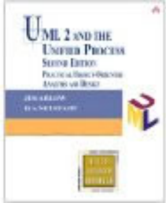
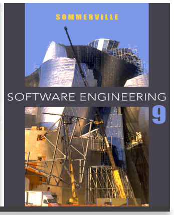




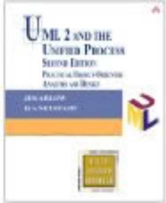
Lecture 1

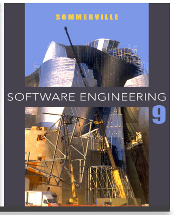
SOFTWARE DEVELOPMENT

PB007 Software Engineering I
Faculty of Informatics, Masaryk University

Fall 2016

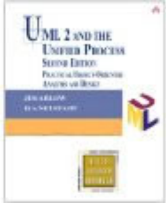
Outline

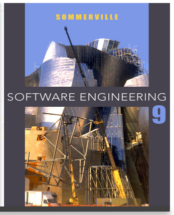




- Course organization
- Software development

- UML in software development
- UML Use Case diagram

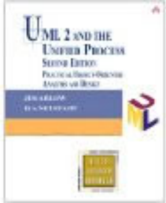


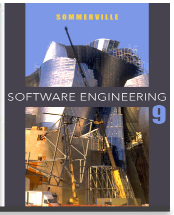


Course Organization

Lecture 1/Part 1

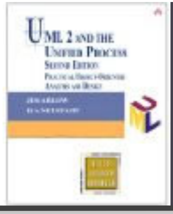
About the lecturer:
Ing. RNDr. Barbora Bühnová, Ph.D.

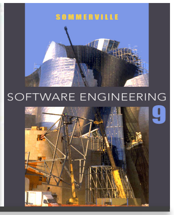




- Industrial experience
- Research
  - Quality of software architecture
  - Lab of Software Architecture and Information Systems (LaSArIS)
- Teaching
  - Courses on UML, software quality, architecture design, programming, algorithm design, and others
- Collaboration with students
  - Bachelor/Master theses (Honeywell, Siemens)
  - Seminar tutoring

About the course:
PB007 Software Engineering I

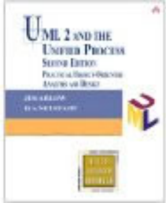


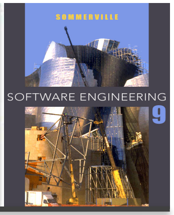


✧ Lectures

1. **Software development**, UML Use Case diagram.
2. **Requirements specification**, UML Activity diagram.
3. System analysis and design, structured vs. object-oriented A&D.
4. **Object oriented analysis**, UML Class, Object and State diagram.
5. **Structured analysis**, data modelling, ERD.
6. **High-level design**, UML Class diagram in design.
7. **Low-level design and implementation**, UML Interaction diagrams
8. **Architecture design**, UML Package, Component and Deployment diagram.
9. **Testing**, verification and validation.
10. **Operation**, maintenance and system evolution.
11. Software development management.
12. Advanced software engineering techniques.

About the course:
PB007 Software Engineering I

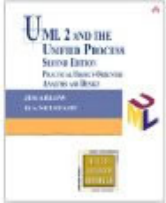


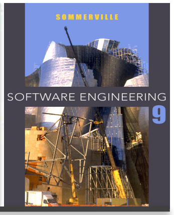


✧ Seminars

1. Visual Paradigm introduction, project assignment.
2. Project start, initial **Use Case diagram**.
3. Detailed **Use Case diagram**, textual specification of UC
4. Specification of use cases, **Activity diagram**.
5. Analytical **Class diagram**, **Object diagram**.
6. Analytical **Class diagram**, update of UC diagram, **interaction of objects**.
7. **State diagram**.
8. Data modelling, **Entity Relationship diagram**.
9. Design-level **Class diagram**, interfaces, implementation details.
10. Refinement of use cases with **Interaction diagrams**.
11. Finalization of **Interaction diagrams**, Class diagram update.
12. Packages, **Component diagram**, **Deployment diagram**.

About the course:
PB007 Software Engineering I





✧ **Prerequisites**

- Basic knowledge of object oriented programming

✧ **Lectures**

- 12 teaching weeks + 1 week free (28. 10. 2016)

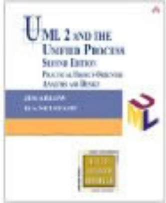
✧ **Seminars**

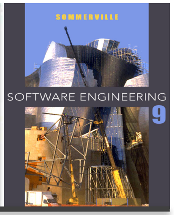
- 12 teaching weeks
- Team project on UML modeling, teams of 3 students (or less)
- Obligatory attendance (one absence ok) and weekly task delivery
- Simple test at the beginning of each seminar (starting in Week 03)
- Penalty for extra absence (-5 points) and late task delivery (-5 points)

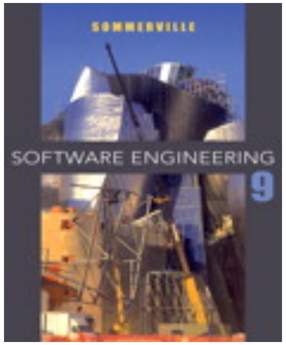
✧ **Evaluation**

- Seminar = project YES/NO, tests (20 points) and penalty recorded in IS notebook
- Exam = test (35 points) + on-site modelling (35 points)
- Grades: F<50, 50<=E<58, 58<=D<66, 66<=C<74, 74<=B<82, 82<=A

Literature





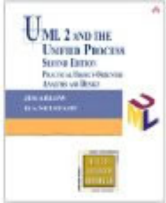


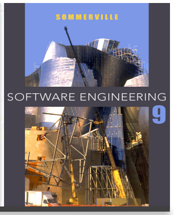
✧ Software Engineering, 9/E

- Author: Ian Sommerville
- Publisher: Addison-Wesley
- Copyright: 2011

✧ UML 2 and the Unified Process, 2/E

- Author: Jim Arlow and Ila Neustadt
- Publisher: Addison-Wesley
- Copyright: 2005

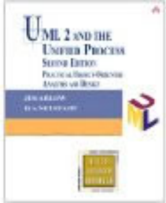


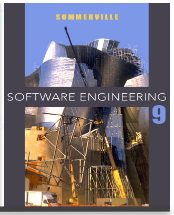


Software Development

Lecture 1/Part 2

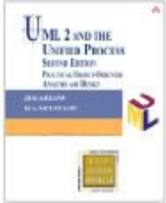
Outline

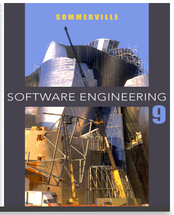




- Software engineering
- Software process activities
- Software process models

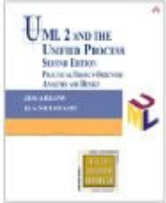
Software and system engineering

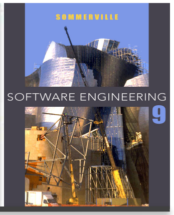




- The **economies** and **human lifes** of ALL developed nations are dependent on software.
- Software engineering is concerned with **theories**, **methods** and **tools** for professional software development.
- Software engineering is concerned with **cost-effective** development of **high-quality** software systems.
- **System engineering** is concerned with all aspects of computer-based systems development including hardware, software and process engineering.

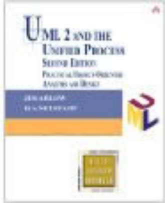
Software products

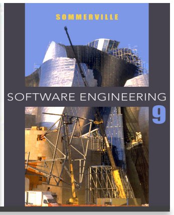




- ✧ Generic products
  - Stand-alone systems that are marketed and sold to **any customer** who wishes to buy them.
  - **Examples** – PC software such as graphics programs, project management tools; CAD software.
- ✧ Customized products
  - Software that is commissioned by a **specific customer** to meet their own needs.
  - **Examples** – embedded control systems, air traffic control software, traffic monitoring systems.

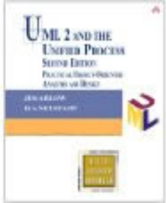
Application types

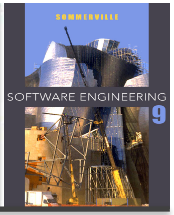




- Stand-alone desktop applications
- Interactive web-based applications
- Embedded control systems
- Batch processing systems
- Computer games
- Systems for modelling and simulation
- Data collection and monitoring systems
- Systems of systems

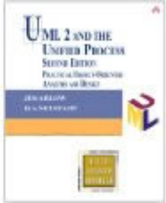
Software engineering fundamentals

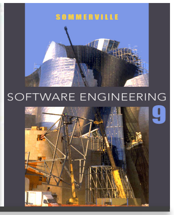




- Some **fundamental principles** apply to all types of software system, irrespective of the development techniques used:
  - Systems should be developed using a **managed and understood development process**. Of course, different processes are used for different types of software.
  - **Dependability and performance** are important for all types of system.
  - Understanding and managing the **software specification and requirements** (what the software should do) are important.
  - Where appropriate, you should **reuse software** that has already been developed rather than write new software.

The software process



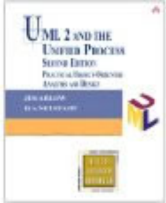


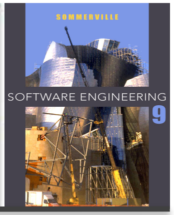
- ✧ A structured set of activities required to develop a software system.
- ✧ Many different software processes but all involve:
  - **Requirements specification**
  - **Analysis and design**
  - **Implementation**
  - **Validation and verification**
  - **Evolution**

Development

- ✧ Is the analysis and design always involved?

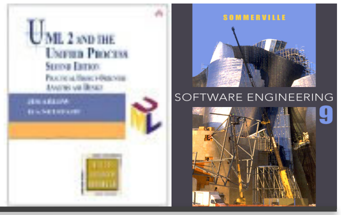
Software process activities

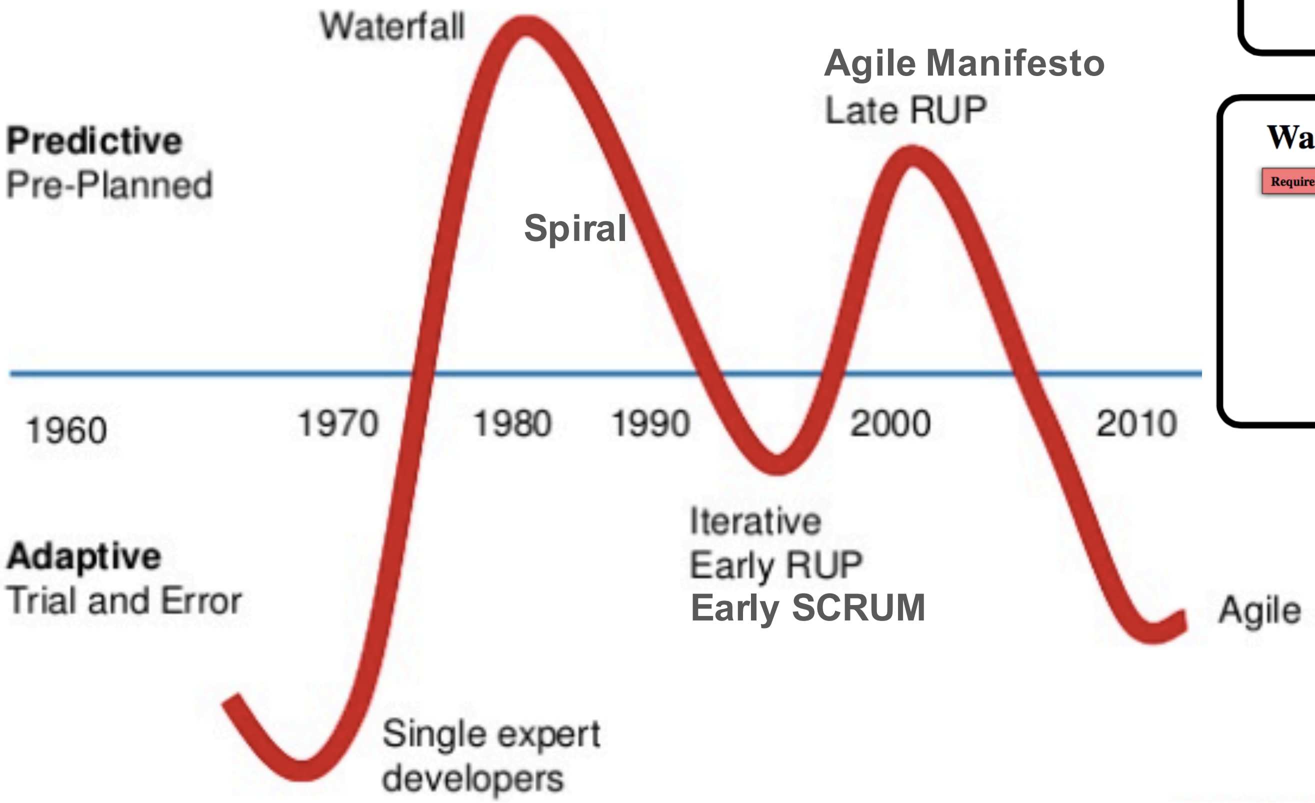


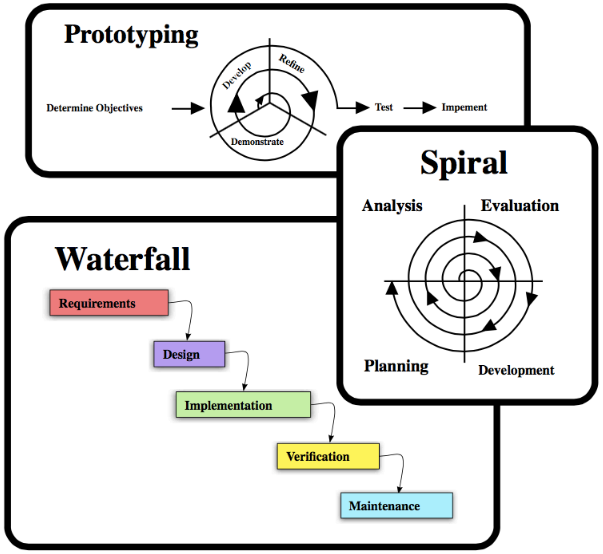


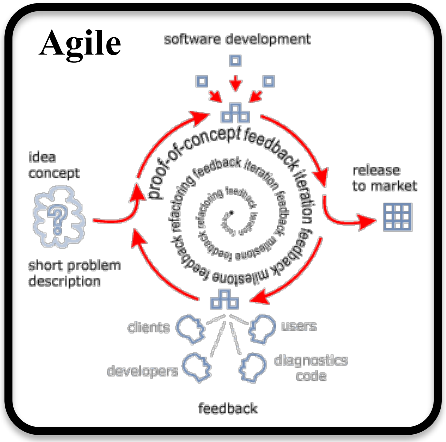
- **Requirements specification**, where customers and engineers define the software and the constraints on its operation.
- **Analysis and design**, where the requirements are refined into system design.
- **Implementation**, where the software is implemented.
- **Validation and verification**, where the software is checked to ensure that it is what the customer requires.
- **Evolution**, where the software is modified to reflect changing customer and market requirements.

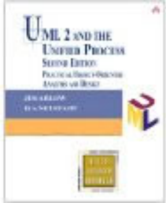
Software process models

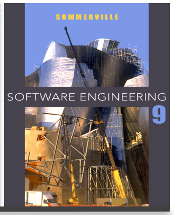








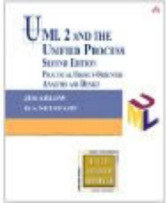


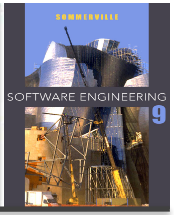


UML in Software Development

Lecture 1/Part 3

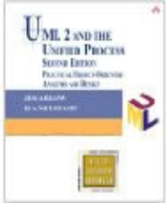
System modeling

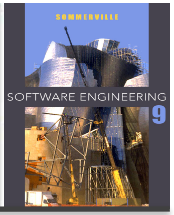




- System modeling is the process of developing **abstract models of a system**, with each model presenting a different view or **perspective** of that system.
- System modeling has now come to mean representing a system using some kind of graphical notation, which is now almost always based on the **Unified Modeling Language (UML).**
- System modelling helps the analyst to **understand the functionality** of the system and models are used to **communicate with colleagues and customers**.

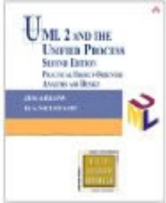
System perspectives

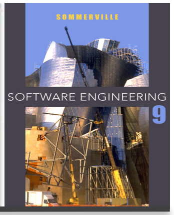




- An **external perspective**, where you model system boundary, the context and/or environment of the system.
- A **structural perspective**, where you model the organization of a system or the structure of the data that is processed by the system.
- An **interaction perspective**, where you model the interactions between a system and its environment, or between the components of a system.
- A **behavioral perspective**, where you model the dynamic behavior of the system and how it responds to events.

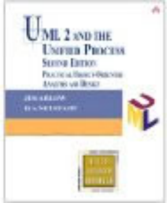
UML diagram types

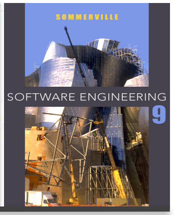




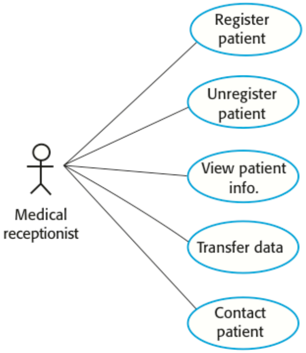
- External perspective
  - **Use case diagram**
- Structural perspective
  - **Class diagram**, Object diagram, Component diagram, Package diagram, Deployment diagram, Composite structure diagram
- Interaction perspective
  - **Sequence diagram**, Communication diagram, Interaction overview diagram, Timing diagram
- Behavioral perspective
  - **Activity diagram**, State diagram

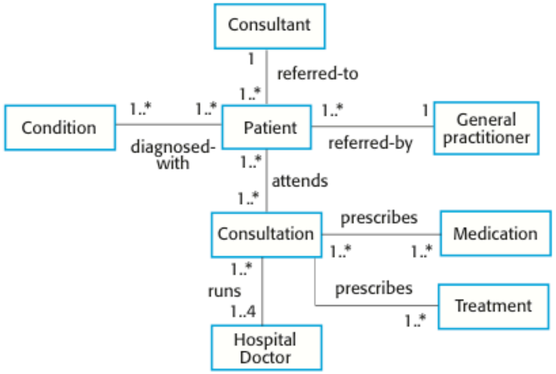
Popular UML diagrams

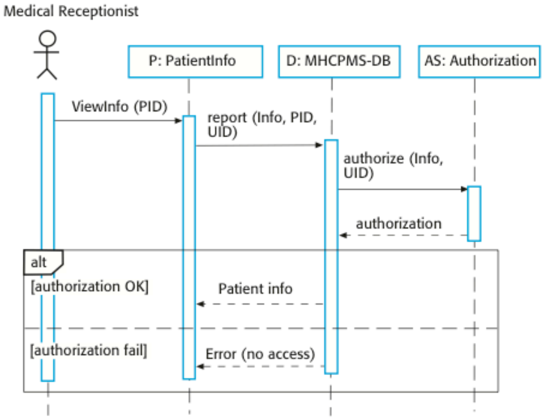


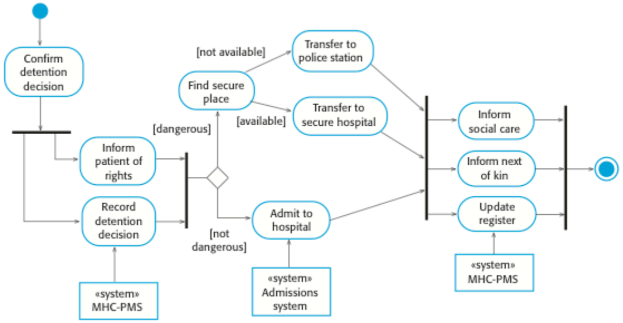


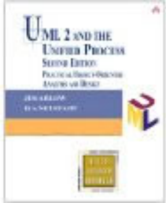
- ✧ **Use case diagrams**, which show the interactions between a system and its environment.
- ✧ **Class diagrams**, which show the object classes in the system and the associations between these classes.
- ✧ **Sequence diagrams**, which show interactions between actors and the system and between system components.
- ✧ **Activity diagrams**, which show the activities involved in a process or in data processing.

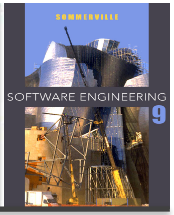








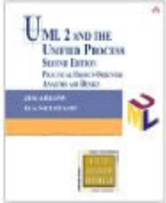


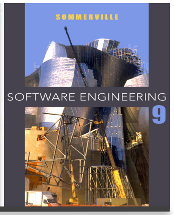


UML Use Case Diagram

Lecture 1/Part 4

Outline





✧ Use Case modelling

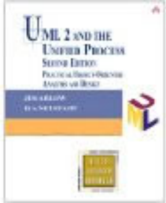
- System boundary – subject
- Use cases
- Actors

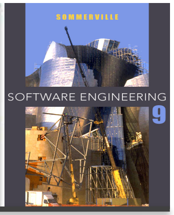
✧ Textual Use Case specification

✧ Advanced Use Case modelling

- Actor generalisation
- Use case generalisation
- «include»
- «extend»

The purpose of Use Case modelling





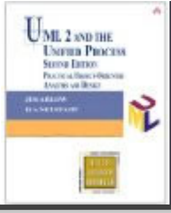
✧ Software specification

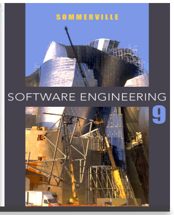
- The process of identifying and establishing system requirements
- Often referred to as **requirements specification** or **requirements engineering**

✧ Use case modelling proceeds as follows:

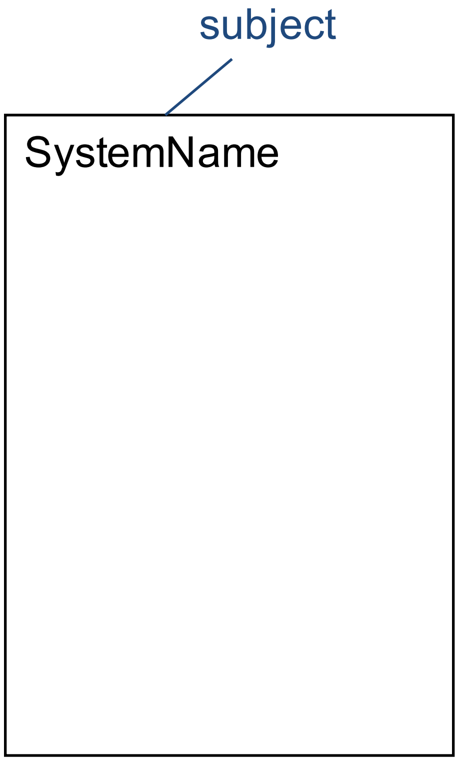
- Find the system boundary
- Find actors – who or what uses the system
- Find use cases – what functions the system should offer
- Specify use cases – with textual specification or UML Activity Diagrams

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

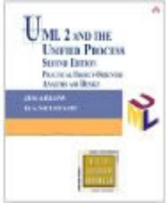


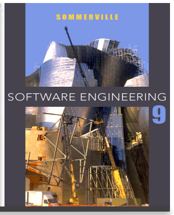
subject

SystemName

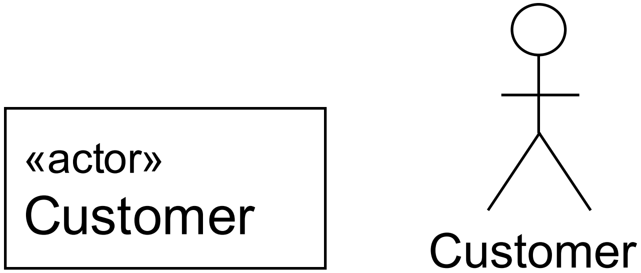
- ✧ Can there be a direct communication relationship between actors?

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

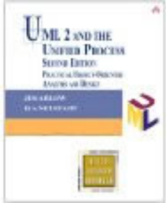


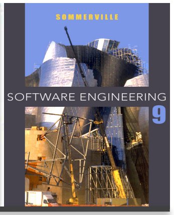
«actor»
Customer

Customer

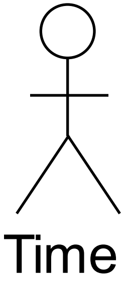
- Actors are **external** to the system
- An Actor specifies a **role** that some external entity adopts when interacting with the system
  - Can one actor represent two physical persons?
  - Can one physical person match to two actors?
  - Can there be two actors with the same name in the model?

Identifying Actors



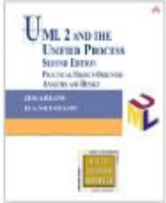


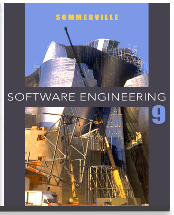
- ✧ When identifying actors ask:
  - Who or what uses the system?
  - What roles do they play in the interaction?
  - Who installs the system?
  - Who starts and shuts down the system?
  - Who maintains the system?
  - What other systems use this system?
  - Who gets and provides information to the system?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



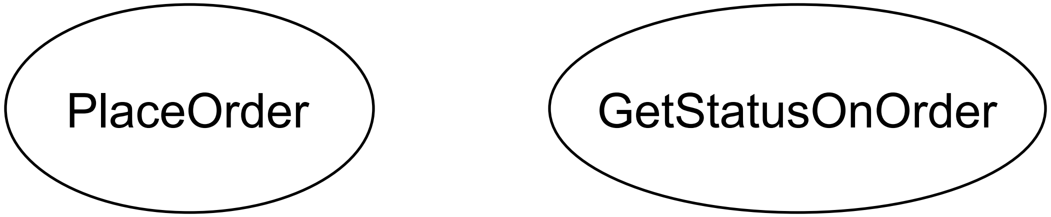
- ✧ What if the actor is not a human? What can it be?

What are use cases?

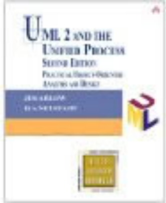


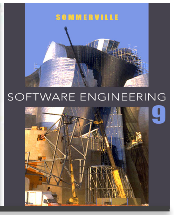


- A use case is something an actor needs the system to do. It is a “case of use” of the system by a specific actor.
- Use cases are always started by an actor
  - The **primary actor** triggers the use case
  - Zero or more **secondary actors** interact with the use case in some way
  - Does the UC diagram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



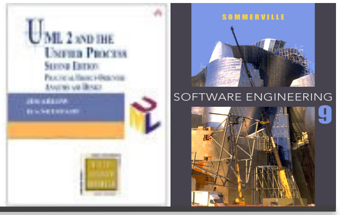
Identifying use cases

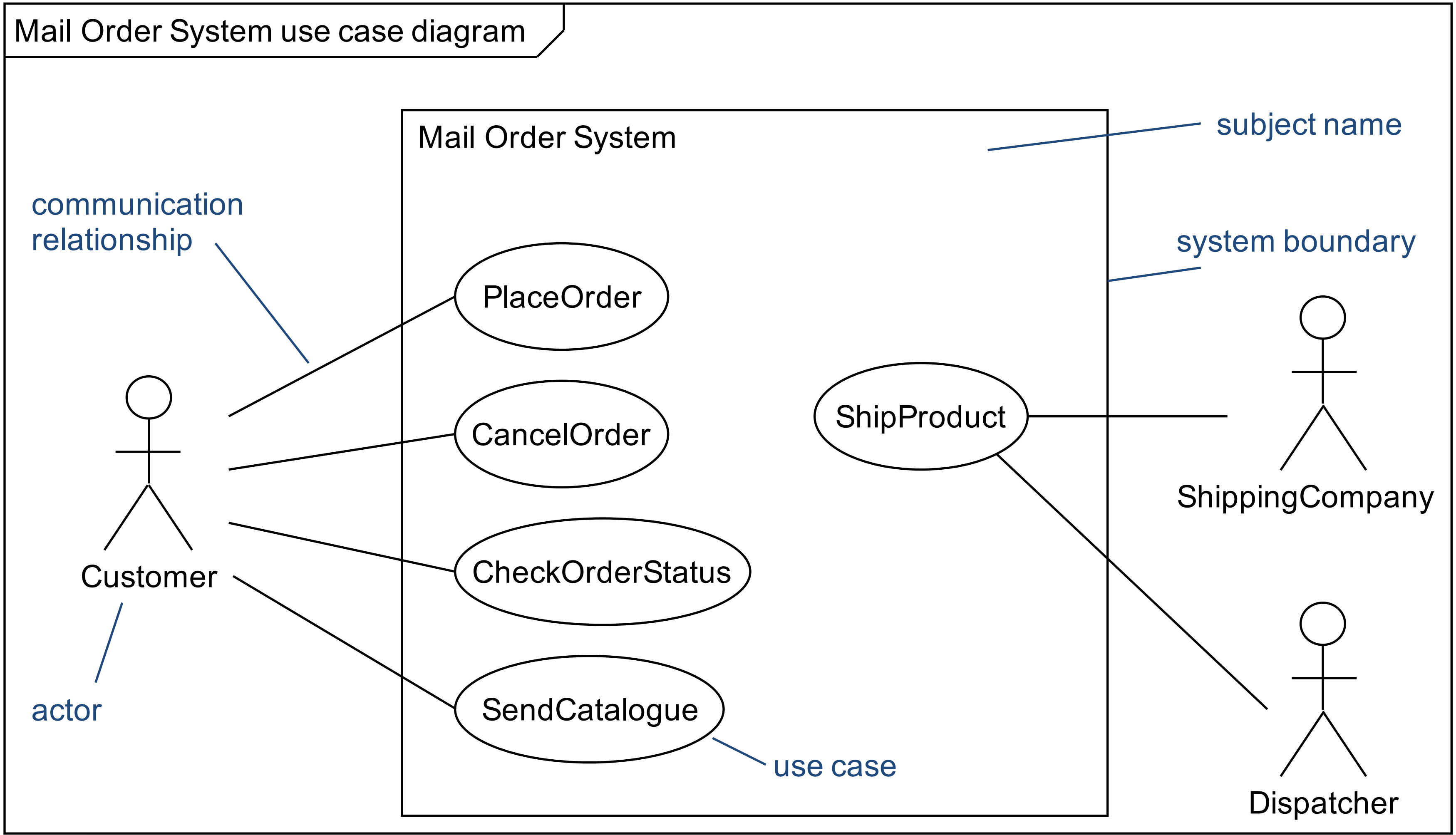




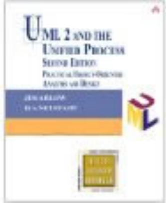
- Start with the list of actors that interact with the system
- When identifying use cases ask:
  - What functions will a specific actor want from the system?
  - Does the system store and retrieve information? If so, which actors trigger this behaviour?
  - What happens when the system changes state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

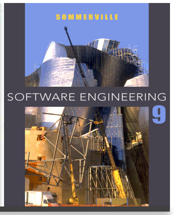
The use case diagram





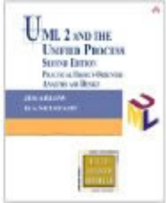
Textual use case specification

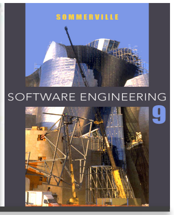




use case name	Use case: PaySalesTax
use case identifier	ID: 1
brief description	Brief description: Pay Sales Tax to the Tax Authority at the end of the business quarter.
the actors involved in the use case	Primary actors: Time
	Secondary actors: TaxAuthority
the system state before the use case can begin	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

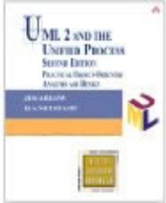


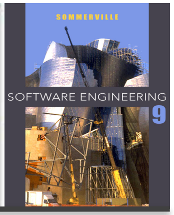


- Use cases describe something that happens
- They are named using **verbs** or **verb phrases**
- Naming standard [1]: use cases are named using UpperCamelCase e.g. PaySalesTax

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

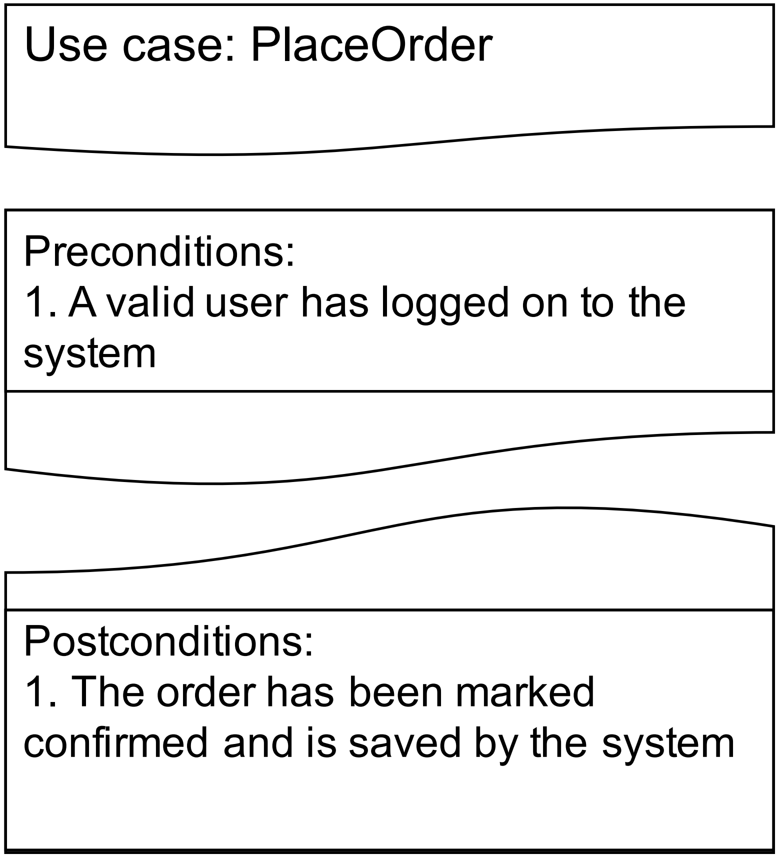
Pre and postconditions





- 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?

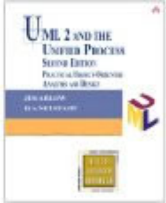


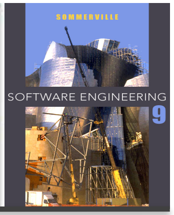
Use case: PlaceOrder

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

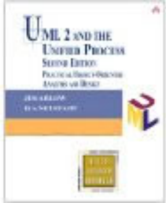


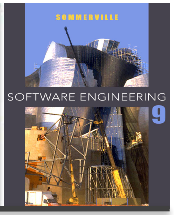


<number> The <something> <some action>

- 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

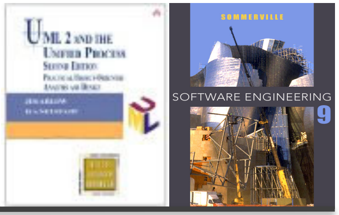




- ✧ Use the keyword IF to indicate alternatives within the flow of events
  - There must be a Boolean expression immediately after IF
- ✧ Use indentation and numbering to indicate the conditional part of the flow
- ✧ Use ELSE to indicate what happens if the condition is false

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.

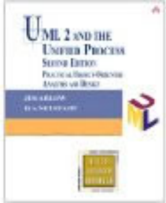
Repetition within a flow: FOR

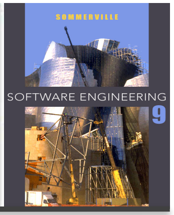


- We can use the keyword FOR to indicate the start of a repetition within the flow of events
- The iteration expression immediately after the FOR statement indicates the number of repetitions of the indented text beneath the FOR statement.

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

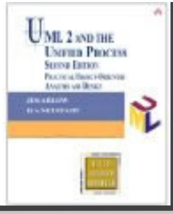


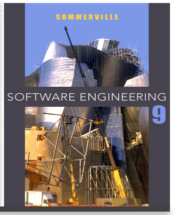


- ✧ We can use the keyword WHILE to indicate that something repeats while some Boolean condition is true

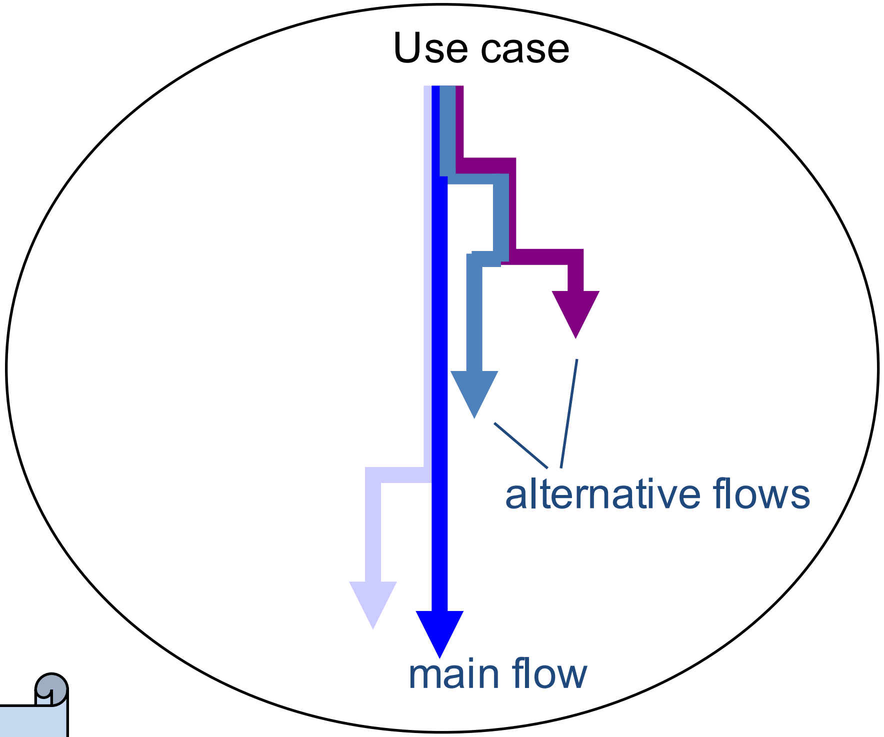
Use case: ShowCompanyDetails
ID: 4
Brief description: The system displays the company details to the Customer.
Primary actors: Customer
Secondary actors: None
Preconditions: None.
Main flow: 1. The use case starts when the Customer selects "show company details". 2. The system displays a web page showing the company details. 3. WHILE the Customer is browsing the company details 3.1. The system plays some background music. 3.2. 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**

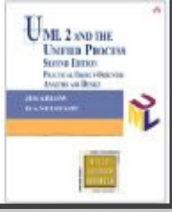

Use case

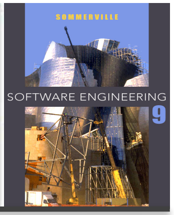
alternative flows

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:

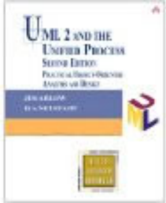
1. The use case begins when the Customer selects "create new customer account".
2. WHILE the Customer details are invalid
   - 2.1. The system asks the Customer to enter his or her details comprising email address, password and password again for confirmation.
   - 2.2 The system validates the Customer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

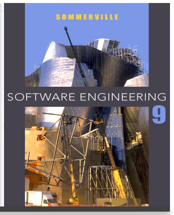
Postconditions:
1. A new account has been created for the Customer.

Alternative flows:
InvalidEmailAddress
InvalidPassword
Cancel

Alternative flows

Advanced Use Case modelling

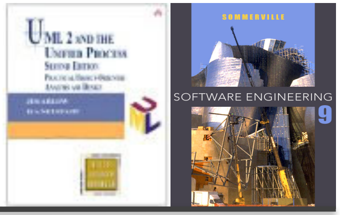




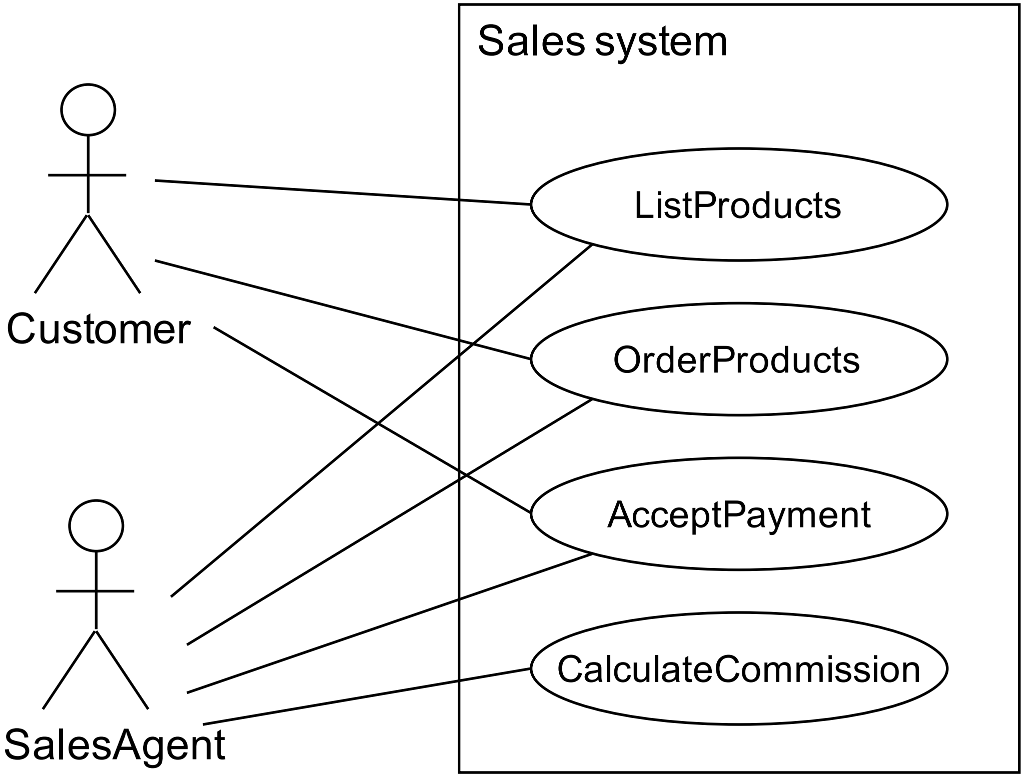
✧ We have studied basic use case analysis, but there are **relationships** that we have still to explore:

- Actor generalisation
- Use case generalisation
- «include» – between use cases
- «extend» – between use cases

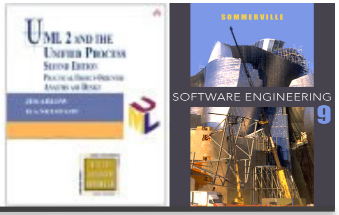
Actor generalization – example



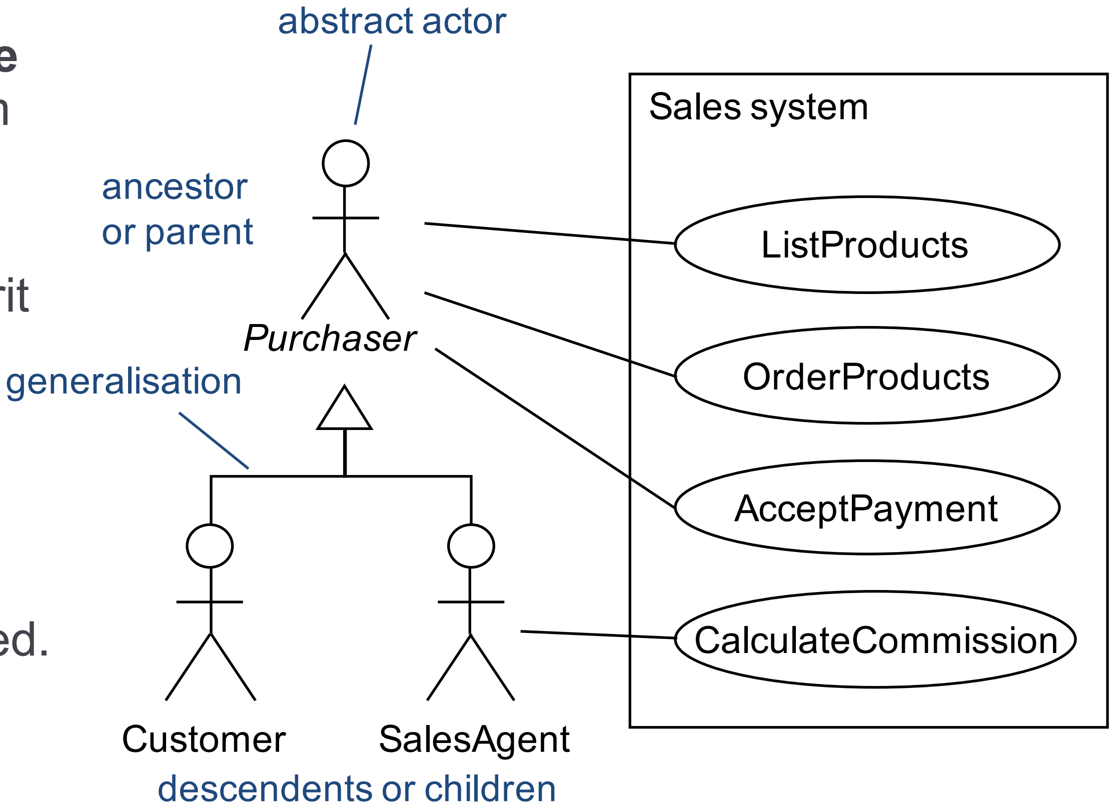
- The Customer and the Sales Agent actors are **very similar**
- They both interact with List products, Order products, Accept payment
- They both can play the **purchaser role**.



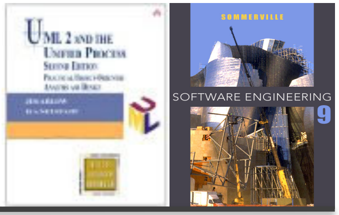
Actor generalisation



- 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**
- Is it always a good idea to generalize two actors sharing some use cases?

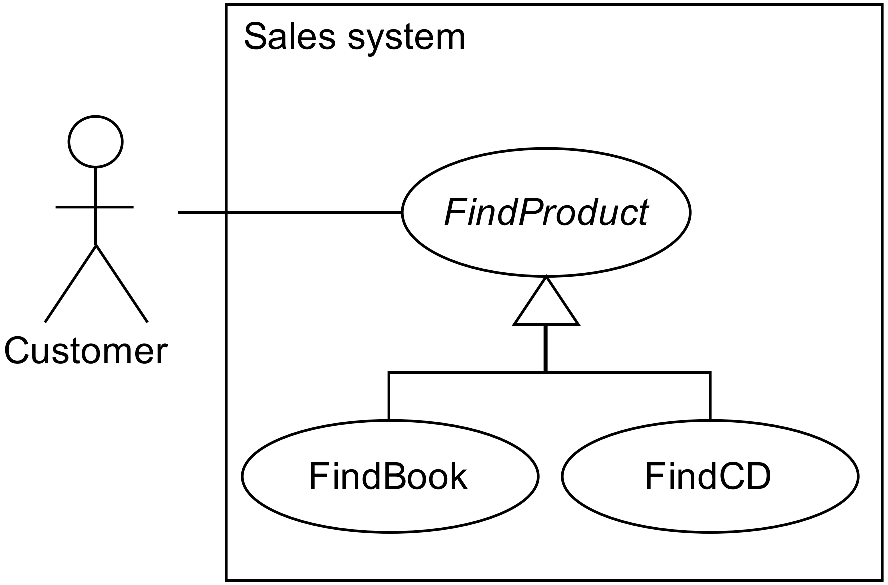


Use actor generalization when it simplifies the model

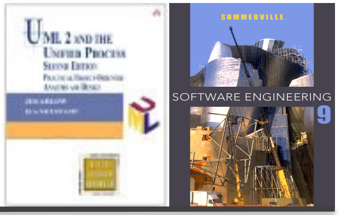


Use case generalisation

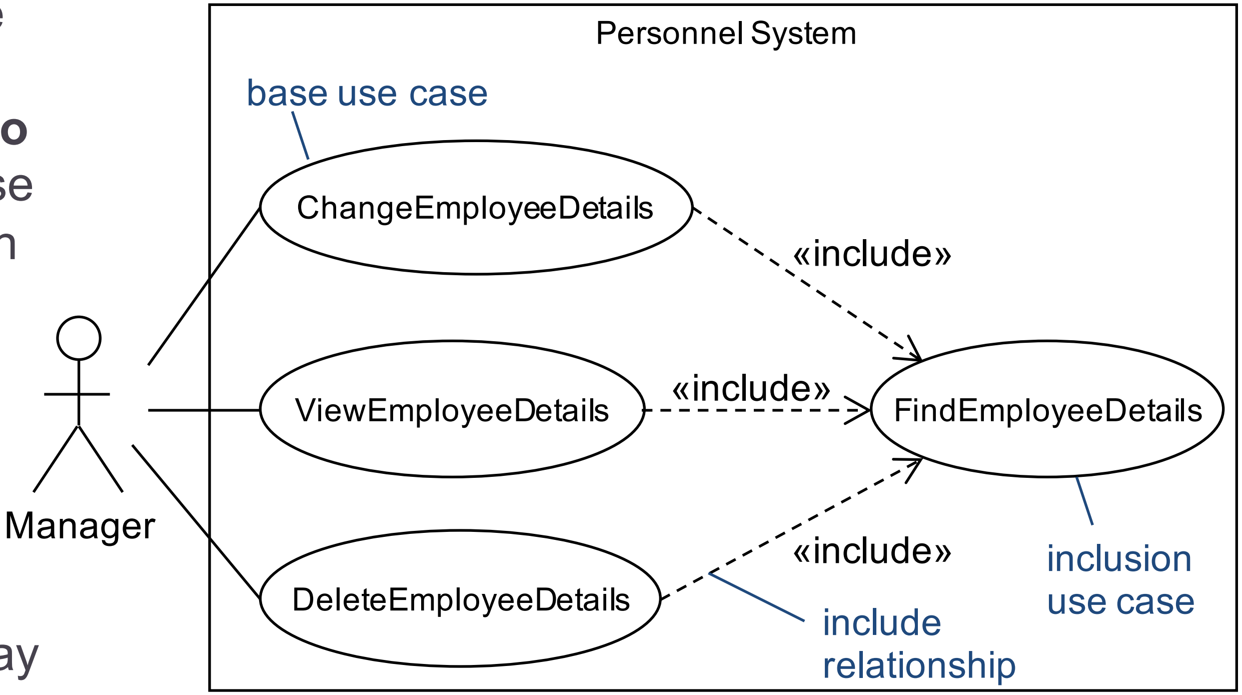
- The ancestor use case must be a more general case of one or more descendant use cases
- Child use cases are more specific forms of their parent
- They can inherit, add and override features of their parent

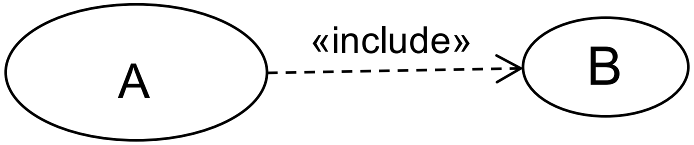


«include»

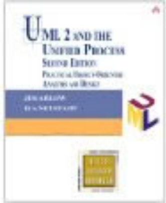


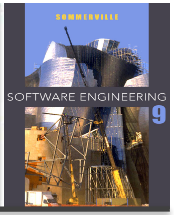
- When use cases share common behaviour we can **factor this out into** a separate inclusion use case and «include» it in base use cases
- Base use cases are **not complete** without the included use cases
- Inclusion use cases may be complete use cases, or they may just specify a **fragment of behaviour**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





«include» example

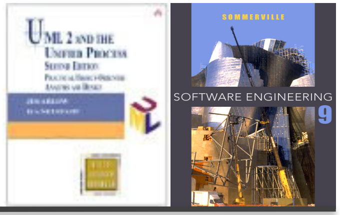




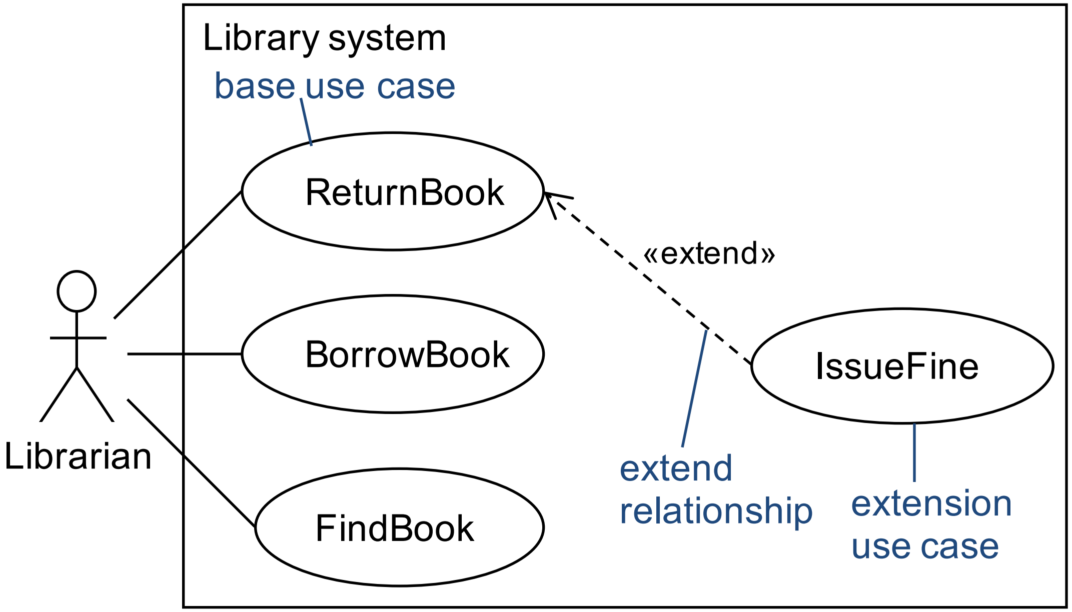
Use case: ChangeEmployeeDetails
ID: 1
Brief description: The Manager changes the employee details.
Primary actors: Manager
Seconday actors: None
Preconditions: 1. The Manager is logged on to the system.
Main flow: 1. include(FindEmployeeDetails). 2. The system displays the employee details. 3. The Manager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

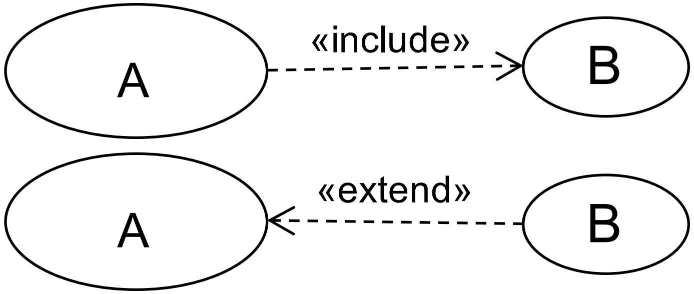
Use case: FindEmployeeDetails
ID: 4
Brief description: The Manager finds the employee details.
Primary actors: Manager
Seconday actors: None
Preconditions: 1. The Manager is logged on to the system.
Main flow: 1. The Manager enters the employee's ID. 2. The system finds the employee details.
Postconditions: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

«extend»

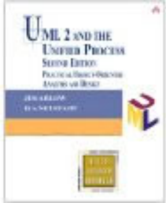


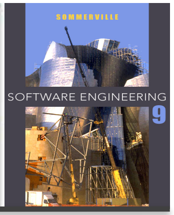
- The extension use case **inserts behaviour** into the base use case.
- The base use case provides **extension points,** but **does not know** about the extensions.
- The base use case is **complete** already without the extensions.
- There may be multiple extension points and multiple extending use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

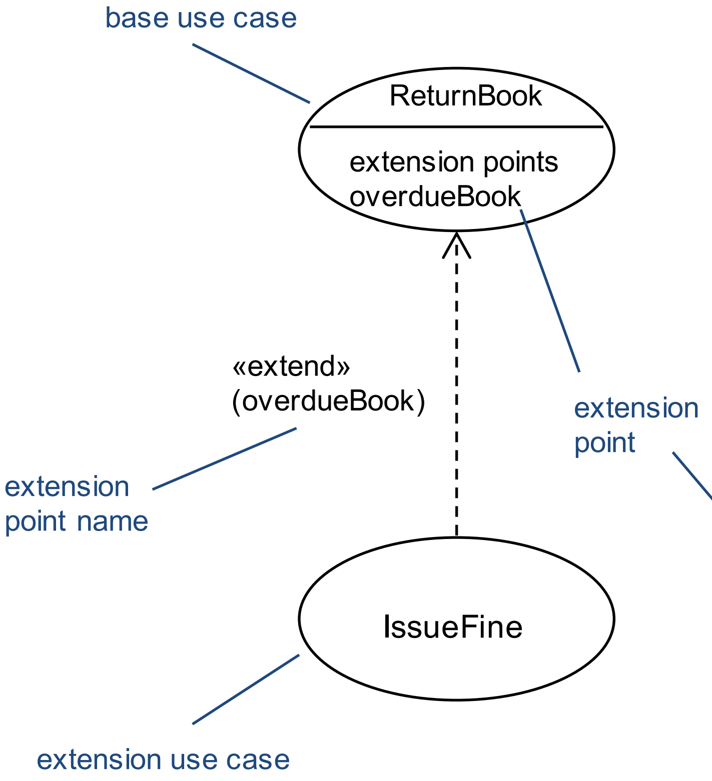




<<extend>> example



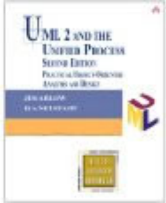


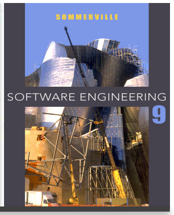


Use case: ReturnBook
ID: 9
Brief description: The Librarian returns a borrowed book.
Primary actors: Librarian
Secondary actors: None.
Preconditions: 1. The Librarian is logged on to the system.
Main flow: 1. The Librarian enters the borrower's ID number. 2. The system displays the borrower's details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

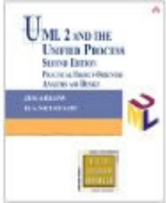
- One use case may cover many individual functional requirements
- One functional requirement may be realised by many use cases

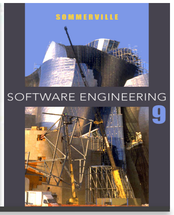
✧ **Requirements Traceability Matrix** can help us to trace if all requirements are covered by our use case model

		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 have highest value 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!