Review

- What is Software Engineering about?
- What is Software Development Life Cycle?
- Can you name any Software Development Life Cycles?
- What are the specific characteristics of Waterfall SDLC?

- Software Engineering is the discipline of designing, implementing, and managing software projects with the intention of creating high-quality software in a cost-effective way.
- The Software Development Life Cycle (SDLC) is a framework that describes the stages involved in an information system development project from an initial feasibility study through maintenance of the completed application.
- Examples of Software Development Life Cycles include Waterfall, Agile, Spiral, Iterative, V-Model, and Big Bang.
- The Waterfall SDLC is characterized by a linear and sequential design process, where progress is seen as flowing steadily downwards (like a waterfall) through phases such as conception, initiation, analysis, design, construction, testing, production/implementation, and maintenance.

- • 软件工程是设计、实施和管理软件项目的学科，旨在以经济有效的方式创建高质量的软件。
- • 软件开发生命周期(SDLC) 是一个框架，描述了信息系统开发项目所涉及的各个阶段，从最初的可行性研究到已完成的应用程序的维护。
- • 软件开发生命周期的示例包括瀑布式、敏捷式、螺旋式、迭代式、V 模型和大爆炸式。
- • 瀑布式 SDLC 的特点是线性和顺序的设计过程，其中的进展被视为稳步向下流动（像瀑布一样），经历概念、启动、分析、设计、构建、测试、生产/实施和维护等阶段。

Review

- What does a class diagram represent?
- Define association, aggregation, and generalization.
- How do you find associations?
- What information does multiplicity provide?

- What is the main purpose of a SD?
- What are the main concepts in a SD?
- What are the communication diagrams?
- What is the difference between SD and communication diagrams?

Class diagrams represent the static structure of a system, showing classes, their attributes, methods, and the relationships among objects.

Association in UML indicates a relationship between two or more objects that shows how objects know about each other. Aggregation is a special form of association that represents a "whole-part" relationship. Generalization is another relationship between a general thing and a more specific kind of that thing, representing inheritance in object-oriented programming.

Associations are typically found by analyzing the problem domain and determining which objects need to collaborate to perform a function.

Multiplicity in UML specifies how many instances of a class can be associated with one instance of another class.

The main purpose of a sequence diagram is to show how objects interact with each other in a particular sequence of time.

The main concepts in a sequence diagram include objects, lifelines, messages, and activations.

Communication diagrams are another type of interaction diagram in UML, showing the interactions between objects in terms of sequenced messages.

The difference between sequence diagrams and communication diagrams is that sequence diagrams focus on the time sequence of messages, whereas communication diagrams focus on the structural organization of the objects that send and receive messages.

• 类图表示系统的静态结构，显示类、它们的属性、方法以及对象之间的关系。

• UML 中的关联表示两个或多个对象之间的关系，显示对象如何相互了解。 聚合是一种特殊的关联形式，代表“整体-部分”关系。 泛化是一般事物与更具体的事物之间的另一种关系，代表面向对象编程中的继承。

• 通常通过分析问题域并确定哪些对象需要协作来执行功能来发现关联。

• UML 中的多重性指定一个类的多少个实例可以与另一类的一个实例相关联。

• 序列图的主要目的是显示对象如何在特定的时间序列中相互交互。

• 序列图中的主要概念包括对象、生命线、消息和激活。

• 通信图是 UML 中的另一种交互图，以排序消息的形式显示对象之间的交互。

• 序列图和通信图之间的区别在于，序列图侧重于消息的时间顺序，而通信图侧重于发送和接收消息的对象的结构组织。

Review

- What are models for?
- What is system behavior?
- What is an actor?
- A use case?
- What is a role?
- How do we know if our requirements are of good quality?

- Models are tools for understanding and constructing the structure and behavior of a system. They are abstractions that help stakeholders agree on design and can guide system implementation.
- System behavior is the way a system acts or reacts to different situations or stimuli.
- An actor in UML is an entity that interacts with the system (usually a user role or any external system).
- A use case is a description of a system's behavior as it responds to a request that originates from outside of that system. In other words, it's a scenario that shows a sequence of events and interactions between a user and a system.
- A role is a set of associated behaviors or a set of expectations about the behavior of an entity within a certain context.
- The quality of requirements can be assessed using certain criteria, such as completeness, clarity, consistency, testability, and traceability.

• 模型是用于理解和构建系统结构和行为的工具。 它们是帮助利益相关者就设计达成一致并可以指导系统实施的抽象。
• 系统行为是系统对不同情况或刺激的行为或反应方式。

• UML 中的参与者是与系统交互的实体（通常是用户角色或任何外部系统）。
• 用例是对系统响应来自系统外部的请求时的行为的描述。 换句话说，它是一个显示用户和系统之间的一系列事件和交互的场景。
• 角色是一组相关的行为或一组关于某个实体在特定上下文中的行为的期望。
• 可以使用某些标准来评估需求的质量，例如完整性、清晰度、一致性、可测试性和可追溯性。

Week 1

Software Engineering

- Software Engineering
  - Engineering: **cost-effective solutions** to practical problems by applying **scientific knowledge** to building **things** for **people**
    - Cost-effective solutions: process and project management, contracts…
    - Scientific knowledge: modelling, proofs, testing, simulation, patterns...
    - Things=software
    - People: customers and end users

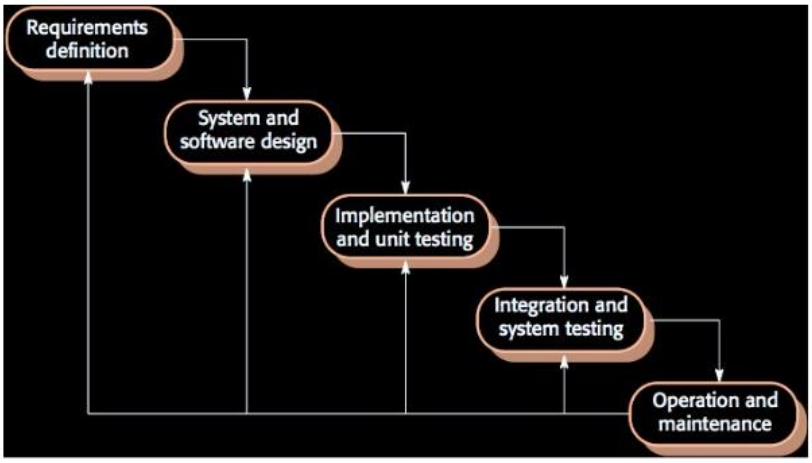
Software Development Tasks

- Requirements Analysis
- Planning
- Design: high level and detailed
- Development
- Testing
- Deployment
- Operation and Maintenance

These to-does are combined in various sequences, making up different Software Development Life Cycle Processes

Software Development Life Cycle (SDLC)

Waterfall Software Development Life Cycle



Design: How to Structure Software?

- What objects, databases, servers, services etc. should we create?
- How are these elements structured into a system?

Why to design?

- Helps to decide where to place requirements
- How the parts of the system will be interacting
- Divide up work between team members

Waterfall SDLC: Advantages and Disadvantages

- Simple to use and understand
- Every phase has a defined result and process review
- Development stages go one by one
- Perfect for projects where requirements are clear and agreed upon
- Easy to determine the key points in the development cycle
- Easy to classify and prioritize tasks

- The software is ready only after the last stage is over
- High risks and uncertainty
- Misses complexity due to interdependence of decisions
- Not suited for long-term projects where requirements will change
- The progress of the stage is hard to measure while it is still in the development
- Integration is done at the very end, which does not give the option of identifying the problem in advance

Verification and Validation

- Verification: check if the software/service complies with a requirements, constraints, and regulations. ("Are you building it right?")
  - Demonstrates that system meets specifications
- Validation: does this meet the needs of the customers/ stakeholders? ("Are you building the right thing?")
  - Demonstrate that system meets user needs
- Can pass verification but fails validation: if built as per the specifications yet the specifications do not address the user's needs.
- Involves checking, reviewing, evaluating & testing

Week 2

What is Agile Software Development?

- Agile is a way of thinking about software development
- In winter 2001, 17 software developers met at a ski resort in Utah and drafted a manifesto outlining an alternative way to develop software to the documentation-driven software development processes of the time
- The manifesto is succinct and puts forward four key values for software development

Agile Manifesto wants people to think about alternative ways of doing aspects of software development

Problems with Agile

- Hard to draw up legally binding contracts - a full specification is never written in advance

● Good for green-field development when you have a clean slate and are not constrained by previous work. However, it's not so effective for brownfield development which involves improving and maintaining legacy systems.
● Works well for small co-located teams, but what about large distributed development?
● Relies on the knowledge of developers in the team but what if they aren't around (holidays, illness, turnover)?

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

Twelve agile principles

Satisfy clients' needs	Satisfy coders' needs
The highest priority is satisfying the client (by delivering working software early and continuously)	Work at a steady, sustainable pace (no heroic efforts)
Embrace change (even late in the development cycle)	Rely on self-organising teams
Collaborate every day with the client	Teams reflect regularly on their performance
Use face to face communication	Progress is measured by the amount of working code produced
Deliver working software frequently	Continuous attention to technical excellence
	Minimise the amount of unnecessary work
	Build teams around motivated individuals

There are various approaches that adhere to Agile values and principles. Different companies choose different approaches.

Popular methods include:

- **Extreme Programming (XP) (the two co-creators were**

signatories of the manifesto)

Extreme Programming Ethos

- Simple design: use the simplest way to implement features
- Sustainable pace: effort is constant and manageable
- Coding standards: teams follow an agreed style and format
- Collective ownership: everyone owns all the code
- Whole team approach: everyone is included in everything

Extreme Programming Practices

- ***Pair programming***: two heads are better than one

Pair programming in more detail

Code is written by two programmers on one machine:

- The helm uses the keyboard and mouse and does the coding
- The tactician thinks about implications and potential problems
- Communication is essential for pair programming to work ● Pair programming facilitates project communication
- The pair doesn't "own" that code - anyone can change it
- Pairings can (and should) evolve at any time
- All code is reviewed as it is written
- The tactician is ideally positioned to recommend refactoring

- **Small releases**: deliver frequently and get feedback from the client
- **Continuous integration**: ensure the system is operational
- **Refactor**: restructure the system when things get messy

○ **Test-driven development (creator was a signatory)**

Test driven: ensure the code runs correctly

Test-driven development in more detail

- Tests are written before any code, and they drive all development
- A programmer's job is to write code to pass the tests
- If there's no test for a feature, then it is not implemented
- Tests are the requirements of the system

The benefits of test-driven development

● Code coverage We can be sure that all code written has at least one test because if there were no test, the code wouldn't exist

● Simplified debugging If a test fails, then we know it must have been caused by the last change

● System documentation Tests themselves are one form of documentation because they describe what the code should be doing

○ **Kanban**

● It's basically a flexible "to do" list tool

● Issues progress through various states from "To do" to "Done"

○ **Scrum (the two co-creators were signatories)**

Scrum is a project management approach.

Some key concepts are:

● The Scrum – a stand-up daily meeting of the entire team

● Scrum Master - team Leader

● Sprint - a short, rapid development iteration

● Product Backlog – To do list of jobs that need doing

● Product Owner – the client (or their representative)

Week 3

Functional requirements specify user interactions with the system, they say what the system is supposed to do:

■ Statements of services the system should provide

■ How the system should react to particular inputs

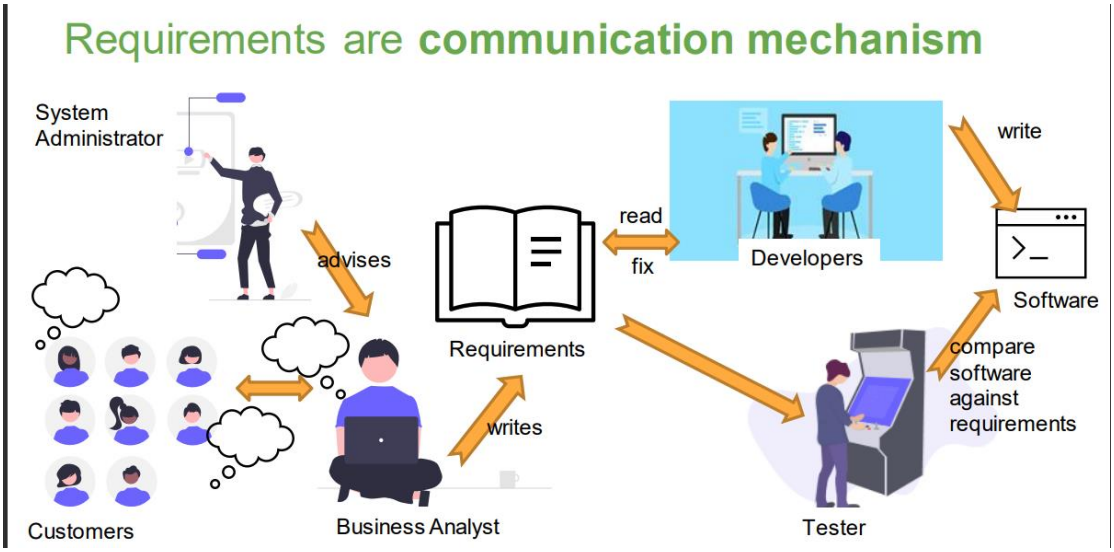
■ How the system should behave in particular situations

■ May also state what the system should NOT do

Non-functional requirements specify other system properties, they say how the functional requirements are realized:

■ Constraints ON the services or functions offered by system

■ Often apply to whole system, not just individual features



Requirements are acceptance criteria

To be able to fairly assess whether the team have produced something that matches what you asked for, the thing that you asked for must be:

○ Unambiguous / Precise

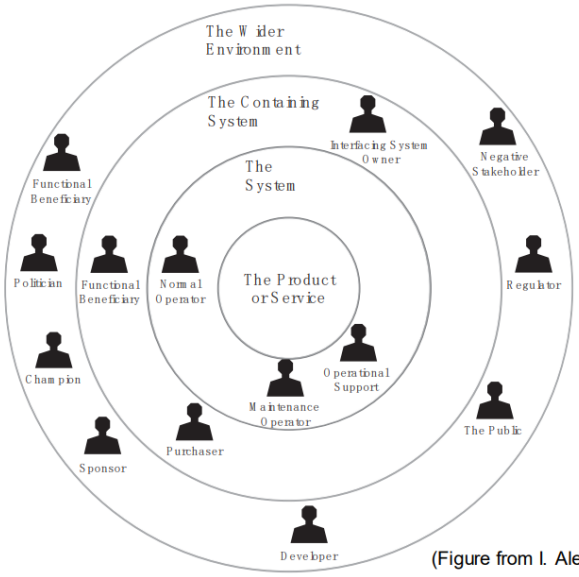
○ Complete

○ Understandable / Clear

Analyzing Requirements

1. Identify stakeholders involved with the system

The Onion model



(Figure from I. Alexander's book)

Identification

• Clients

• Documentation, e.g., organization chart

• Templates (e.g., onion model)

• Similar projects

• Analyzing the context of the project

Keeping in mind:

• Surrogate stakeholders (e.g., legal, unavailable at present, mass product users)

• Negative stakeholders

2. **Identify top-level user needs (e.g., as NFRs or "user stories")**

Identify "User Stories"

A popular way to record user needs…

As a < type of user >, I want to < some goal > so that < some reason >.

As a student, I want to be able to register for a module, so that I can learn about topics of interest to me.

As a customer, I want to be able to pay for a university course, so that I can attend the lectures to get a degree.

As a customer, I want to have my data kept securely, so that my privacy is protected.

3. **Break down stories into individual steps / Refine requirements**

Use-case specification

● A requirements document that contains the text of a use case, including:

○ A description of the flow of events describing the interaction between actors and the system

○ Other information, such as:

■ Preconditions

■ Postconditions

■ Special requirements

- Key scenarios
- Subflows

Checkpoints for use cases

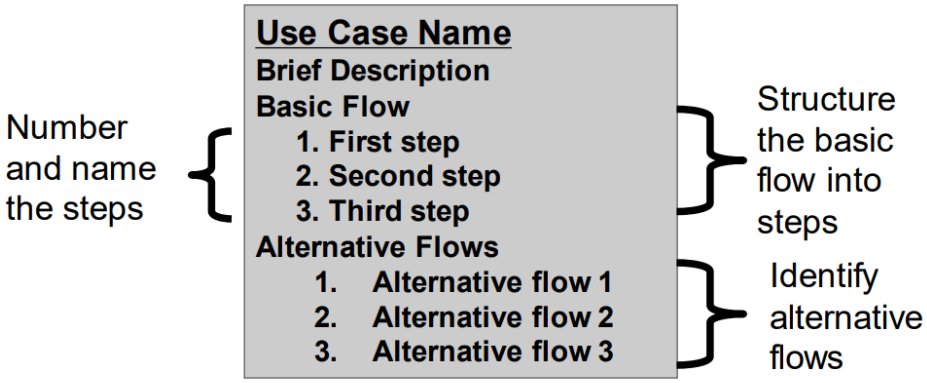
✓ Each use case is independent of the others

✓ No use cases have very similar behaviors or flows of events

✓ No part of the flow of events has already been modeled as another use case

Outline each use case

- An outline captures use case steps in short sentences, organized sequentially



Flows of events (basic and alternative)

- A flow is a sequence of steps
- One basic flow
  - Successful scenario from start to finish
- Many alternative flows
  - Regular variants

- Odd cases
- Exceptional (error) flows

What is a use-case scenario?

▪ An instance of a use case

▪ An ordered set of actions from the start of a use case to one of its end points

4. Specify atomic requirements (e.g., for each step-in user stories)

Quality Attributes of Requirements

● Consistency: Are there conflicts between requirements?

● Completeness: Have all features been included?

● Comprehendability: Can the requirement be understood?

● Traceability: Is the origin of requirement clearly recorded?

● Realism: Can the requirements be implemented given available resources and technology?

● Verifiability: Can requirements be “ticked off”?

- 一致性：需求之间是否存在冲突？
- 完整性：是否包含所有功能？
- 可理解性：需求能否被理解？
- 可追溯性：需求的来源是否有清晰的记录？
- 现实性：在现有资源和技术的情况下能否实现这些要求？
- 可验证性：需求是否可以“勾选”？

Verifiability of Requirements

We should ensure that requirements are ***verifiable.***

No point specifying something that can't be "ticked off."

For example, the following is an ***unverifiable*** "objective": The system must be easy to use by waiting staff and should be organized so that user errors are minimized.

In comparison, the following is a ***testable*** requirement: Waiting staff shall be able to use all 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.

我们应该确保需求是可验证的。
没有必要指定无法“勾选”的内容。
例如，以下是一个无法验证的“目标”：系统必须易于等候人员使用，并且应进行组织以尽量减少用户错误。
相比之下，以下是一个可测试的要求： 服务员经过四个小时的培训后应能够使用所有系统功能。 经过这次培训后，有经验的用户在系统使用过程中平均犯错的次数不得超过每小时两次。

Requirements Elicitation Techniques

● Interviews ● Observations ● Surveys ● Current documentation ● Similar products and solutions ● Co-design ● Prototyping

What Is System Behavior?

- **System behavior is how a system acts and reacts.**
  - It comprises the actions and activities of a system.
- **System behavior is captured in use cases.**

○ Use cases describe the interactions between the system and (parts of) its environment

What is a use-case model?

● Describes the functional requirements of a system in terms of use cases

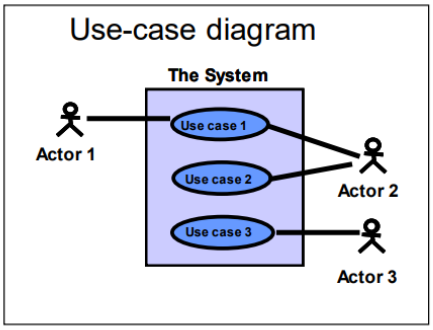
● Links stakeholder needs to software requirements

● Serves as a planning tool

● Consists of actors and use cases

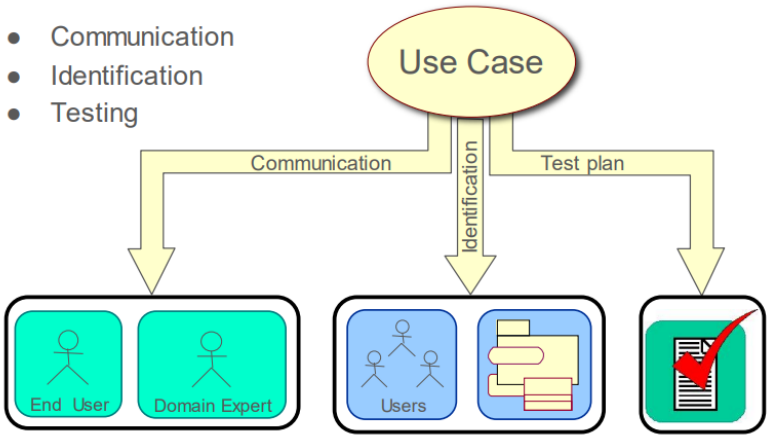
Use-case diagram

- Shows a set of use cases and actors and their relationships
- Defines clear boundaries of a system
- Identifies who or what interacts with the system
- Summarizes the behavior of the system



What Are the Benefits of a Use-Case Model?

- Communication
- Identification
- Testing



Major Concepts in Use-Case Modeling

● An actor represents anything that interacts with the system.

● A use case describes a sequence of events, performed by the system, that yields an observable result of value to a particular actor.

What Is an Actor?

• Actors represent roles a user of the system can play.

• They can represent a human, a machine, or another system.

• They can actively interchange information with the system.

• They can be a giver of information.

• They can be a passive recipient of information.

• Actors are not part of the system.

  • Actors are EXTERNAL.

What Is a Use Case? Use Case

■ Defines a set of use-case instances.

Where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor.

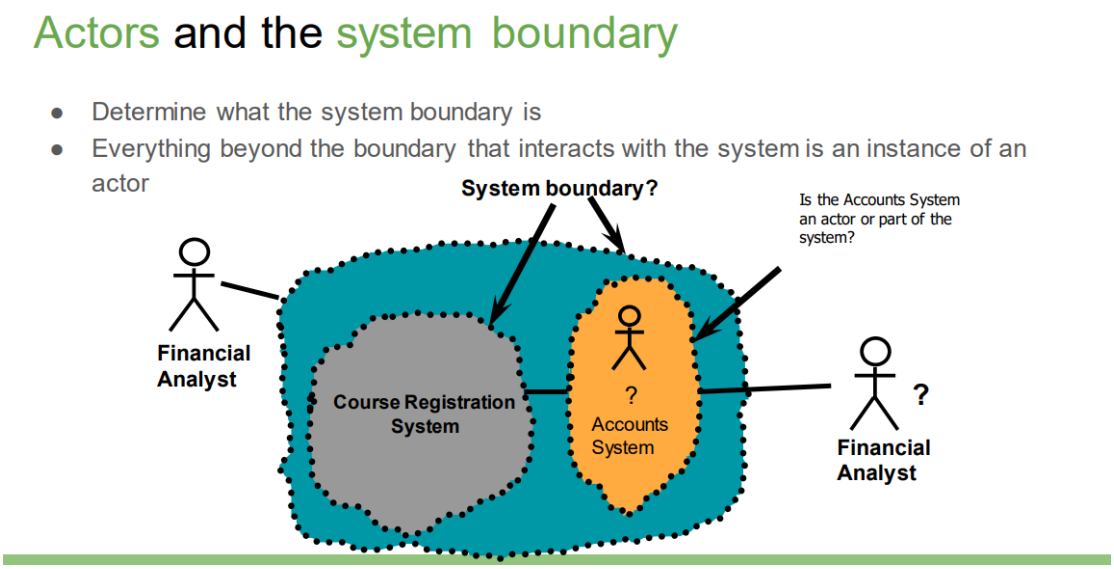
其中每个实例都是系统执行的一系列操作，这些操作为特定参与者产生有价值的可观察结果。

■ A use case models a dialogue between one or more actors and the system

■ A use case describes the actions the system takes to deliver

something of value to the actor

■A use case is initiated by an actor to invoke a certain functionality in the system



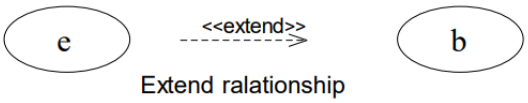
Modelling with Use Cases

● Generalization between Use Cases means that the child is a more specific than the parent; the child inherits all attributes and associations of the parent, but may add new features

用例之间的泛化意味着子用例比父用例更具体； 子级继承父级的所有属性和关联，但可以添加新功能

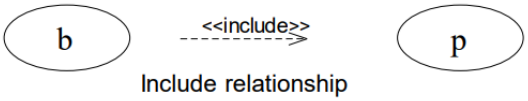
Use Cases

Specifies how the behaviour of the extension use cases `e` can be inserted into the behaviour of the base use case `b`.
`e` is optional.



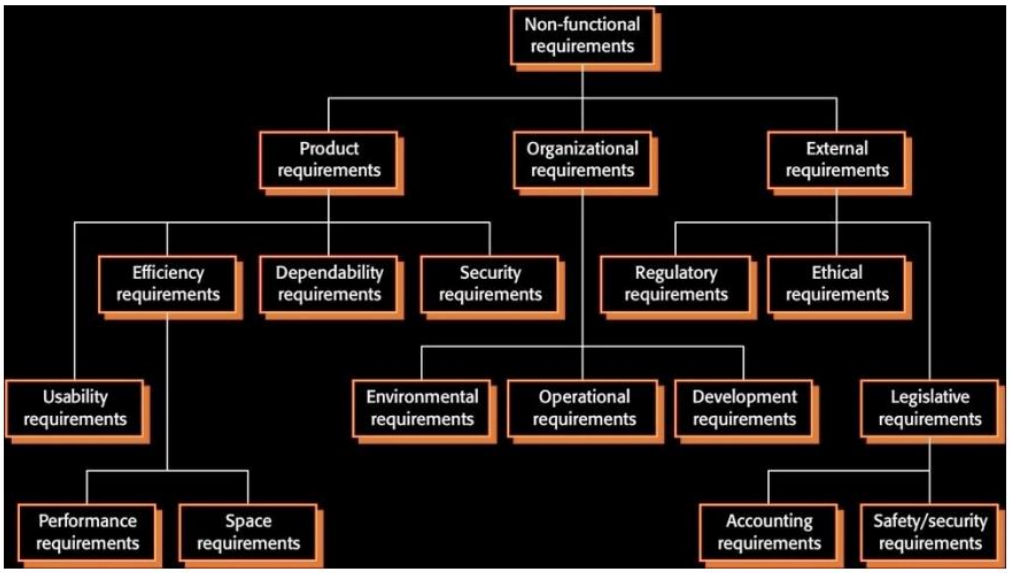
Extend ralationship

Specifies how the behaviour of the included Use Case `p` contributes to the behaviour of the base use case `b`.



Include relationship

But also: Non-functional Requirements



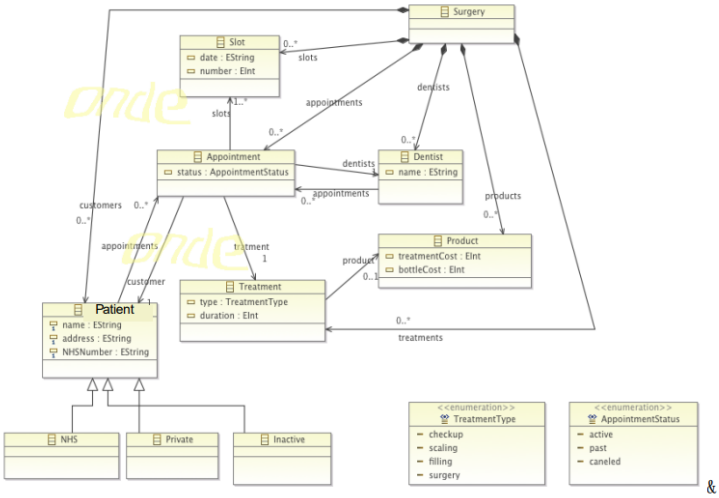
Week 4

Why OO Design?

- Organize ideas
- Plan work
- Build understanding of the system structure and behavior
- Communicate with development team
- Help (future) maintenance team to understand

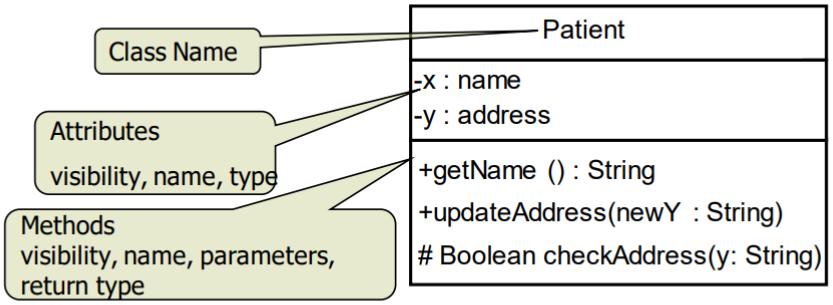
What Is a Class Diagram?

- Static view of a system



Class Diagrams

Class can be understood as a template for creating objects with own functionality



Visiblity:
+ public # protected - private

Notation for Attributes

[visibility] name [: type] [multiplicity] [=value] [{property}]

- visibility
  - other package classes
  - - private : available only within the class
  - + public: available for the world
  - # protected: available for subclasses and other package classes
  - ~ package: available only within the package
- [multiplicity], by default 1
- properties: readOnly, union, subsets<property-name>, redefined<property-name>, ordered, bag, seq, composite.
- static attributes appear underlined

Notation for Operations

[visibility] name ([parameter-list]) : [{property}]

- visibility
- method name
- formal parameter list, separated by commas:
  - direction name : type [multiplicity] = value [{property}]
  - static operations are underlined
- Examples:
  - display()
  - - hide()
  - + toString() : String
  - createWindow (location: Coordinates, container: Container): Window

How do we find Classes: Grammatical Parse

Classes

- Identify nouns from existing text
- Narrow down to remove
  - Duplicates and variations (e.g., synonyms)
  - Irrelevant
  - Out of scope
  - 重复和变体（例如同义词）
  - 不相关
  - 超出范围

Structural Relationships in Class Diagrams

What Is an Association?

- The semantic relationship between two or more classifiers that specifies connections among their instances.
- A structural relationship specifying that objects of one thing are connected to objects of another thing.
- **两个或多个分类器之间的语义关系，指定其实例之间的连接。**
- **一种结构关系，指定一个事物的对象与另一个事物的对象相连接。**

What Is Multiplicity?

● Multiplicity is the number of instances one class relates to ONE instance of another class.

● For each association, there are two multiplicity decisions to make, one for each end of the association.

● 多重性是指一个类与另一类的一个实例相关的实例数。
● 对于每个关联(association)，需要做出两个多重决策，关联的每一端都有一个决策。

○ For each instance of patient, many or no Appointments can be made.

○ For each instance of Appointment, there will be one Patient to see.

Multiplicity Indicators

Unspecified	
Exactly One	1
Zero or More	0..*
Zero or More	*
One or More	1..*
Zero or One (optional value)	0..1
Specified Range	2..4
Multiple, Disjoint Ranges	2, 4..6

What Is an Aggregation?

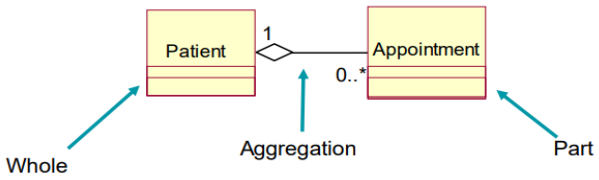
- A special form of association that models a whole-part relationship between the aggregate (the whole) and its parts.
    - An aggregation is an “is a part-of” relationship.
- Multiplicity is represented like other associations.

What Is Composition?

- A form of aggregation with strong ownership and coincident lifetimes
    - The parts cannot survive the whole/aggregate

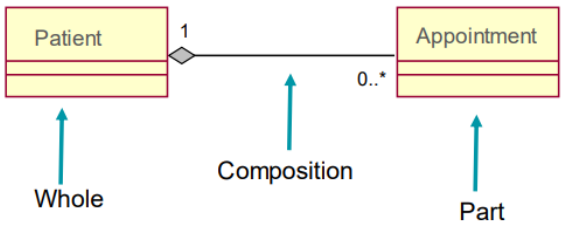
What Is an Aggregation?

- A special form of association that models a whole-part relationship between the aggregate (the whole) and its parts.
    - An aggregation is an “is a part-of” relationship.
- Multiplicity is represented like other associations.



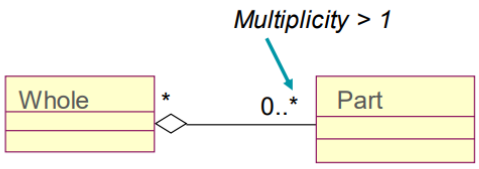
What Is Composition?

- A form of aggregation with strong ownership and coincident lifetimes
    - The parts cannot survive the whole/aggregate

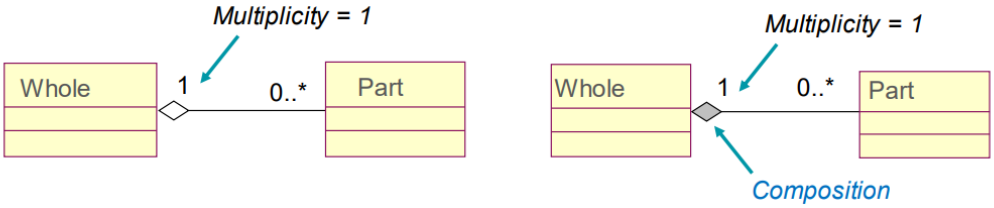


Aggregation: Shared vs. Non-shared

- Shared Aggregation



- Non-shared Aggregation



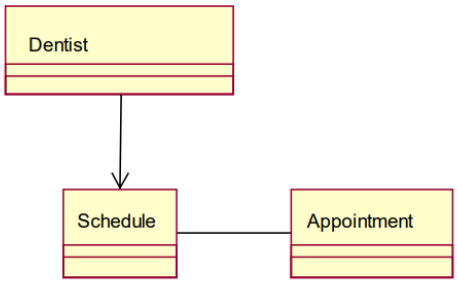
What Is Navigability?

● Indicates that it is possible to navigate from an associating class to the target class using the association

表示可以使用关联从关联类导航到目标类

What Is Navigability?

- Indicates that it is possible to navigate from an associating class to the target class using the association



What Is Generalization?

● A relationship among classes where one class shares the properties and/or behavior of one or more classes.

● Defines a hierarchy of abstractions where a subclass inherits from one or more superclasses.

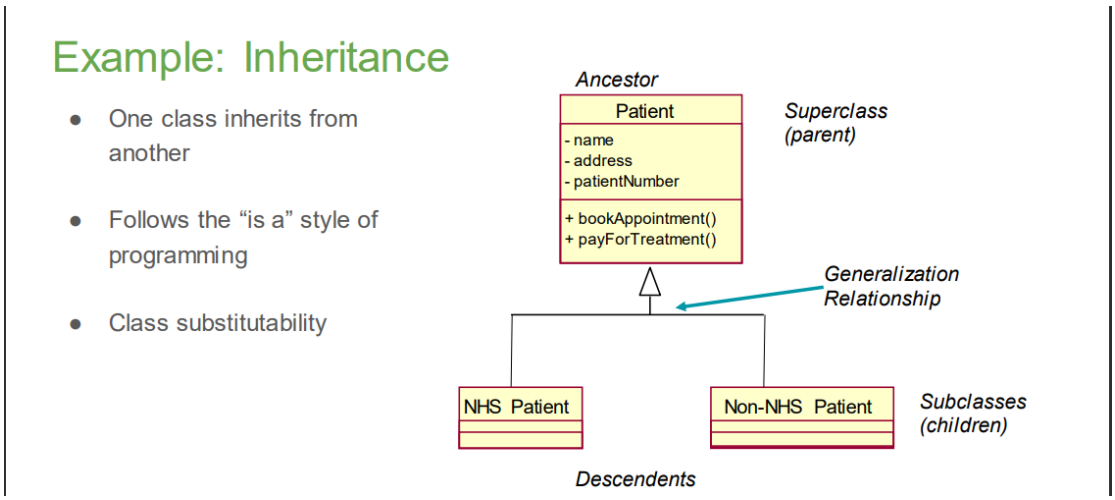
● Is an “is a kind of” relationship.

● 类之间的一种关系，其中一个类共享一个或多个类的属性和/或行为。

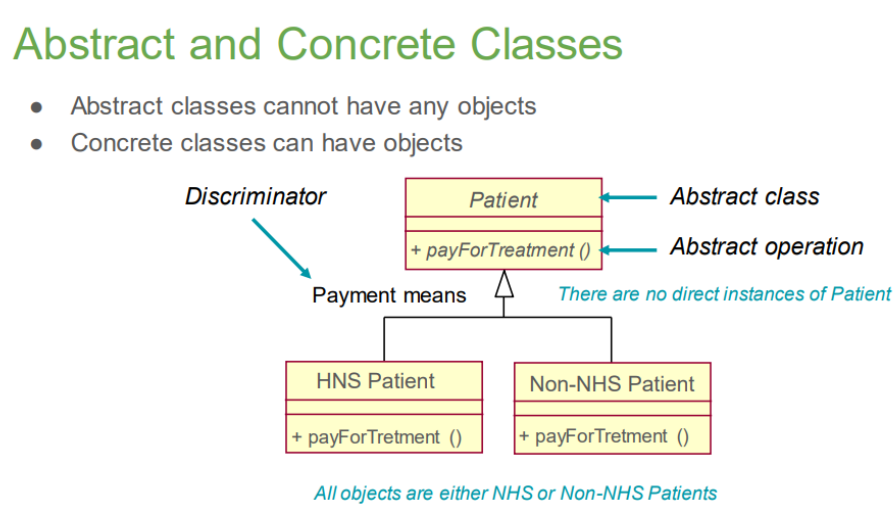
● 定义一种抽象层次结构，其中子类继承一个或多个超类。

● 是一种“是一种”关系。

Example: Inheritance



Abstract and Concrete Classes



Generalization vs. Aggregation

- Generalization and aggregation are often confused
    - Generalization represents an “is a” or “kind-of” relationship
    - Aggregation represents a “part-of” relationship

Behaviour Modelling

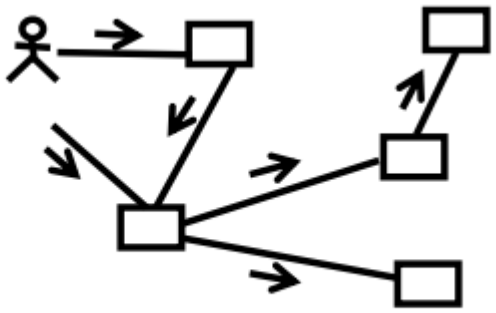
Objects Need to Collaborate

- Objects are useless unless they can collaborate to solve a problem.
    - Each object is responsible for its own behavior and status.
    - No one object can carry out every responsibility on its own.
- How do objects interact with each other?
    - They interact through messages.
    - Message shows how one object asks another object to perform some activity.

- 除非对象能够协作解决问题，否则它们毫无用处。
    - 每个对象对其自己的行为和状态负责。
    - 没有任何一个对象能够独自承担所有责任。
- 对象之间如何相互作用?
    - 他们通过消息进行交互。
    - 消息显示一个对象如何要求另一个对象执行某些活动。

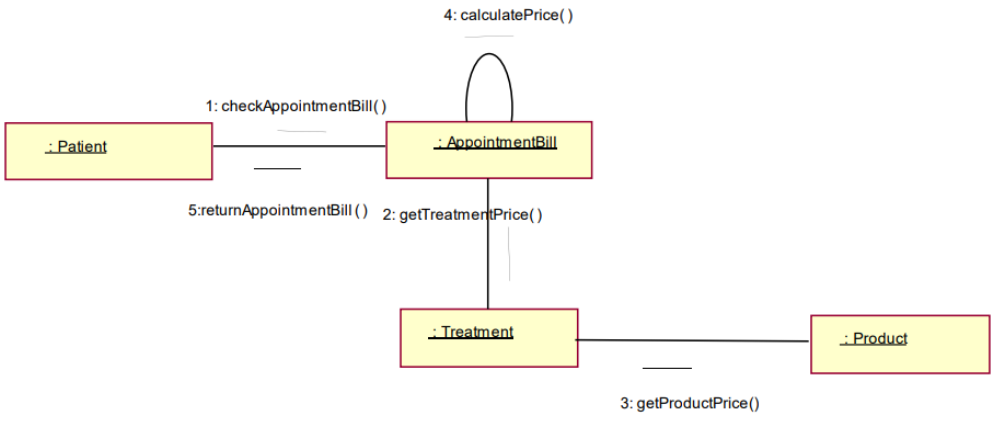
What Is a Communication Diagram?

- A communication diagram emphasizes the organization of the objects that participate in an interaction.
- The communication diagram shows:
  - The objects participating in the interaction.
  - Links between the objects.
  - Messages passed between the objects.
- 通信图强调参与交互的对象的组织。
- 通讯图所示：
  - 参与交互的对象。
  - 对象之间的链接。
  - 对象之间传递的消息。



Communication Diagrams

Example: Communication Diagram

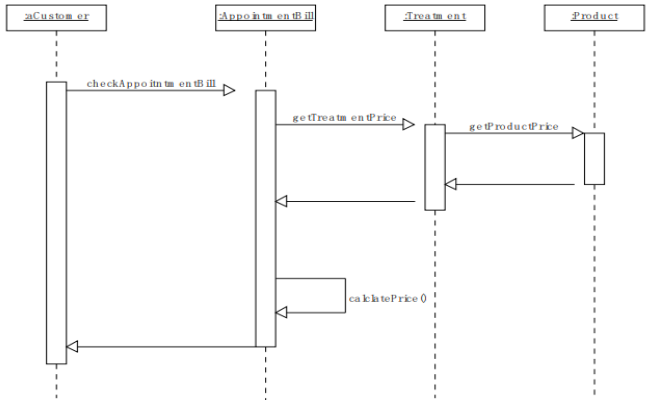


Sequence Diagrams: Basic Elements

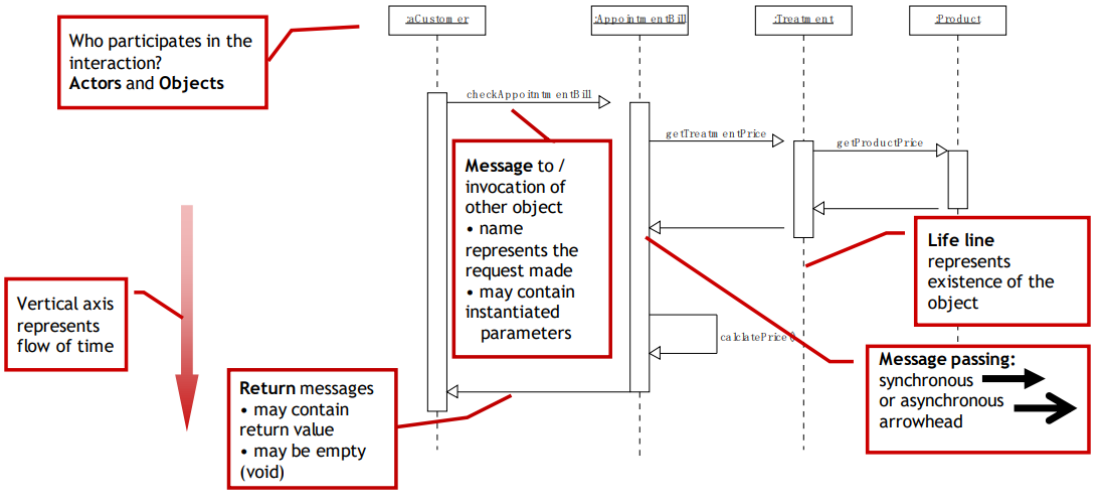
- A set of participants arranged in time sequence
- Good for real-time specifications and complex scenarios
- Who participates in the interaction? Actors and Objects
- Return messages
  - may contain return value
  - may be empty (void)
- Message to / invocation of other object
  - name represents the request made
  - may contain instantiated parameters
- Life line represents existence of the object

Sequence Diagrams: Basic Elements

- A set of participants arranged in time sequence
- Good for real-time specifications and complex scenarios



Sequence Diagrams: Basic Elements



Method for Analysis Sequence Diagrams

- for each scenario (high-level sequence diagram)
  - decompose to show what happens to objects inside the system
    - ➔ objects and messages
  - Which tasks (operation) does the object perform?
    - ➔ label of message arrow
  - Who is to trigger the next step?
    - ➔ return message or pass on control flow

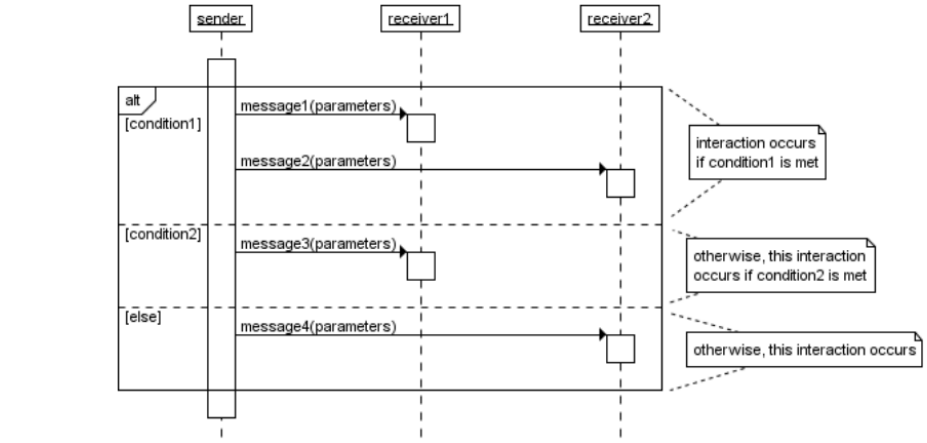
Sequence Diagrams

- Sequence Diagrams can model simple sequential flow, branching, iteration, recursion and concurrency
- They may specify different scenarios/runs
  - Primary
  - Variant
  - Exceptions

- 序列图可以对简单的顺序流、分支、迭代、递归和并发进行建模
- 他们可能指定不同的场景/运行
  - 主要的
  - 变体
  - 例外情况

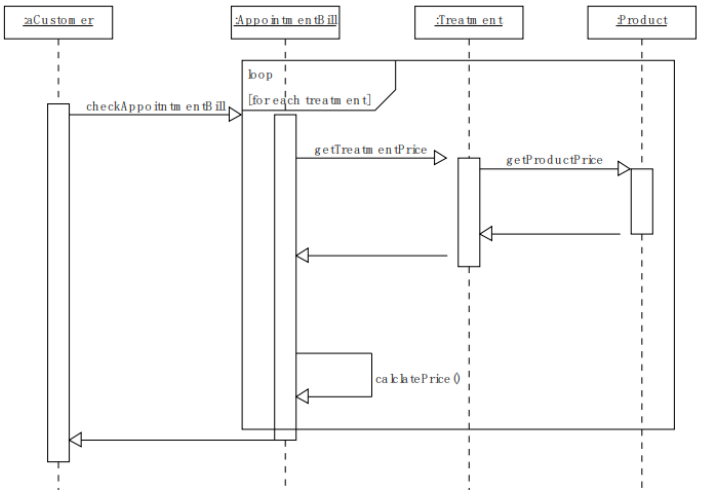
Interaction frames: alt and loop

Interaction frames: alt



https://web.archive.org/web/20231018070441/http://www.tracemodeler.com/articles/a_quick_introduction_to_uml_sequence_diagrams/

Interaction frames: loop



Sequence and Communication Diagram Similarities

- Semantically equivalent
    - Can convert one diagram to the other without losing any information
- Model the dynamic aspects of a system
- Model a use-case scenario

- 语义等效
  - 可以将一个图表转换为另一个图表，而不会丢失任何信息
- 对系统的动态方面进行建模
- 对用例场景进行建模

Sequence and Communication Diagram Differences

Sequence diagrams	Communication diagrams
Show the explicit sequence of message	Show relationships in addition to interactions
Show execution occurrence	Better for visualizing patterns of communication
Better for visualizing overall flow	Better for visualizing all of the effects on a given object
Better for real-time specifications and for complex scenarios	Easier to use for brainstorming sessions