Software Development

But have you heard of ***Software Crisis***

- Software is getting larger and larger
- Software is getting more complex
  - Complex domains (e.g., human behaviours)
  - Systems dependency (e.g., energy and cars)
- Time to market is shorter than ever

Why Do Software Projects Fail?

- Bad developers – not the main problem!
- Over budget
  - Poor requirements
  - Over-ambitious requirements
  - Unnecessary requirements
- Contract management
- End-user training
- Operational management

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

Collaboration Tools and Techniques

- UML Designs: Diagramming to reach a consensus on design
- GitHub: Sharing of documents for effective collaboration
- Kanban: Task allocation and progress monitoring
- Test-driven development: Stop other people breaking your code !
- Other techniques are encouraged: Feel free to experiment and explore

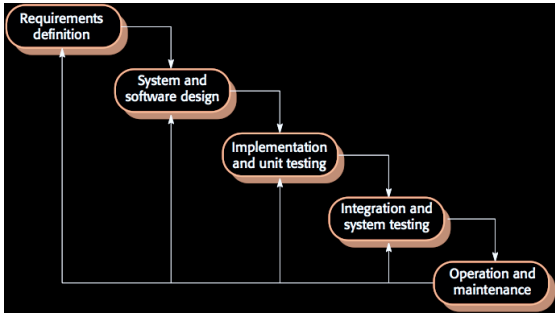
Software Development Tasks

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

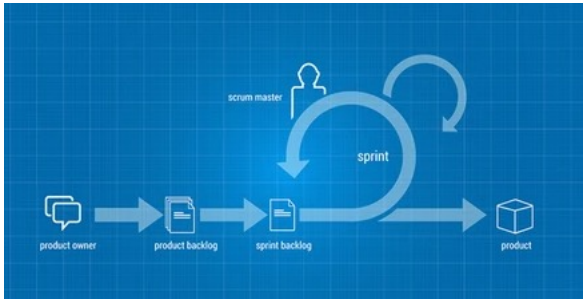
These to-does are combined in various sequences, making up different Software Development Life Cycle Processes

Software Development Life Cycle Examples

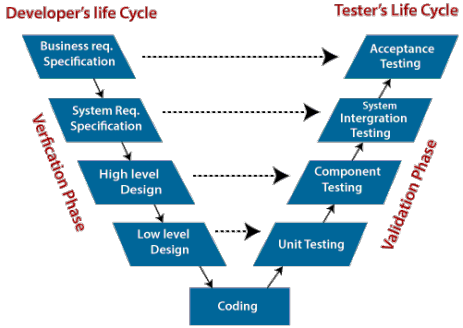
Waterfall



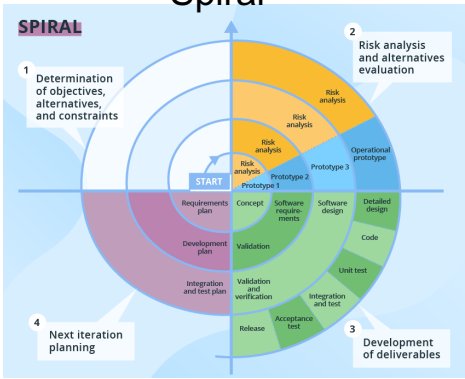
Agile



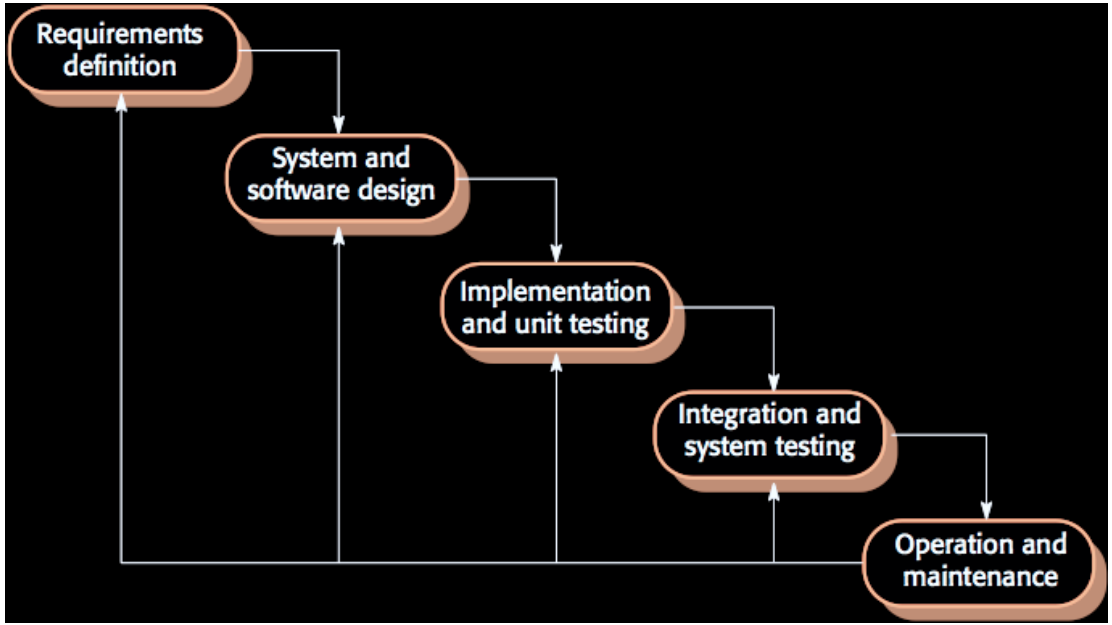
V-Model



Spiral



Waterfall Software Development Life Cycle



1. Requirements definition
2. System and software design
3. Implementation and unit testing
4. Integration and system testing
5. Operation and maintenance

Requirements: What to Implement?

- What should the system do ?
- What are the needs of the users ?
- What are the needs of the host organisation ?
- What are interoperability needs of existing systems

We need to think about functional elements: what the system should do?
And also non-functional elements: quality properties of system operation, e.g., security, ease of use, response time etc.

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

Implementation

Not just programming, but also…

- Parallel working and code sharing
- API partitioning and “firewalling”
- Versioning, integration and config management
- Development environments and automated build
- Automated testing
- Docs and training material

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

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

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?

Agile Software Development

Dr Jon Bird
jon.bird@bristol.ac.uk

Thanks to Dr Simon Lock who developed many of these slides for an earlier version of this unit.

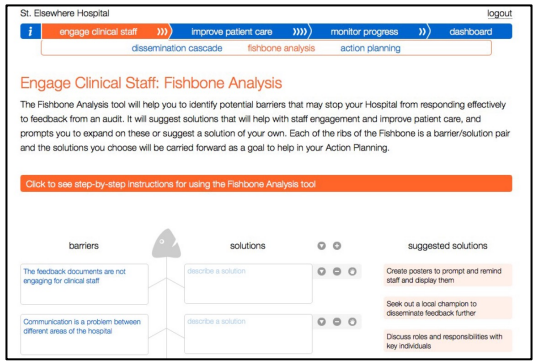
Images are royalty free from www.pexels.com

Today’s Lecture

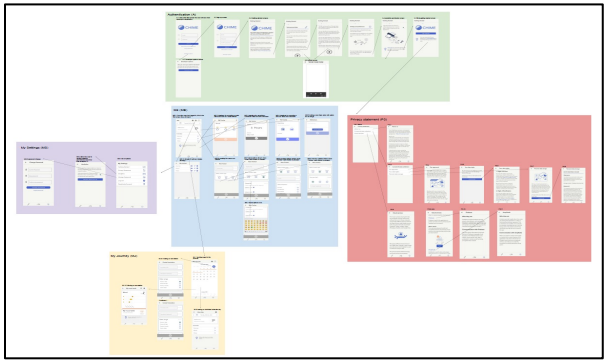
- A digital health project developed using the waterfall approach, which was the focus of last week
- Waterfall versus agile approaches to software development
- Agile software development, including: extreme programming (which includes pair programming); test-driven development; scrum; and Kanban
- A digital health project developed using an agile approach
- Recommended reading

Digital health projects

- My research involves developing digital technologies to address health issues
- What software development process do I employ in my research?



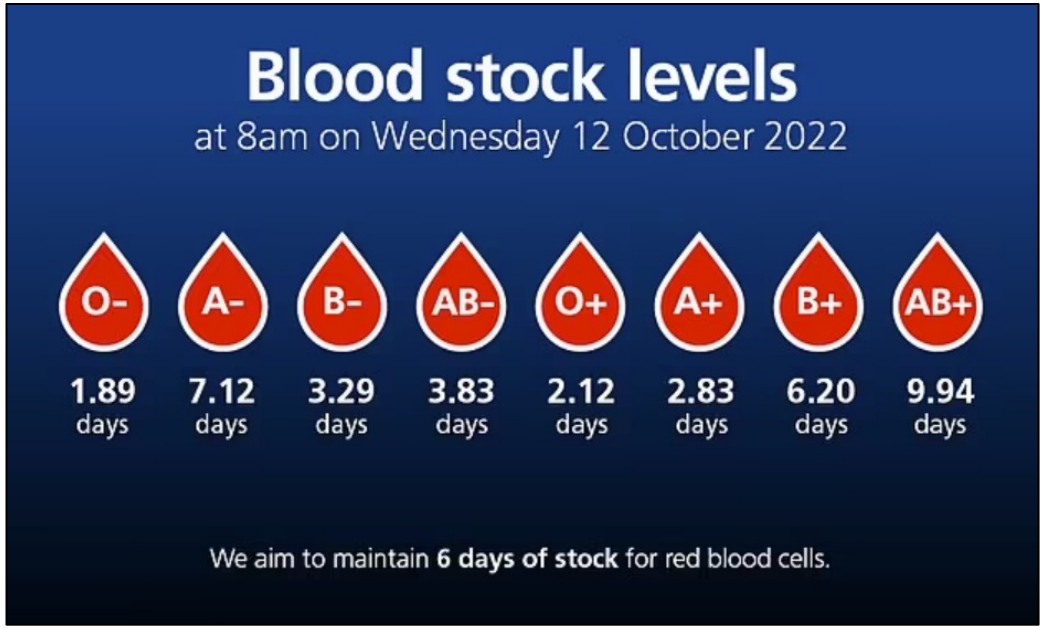
Affinitie



CHIME

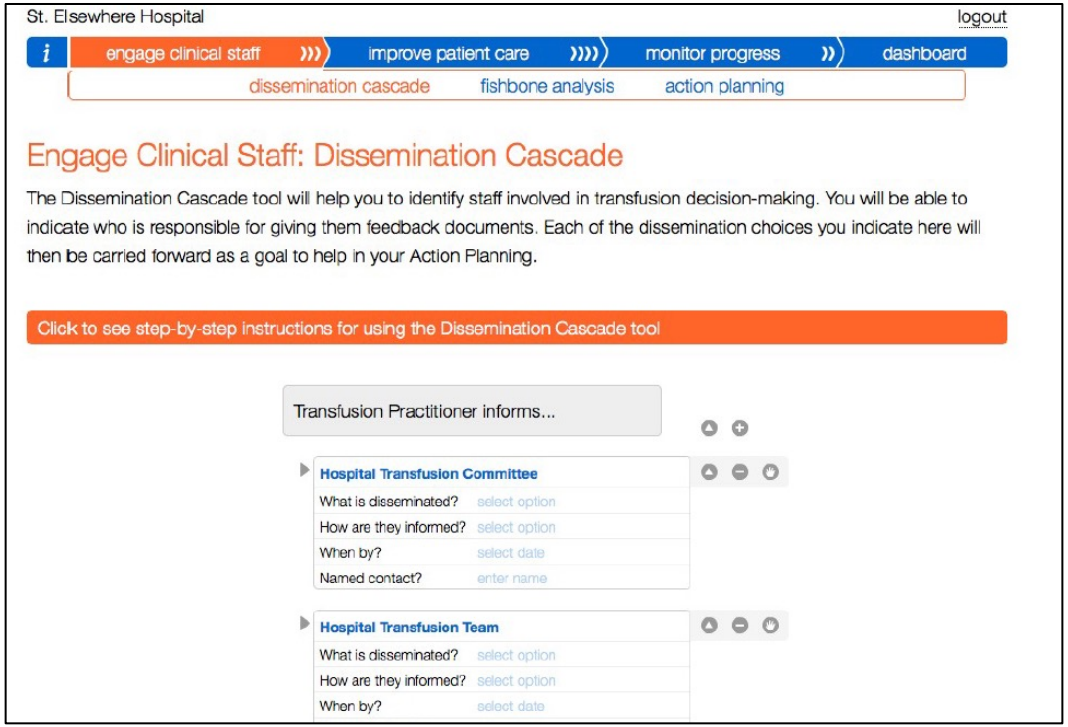
Affinitie - motivation

- There are 3 million blood transfusions annually in the UK
- Around 20% are unnecessary
- Blood components are a scarce resource
- There are health risks associated with blood transfusions
- How can we reduce unnecessary blood transfusions?



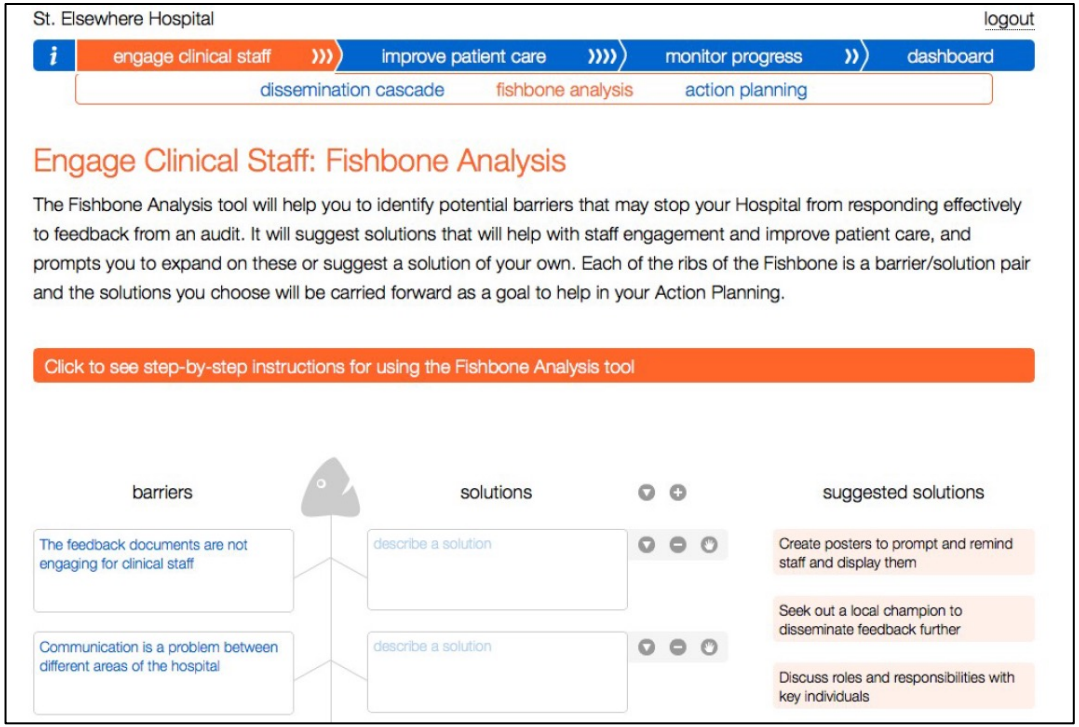
Affinitie – software

- I was asked to join an existing research project to develop a web toolkit for transfusion practitioners
- The research team consisted of health psychologists, statisticians, doctors and NHS Blood and Transplant
- The aim was to help transfusion practitioners disseminate blood transfusion audit results to everyone in the hospital

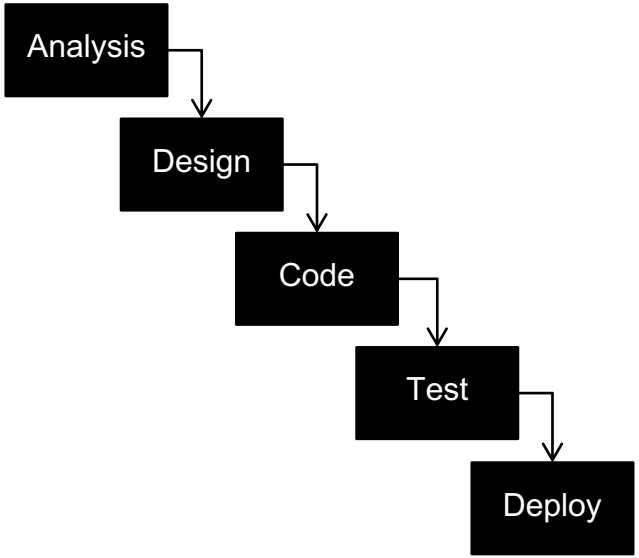


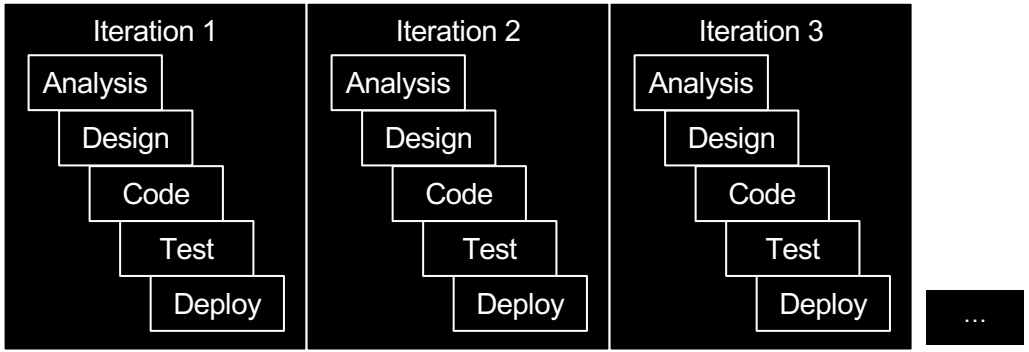
Affinitie – software development

- We were given **a clear set of requirements** by the research team
- They had already developed a set of paper-based tools for transfusion practitioners to help them disseminate blood transfusion audit results
- Software development approach: **waterfall**



Waterfall versus agile life cycles





What is Agile Software Development?

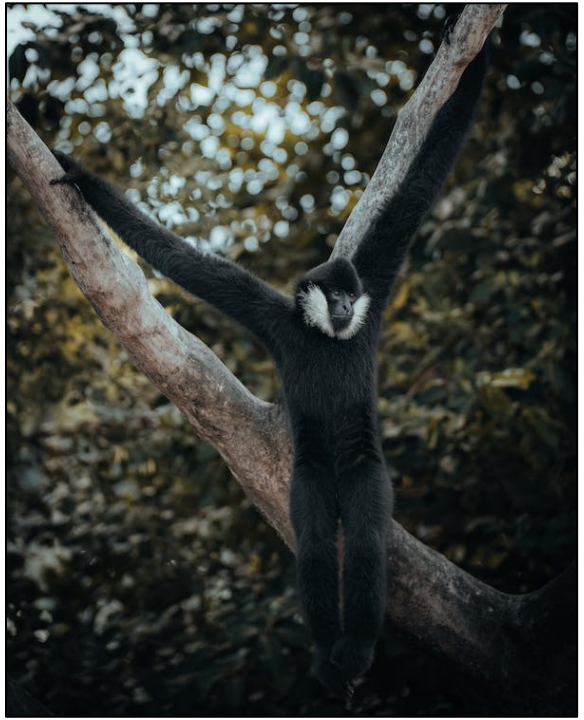
- Agile is a way of thinking about software development
- In winter 2001, 17 software developers met at a ski resort in Utah and drafted a manifesto outlining an alternative way to develop software to the documentation-driven software development processes of the time
- The manifesto is succinct and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

The Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

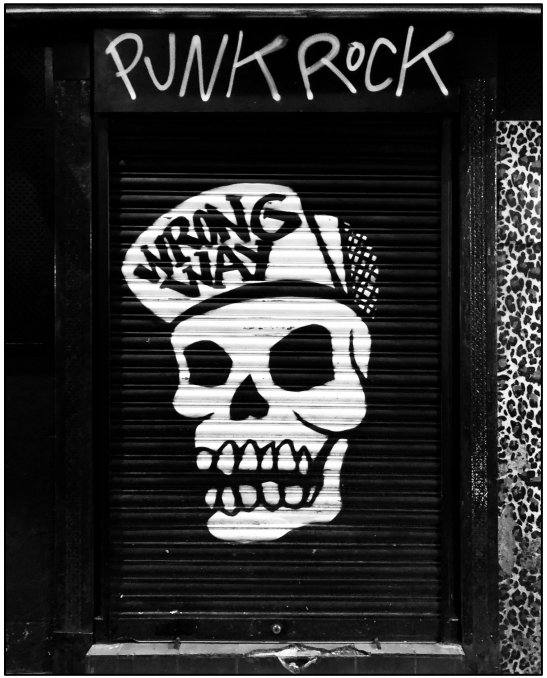
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more



How should you read the manifesto?

- The values are purposefully provocative in order to get people to think about software development
- The Agile Manifesto is **not** proposing that we disregard aspects of software development like processes, tools and documentation
- Rather, the Agile Manifesto wants people to think about alternative ways of doing aspects of software development



Agile was created by coders for coders

Coders like
Writing quality code
Ticking things off their to do list
Impressing clients by showing them working software

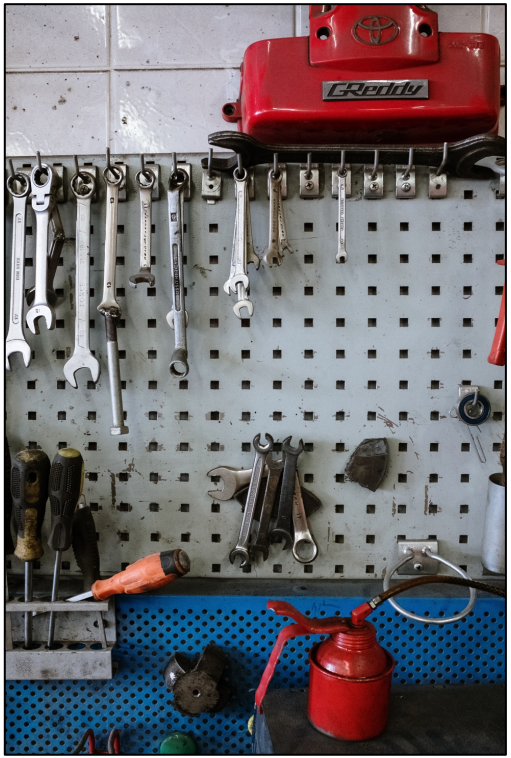
Coders dislike
Writing extensive documentation
Committing to a final design in advance
Being micromanaged
Working to big, immovable deadlines

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

Agile Methods

- 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)
    - Test-driven development (creator was a signatory)
    - Kanban
    - Scrum (the two co-creators were signatories)
- We'll introduce some key practices from each of these methods that we think you will be useful in this unit and your summer projects
- There are links in the reading to some of these methods



Hands up if any of these apply to you

- Your code structure is complex and “sophisticated”
- You work primarily on your own
- In group projects you are responsible for just your own code
- You write code in your own style
- You work some weekends and so some “all-nighters”
- You code develops in “heroic bursts”

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
- **Test driven**: ensure the code runs correctly
- **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

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

The impact of pair programming

Research studies have assessed the impact of pair programming and identified a number of benefits

Single programmer 77 source lines per month versus pair programming 175 source lines per month

[Jensen, 2005]

15% increase in development-time costs but improves design quality, reduces defects, reduces staffing risk, enhances technical skills, improves team communications and is considered more enjoyable at statistically significant levels.

[Cockburn & Williams, 2000]

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

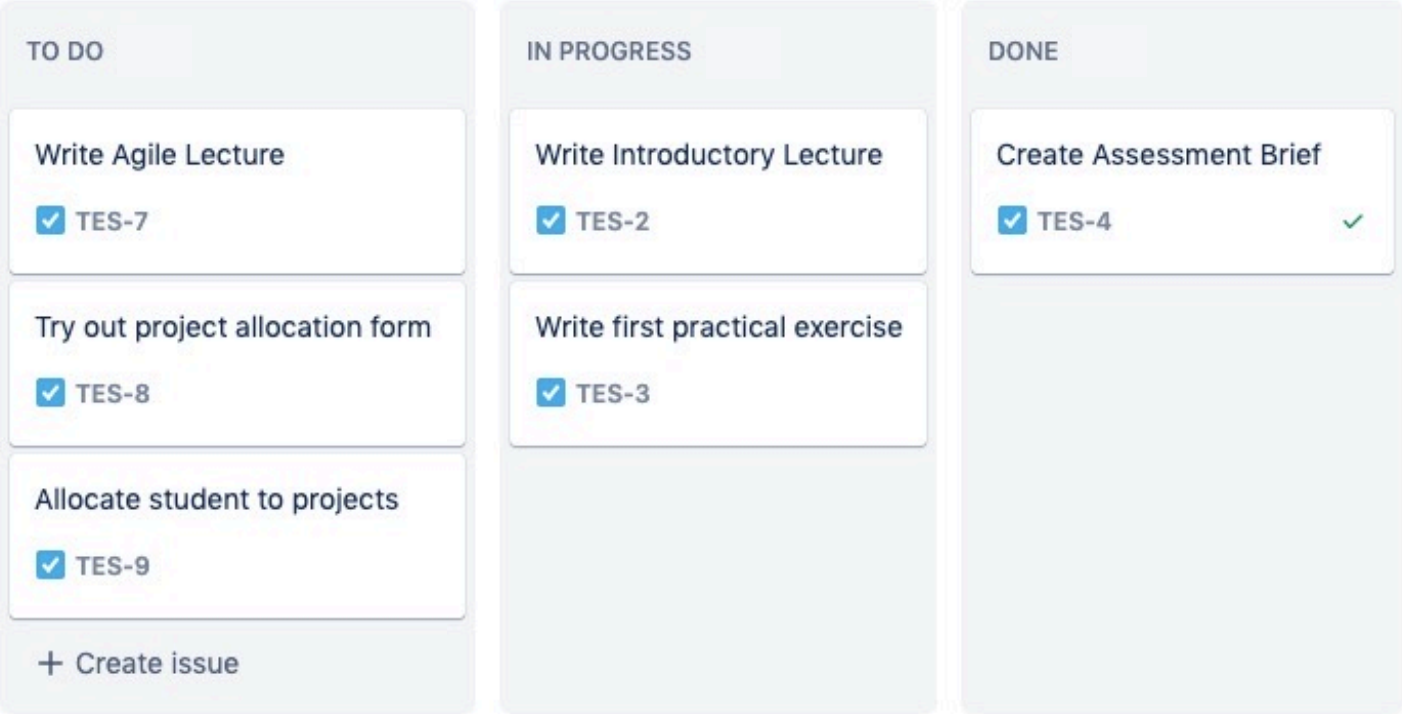
Scrum

- 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)

Kanban Board

- A concept taken from the Kanban method which was first defined in 2007 but came out of a scheduling system developed by Toyota in the 1950s for just-in-time manufacturing
- In Japanese “kanban” means “visual board” or “sign”
- It’s basically a flexible “to do” list tool
- Issues progress through various states from ”To do” to “Done”
- It was originally implemented as post-it notes on a whiteboard
- Various digital tools now fulfil the same function e.g. Jira

Jira Kanban Board

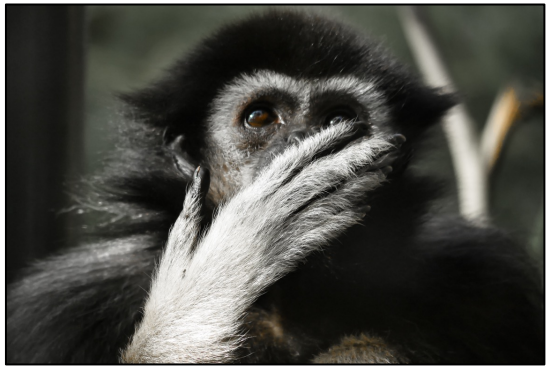


Columns (Swim lanes)

- It is common to use three columns
- But Jira allows you to customize the layout
- For example, you might have columns for:
  - Backlog
  - Being Verified
  - Awaiting integration
- Do what works for your team but make sure you have a “Done” column

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) ?



CHIME - motivation

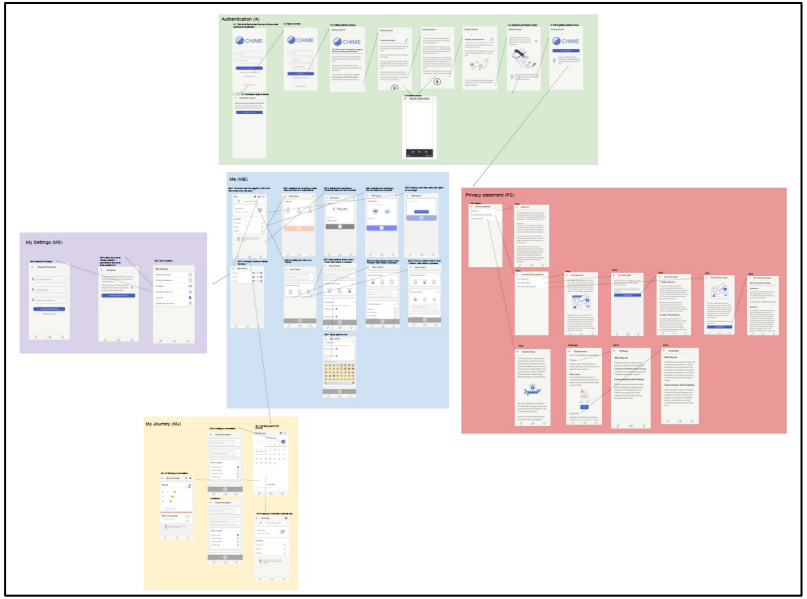
- Tracking health and lifestyle data can help people manage long term conditions
- Sharing these data with healthcare professionals can improve clinical decision making

CHIME - software

- Chime is an app designed for people living with HIV (PLHIV)
- It was developed by researchers at a number of UK universities in collaboration with the Terrence Higgins Trust, the leading HIV charity in the UK

CHIME – software development

- I project manged two programmers who developed the app code at UoB
- We initially designed and built a prototype app based on the requirements identified by other researchers on the project
- The app was evaluated by stakeholders who identified more requirements
- We then carried out a series of four two-week sprints and presented working code at the end of each sprint to other researchers
- The app was then evaluated by PLHIV
- Software development approach: **agile**



Requirements Engineering

Lecture 3

Ruzanna Chitchyan, Jon Bird, Pete Bennett
TAs: Alex Elwood, Alex Cockrean, Casper Wang

Overview

- What are requirements?
- Stakeholder Identification
- Functional and Non-Functional Requirements
- Describe system behavior and capture it in a model with Use Case Model
  - Use Case Diagram
  - Use Case Specification
- Requirements Quality

Requirements

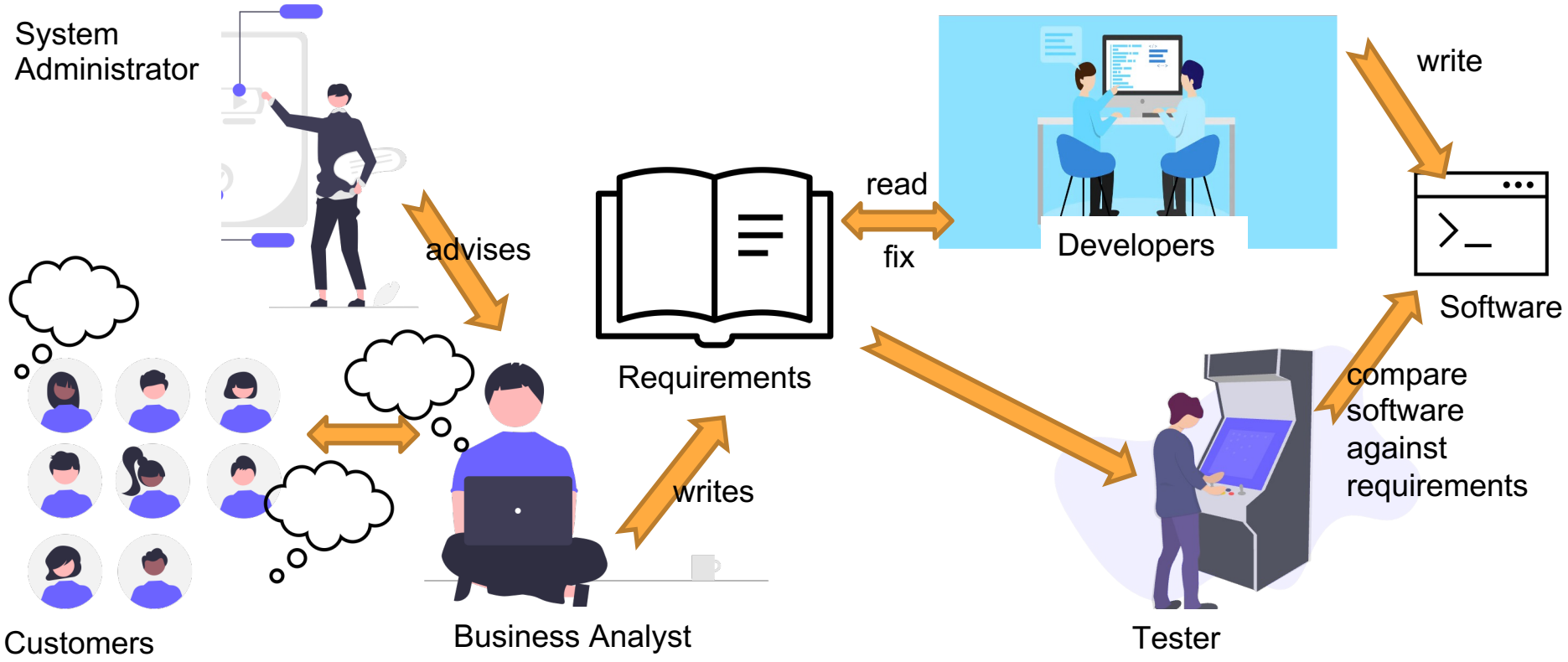
- **System requirements** specify a system, not in terms of system implementation, but in terms of user observation. Requirements record description of the system features and constraints.
  - **Functional requirements** specify user interactions with the system, they say **what** the system is supposed 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 realised:
    - Constraints ON the services or functions offered by system
    - Often apply to whole system, not just individual features

Why do we need Requirements Engineering?

General Problems and Requirements Engineering

- Inconsistent terminology: people express needs in **their own words**
- **Conflicting** needs for the same system
- People frequently **don’t know** what they want (or at least can't explain !)
- Requirements **change** quite frequently
- Relevant people/information may **not be accessible**

Requirements are **communication mechanism**



Requirements are **instructions**

- On your own or in pairs
- Follow some instructions to draw.
- I'll then judge whether you followed the instructions correctly.
- NOT your artistic skill!

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

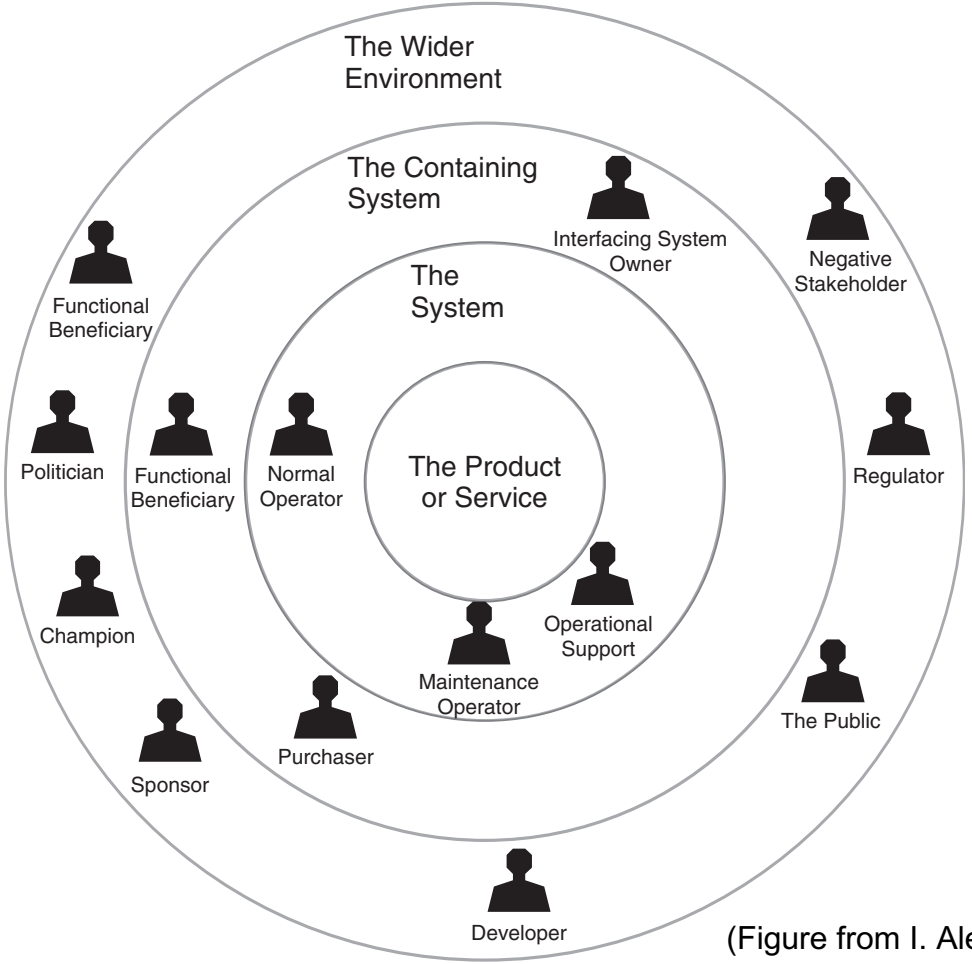
Analysing Requirements

1. Identify stakeholders involved with the system
2. Identify top-level user needs (e.g., as NFRs or “user stories”)
3. Break down stories into individual steps / Refine requirements
4. Specify atomic requirements (e.g., for each step in user stories)

Let’s look at each of these stages in turn…

1. Identify Stakeholders

The Onion model



(Figure from I. Alexander's book)

Stakeholders

Identification

- Clients
- Documentation, e.g., organisation chart
- Templates (e.g., onion model)
- Similar projects
- Analysing 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 needs/concerns

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.

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.

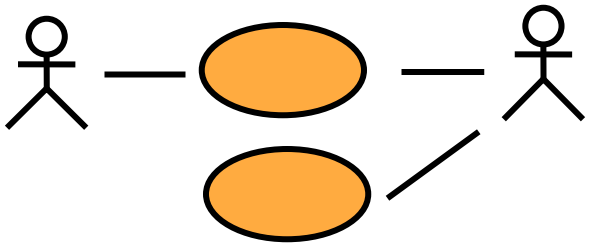
What is a use-case model?

- 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**

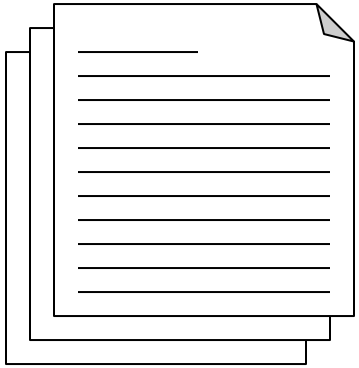
Capture a use-case model

- A use-case model is comprised of:

Use-case diagrams
(visual representation)

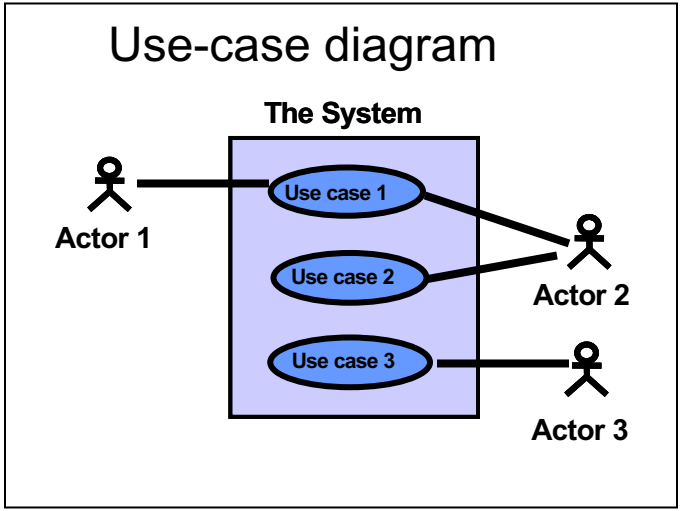


Use-case specifications
(text representation)



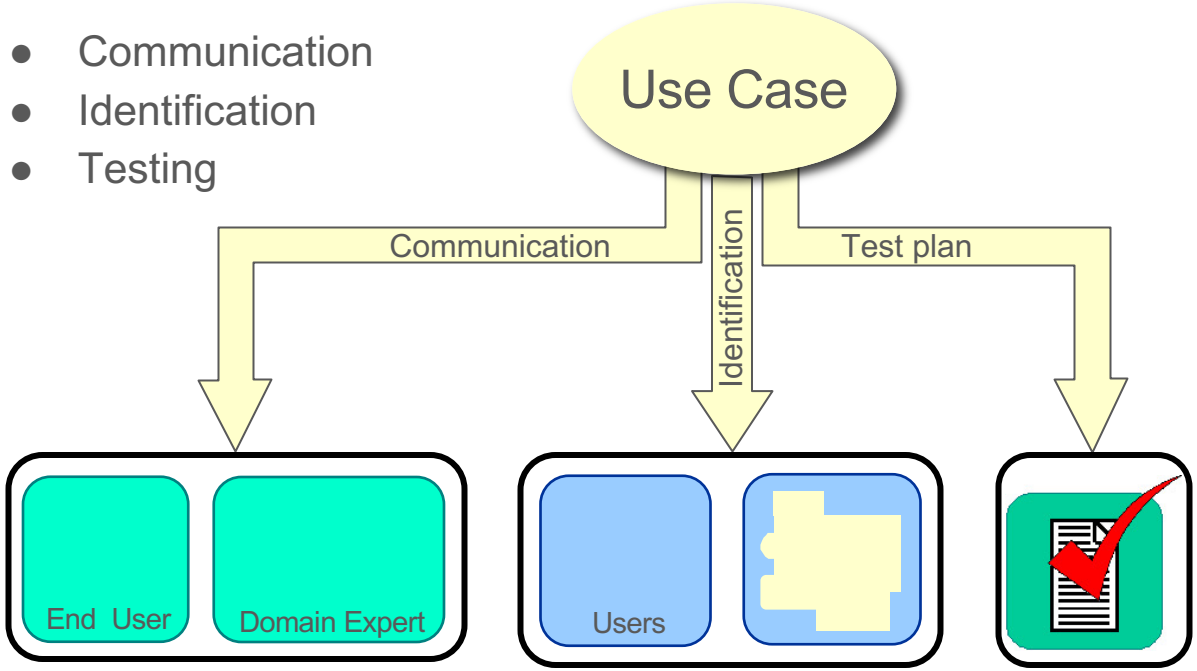
Use-case diagram

- 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

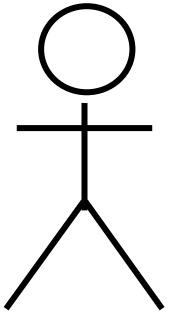
- An actor represents anything that interacts with the system.

Actor

- 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?

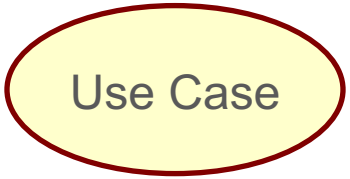
- Actors represent roles a user of the system can play.
- They can represent a human, a machine, or another system.
- They can actively interchange information with the system.
- They can be a giver of information.
- They can be a passive recipient of information.
- Actors are not part of the system.
  - Actors are EXTERNAL.



Actor

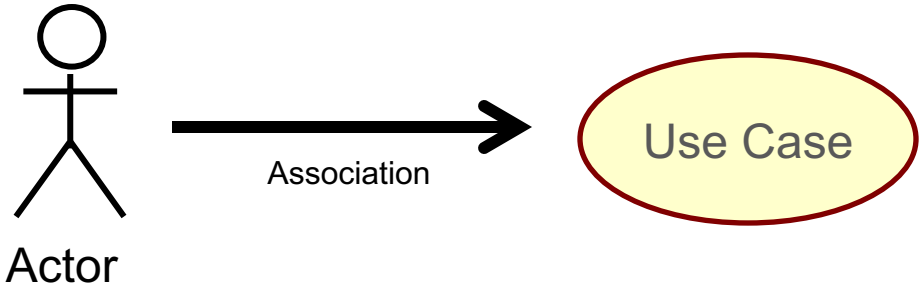
What Is a Use Case?

- 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.
  - 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



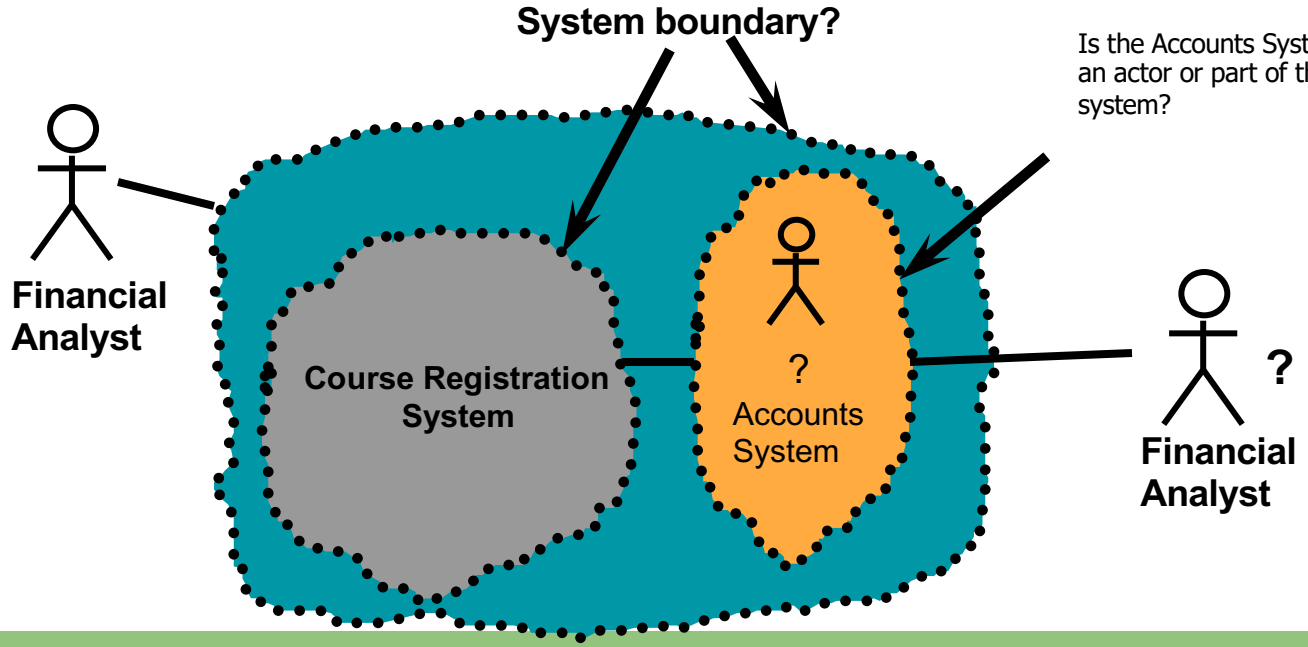
Use Cases and Actors

- A use case models a dialog between actors and the system.
- A use case is initiated by an actor to invoke a certain functionality in the system.



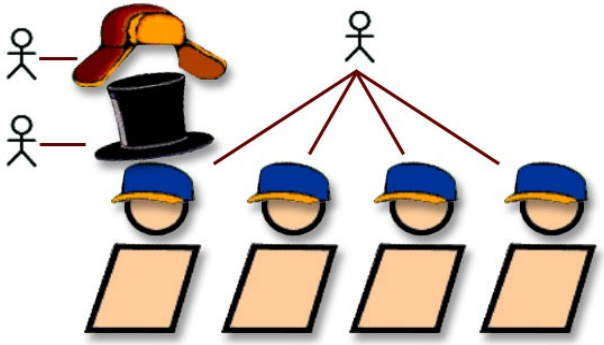
Actors and the system boundary

- Determine what the system boundary is
- Everything beyond the boundary that interacts with the system is an instance of an actor

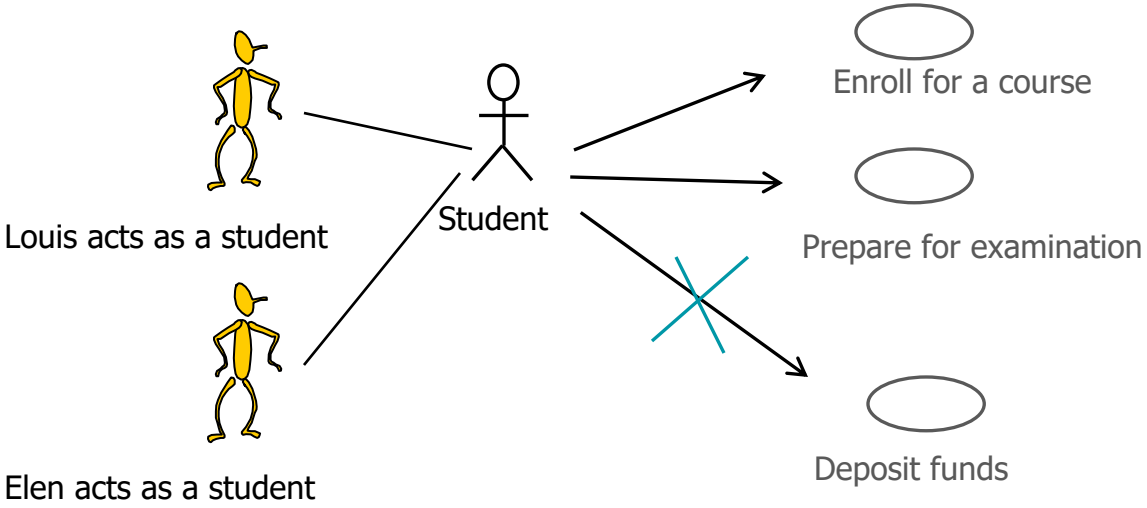


Actors and roles

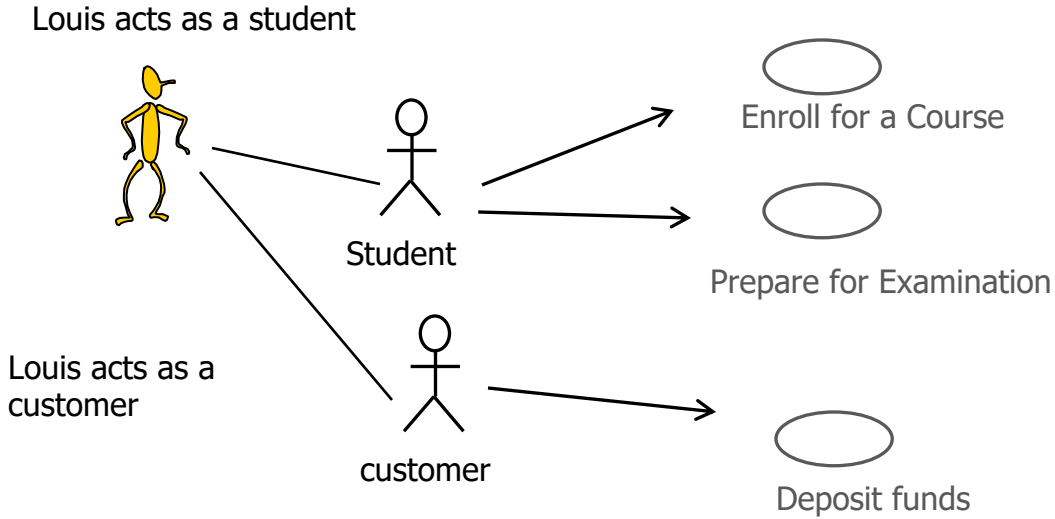
- An actor represents a role that a human, hardware device, or another system can plan in relation to the system.



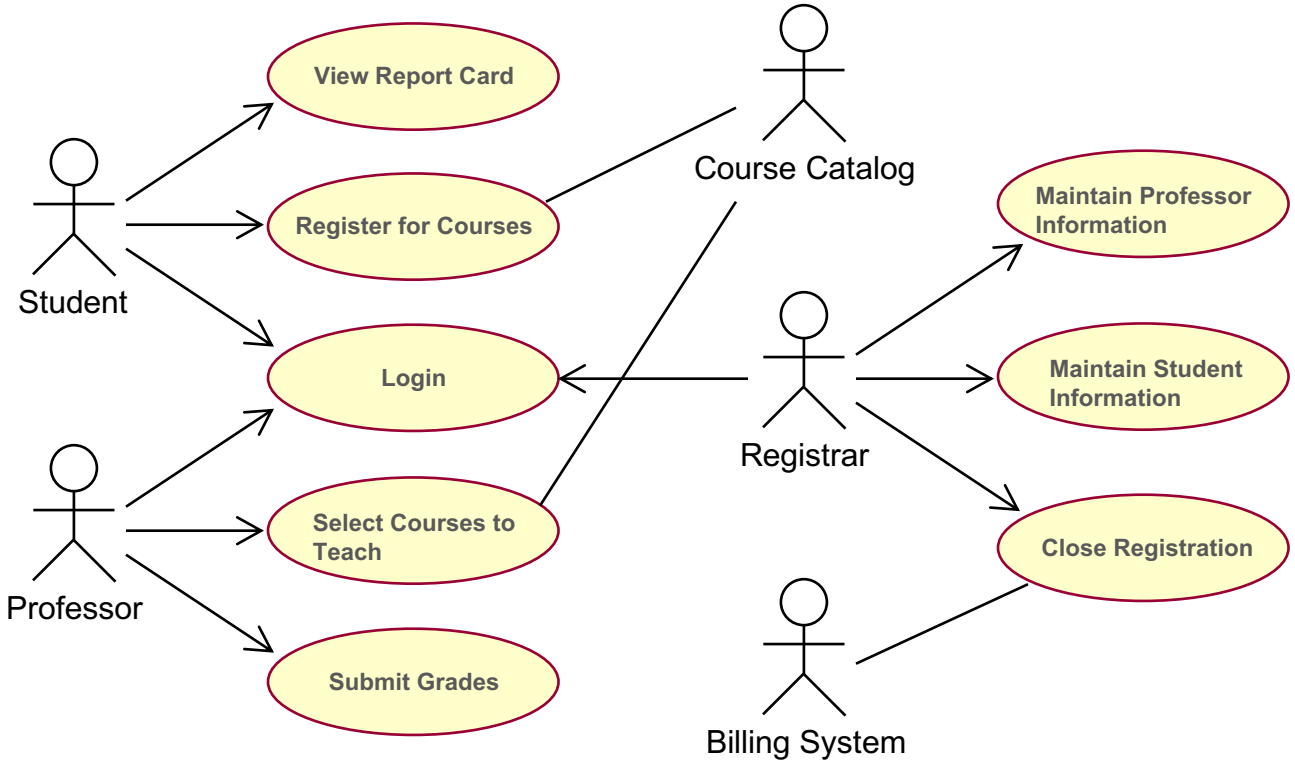
Actors


Louis acts as a student
Elen acts as a student
Student
Enroll for a course
Prepare for examination
Deposit funds

Actors

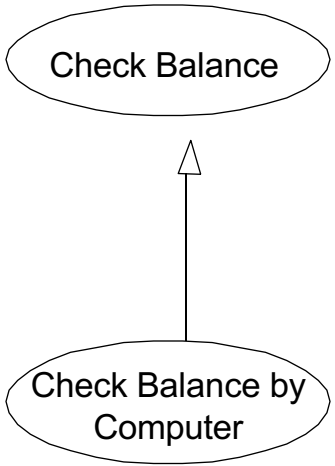

Louis acts as a student
Student
Enroll for a Course
Prepare for Examination
Louis acts as a customer
customer
Deposit funds

How Would You Read This Diagram?

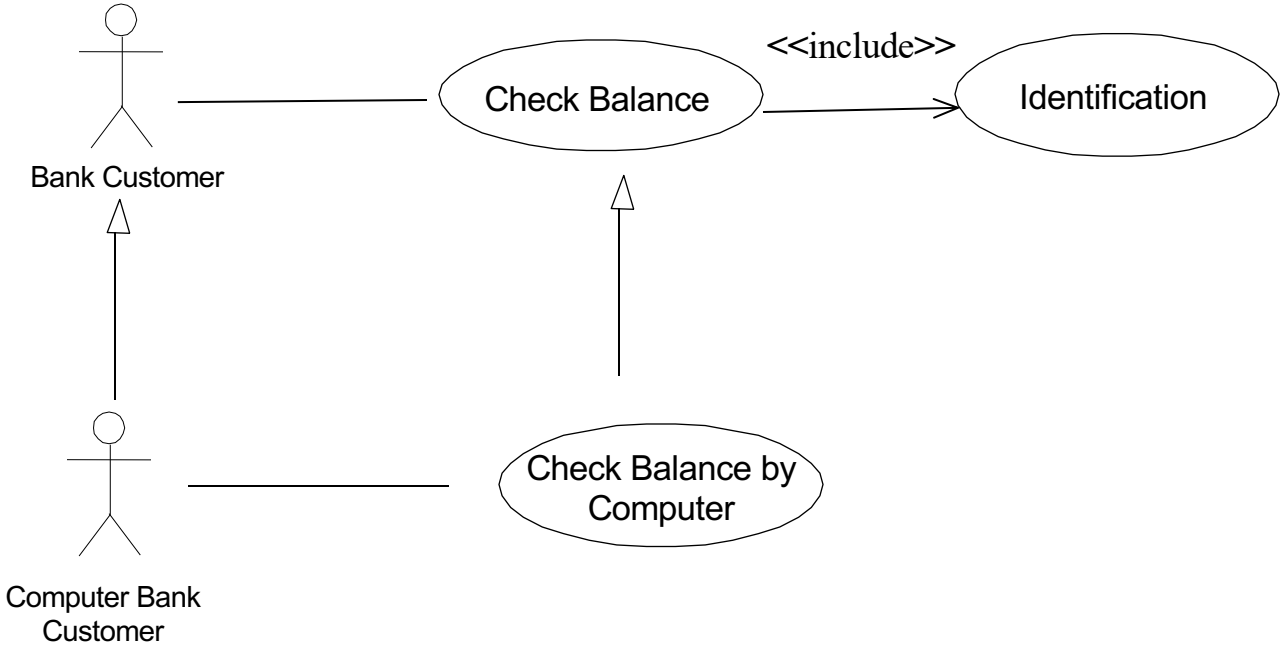


Modelling with Use Cases

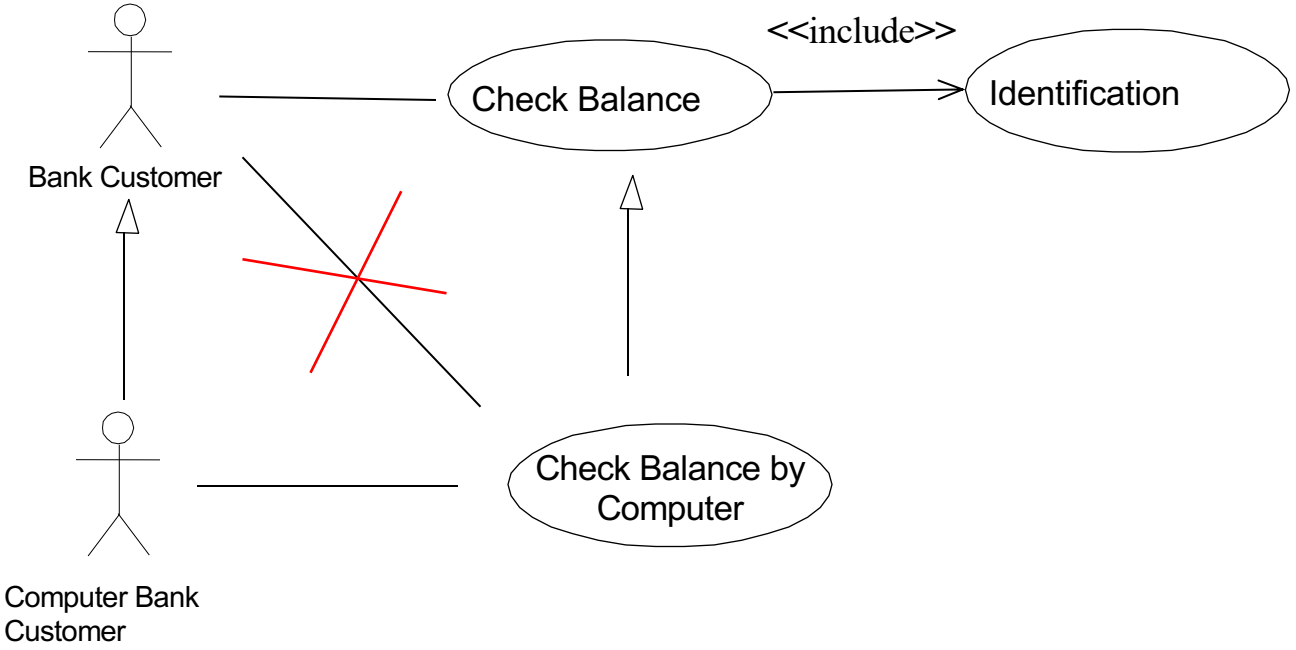
- Generalization between Use Cases means that the child is a more specific than the parent; the child inherits all attributes and associations of the parent, but may add new features



Structuring Use Case Diagrams

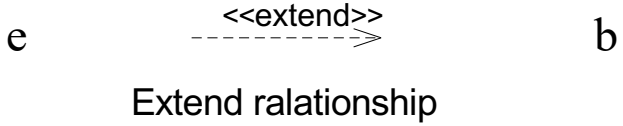


Use Case Diagram: Structuring



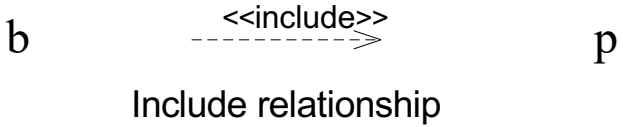
Use Cases

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.



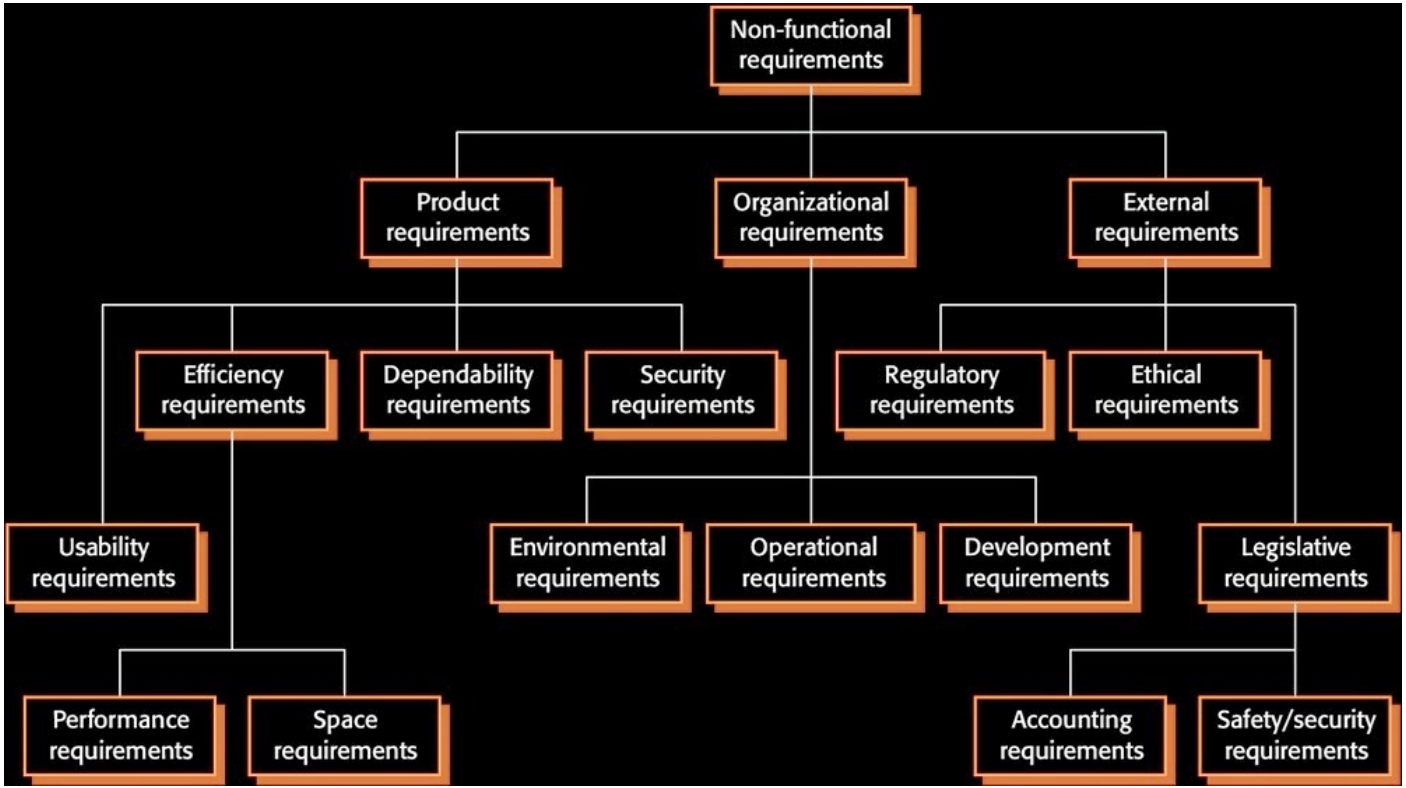
Extend ralationship

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



Include relationship

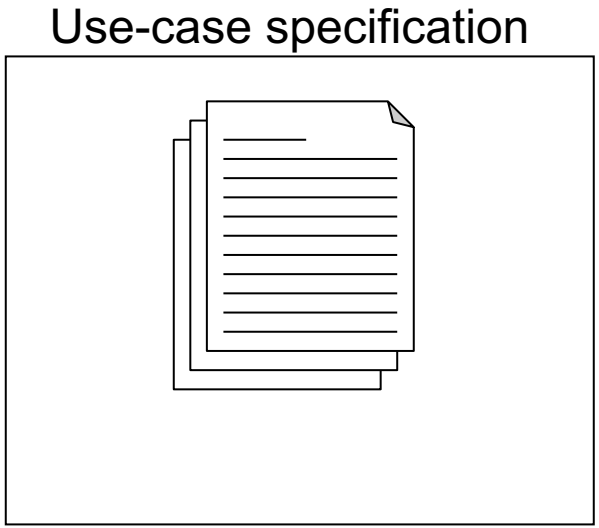
But also: Non-functional Requirements



3. Break down stories into individual steps / Refine requirements

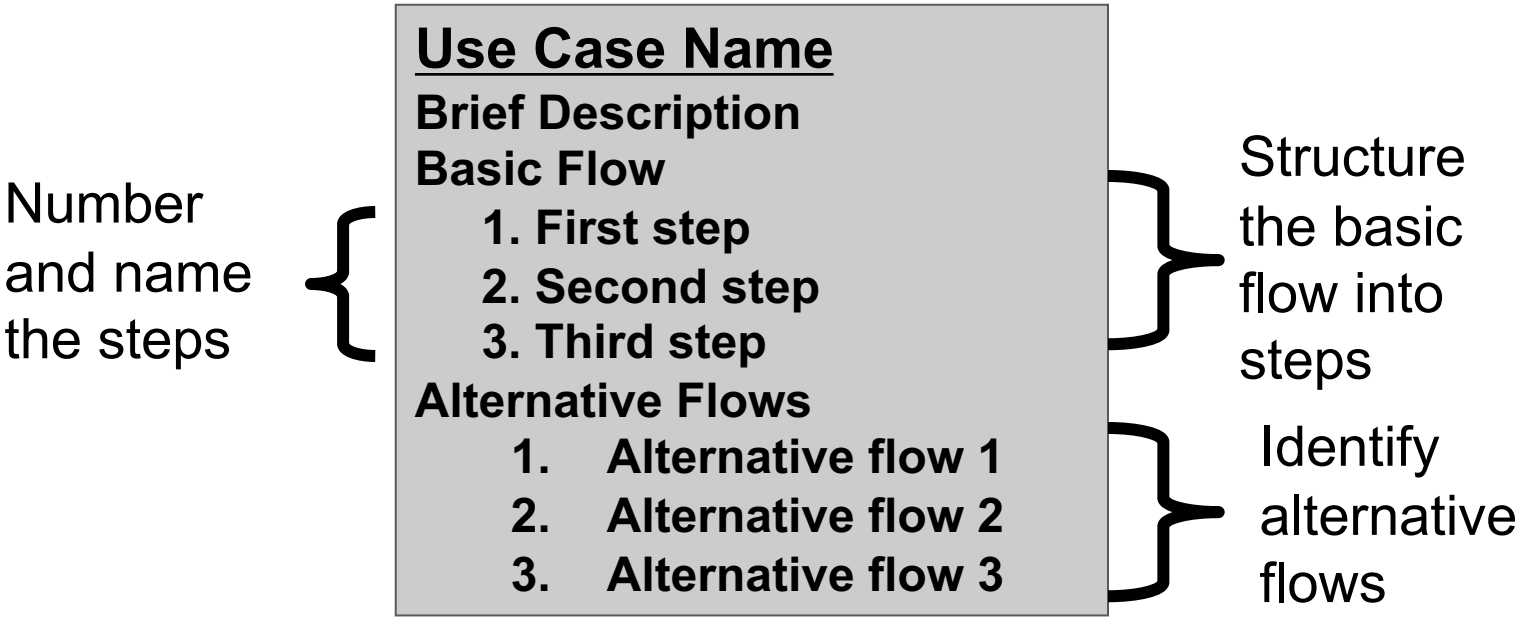
Use-case specification

- 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



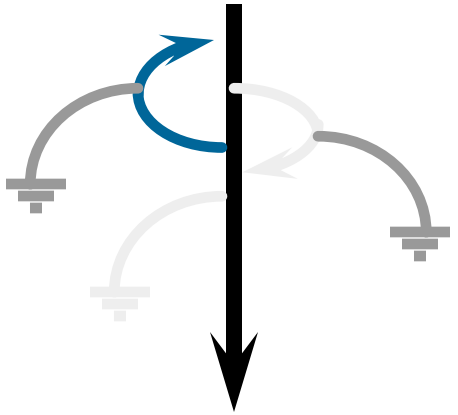
Outline each use case

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



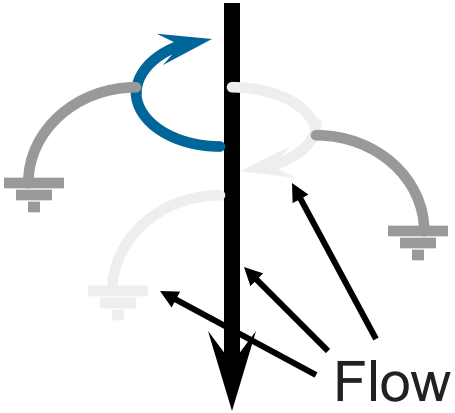
Flows of events (basic and alternative)

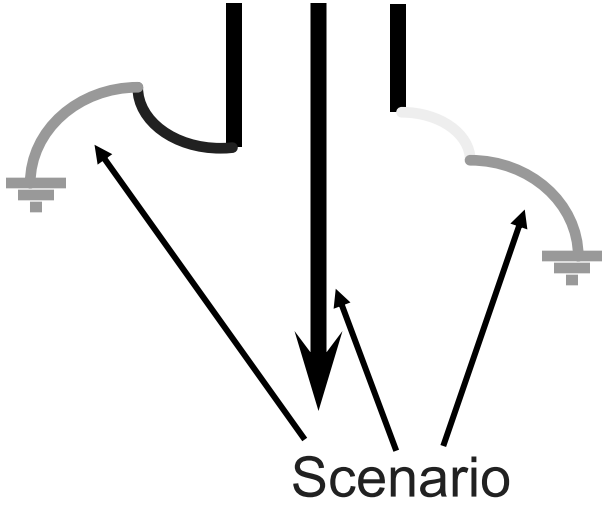
- A flow is a sequence of steps
- One basic flow
  - Successful scenario from start to finish
- Many alternative flows
  - Regular variants
  - Odd cases
  - Exceptional (error) flows



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





Note: This diagram illustrates only some of the possible scenarios based on the flows.

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

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

- ✓Not detailed in this course, e.g.:
  - ✓Structured language
  - ✓Formal methods

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” ?

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 organised so that user errors are minimised.

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.

Requirements Elicitation Techniques

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

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?

Object Oriented Design

Lecture 4

Ruzanna Chitchyan, Jon Bird, Pete Bennett
TAs: Alex Elwood, Alex Cockrean, Casper Wang

Overview

- Why would we do OO design?
- Class Diagrams
  - Associations: Composition and Aggregation
  - Generalisation: Inheritance
  - Navigability
- Modelling behaviour
  - Communication diagrams
  - Sequence Diagrams

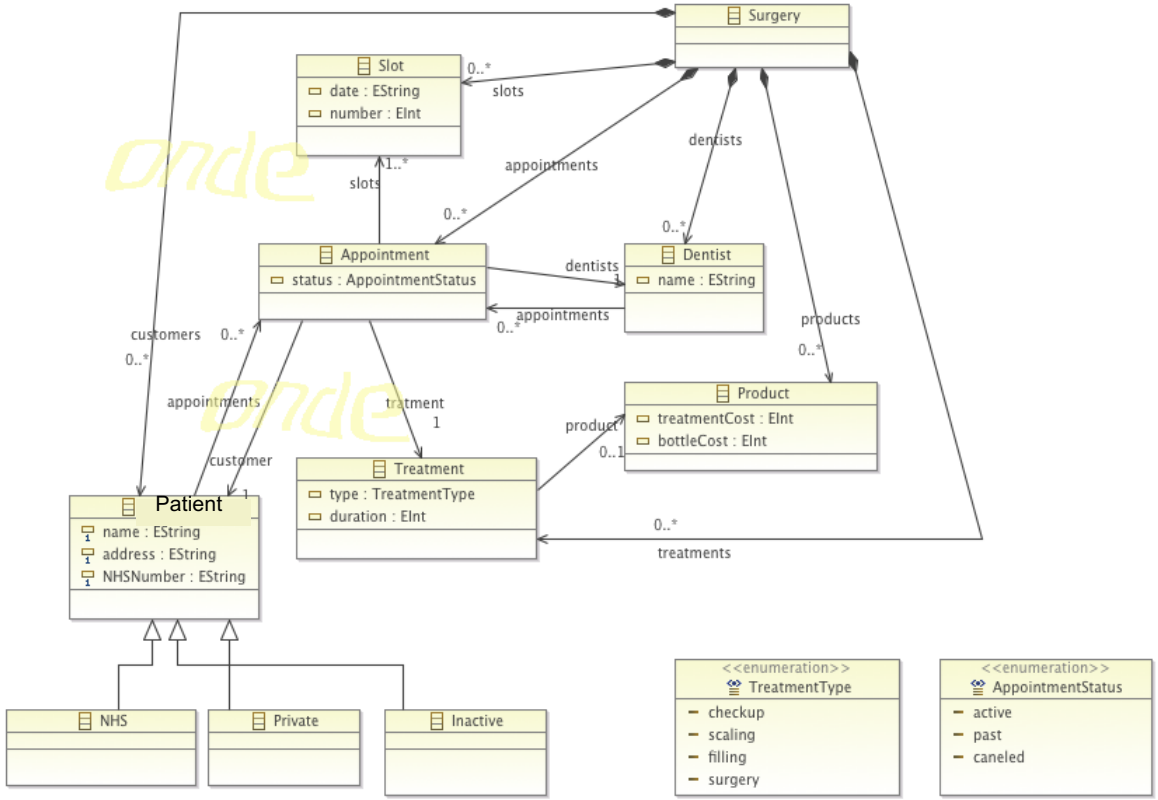
Why OO Design?

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

Class Diagrams

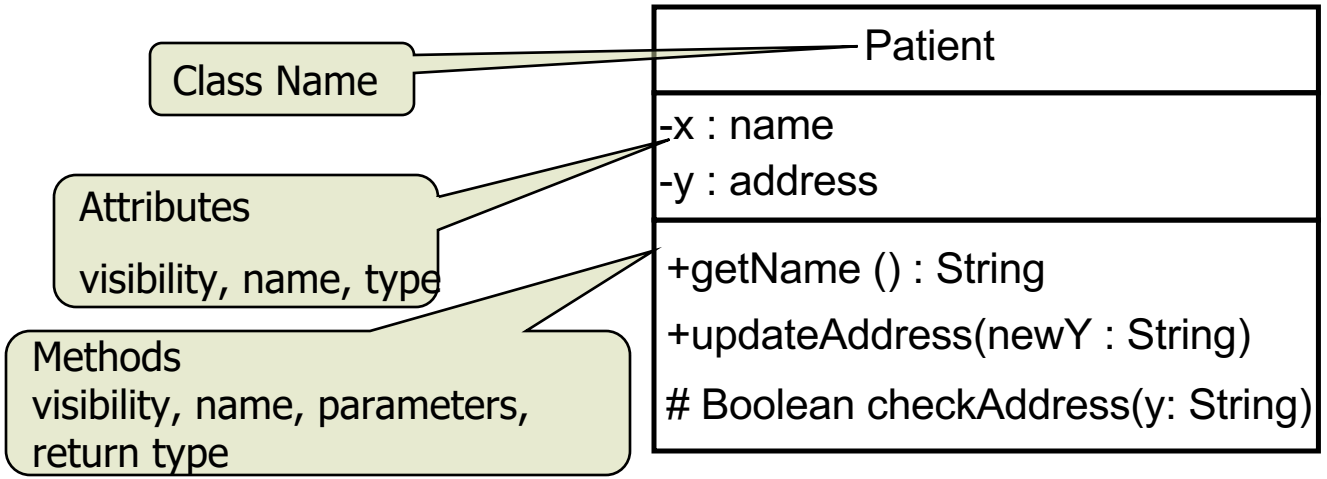
What Is a Class Diagram?

- 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

Grammatical Parse: Dental Surgery Example

You are responsible for development of a software system for keeping track of the appointments and services of a dental surgery. This business employs several dentists, provides treatments to NHS and non NHS patients, and allows for patients to buy products (e.g., toothbrush, paste, etc.) when they pay for the received services (such as periodontal therapy with plague removal and scaling, and dental surgery).

Grammatical Parse: Dental Surgery Example

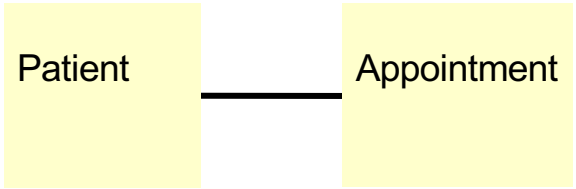
You are responsible for development of a software system for keeping track of the appointments and services of a dental surgery. This business employs several dentists, provides treatments to NHS and non NHS patients, and allows for patients to buy products (e.g., toothbrush, paste, etc.) when they pay for the received services (such as periodontal therapy with plague removal and scaling, and dental surgery).

Structural Relationships in Class Diagrams

What Is an Association?

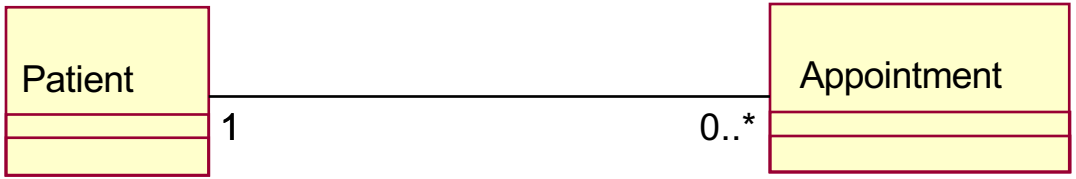
- The semantic relationship between two or more classifiers that specifies connections among their instances.

- A structural relationship specifying that objects of one thing are connected to objects of another thing.



What Is Multiplicity?

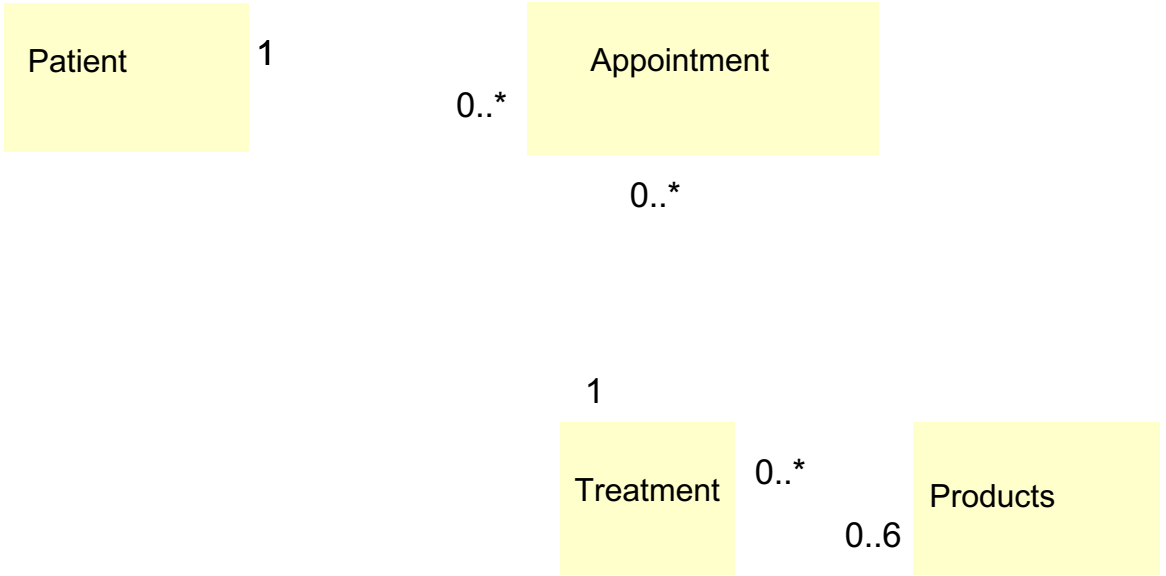
- Multiplicity is the number of instances one class relates to ONE instance of another class.
- For each association, there are two multiplicity decisions to make, one for each end of the association.
    - For each instance of Patient, many or no Appointments can be made.
    - For each instance of Appointment, there will be one Patient to see.



Multiplicity Indicators

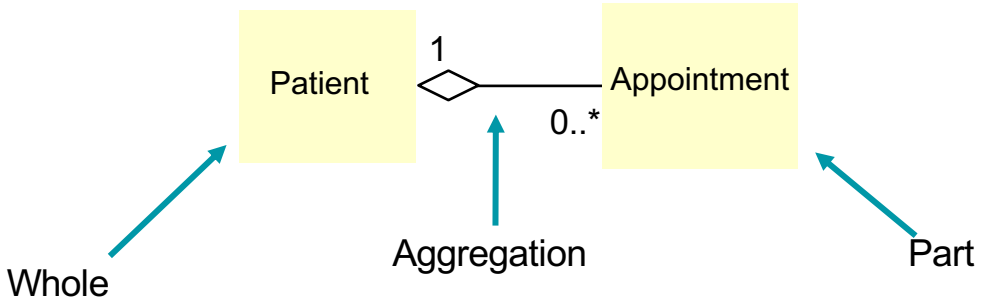
Unspecified	
Exactly One	1
Zero or More	0..*
Zero or More	*
One or More	1..*
Zero or One (optional value)	0..1
Specified Range	2..4
Multiple, Disjoint Ranges	2, 4..6

Example: Multiplicity



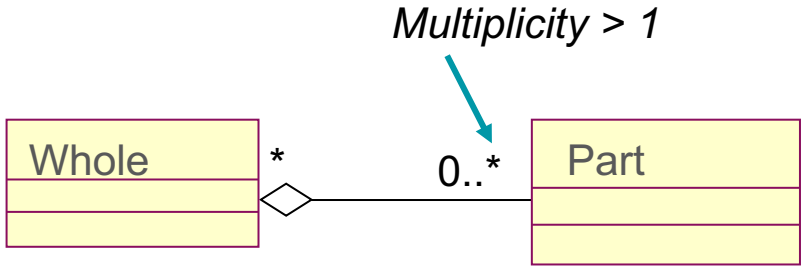
What Is an Aggregation?

- 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.

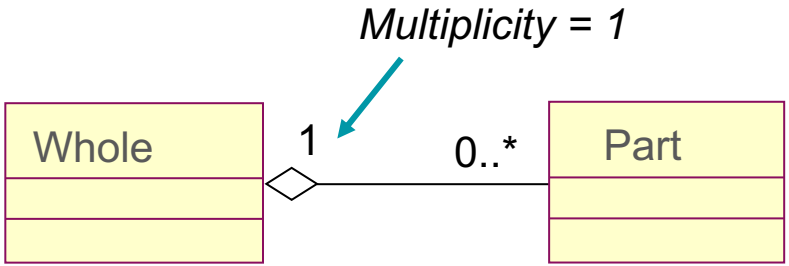


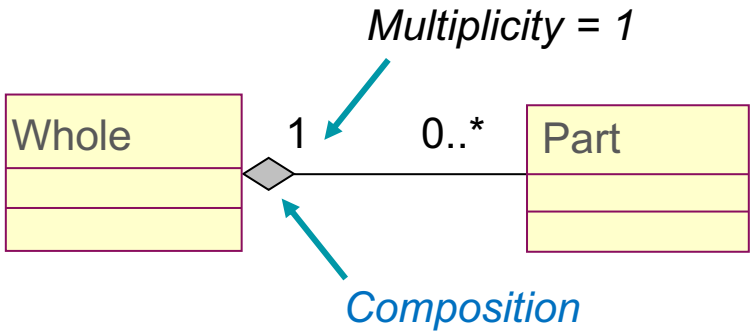
Aggregation: Shared vs. Non-shared

- Shared Aggregation



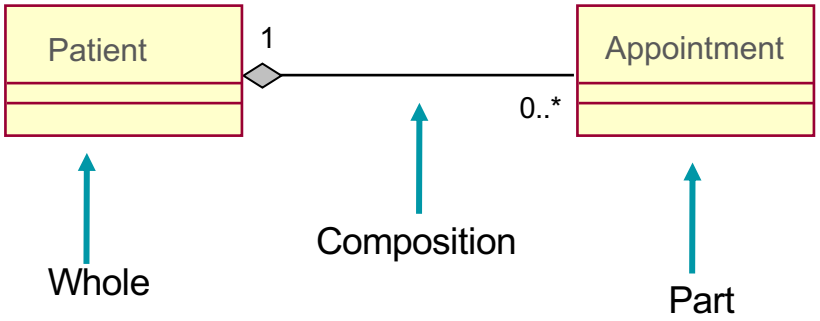
- Non-shared Aggregation





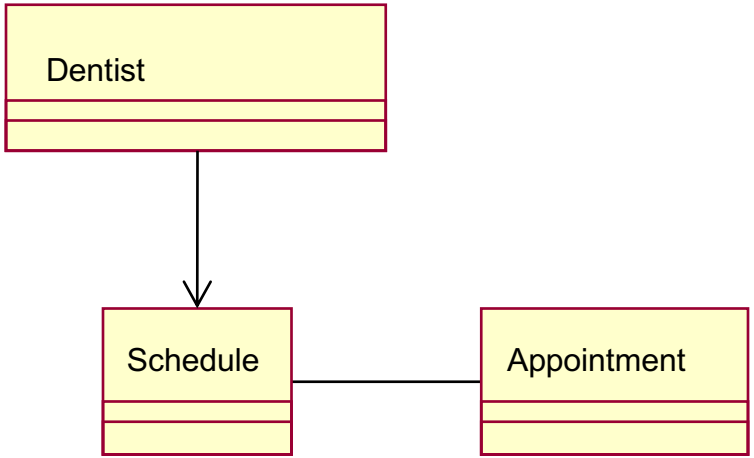
What Is Composition?

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



What Is Navigability?

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



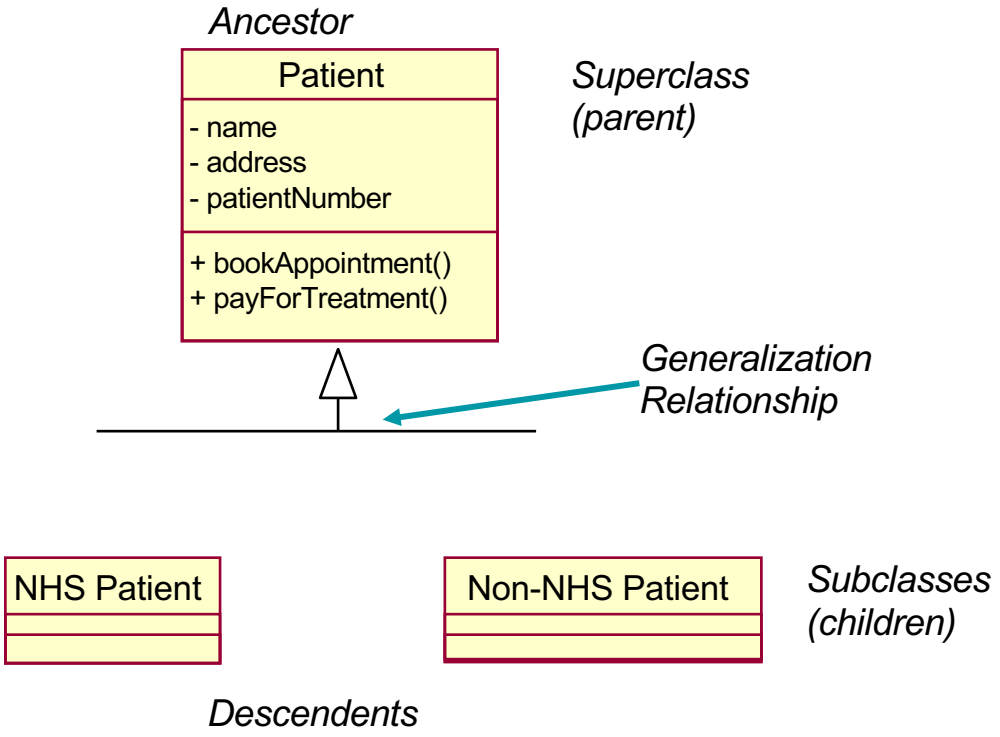
What Is Generalization?

- 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.

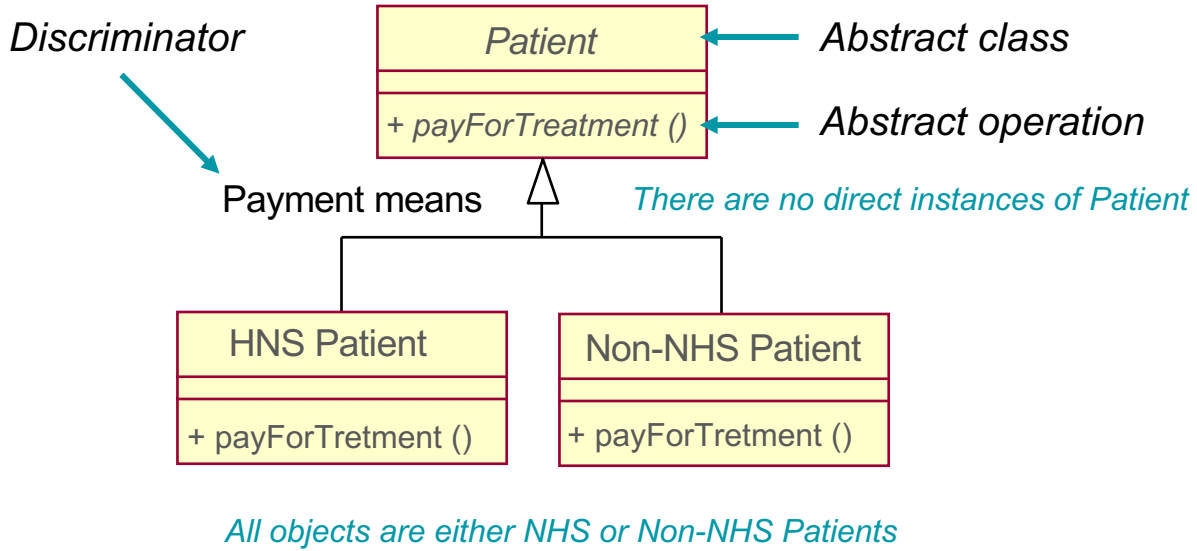
Example: Inheritance

- One class inherits from another
- Follows the “is a” style of programming
- Class substitutability



Abstract and Concrete Classes

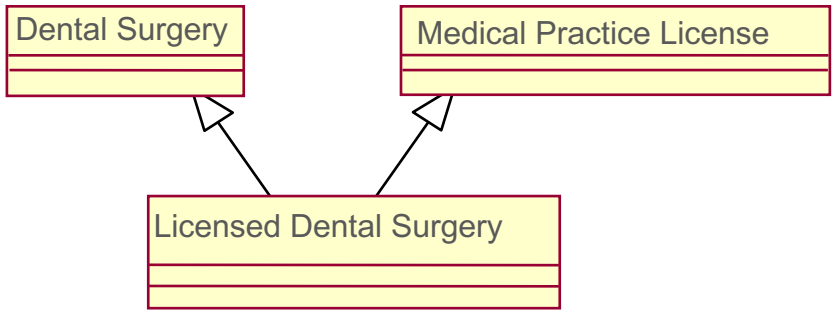
- Abstract classes cannot have any objects
- Concrete classes can have objects



Generalization vs. Aggregation

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

Is this correct?



Behaviour Modelling

Objects Need to Collaborate

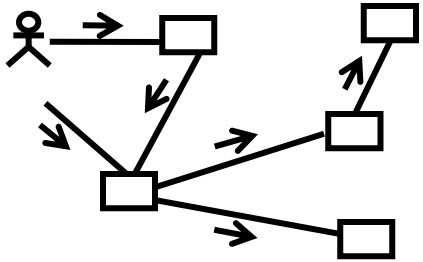
- Objects are useless unless they can collaborate to solve a problem.
  - Each object is responsible for its own behavior and status.
  - No one object can carry out every responsibility on its own.

- How do objects interact with each other?
  - They interact through messages.
  - Message shows how one object asks another object to perform some activity.

Communication Diagrams

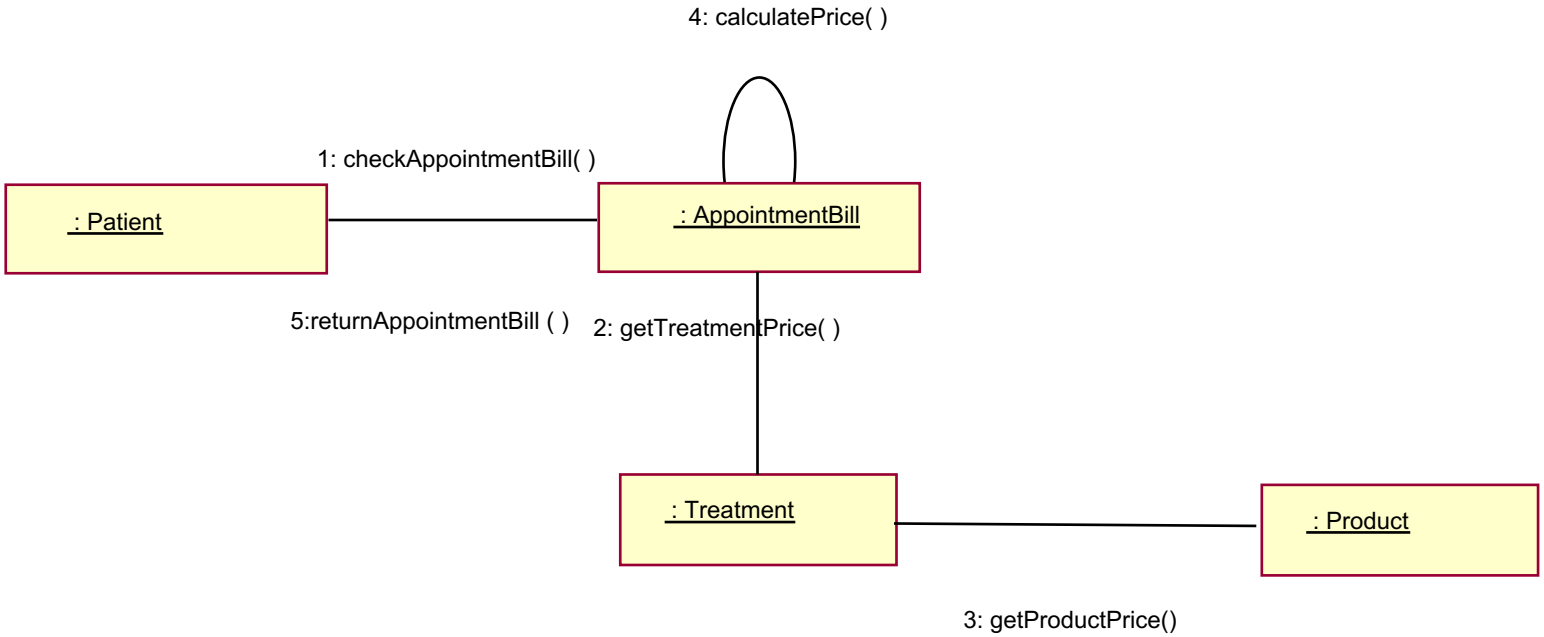
What Is a Communication Diagram?

- A communication diagram emphasizes the organisation 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.



Communication Diagrams

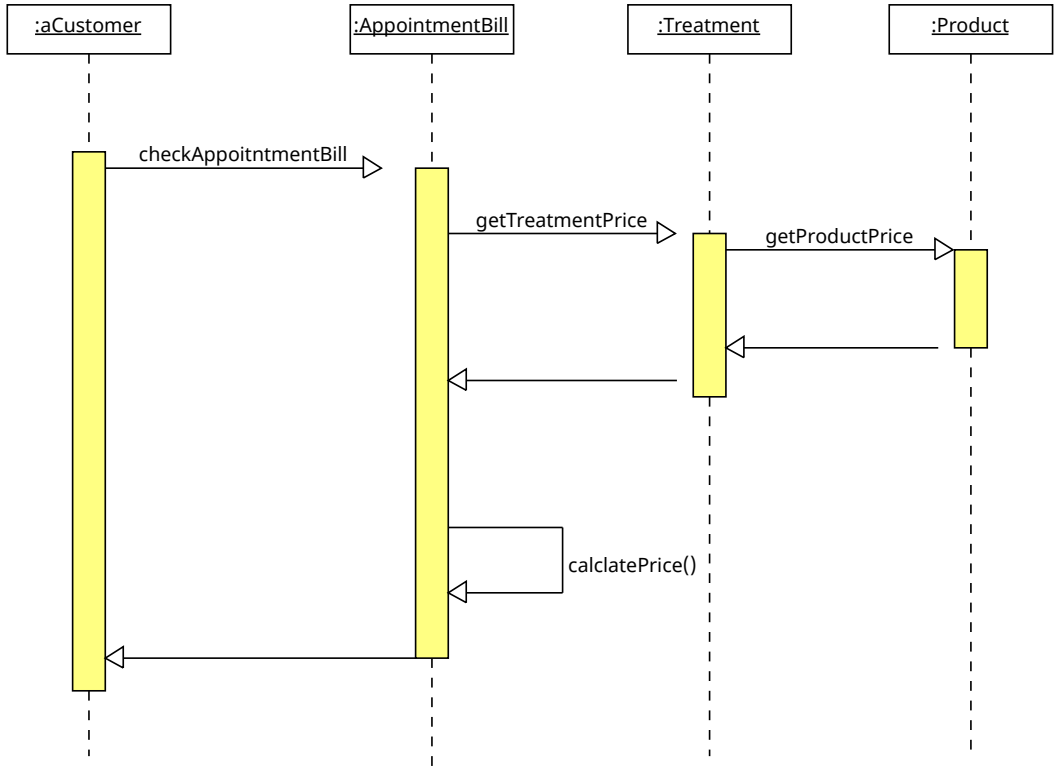
Example: Communication Diagram



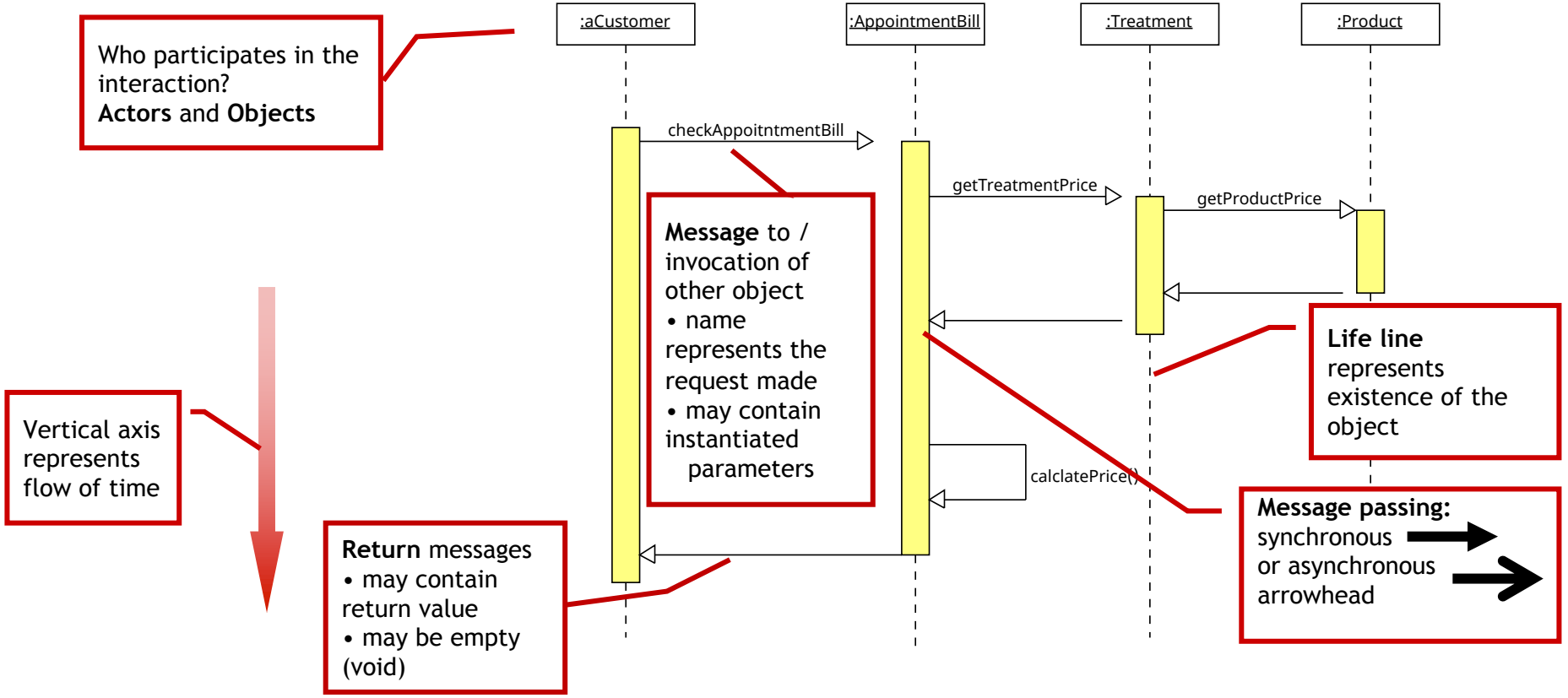
Sequence Diagrams

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?

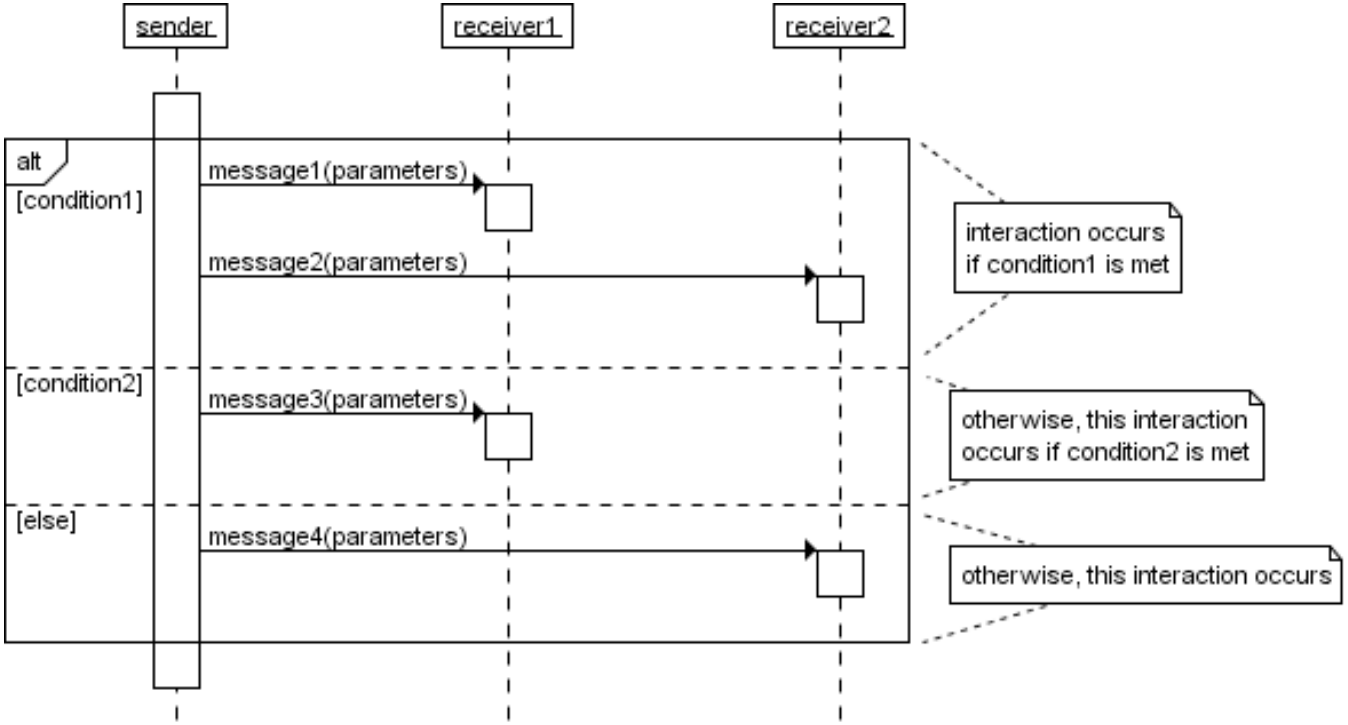
    ➔ label of message arrow
  - Who is to trigger the next step?

    ➔ return message or pass on control flow

Sequence Diagrams

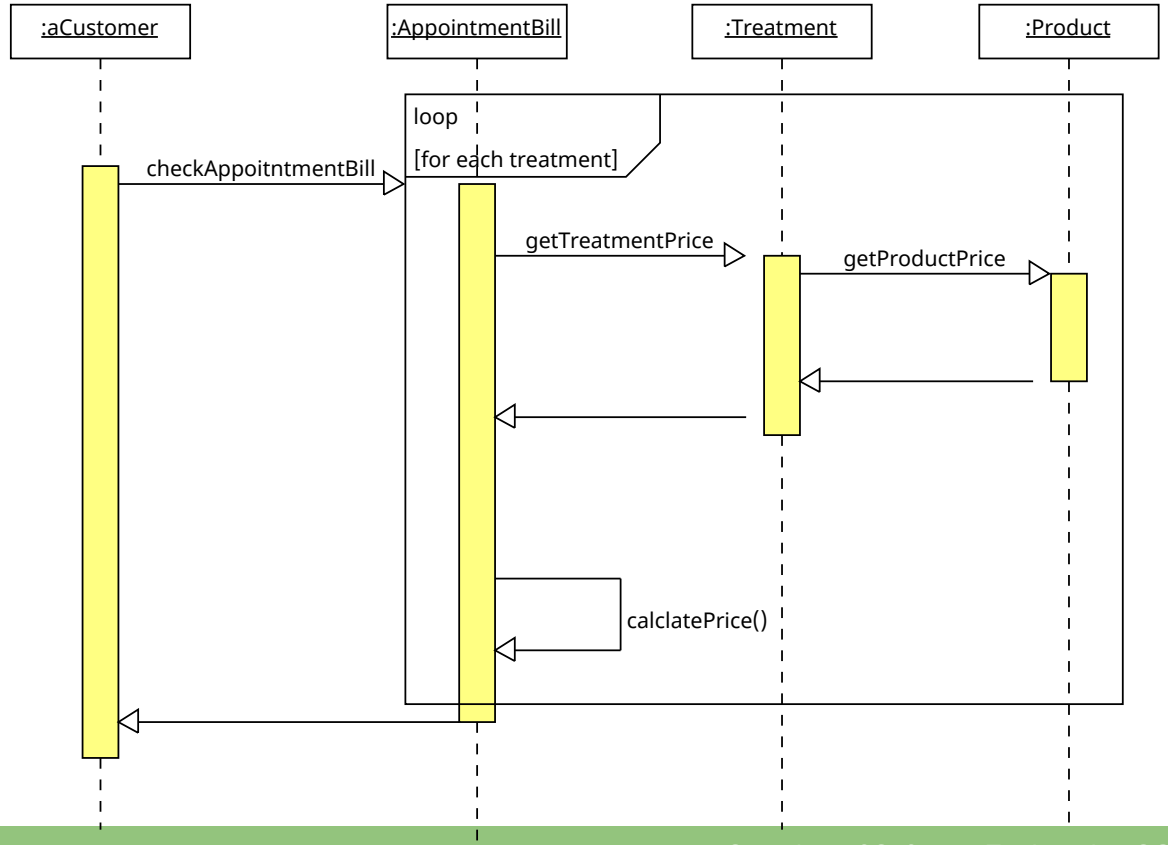
- Sequence Diagrams can model simple sequential flow, branching, iteration, recursion and concurrency
- They may specify different scenarios/runs
    - Primary
    - Variant
    - Exceptions

Interaction frames: alt



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

Interaction frames: loop



Comparison: Communication and Sequence Diagrams

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

Sequence and Communication Diagram Differences

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

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?