listing under Alternative flows (see later)

Branching within a flow: IF

- Use the keyword IF to indicate alternatives within the flow of events
  - There must be a Boolean expression immediately after IF
- Use indentation and numbering to indicate the conditional part of the flow
- Use ELSE to indicate what happens if the condition is false

Use case: ManageBasket
ID: 2
Brief description: The Customer changes the quantity of an item in the basket.
Primary actors: Customer
Secondary actors: None.
Preconditions: 1. The shopping basket contents are visible.
Main flow: 1. The use case starts when the Customer selects an item in the basket. 2. IF the Customer selects "delete item" 2.1 The system removes the item from the basket. 3. IF the Customer types in a new quantity 3.1 The system updates the quantity of the item in the basket.
Postconditions: None.
Alternative flows: None.

Repetition within a flow: FOR

- We can use the keyword FOR to indicate the start of a repetition within the flow of events
- The iteration expression immediately after the FOR statement indicates the number of repetitions of the indented text beneath the FOR statement.

Use case: FindProduct
ID: 3
Brief description: The system finds some products based on Customer search criteria and displays them to the Customer.
Actors: Customer
Preconditions: None.
Main flow: 1. The use case starts when the Customer selects "find product". 2. The system asks the Customer for search criteria. 3. The Customer enters the requested criteria. 4. The system searches for products that match the Customer's criteria. 5. FOR each product found 5.1. The system displays a thumbnail sketch of the product. 5.2. The system displays a summary of the product details. 5.3. The system displays the product price.
Postconditions: None.
Alternative flows: NoProductsFound

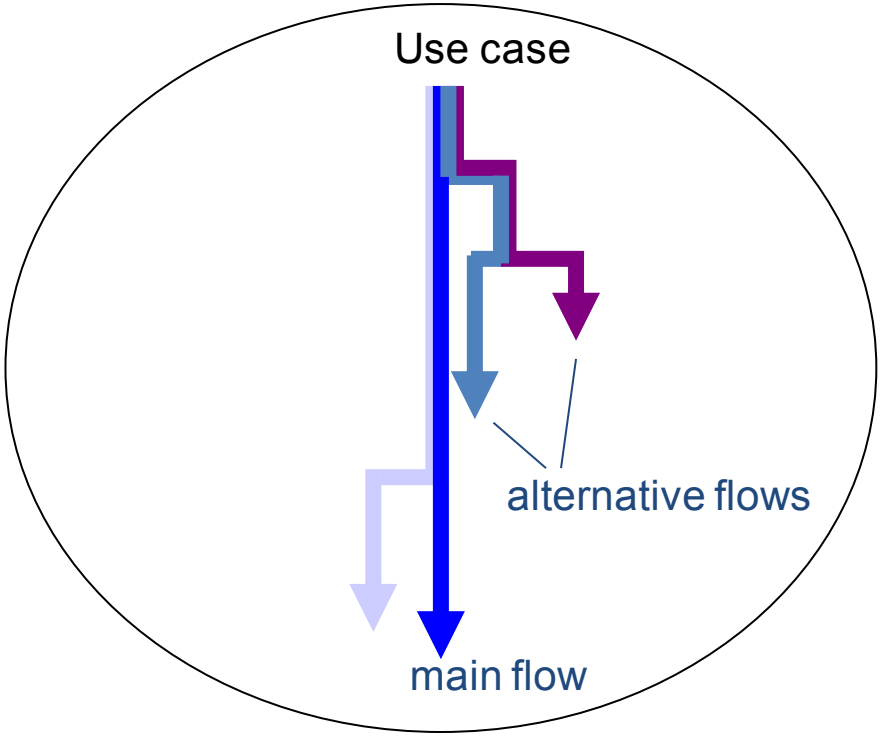
Repetition within a flow: WHILE

- We can use the keyword WHILE to indicate that something repeats while some Boolean condition is true

Use case: ShowCompanyDetails
ID: 4
Brief description: The system displays the company details to the Customer.
Primary actors: Customer
Secondary actors: None
Preconditions: None.
Main flow: 1. The use case starts when the Customer selects "show company details". 2. The system displays a web page showing the company details. 3. WHILE the Customer is browsing the company details 4. The system searches for products that match the Customer's criteria. 4.1. The system plays some background music. 4.2. The system displays special offers in a banner ad.
Postconditions: 1. The system has displayed the company details. 2. The system has played some background music. 3. The systems has displayed special offers.
Alternative flows: None.

Branching: Alternative flows

- We may specify one or more *alternative flows* through the flow of events:
  - Alternative flows capture errors, branches, and interrupts
  - Alternative flows *never* return to the main flow
- Potentially very many alternative flows! You need to manage this:
  - Pick the most important alternative flows and document those.
  - If there are groups of similar alternative flows - document one member of the group as an exemplar and (if necessary) add notes to this explaining how the others differ from it.



Only document enough alternative flows to clarify the requirements!

Referencing alternative flows

- List the names of the alternative flows at the end of the use case
- Find alternative flows by examining each step in the main flow and looking for:
  - Alternatives
  - Exceptions
  - Interrupts

Use case: CreateNewCustomerAccount
ID: 5
Brief description: The system creates a new account for the Customer.
Primary actors: Customer
Secondary actors: None.
Preconditions: None.

Main flow:

1. The use case begins when the Customer selects "create new customer account".
2. WHILE the Customer details are invalid
   - 2.1. The system asks the Customer to enter his or her details comprising email address, password and password again for confirmation.
   - 2.2 The system validates the Customer details.
3. The system creates a new account for the Customer.

Postconditions:
1. A new account has been created for the Customer.

Alternative flows:
InvalidEmailAddress
InvalidPassword
Cancel

Alternative flows

An alternative flow example

notice how we name and number alternative flows

Alternative flow: CreateNewCustomerAccount:InvalidEmailAddress
ID: 5.1
Brief description: The system informs the Customer that they have entered an invalid email address.
Primary actors: Customer
Secondary actors: None.
Preconditions: 1. The Customer has entered an invalid email address
Alternative flow: 1. The alternative flow begins after step 2.2. of the main flow. 2. The system informs the Customer that he or she entered an invalid email address.
Postconditions: None.

always indicate how the alternative flow begins. In this case it starts after step 2.2 in the main flow

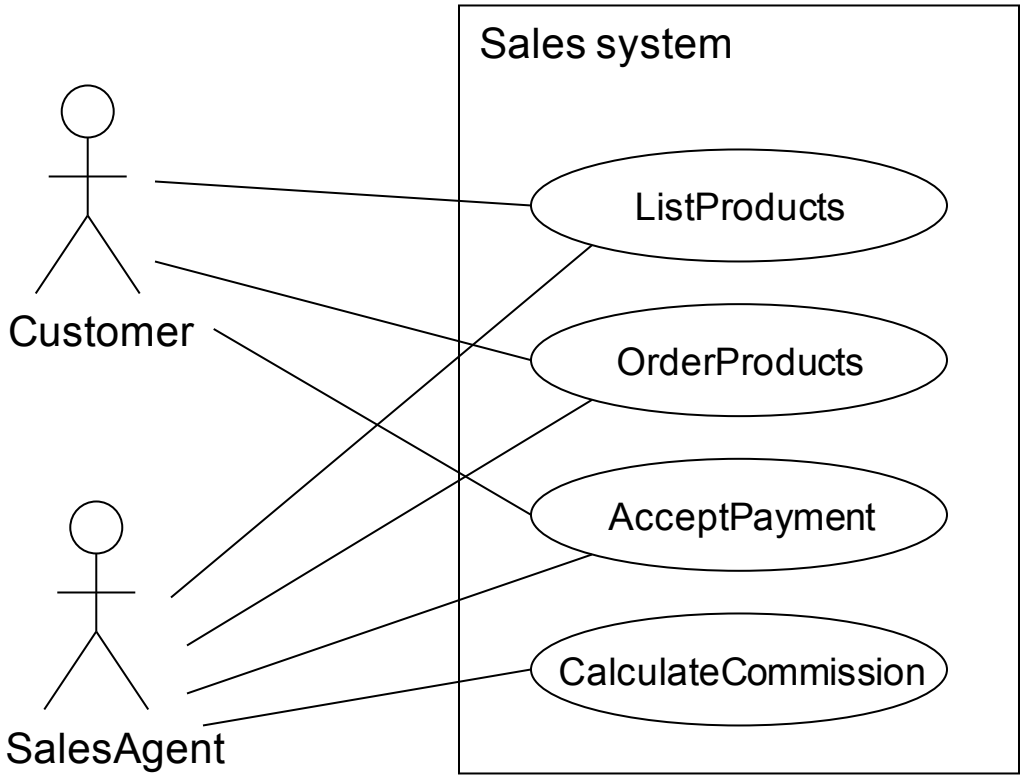
- The alternative flow may be triggered *instead* of the main flow - started by an actor
- The alternative flow may be triggered *after a particular step* in the main flow - after
- The alternative flow may be triggered *at any time* during the main flow - at any time

Advanced Use Case modelling

- We have studied basic use case analysis, but there are **relationships** that we have still to explore:
  - Actor generalisation
  - Use case generalisation
  - «include» – between use cases
  - «extend» – between use cases

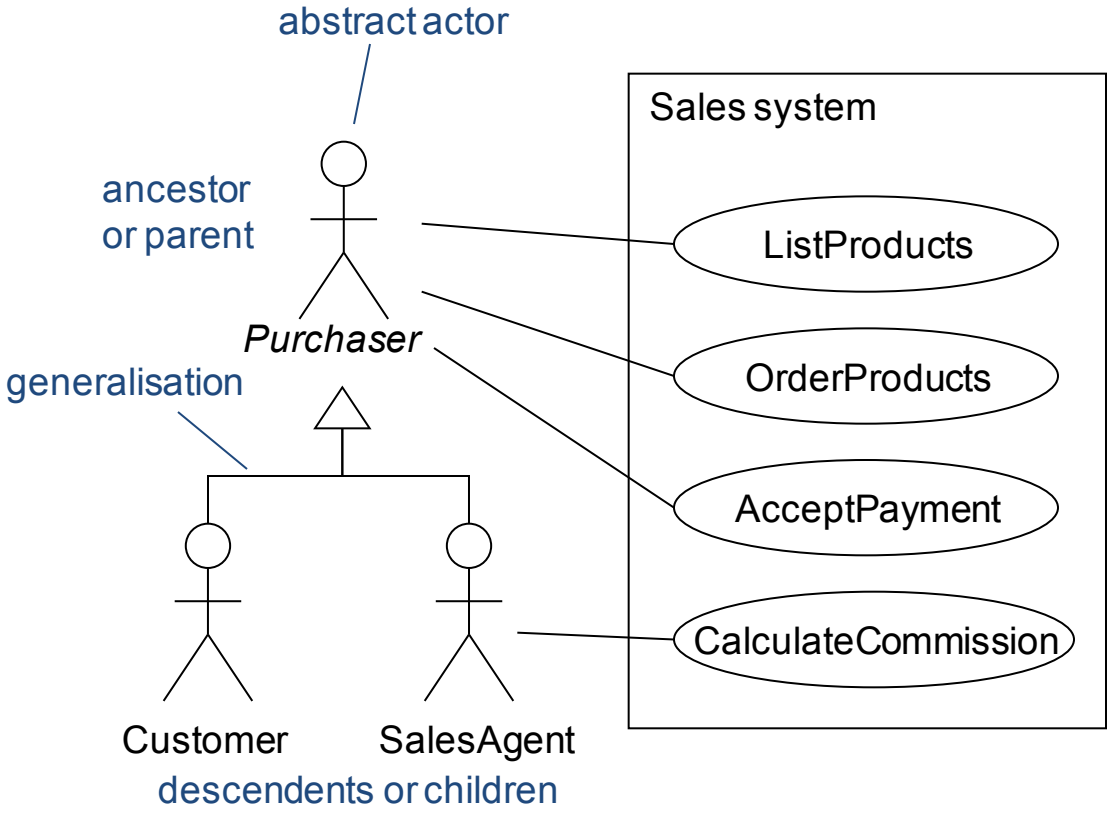
Actor generalization - example

- The Customer and the Sales Agent actors are *very* similar
- They both interact with List products, Order products, Accept payment
- Additionally, the Sales Agent interacts with Calculate commission
- Our diagram is a *mess* – can we simplify it?



Actor generalisation

- If two actors communicate with the same set of use cases in the same way, then we can express this as a generalisation to another (possibly abstract) actor
- The descendent actors inherit the roles and relationships to use cases held by the ancestor actor
- We can substitute a descendent actor anywhere the ancestor actor is expected. This is the *substitutability principle*

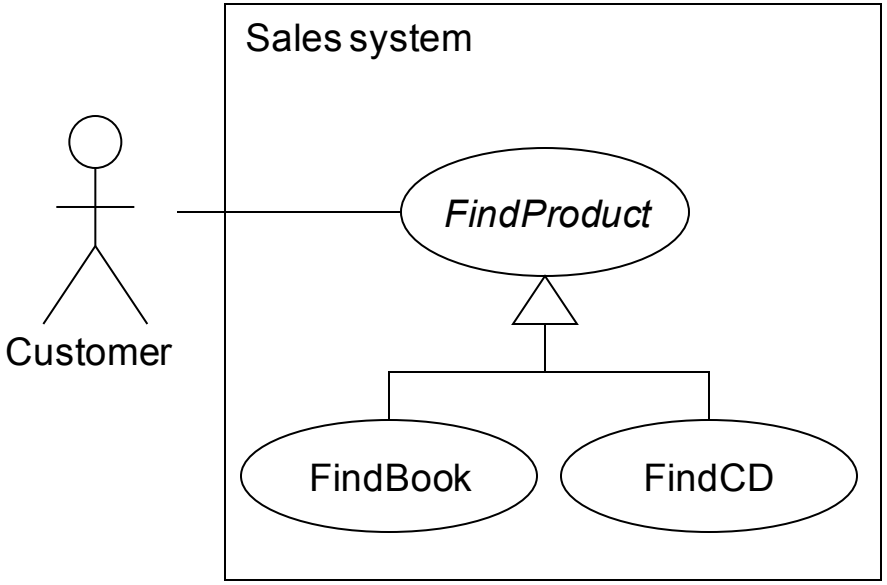


Use actor generalization when it simplifies the model

Use case generalisation

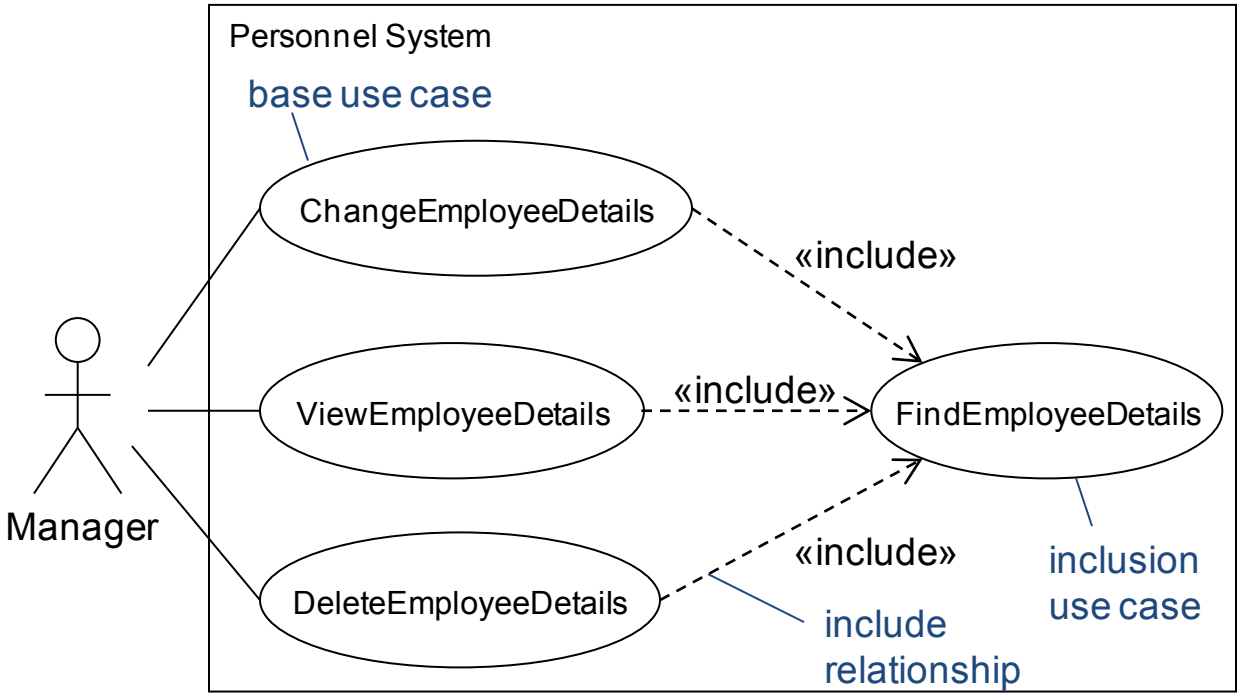
- The ancestor use case must be a more general case of one or more descendant use cases
- Child use cases are more specific forms of their parent
- They can inherit, add and override features of their parent

Use case generalization semantics			
Use case element	Inherit	Add	Override
Relationship	Yes	Yes	No
Extension point	Yes	Yes	No
Precondition	Yes	Yes	Yes
Postcondition	Yes	Yes	Yes
Step in main flow	Yes	Yes	Yes
Alternative flow	Yes	Yes	Yes



«include»

- The base use case executes until the point of inclusion: include(InclusionUseCase)
  - Control passes to the inclusion use case which executes
  - When the inclusion use case is finished, control passes back to the base use case which finishes execution
- Note:
  - Base use cases are *not complete* without the included use cases
  - Inclusion use cases may be complete use cases, or they may just specify a fragment of behaviour for inclusion elsewhere



When use cases share common behaviour we can factor this out into a separate inclusion use case and «include» it in base use cases

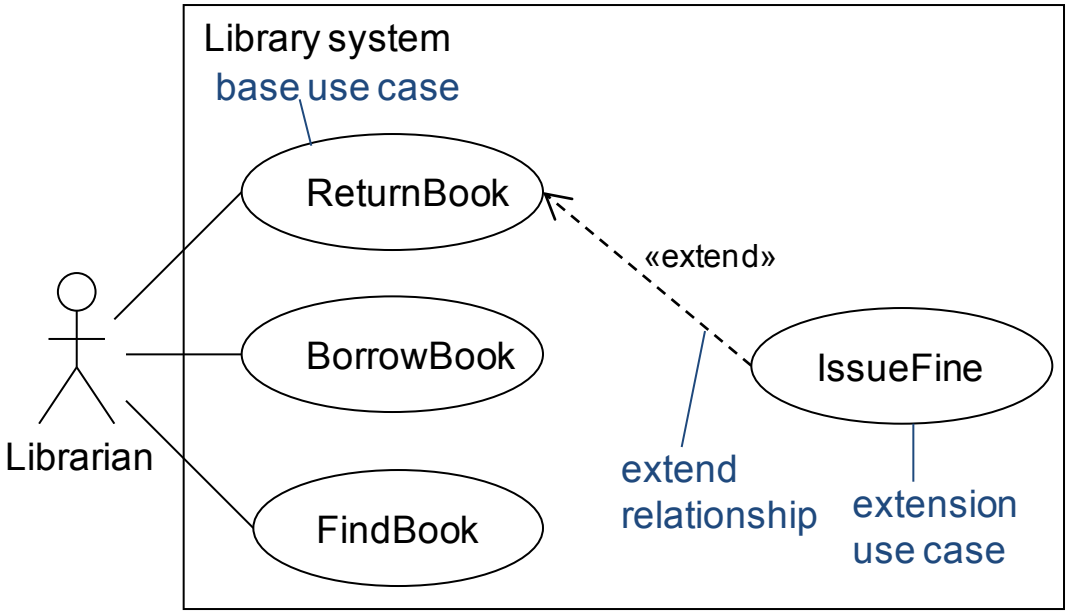
«include» example

Use case: ChangeEmployeeDetails
ID: 1
Brief description: The Manager changes the employee details.
Primary actors: Manager
Seconday actors: None
Preconditions: 1. The Manager is logged on to the system.
Main flow: 1. include(FindEmployeeDetails). 2. The system displays the employee details. 3. The Manager changes the employee details. ...
Postconditions: 1. The employee details have been changed.
Alternative flows: None.

Use case: FindEmployeeDetails
ID: 4
Brief description: The Manager finds the employee details.
Primary actors: Manager
Seconday actors: None
Preconditions: 1. The Manager is logged on to the system.
Main flow: 1. The Manager enters the employee's ID. 2. The system finds the employee details.
Postconditions: 1. The system has found the employee details.
Alternative flows: None.

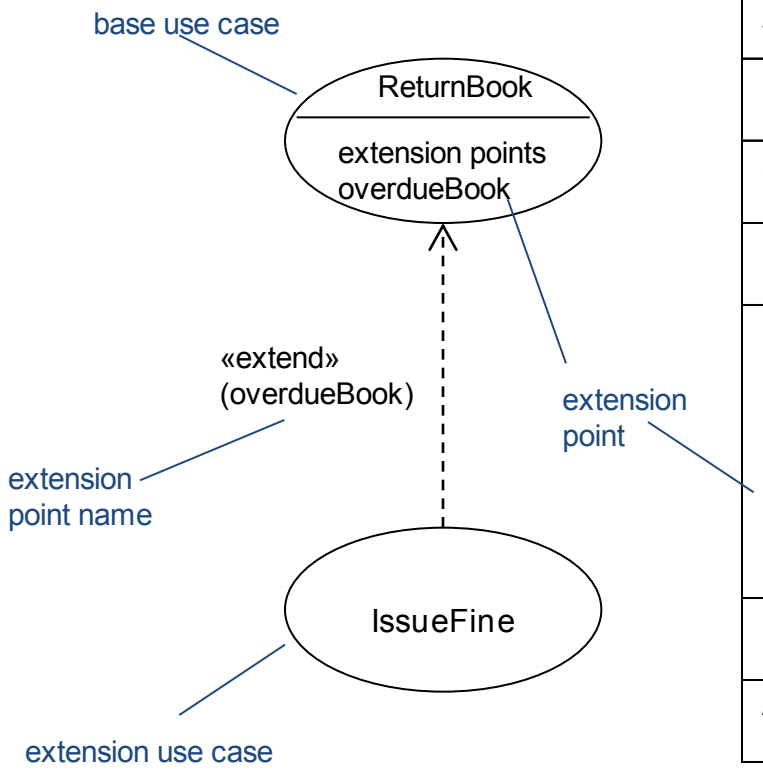
«extend»

- «extend» is a way of adding new behaviour into the base use case by inserting behaviour from one or more extension use cases
  - The base use case specifies one or more extension points in its flow of events
- The extension use case may contain several insertion segments
- The «extend» relationship may specify *which* of the base use case extension points it is extending



The extension use case inserts behaviour into the base use case.
The base use case provides extension points, but *does not know* about the extensions.

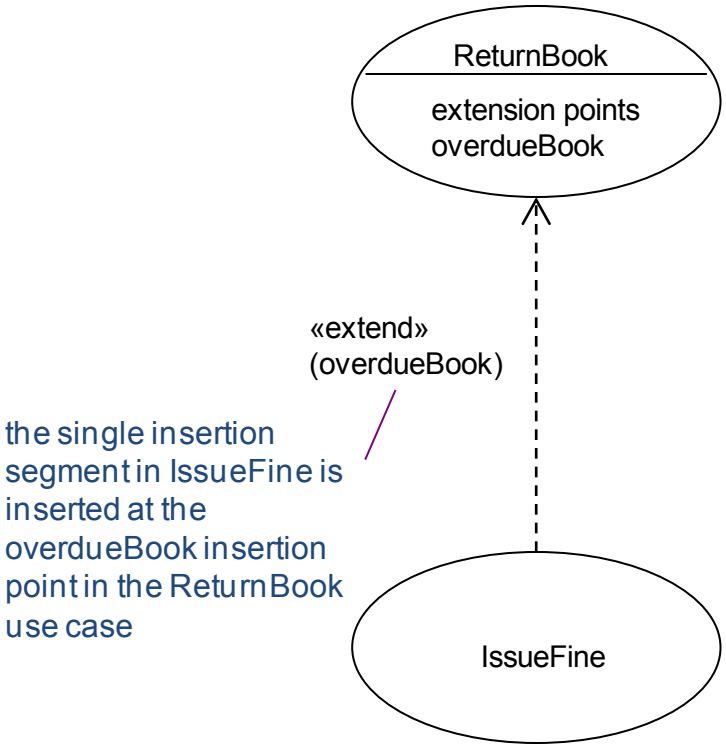
Base use case



Use case: ReturnBook
ID: 9
Brief description: The Librarian returns a borrowed book.
Primary actors: Librarian
Secondary actors: None.
Preconditions: 1. The Librarian is logged on to the system.
Main flow: 1. The Librarian enters the borrower's ID number. 2. The system displays the borrower's details including the list of borrowed books. 3. The Librarian finds the book to be returned in the list of books. extension point: overdueBook 4. The Librarian returns the book. …
Postconditions: 1. The book has been returned.
Alternative flows: None.

- There is an extension point overdueBook just before step 4 of the flow of events
- Extension points are *not* numbered, as they are *not* part of the flow

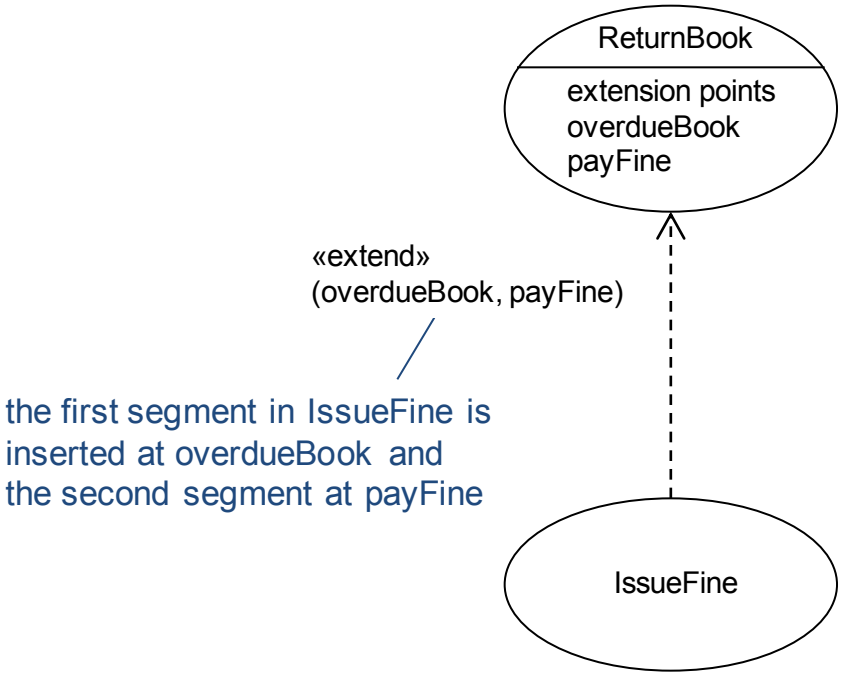
Extension use case



Extension Use case: IssueFine
ID: 10
Brief description: Segment 1: The Librarian records and prints out a fine.
Primary actors: Librarian
Secondary actors: None.
Segment 1 preconditions: 1. The returned book is overdue.
Segment 1 flow: 1. The Librarian enters details of the fine into the system. 2. The system prints out the fine.
Segment 1 postconditions: 1. The fine has been recorded in the system. 2. The system has printed out the fine.

- Extension use cases have one or more *insertion segments* which are behaviour fragments that will be inserted at the specified extension points in the base use case

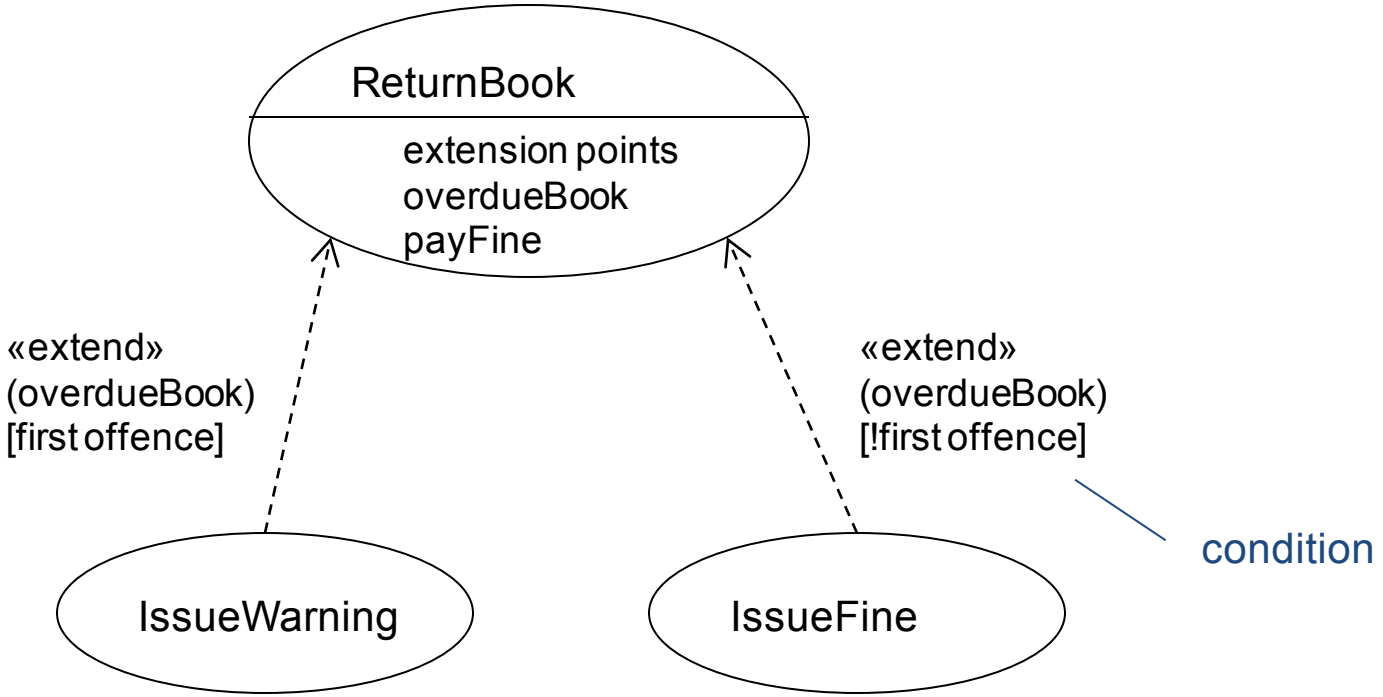
Multiple insertion points



- If more than one extension point is specified in the «extend» relationship then the extension use case must have the *same number* of insertion segments

Extension Use case: IssueFine
ID: 10
Brief description: Segment 1: The Librarian records and prints out a fine. Segment 2: The Librarian accepts payment for a fine.
Primary actors: Librarian
Secondary actors: None.
Segment 1 preconditions: 1. The returned book is overdue.
Segment 1 flow: 1. The Librarian enters details of the fine into the system. 2. The system prints out the fine.
Segment 1 postconditions: 1. The fine has been recorded in the system. 2. The system has printed out the fine.
Segment 2 preconditions: 1. A fine is due from the borrower.
Segment 2 flow: 1. The Librarian accepts payment for the fine from the borrower. 2. The Librarian enters the paid fine in the system. 3. The system prints out a receipt for the paid fine.
Segment 2 postconditions: 1. The fine is recorded as paid. 2. The system has printed a receipt for the fine.

Conditional extensions



- We can specify conditions on «extend» relationships
  - Conditions are Boolean expressions
  - The insertion is made if and only if the condition evaluates to true

Requirements tracing

- Given that we can capture functional requirements in a requirements model *and* in a use case model we need some way of relating the two
- There is a many-to-many relationship between requirements and use cases:
  - One use case covers many individual functional requirements
  - One functional requirement may be realised by many use cases
- Hopefully we have CASE support for requirements tracing:
  - With UML tagged values, we can assign numbered requirements to use cases
  - We can capture use case names in our Requirements Database
- If there is no CASE support, we can create a Requirements Traceability matrix

Requirements	Use cases			
	U1	U2	U3	U4
R1	■			
R2		■	■	
R3			■	
R4				■
R5	■			

Requirements Traceability Matrix

When to use use case analysis

- Use cases describe system behaviour from the point of view of one or more actors. They are the *best* choice when:
  - The system is dominated by functional requirements
  - The system has many types of user to which it delivers different functionality
  - The system has many interfaces
- Use cases are designed to capture *functional* requirements. They are a *poor* choice when:
  - The system is dominated by non-functional requirements
  - The system has few users
  - The system has few interfaces

Key points

- We have seen how to capture functional requirements with use cases
- We have looked at:
  - Use cases
  - Actors
  - Branching with IF
  - Repetition with FOR and WHILE
  - Alternative flows

Key points

- We have learned about techniques for advanced use case modelling:
  - Actor generalisation
  - Use case generalisation
  - «include»
  - «extend»
- Use advanced features with discretion only where they simplify the model!