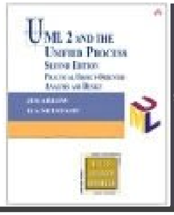
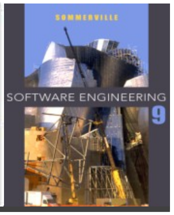


UML 2 and the Unified Process

Second Edition



SOMMERVILLE

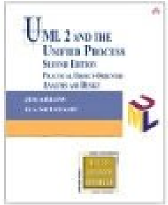
SOFTWARE ENGINEERING

9

Requirements Engineering

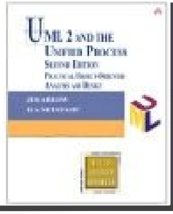
Lecture 2

Requirements engineering (RE)

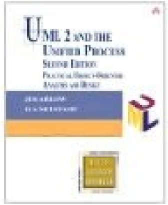


- **Requirements** are descriptions of system **services** and **constraints** under which the system operates and is developed.
  - It may range from a **high-level abstract statement** of a service or of a system constraint to a detailed **mathematical functional specification**.
- **Requirements engineering**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

Outline



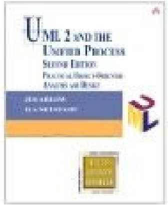
- ✧ Requirements and their types
- ✧ Requirements engineering process
    - Requirements elicitation and analysis
    - Requirements validation
    - Requirements management
- ✧ UML Use Case diagram

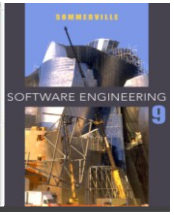


Requirements and their Types

Lecture 2/Part 1

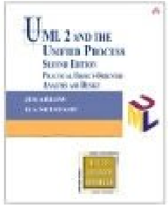
Types of requirements

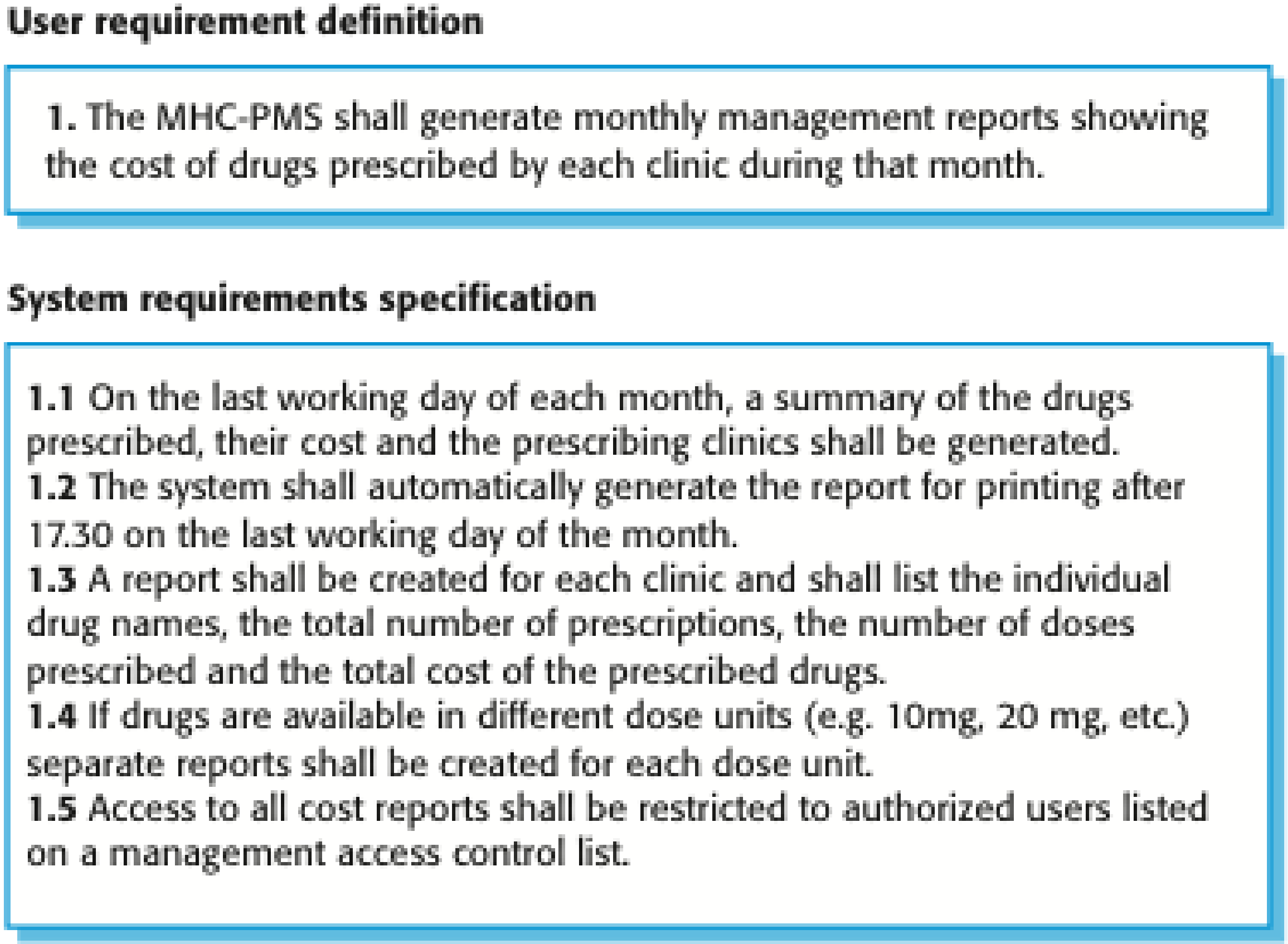




- ✧ User requirements
  - Statements of the services the system provides to the users and its operational constraints.
    - In natural language and/or diagrams.
  - For client managers, client engineers and system architects.
- ✧ System requirements
  - **Detailed descriptions** of the system’s functions, services and operational constraints. Define **what should be implemented**.
    - In structured document and/or diagrams.
  - For client engineers, system architects and system developers.
- ✧ Which of them are more abstract/concrete?

User and system requirements





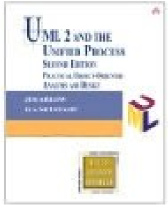
User requirement definition

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

System requirements specification

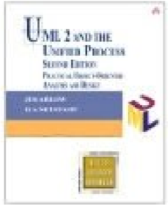
1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic 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.

Functional and non-functional requirements



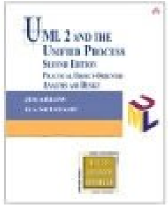
- Functional requirements
  - Statements of **services the system provides**, how the system should react to particular inputs and how the system should **behave** in particular situations.
  - E.g. *A user **shall be able to** search the appointments lists for all clinics.*
- Non-functional requirements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

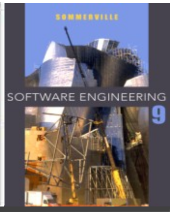
Requirements quality criteria



- ✧ Precise
  - Is there only **one interpretation** for the requirement in the system context?
- ✧ **Complete**
  - Are **all functions** required by the customer **included**?
- ✧ **Consistent**
  - Are there any requirements **conflicts** or **contradictions**?

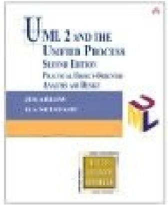
Additional quality criteria



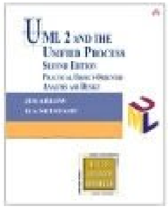


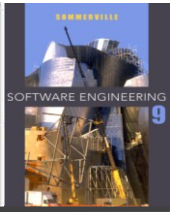
- ✧ Comprehensibility
  - Are all requirements properly understood?
- ✧ Realism
  - Can the requirements be implemented given available budget and technology?
- ✧ Verifiability
  - Can the requirements be checked?
- ✧ Traceability
  - Is the origin of the requirement clearly stated?
- ✧ Adaptability
  - Can the req. be changed without a large impact on other req.?

Key points



- Requirements = services + constraints
- Types of requirements
  - Vertically – user vs. system requirements
  - Horizontally – functional vs. non-functional requirements
- Quality criteria
  - Precision, completeness, consistency.
  - Comprehensibility, realism, verifiability, traceability, adaptability.

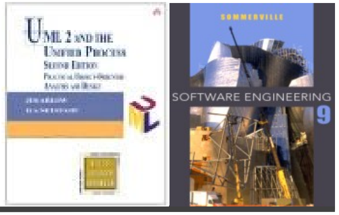




Requirements Engineering Process

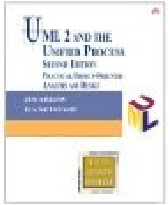
Lecture 2/Part 2

Outline



- Requirements elicitation and analysis
- Requirements validation
- Requirements management

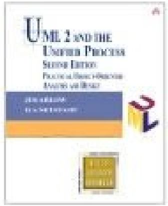
Requirements engineering processes

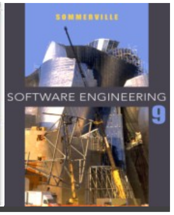


- The processes used for RE vary widely depending on the **application domain**, the **people** involved and the **organisation** developing the requirements.
- However, there are a number of generic activities **common to all processes**
  - Requirements elicitation and analysis;
  - Requirements validation;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

- Is there a relation to Boehm’s model from Lecture 1?

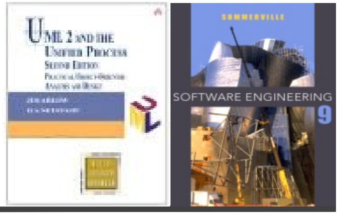
Requirements elicitation and analysis

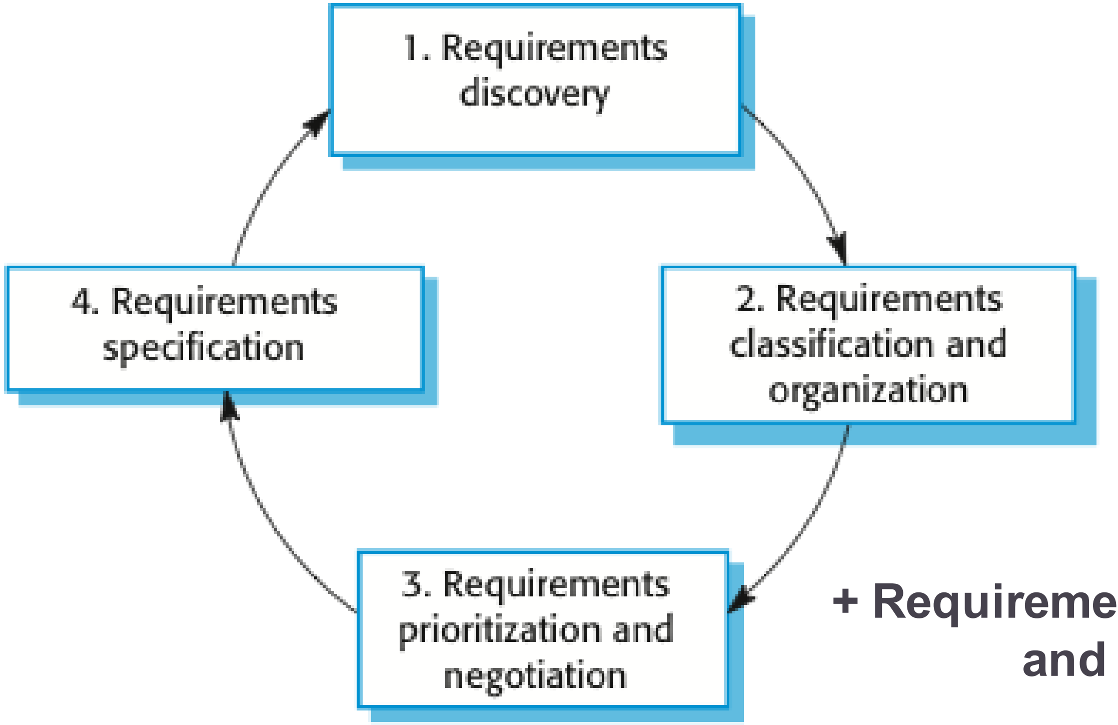




- Software engineers work with system **stakeholders**:
  - end-users, managers, maintenance engineers, domain experts, trade unions, etc.
- To find out about:
  - the application domain,
  - the services that the system should provide,
  - the required system performance,
  - hardware constraints,
  - other systems, etc.
- As far as possible, it should set of **WHAT the system should do** rather than HOW it will do (implement) it.

The requirements elicitation and analysis process



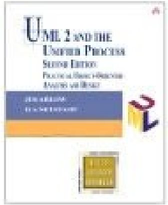


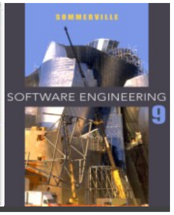
1. Requirements discovery
2. Requirements classification and organization
3. Requirements prioritization and negotiation
4. Requirements specification

+ Requirements verification and validation

Where shall this activity come?

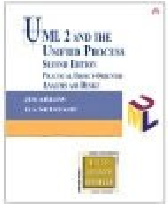
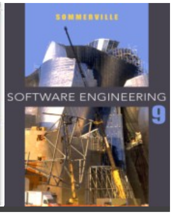
Process activities





- Requirements discovery
  - Interacting with stakeholders and studying existing processes and needs to discover their requirements.
- Requirements classification and organisation
  - Groups related requirements and organises them into clusters.
- Prioritisation and negotiation
  - Prioritising requirements and resolve stakeholder conflicts.
- Requirements specification
  - Requirements are documented and input into the next iteration.
- Requirements verification and validation
  - Identify and resolve requirements quality problems

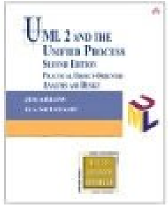
Requirements discovery

- Questionnaires
- Interviews
  - Small number of software engineers and stakeholders
- Workshops
  - Large group of interested parties; free brainstorming
- Ethnography
  - Observe existing processes

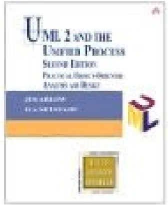
- Is there a recommended order if the techniques are combined?

Interviews and workshops



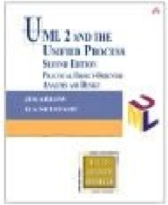
- ✧ Formal or informal interviews with stakeholders are part of most RE processes.
- ✧ Types of interview
  - **Closed interviews** based on pre-determined list of questions
  - **Open interviews** where various issues are explored
  - **Workshops** with brainstorming of all involved stakeholders
- ✧ 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.

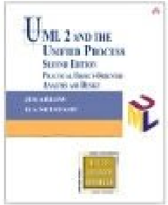
Requirements classification and prioritisation



- 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

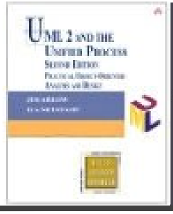
Requirements specification



Notation	Description
Natural language	Numbered sentences in natural language, each sentence expressing one requirement. *E.g. Project assignment.*
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. *E.g. Textual specification of UML use cases. (the table view)*
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. *E.g. Main flow in the UML UC textual specification.*
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system. *E.g. UML use case and activity diagrams.*
Mathematical specifications	Notations based on mathematical concepts; *E.g. finite-state machines or sets*. Although they can reduce the ambiguity in a requirements document, most customers don’t understand them and are reluctant to accept it as a system contract

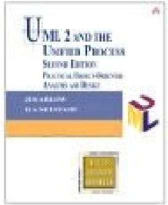
✧ When shall we choose mathematical specification?

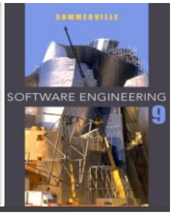
Requirements verification and validation



- ✧ Requirements verification
  - Concerned with checking requirements precision, completeness, consistency, comprehensibility, realism, verifiability, traceability, and adaptability (to the expected extent).
- ✧ Requirements validation
  - Concerned with checking that the requirements **define the system that the customer really wants**.
- ✧ Requirements error costs are high so verification and 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 techniques





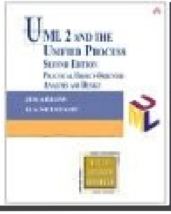
✧ Requirements reviews

- Systematic manual analysis of the requirements.
- Both client and contractor staff should be involved.
- Reviews may be **formal** (with completed documents) or **informal** (relying on good communications between developers, customers and users).

✧ Prototyping

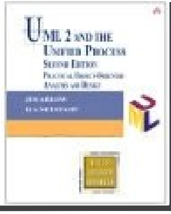
- Using an executable model of the system to check customer satisfaction.

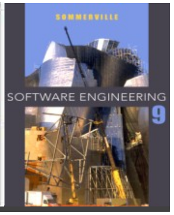
Requirements management

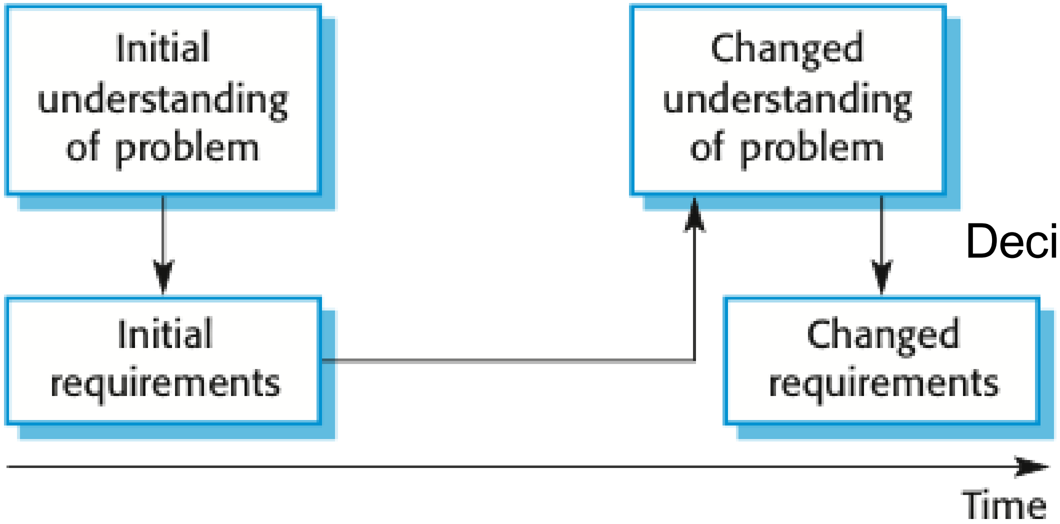


- Requirements management is the process of **managing changing requirements** during the requirements engineering process and system development.
- New requirements emerge as a system is being developed and tested by the users. Some due to **business**, **organizational** and **technical changes**.
- Traceability and maintenance of **links between dependent requirements** is important to assess the impact of requirements changes.
- We may need a formal process for making **change proposals** and linking these to system requirements.

Requirements evolution





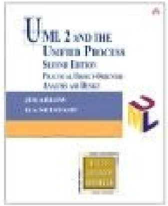

Initial understanding of problem
Changed understanding of problem
Initial requirements
Changed requirements
Time

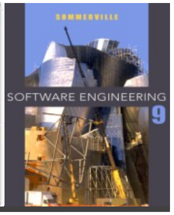
Decide what changes shall be accepted

✧ Each requirements change should be analysed before deciding whether to accept it.

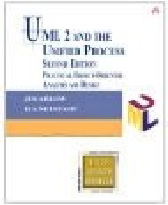
- Analyse the problem, check the validity of the change proposal
- Asses the effects of the change, via traceability information
- Integrate the change in the specification documents

Key points





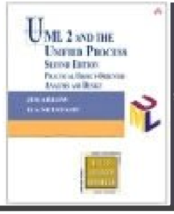
- Requirements engineering is an **iterative activity**.
- Requirements **discovery**
  - Questionnaires, interviews, workshops, ethnography
- Requirements **prioritization**
  - MoSCoW, RUP attributes
- Requirements **specification**
- Requirements **verification and validation**
- Requirements **management** and **evolution**



UML Use Case Diagram

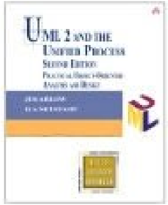
Lecture 2/Part 3

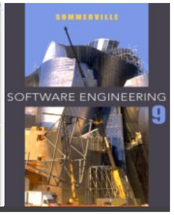
Outline



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

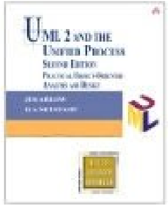
The purpose of Use Case modelling



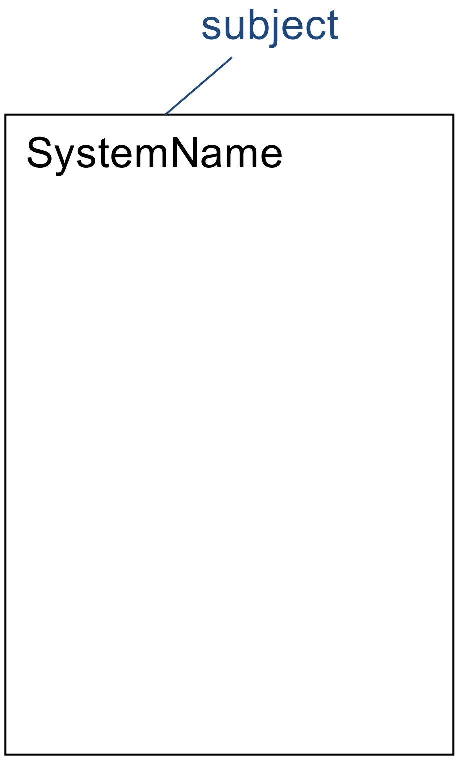


- ✧ Use case modelling is a form of requirements engineering
- ✧ Use case modelling proceeds as follows:
  - Find the system boundary
  - Find actors – who or what uses the system
  - Find use cases – what functions the system should offer
  - Specify use cases – with textual specification or UML Activity Diagrams

The subject

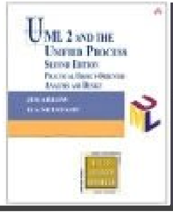


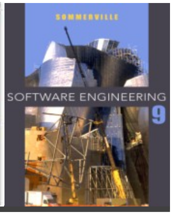
- We create a Use Case model containing:
  - **Subject** – the edge of the system
    - also known as the system boundary
  - **Actors** – who or what uses the system
  - **Use Cases** – things actors do with the system; functions the system should offer to its users
  - **Relationships** – between actors and use cases



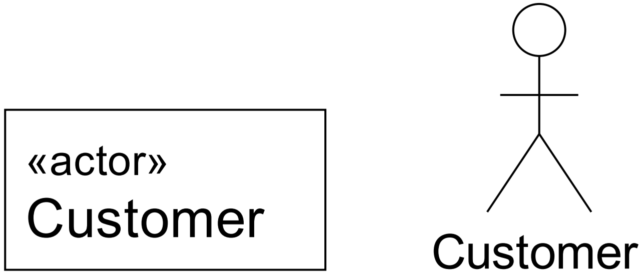
- Can there be a direct relationship between actors?

What are actors?

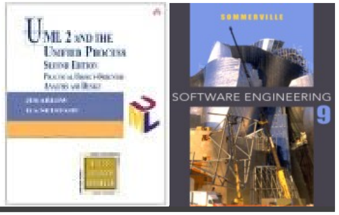




- An actor is anything that interacts **directly** with the system
  - Actors identify who or what uses the system and so indicate where the system boundary lies
- Actors are **external** to the system
- An Actor specifies a **role** that some external entity adopts when interacting with the system
  - Can one actor represent two physical persons?
  - Can one physical person match to two actors?
  - Can there be two actors with the same name in the model?

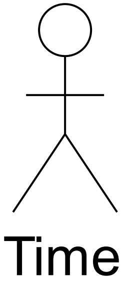


Identifying Actors



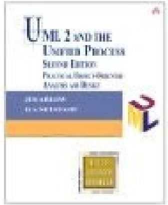
✧ When identifying actors ask:

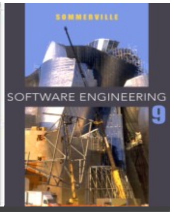
- Who or what uses the system?
- What roles do they play in the interaction?
- Who installs the system?
- Who starts and shuts down the system?
- Who maintains the system?
- What other systems use this system?
- Who gets and provides information to the system?
- Does anything happen at a fixed time?



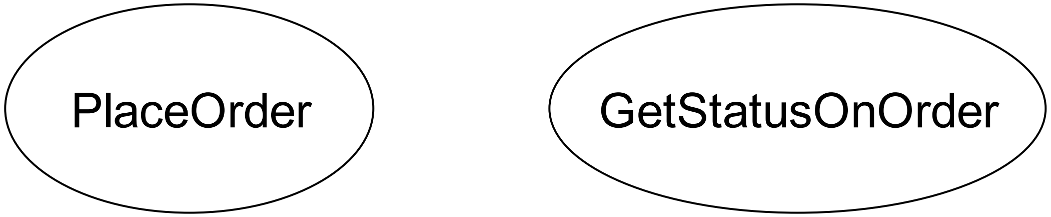
✧ What if the actor is not a human? What can it be?

What are use cases?



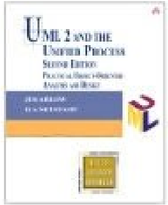


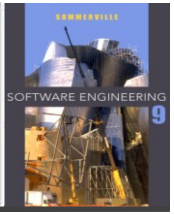
- ✧ A use case is something an actor needs the system to do. It is a “case of use” of the system by a specific actor.
- ✧ Use cases are always started by an actor
  - The **primary actor** triggers the use case
  - Zero or more **secondary actors** interact with the use case in some way
  - Does the UC diagram tell me which actor is primary/secondary?
- ✧ Use cases are always written from the **point of view of the actors**.


PlaceOrder

GetStatusOnOrder

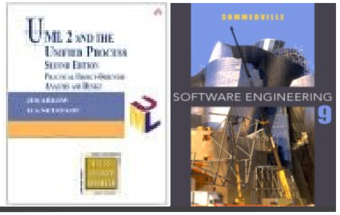
Identifying use cases

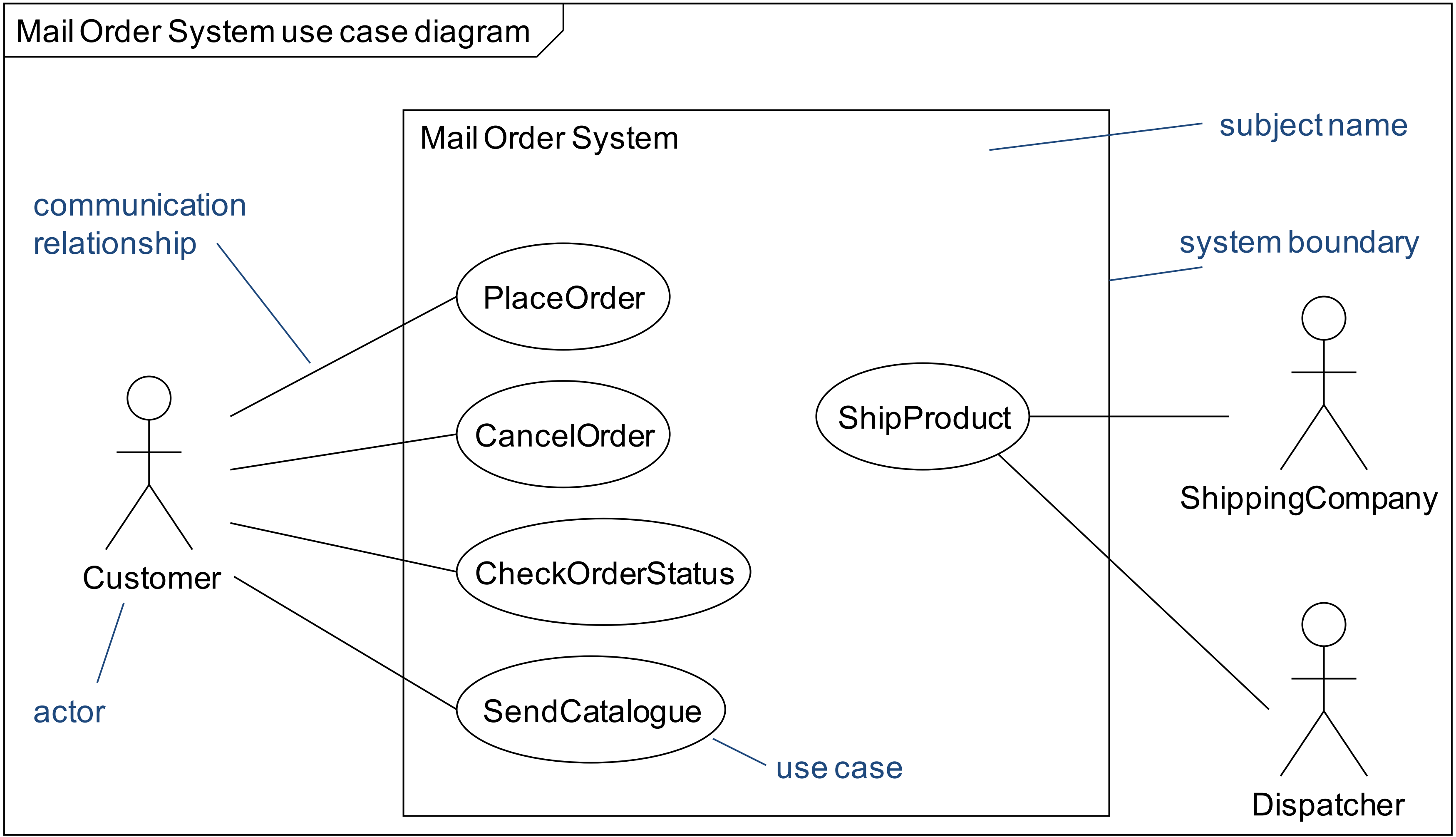




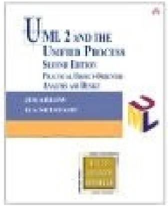
- Start with the list of actors that interact with the system
- When identifying use cases ask:
  - What functions will a specific actor want from the system?
  - Does the system store and retrieve information? If so, which actors trigger this behaviour?
  - What happens when the system changes state (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?

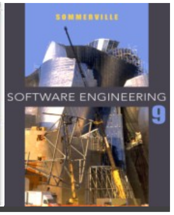
The use case diagram





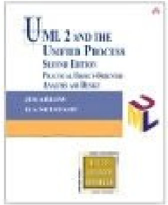
Textual use case specification





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: implicit time actor 1. The use case starts when it is the end of the business quarter. 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.

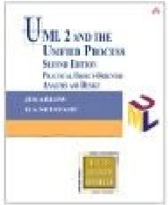
Naming use cases

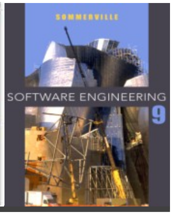


- ✧ Use cases describe something that happens
- ✧ They are named using **verbs** or **verb phrases**
- ✧ Naming standard [1]: use cases are named using UpperCamelCase e.g. PaySalesTax

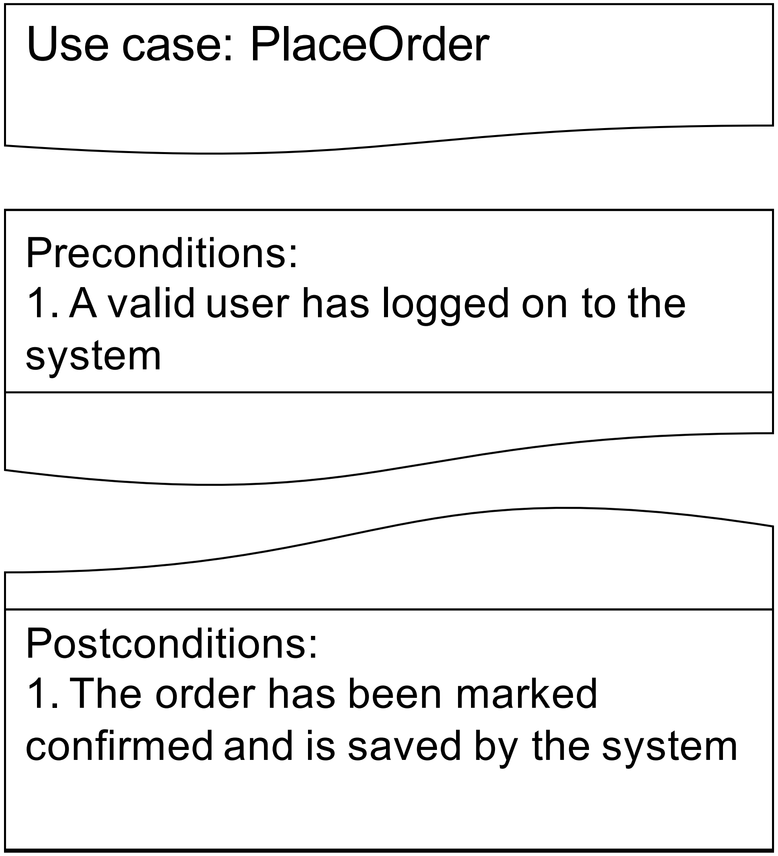
[1] UML 2 does not specify any naming standards.
All naming standards here are based on industry best practice.

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
- ✧ What pre/postconditions does a delete of a product have?
- ✧ What about if the deletion is not successful?

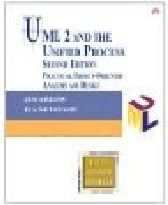


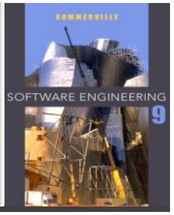
Use case: PlaceOrder

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

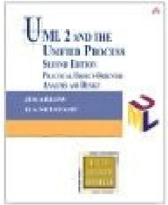




<number> The <something> <some action>

- 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* scenario
  - Everything goes as expected, without errors, deviations and interrupts
  - Alternatives can be shown by branching or by listing under Alternative flows (see later)

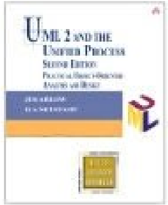
Branching within a flow: IF

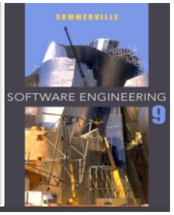


- ✧ Use the keyword IF to indicate alternatives within the flow of events
  - There must be a Boolean expression immediately after IF
- ✧ Use indentation and numbering to indicate the conditional part of the flow
- ✧ Use ELSE to indicate what happens if the condition is false

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

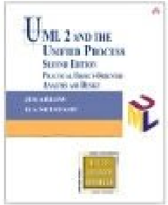
Preconditions:
None.

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

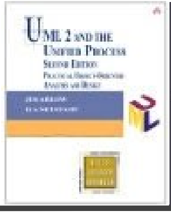
Repetition within a flow: WHILE

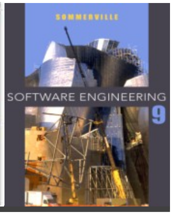


- ✧ We can use the keyword WHILE to indicate that something repeats while some Boolean condition is true

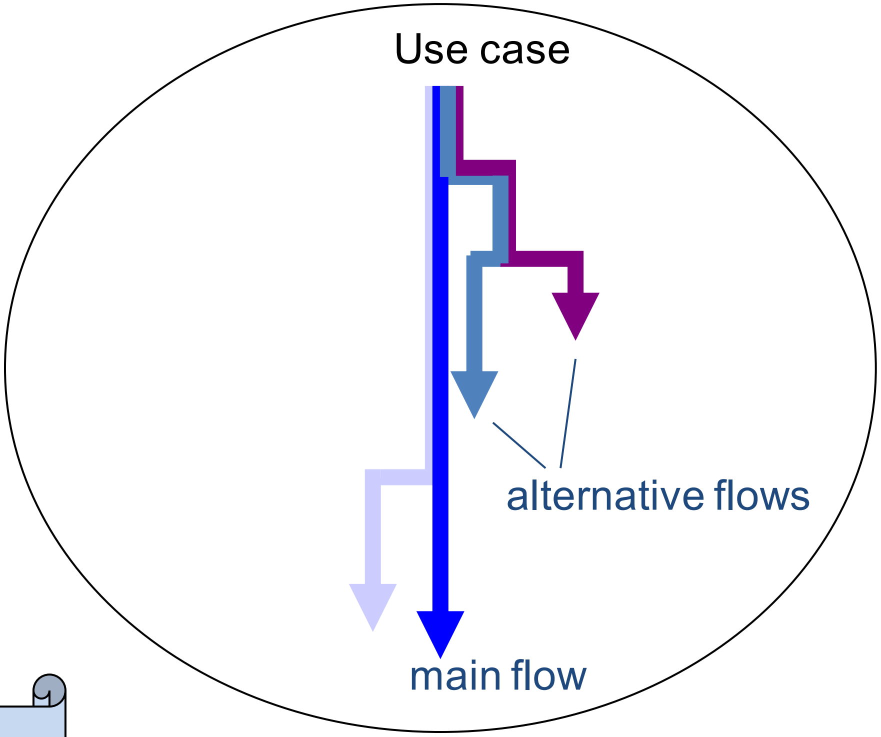
Use case: ShowCompanyDetails
ID: 4
Brief description: The system displays the company details to the Customer.
Primary actors: Customer
Secondary actors: None
Preconditions: None.
Main flow: 1. The use case starts when the Customer selects "show company details". 2. The system displays a web page showing 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



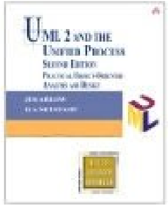


- ✧ Alternative flows capture errors, branches, and interrupts
- ✧ They can often be **triggered *at* any time** during the main flow
- ✧ Alternative flows **never return to the main flow**



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:

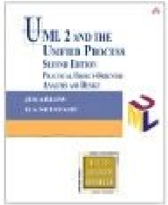
1. The use case begins when the Customer selects "create new customer account".
2. WHILE the Customer details are invalid
   - 2.1. The system asks the Customer to enter his or her details comprising email address, password and password again for confirmation.
   - 2.2 The system validates the Customer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

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

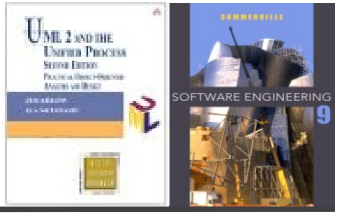
Alternative flows:
InvalidEmailAddress
InvalidPassword
Cancel

Alternative flows

Advanced Use Case modelling



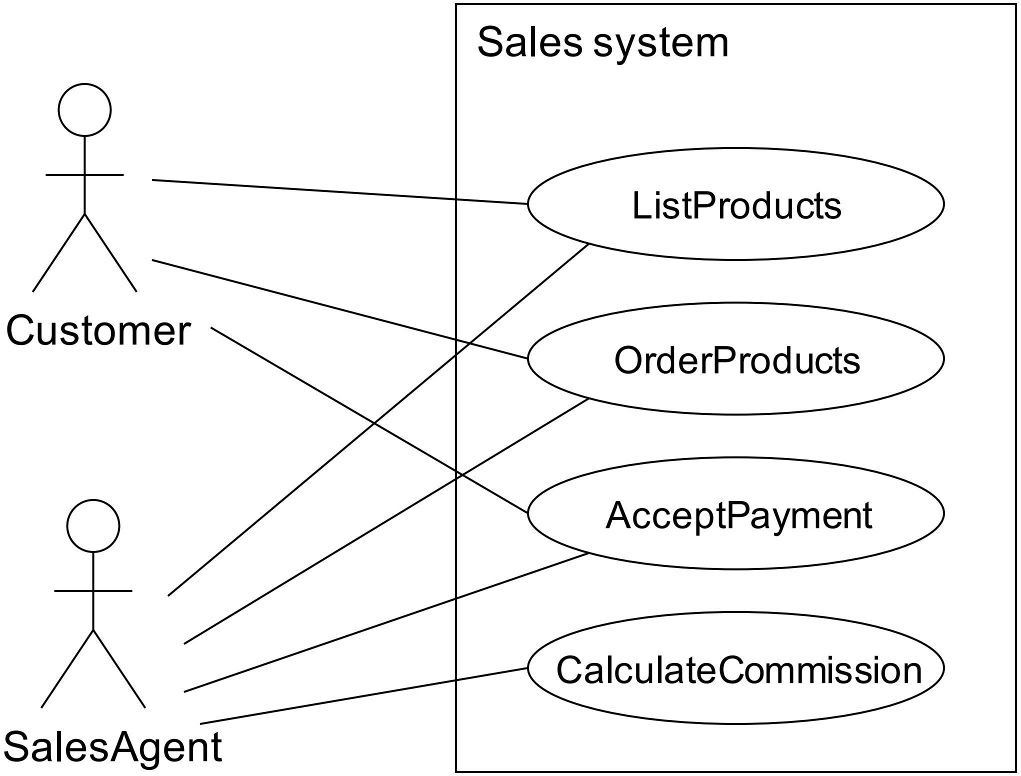
- We have studied basic use case analysis, but there are **relationships** that we have still to explore:
  - Actor generalisation
  - Use case generalisation
  - «include» – between use cases
  - «extend» – between use cases



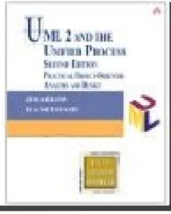
Actor generalization - example

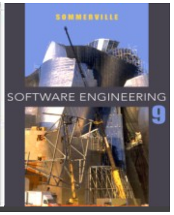
- The Customer and the Sales Agent actors are **very similar**
- They both interact with List products, Order products, Accept payment
- They both can play the **purchaser role**.

- Can we always generalize two actors sharing some use cases?

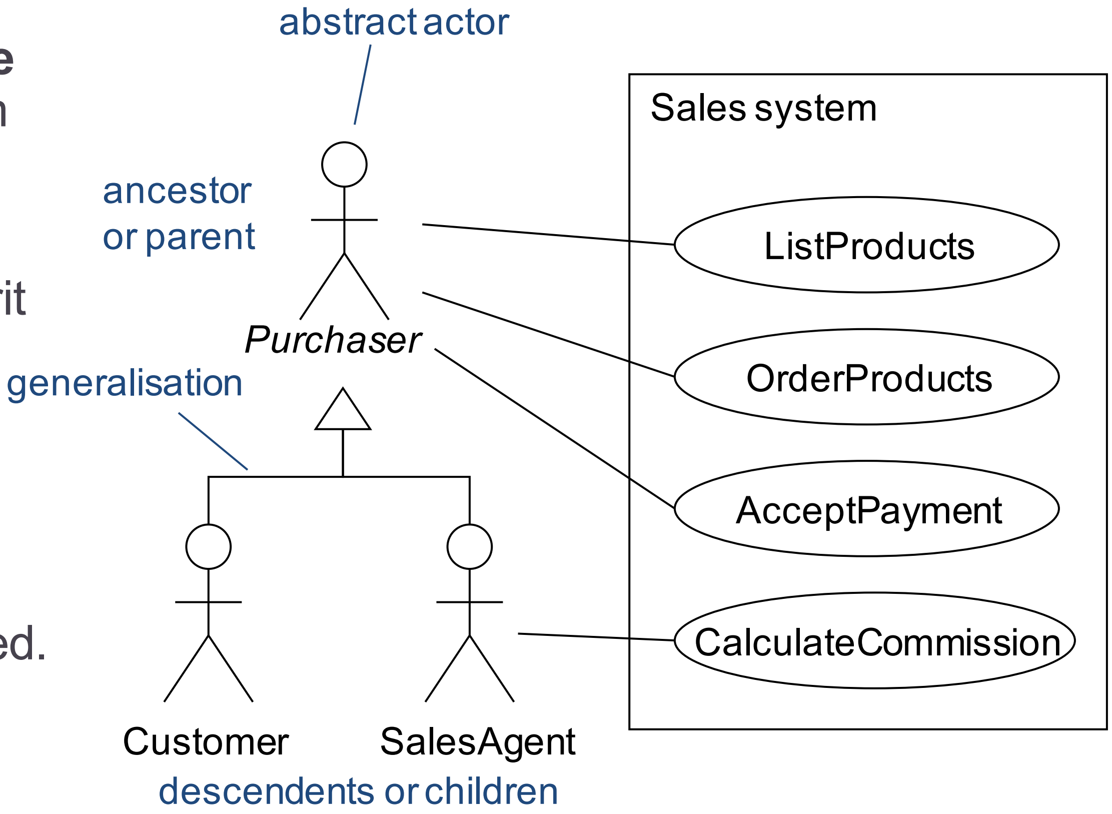


Actor generalisation



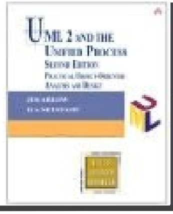


- If two actors **share the same sub-role**, which makes them communicate with the same set of use cases
- The descendent actors inherit the roles and relationships to use cases held by the **ancestor** actor
- We can substitute a descendent actor anywhere the ancestor actor is expected. This is the **substitutability principle**
- Can we always generalize two actors sharing some use cases?

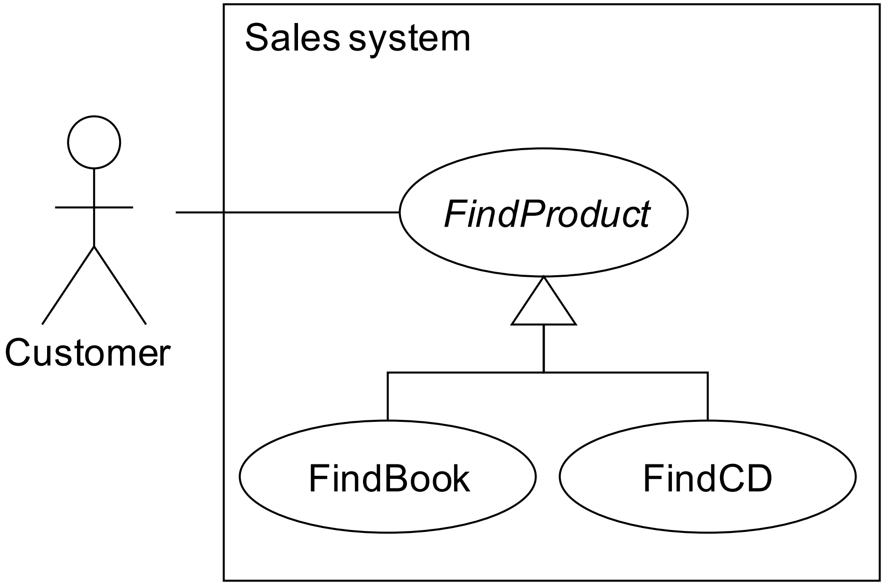


Use actor generalization when it simplifies the model

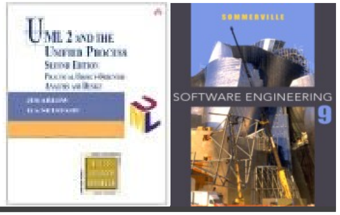
Use case generalisation



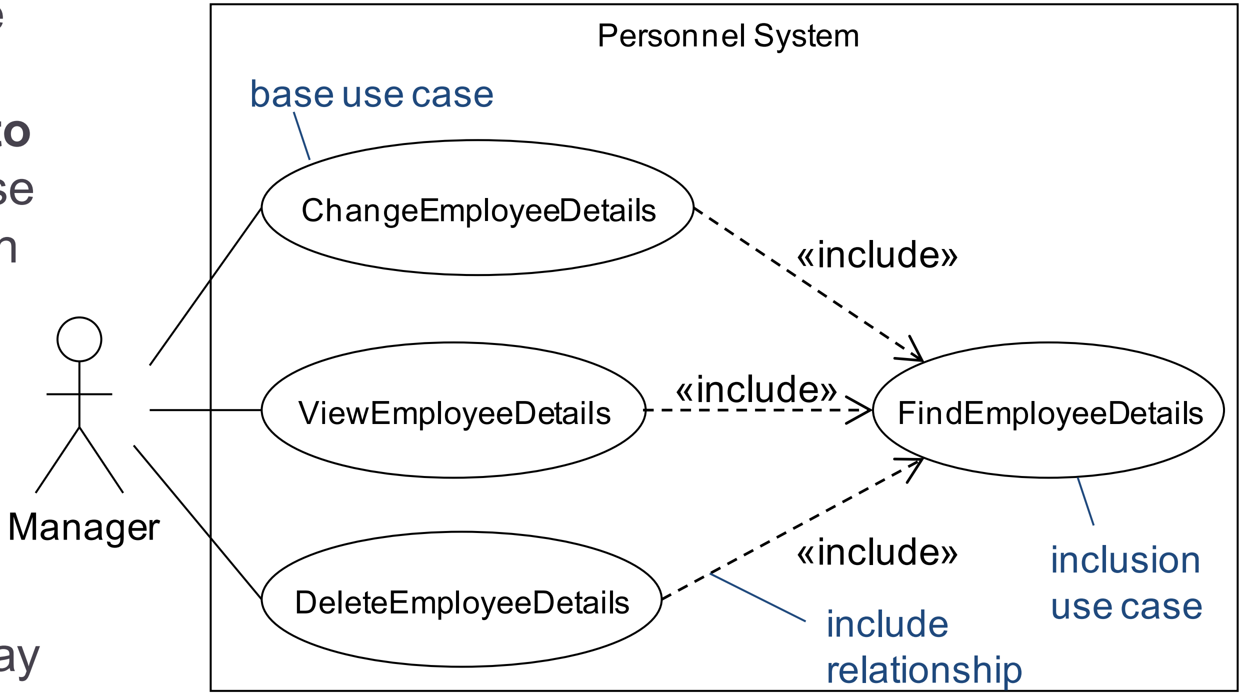
- The ancestor use case must be a more general case of one or more descendant use cases
- Child use cases are more specific forms of their parent
- They can inherit, add and override features of their parent

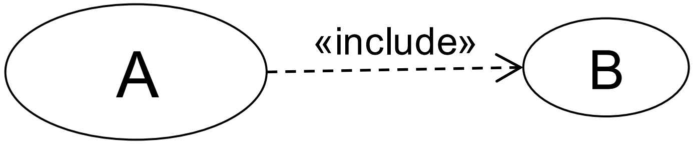


«include»

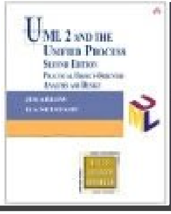


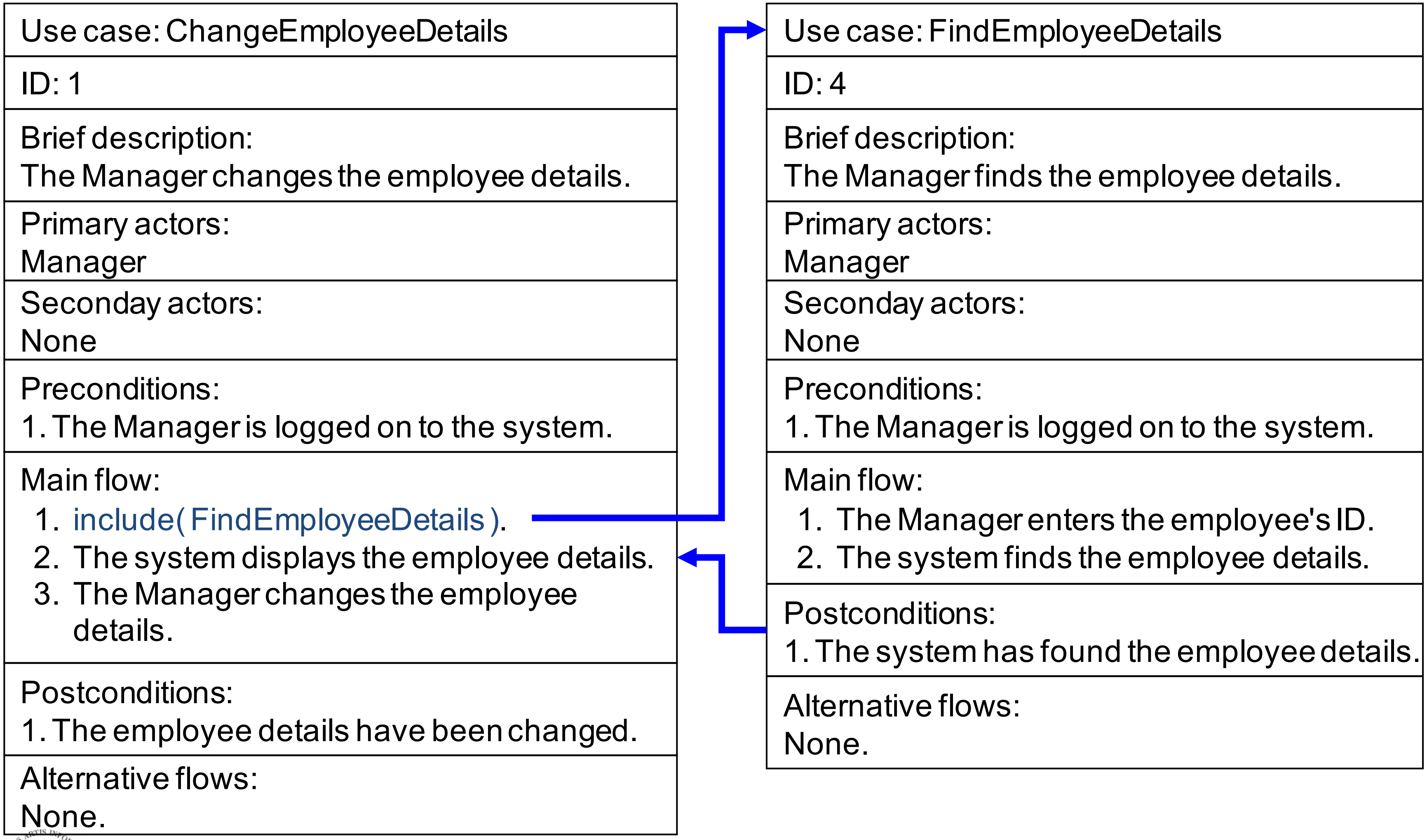
- When use cases share common behaviour we can **factor this out into** a separate inclusion use case and «include» it in base use cases
- Base use cases are **not complete** without the included use cases
- Inclusion use cases may be complete use cases, or they may just specify a **fragment of behaviour** for inclusion elsewhere





«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.

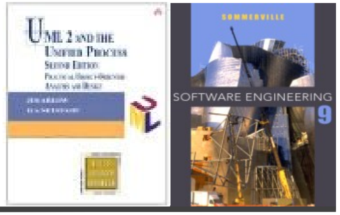
Main flow:

1. The Manager enters the employee's ID.
2. The system finds the employee details.

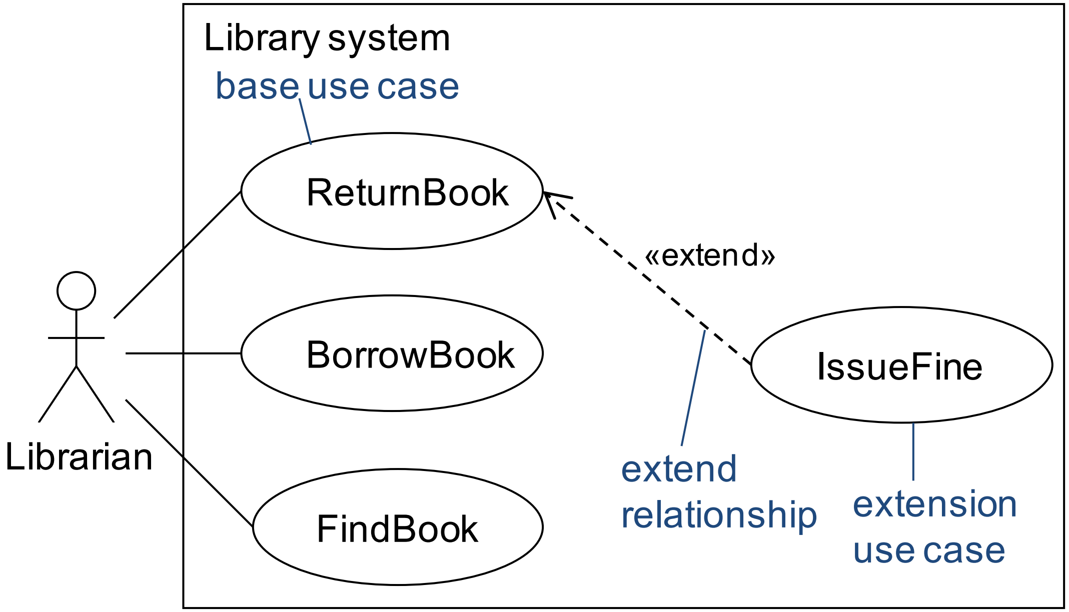
Postconditions:
1. The system has found the employee details.

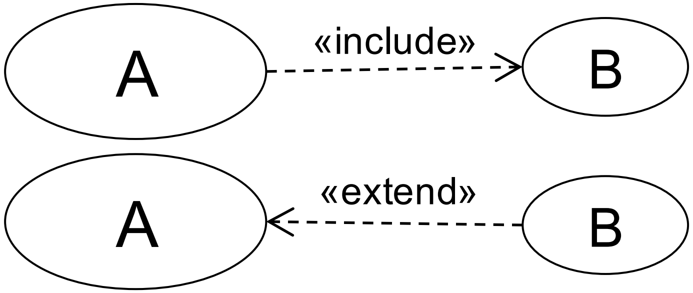
Alternative flows:
None.

«extend»

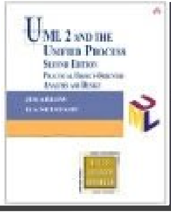


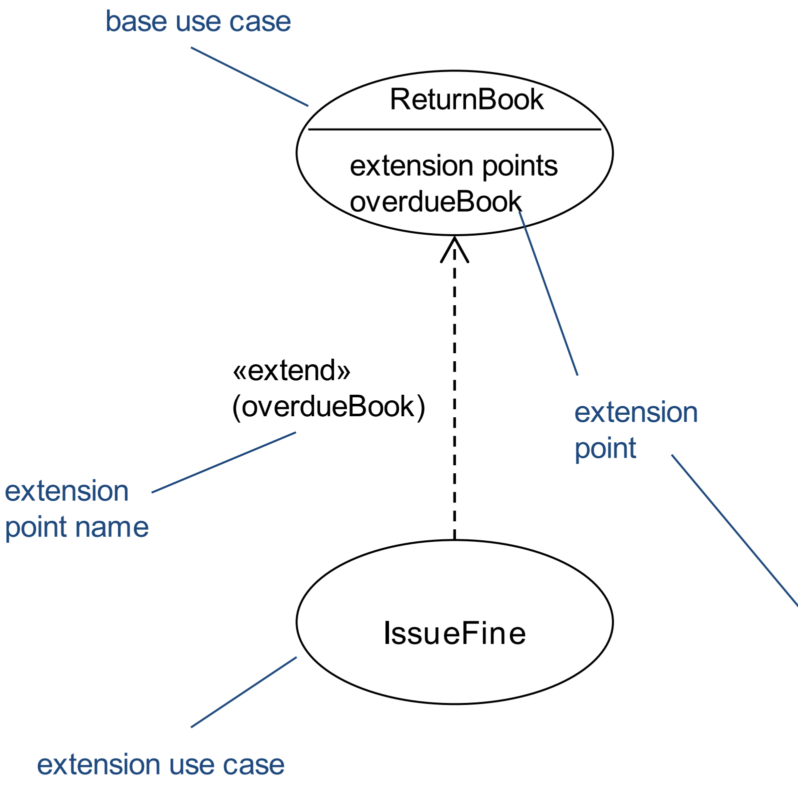
- The extension use case **inserts behaviour** into the base use case.
- The base use case provides **extension points,** but **does not know** about the extensions.
- The base use case is **complete** already without the extensions.
- There may be multiple extension points and multiple extending use cases.





<<extend>> example

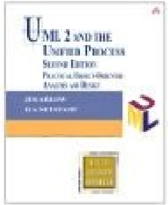


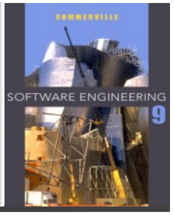


Use case: ReturnBook
ID: 9
Brief description: The Librarian returns a borrowed book.
Primary actors: Librarian
Secondary actors: None.
Preconditions: 1. The Librarian is logged on to the system.
Main flow: 1. The Librarian enters the borrower's ID number. 2. The system displays the borrower's details 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.

- Extension points are *not* numbered, as they are *not* part of the flow

Requirements tracing





There is a many-to-many relationship between requirements and use cases:

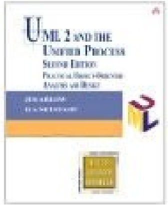
- One use case covers many individual functional requirements
- One functional requirement may be realised by many use cases

✧ **Requirements Traceability Matrix** can help us to trace if all requirements are covered by our use case model

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

Requirements Traceability Matrix

Key points



- Use cases describe system behaviour from the **point of view of 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**
- We have discussed:
  - **Actors**, **use cases** and their **textual specification**
  - Actor and use case **generalization**
  - Advanced relationships between use cases (**include, extend**)
- Use advanced features only where they simplify the model!