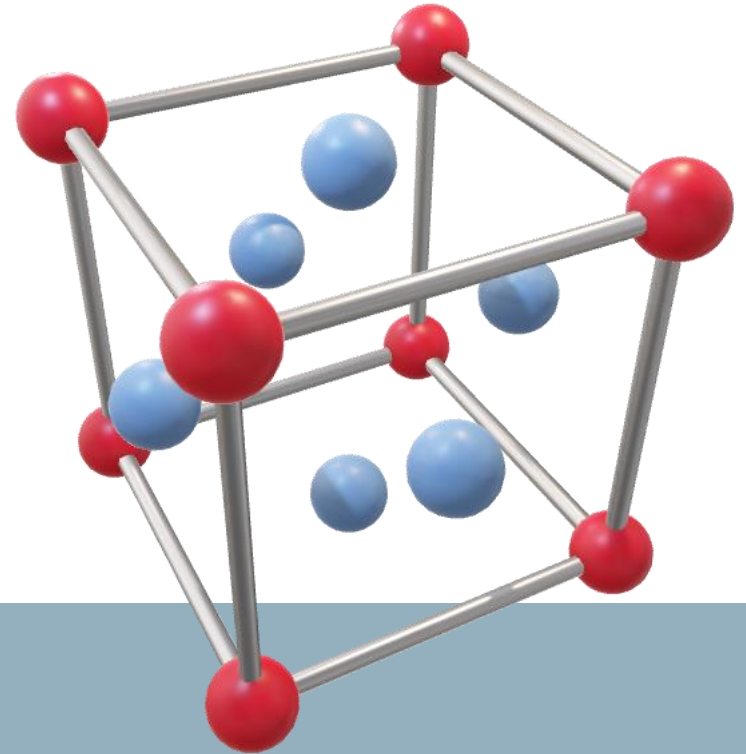


COMP2100/6442

Software Design Methodologies / Software Construction

UML & Use Cases



Sergio J. Rodríguez Méndez

Outline

- > OOAD & Visual Modelling
- > UML: Evolution, purpose, brief, diagrams
- > Modelling the system's behaviour
- > Use Cases
 - >> Main concepts, What?, Benefits
 - >> Actors, Use Case, Diagrams, Documentation
 - >> Event Flow, Audience, Structure, Scenarios
- > Functional and Non-Functional Requirements
- > (Complementary reading)

Object-Oriented Analysis and Design (OOAD)

- > Ensures that the purpose and requirements of a system are thoroughly captured and documented before the system is built.
- > Allows a detailed system model to be developed based on user requirements.
- > Provides abstraction from the underlying complexity of the system and allows the system to be viewed as a whole.

Visual Modelling

- > Models are usually represented visually by some type of notation.
- > The notation takes the form of graphical symbols and connections.
- > A graphical notation:
 - >> facilitates portraying a structure of a complex system.
 - >> provides consistency throughout the development process.

Visual Modelling

> A good notation should:

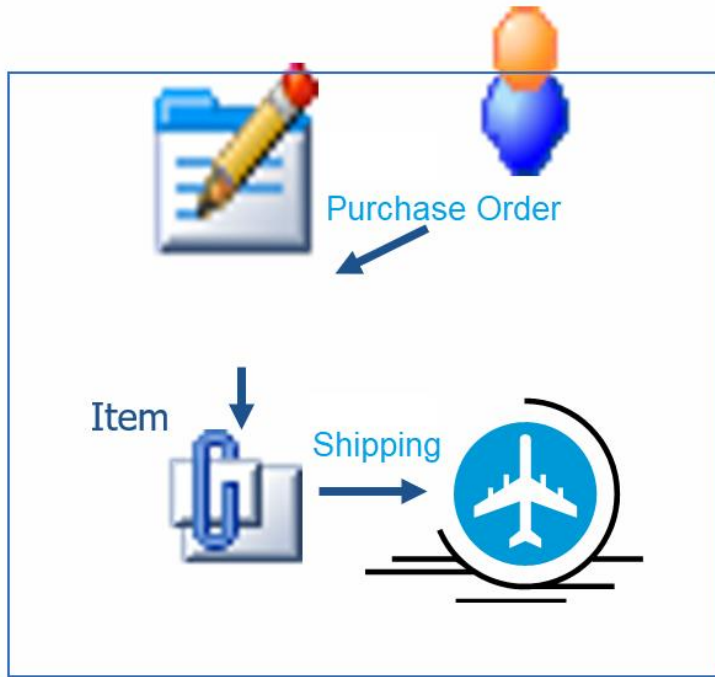
>> allow an accurate description of the system it represents.

>> be as simple as possible, without being oversimplified.

>> be easy to update and communicate with others.

> UML is a robust notation that is used to build OOAD models.

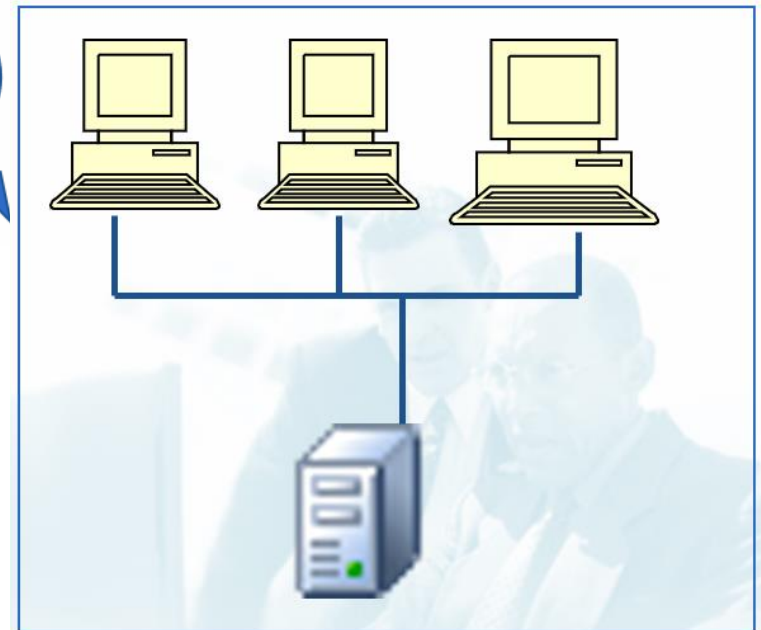
Visual Modelling



Business Process



> “Visual modelling captures the system’s essential parts”
— Dr James Rumbaugh

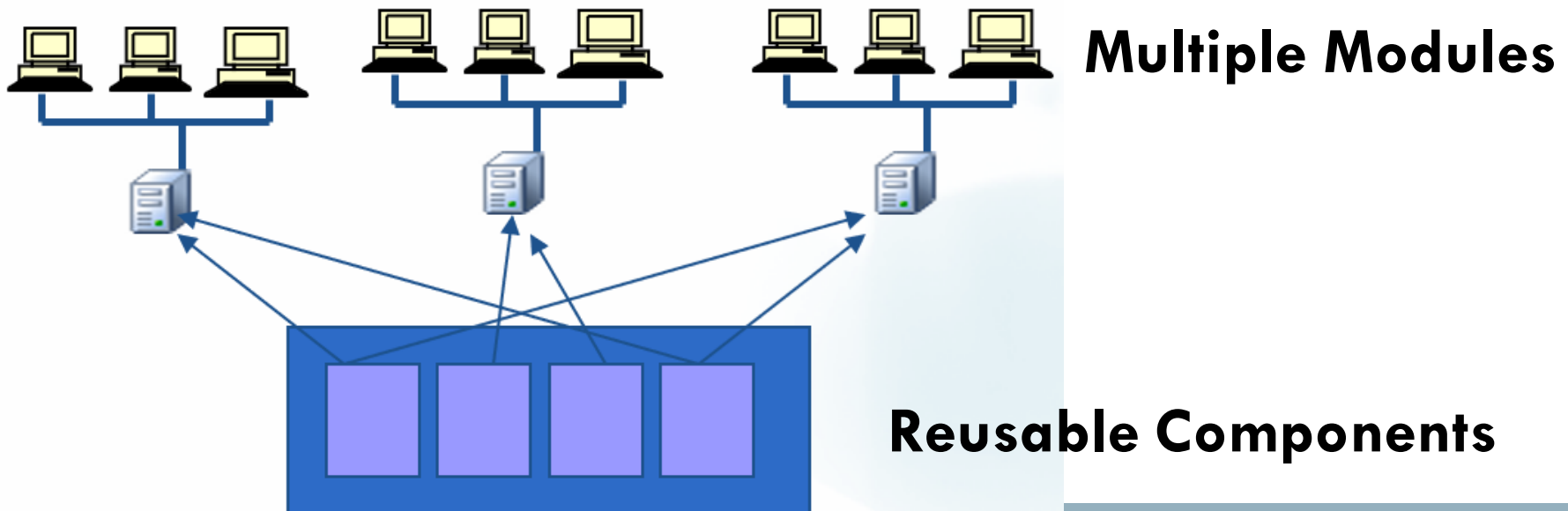


Information System

> Visual modelling is to model using standardised graphical notations.

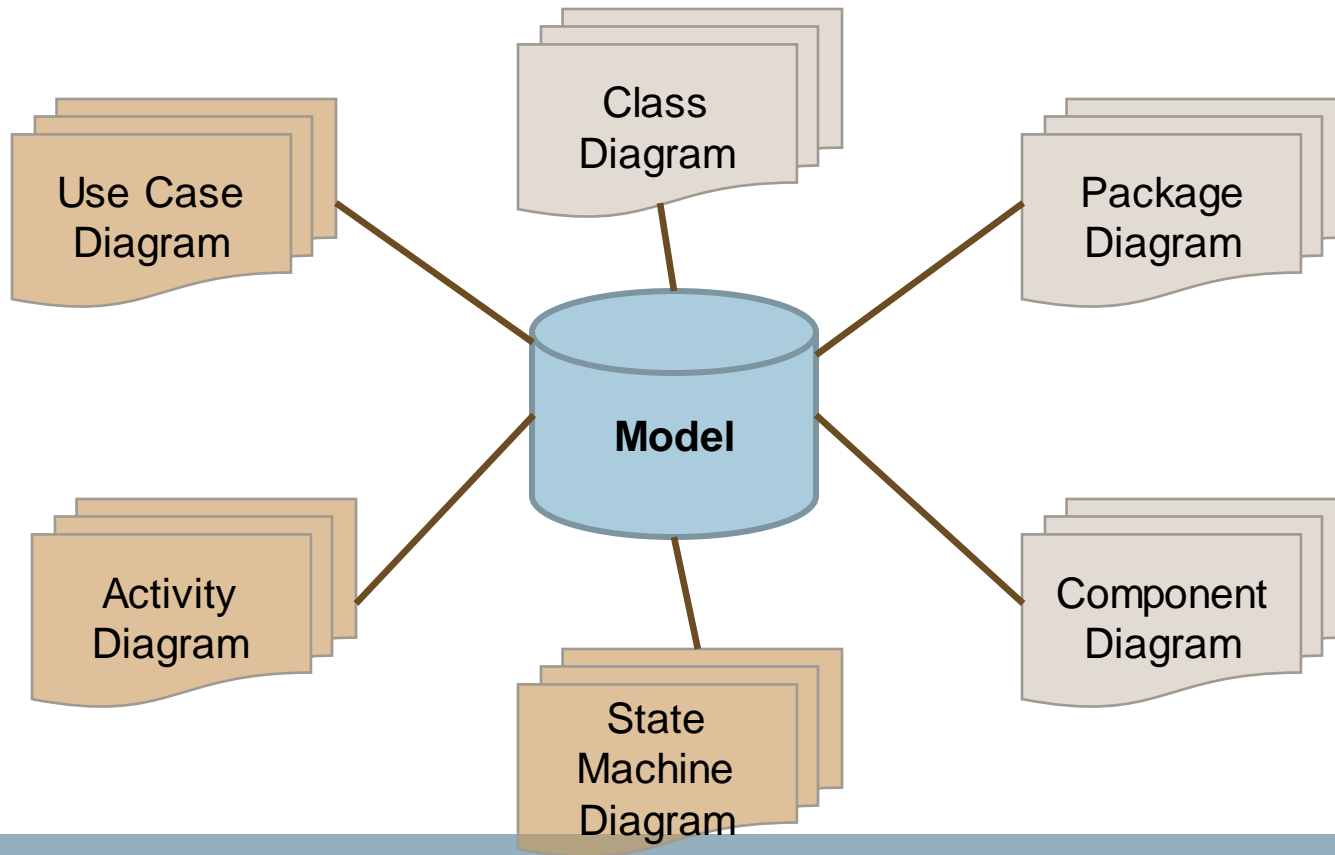
Visual Modelling (VM)

- > VM is a *communication tool* used to analyse and design a system: **It visually abstracts the complexity of a system.**
- > VM helps capture different aspects of reality (objective and subjective), such as business processes.
- > The system's analysis must be done from different frames of reference.
- > A system's model is programming-language agnostic.
- > VM helps to establish the system's architecture (skeleton) by promoting **reusable components**.



Visual Modelling

- > Modelling a system from several different but related perspectives permits it to be understood for different purposes.
- > The **diagrams** are different **views** of a model.
- > The diagrams depict relevant/essential aspects.



OOAD & VM & UML

- > UML is a robust notation that is used to build OOAD models.
- > UML consists of a series of diagrams that represent the different views of a software system in analysis and design.
- > It is important to accurately record all the diagrams developed during the system's analysis and design.
- > This ensures consistency in the model.

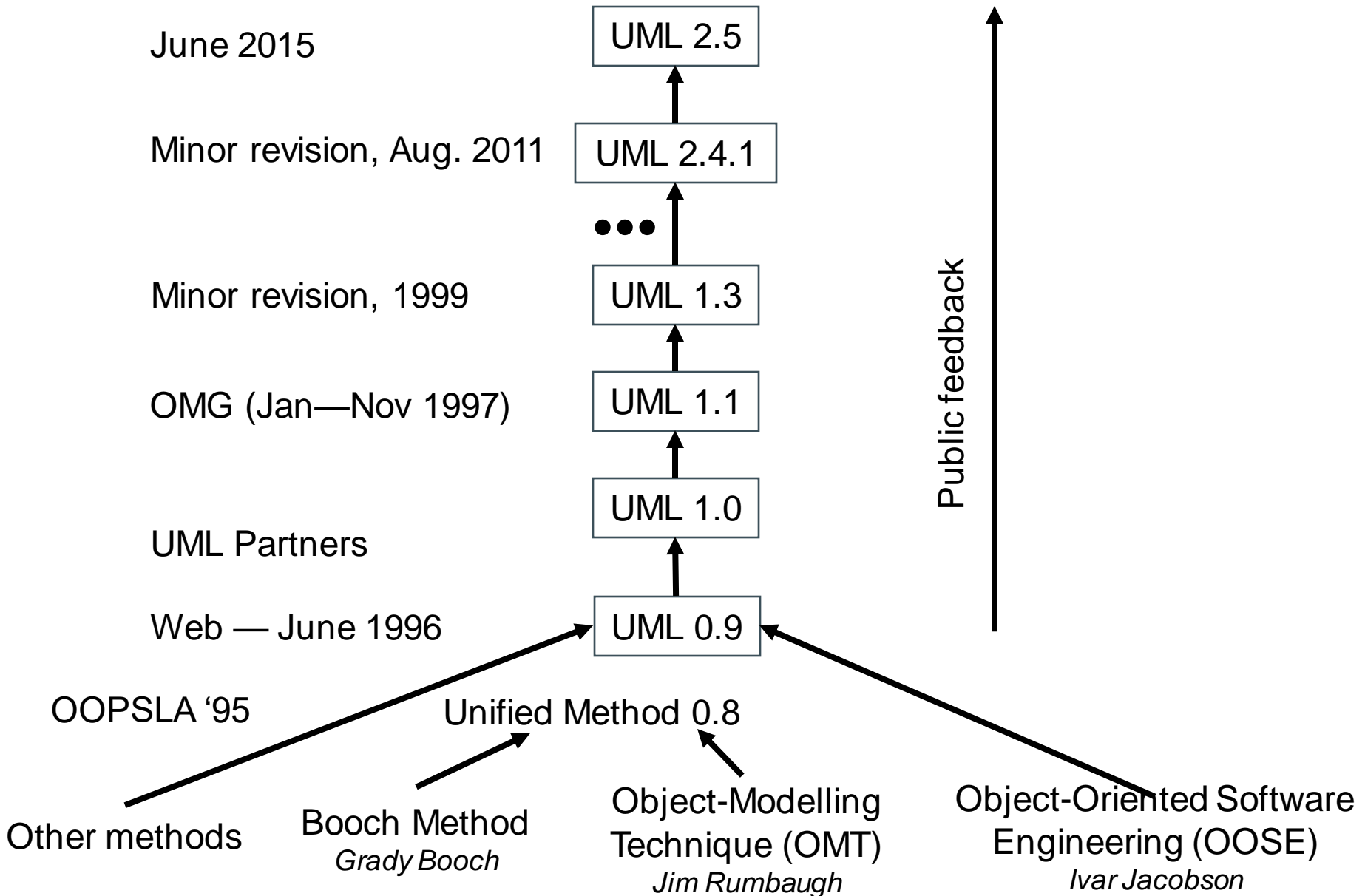
UML: *Unified Modelling Language*

- > It combines the best characteristics:
 - >> Data modelling concepts (Entity-Relationship, Relational).
 - >> Business modelling (Workflow).
 - >> Object modelling.
 - >> Component modelling.
- > UML is the standard language for **visualising, specifying, building, and documenting** *artefacts* of software systems.
- > UML provides a shared vocabulary for conveying the *software OOAD*.
- > UML audience: all the project stakeholders and participants, from business analysts to software architects and developers.
- > UML is not a programming language. It is recognised as a **modelling language**, not a methodology or method.

UML: *Unified Modelling Language*

- > UML emerged in the 1990s following the “method wars”.
- > **Unified** — it was initially conceived by Rational Software Corp. and three of the most prominent methodologists in the formation Systems and technology industry, Booch, Rumbaugh, and Jacobson.
- > **Modelling** — with an emphasis on modelling, it focuses on how we understand the world around us.
- > **Language** — it functions as the means for expressing and communicating knowledge.
- > UML has four distinguishing features in comparison to other modelling languages:
 - >> It is general-purpose
 - >> It is tool supported
 - >> It is broadly applicable
 - >> It is industry standardised

UML Evolution



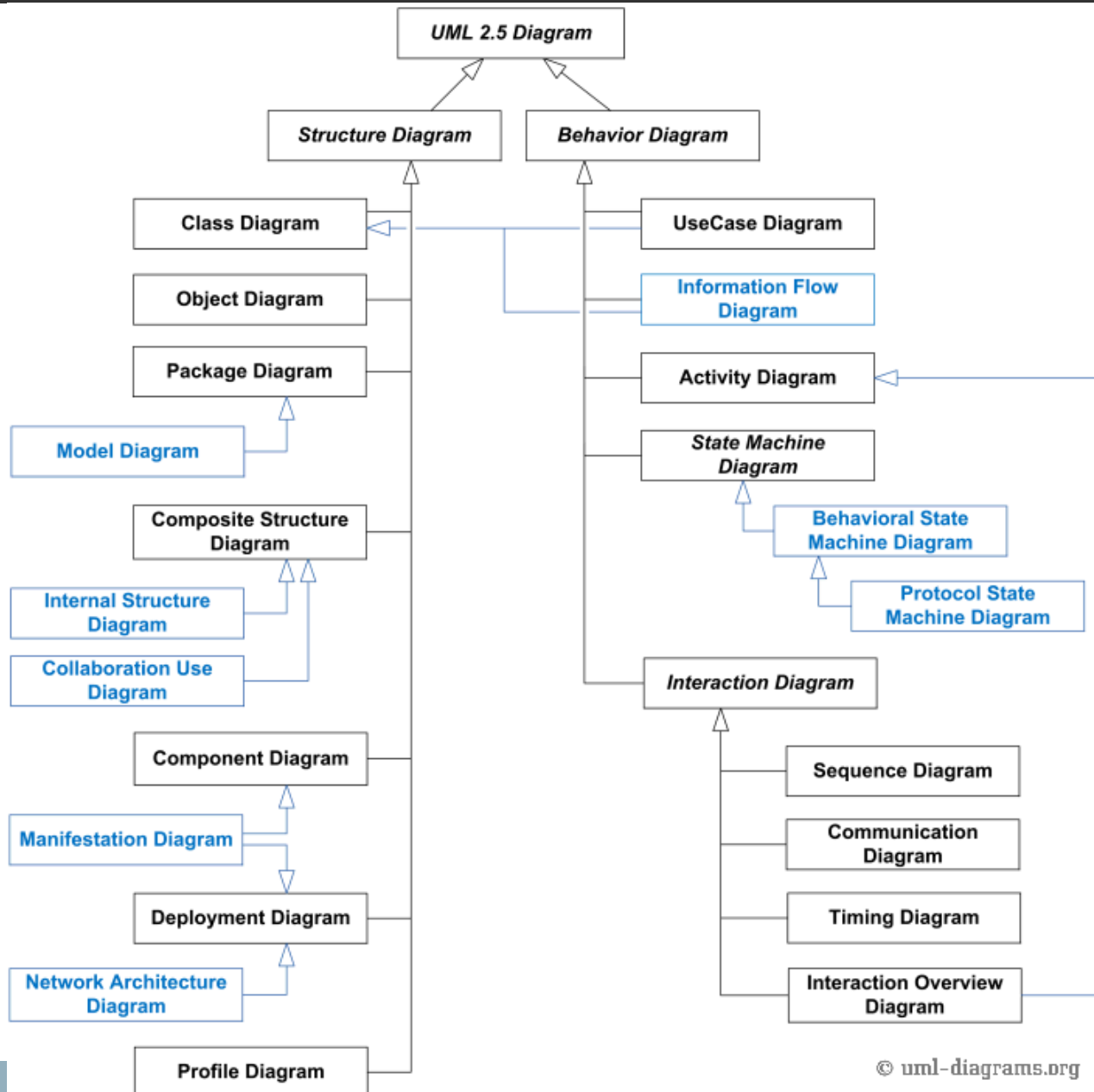
UML Purpose

- > System interaction with the external world: **Use Cases**.
- > System behaviour.
- > System structure.
- > Organisation's architecture.
- > System components.
- > *The heart of an OOAD (solution) is the model construction.*
- > *The model abstracts the problem's essential details by adequately managing the solution's complexity.*
- > UML is used to **understand, design, browse, configure, maintain, and control** *information about systems*.
- > It is intended for use with all development methods, lifecycle stages, application domains and media.

UML Brief

- > It includes semantic concepts, notations, and guidelines.
- > UML is:
 - >> a general-purpose modelling language used by people and machines.
 - >> meant to support good practices for OOAD.
 - >> not a complete development method.
 - >> aimed to be as simple as possible while modelling the full range of the practical system's views (accurately describe conceptual and executable artefacts).
- > UML captures information about a system's static structure and dynamic behaviour.
 - >> The static structure defines the kind of objects important to a system and its implementation, as well as the relationships among the objects.
 - >> The dynamic behaviour defines the history of objects over time and the communications among objects to achieve goals.

UML 2.5 Diagrams



> System's behaviour is:

>> How the *system acts and reacts* when performing its *expected functions* and *interacting* with *external entities*.

>> The set of *externally visible activities* you can *test* (black-box testing).

>> Captured and depicted in the **Use Cases**.

> **Use Cases** describe the system, its environment (context), and the relationships between them from the perspective of an external observer.

> **Use Cases** is a VM modelling technique that captures business processes from the user's perspective, identifying their event flows or "paths" (main/critical path and exceptions).

> The emphasis is on modelling **WHAT** the system does instead of **HOW**.

Use Cases: important concepts

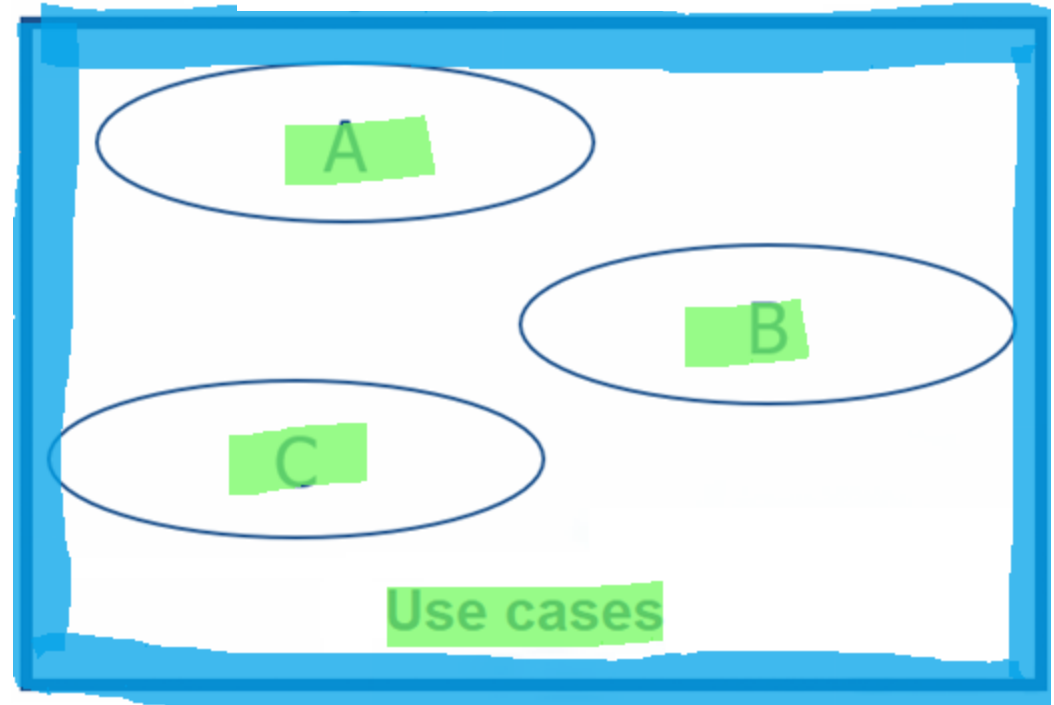
The system boundary is also called "**subject**"

System boundary



Actor

The **(business) actor** is an external entity (someone or something) outside the system that interacts with it.



A **(business) Use Case** is a sequence of *transactions/actions* that a system executes and finishes with an *observable result* (a system's exit) to a *particular actor*.

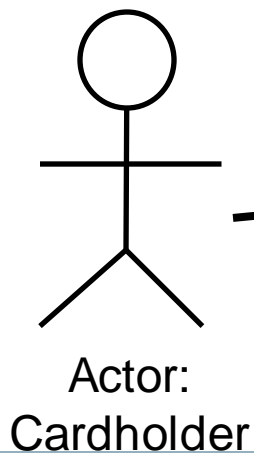
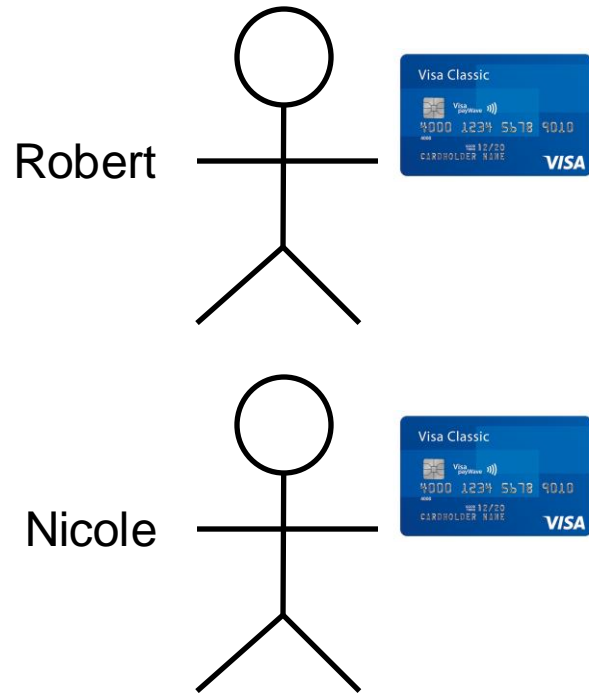
Use Cases: In a nutshell

- > It models the desired/expected system's functions (use cases) and what surrounds them (actors).
- > The use case models are used across the system's life cycle, mainly in the analysis, design, and testing stages.
- > The primary purpose is to convey the system's functionality and behaviour to the client/customer, domain experts, and end users.
- > Provides *acceptance* from the early stages of the system's construction.
- > Identifies the answers to:
 - >> **Who** will interact with the **system**, and **what** should the **system** do?
 - >> What **interfaces** should the **system** have?
- > It is used to verify the following:
 - >> All requirements have been captured.
 - >> All project stakeholders have understood the requirements.

(Business) Actors

- > External entities (human or non-human) that initiate a series of events involved in the use case tasks.
- > Do not form part of the system.
- > Represent roles the end users can “play”.
- > Actively interchanges information with the system.
- > Can be an *active* or *passive* information receptor.
- > Can represent a human, a machine, or another system.
- > How to identify them?
 - >> Who is interested in a particular system requirement?
 - >> Who will use a specific system function?
 - >> What actors do the use cases require? Who instantiates the use cases?
 - >> Where is the system used? Within an organisation? Publicly accessible?
 - >> Who will input/use/delete information from the system?
 - >> Who will provide support and maintenance to the system?
 - >> Does the system use an external resource?
 - >> Does an actor play diverse roles? Do various actors exist with the same role?

Instances of actors



Change PIN

Use Case:
Change PIN

Roles



Carlos as **client**

Carlos as **operator**

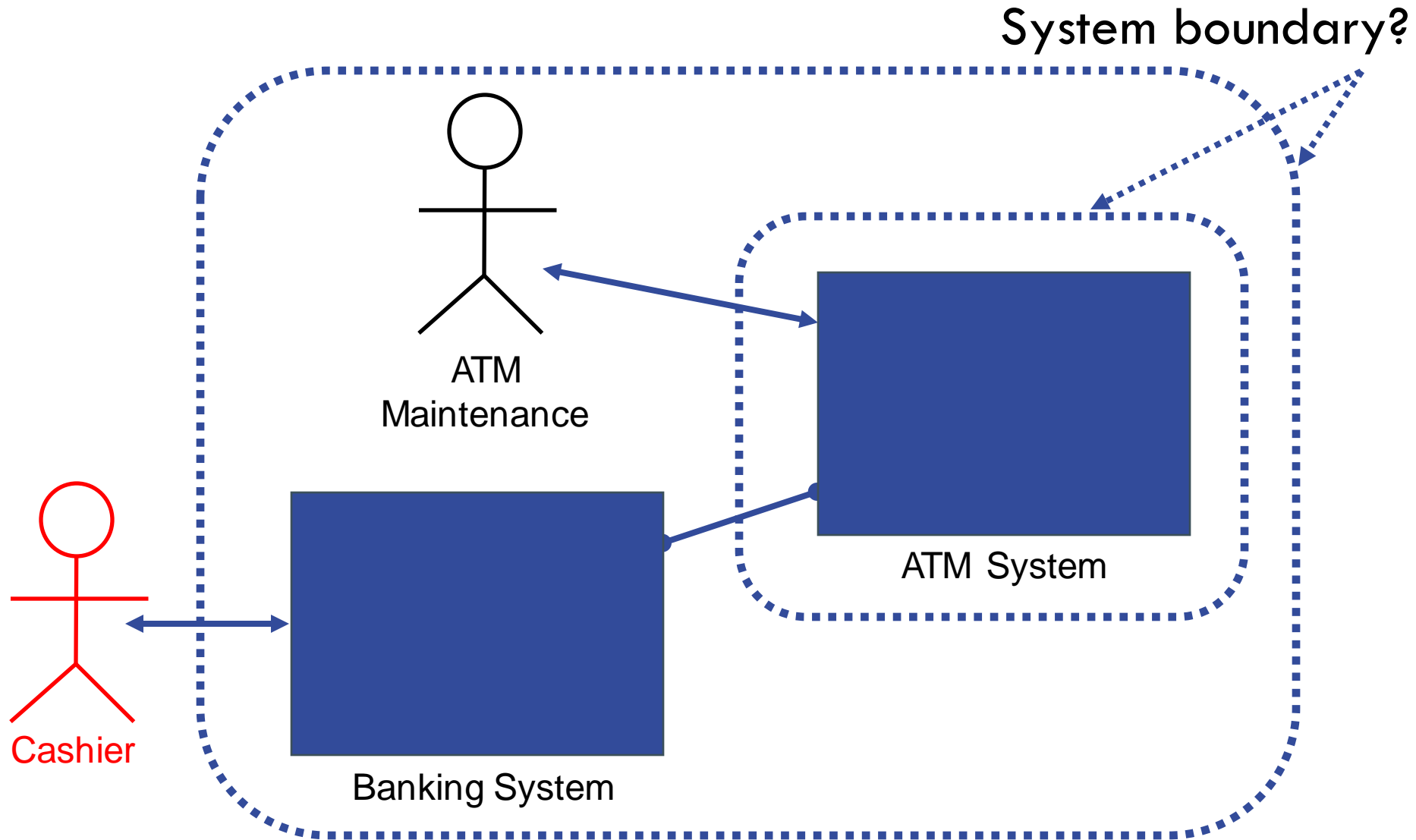
Actor:
Client

Actor:
Operator

ABC

Use Case:
ABC

Actors and system boundary



The actors play based on the system boundary

(Business) Use Cases

- > Model a conversation between actors and the system.
- > Are instantiated/started by an actor when it invokes a specific system's functionality.
- > A use case is a complete and significant event flow in the system.
- > Taken together, the use cases constitute all possible ways of using the system.
- > How to identify them?
 - >> What are the **tasks** for each **actor**?
 - >> Will the **actor** Create/Read/Update/Delete (CRUD) **system**'s information?
 - >> What **use cases** CRUD each **actor's information**?
 - >> Does the **actor** required to notify the **system** about sudden external changes?
 - >> Does the **actor** need to be informed about **specific events** in the **system**?
 - >> Does the **system** provides the organisation with the expected behaviour?
 - >> What **use cases** will provide support and maintenance to the **system**?
 - >> Can the **use cases** execute all the **functional requirements**?
- > Purpose: to **abstract functions** of the **actors**.

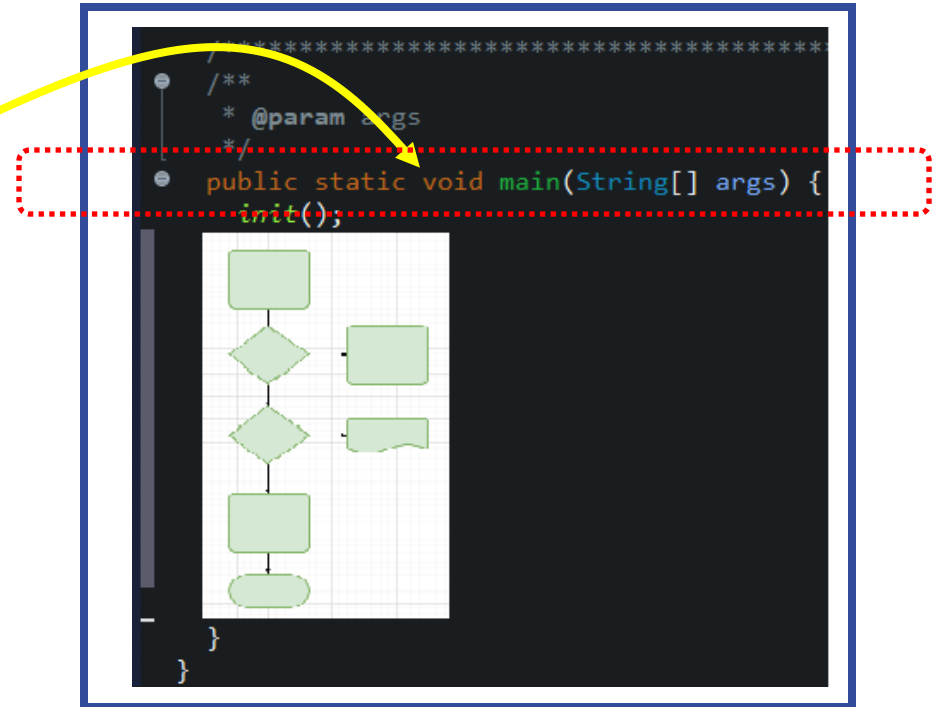
Use Cases: important concepts



Actor

The **(business) actor** is an external entity (someone or something) outside the system that interacts with it.

System boundary

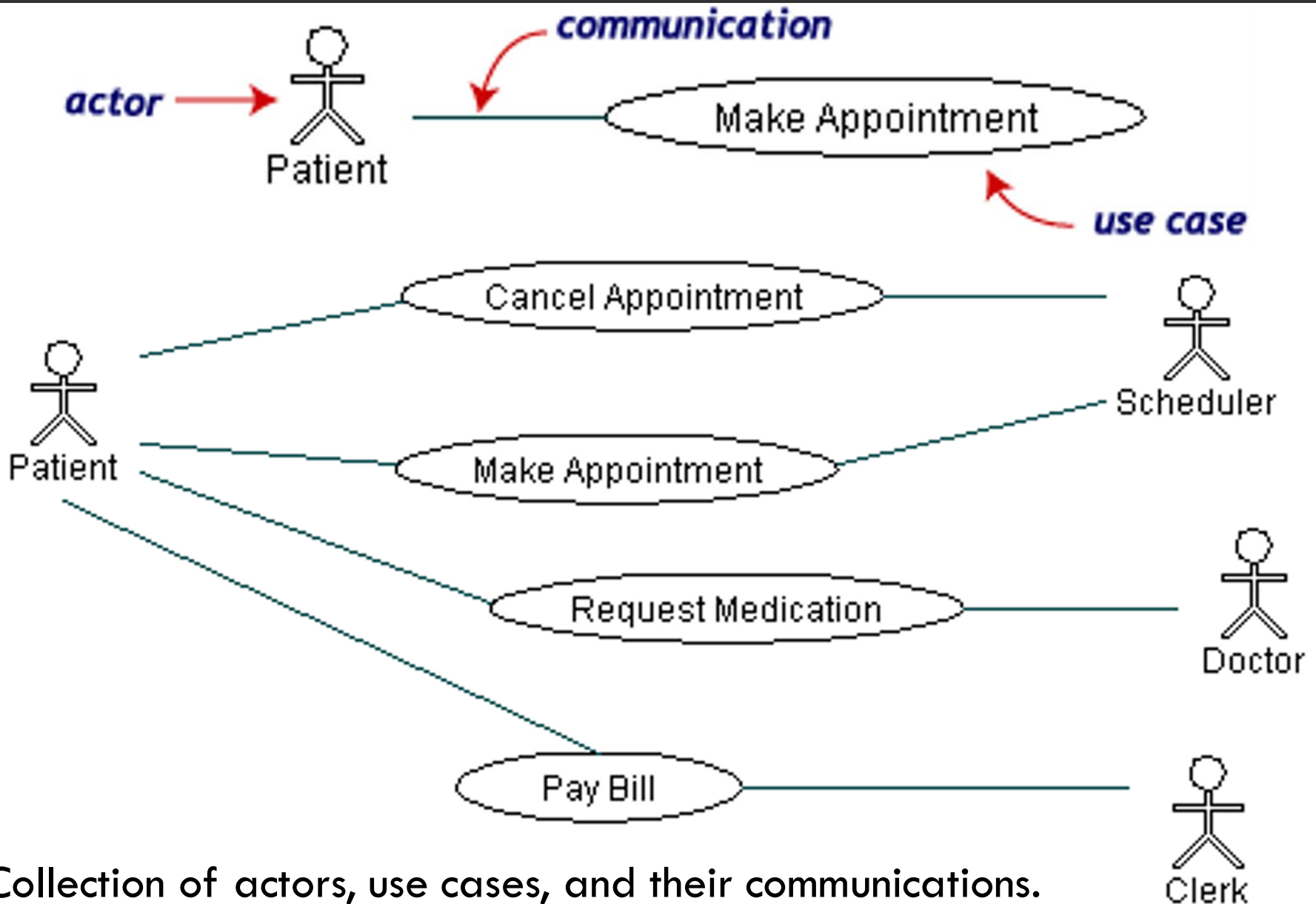


A **(business) Use Case** is a sequence of *transactions/actions* that a system executes and finishes with an *observable result* (a system's exit) to a *particular actor*.

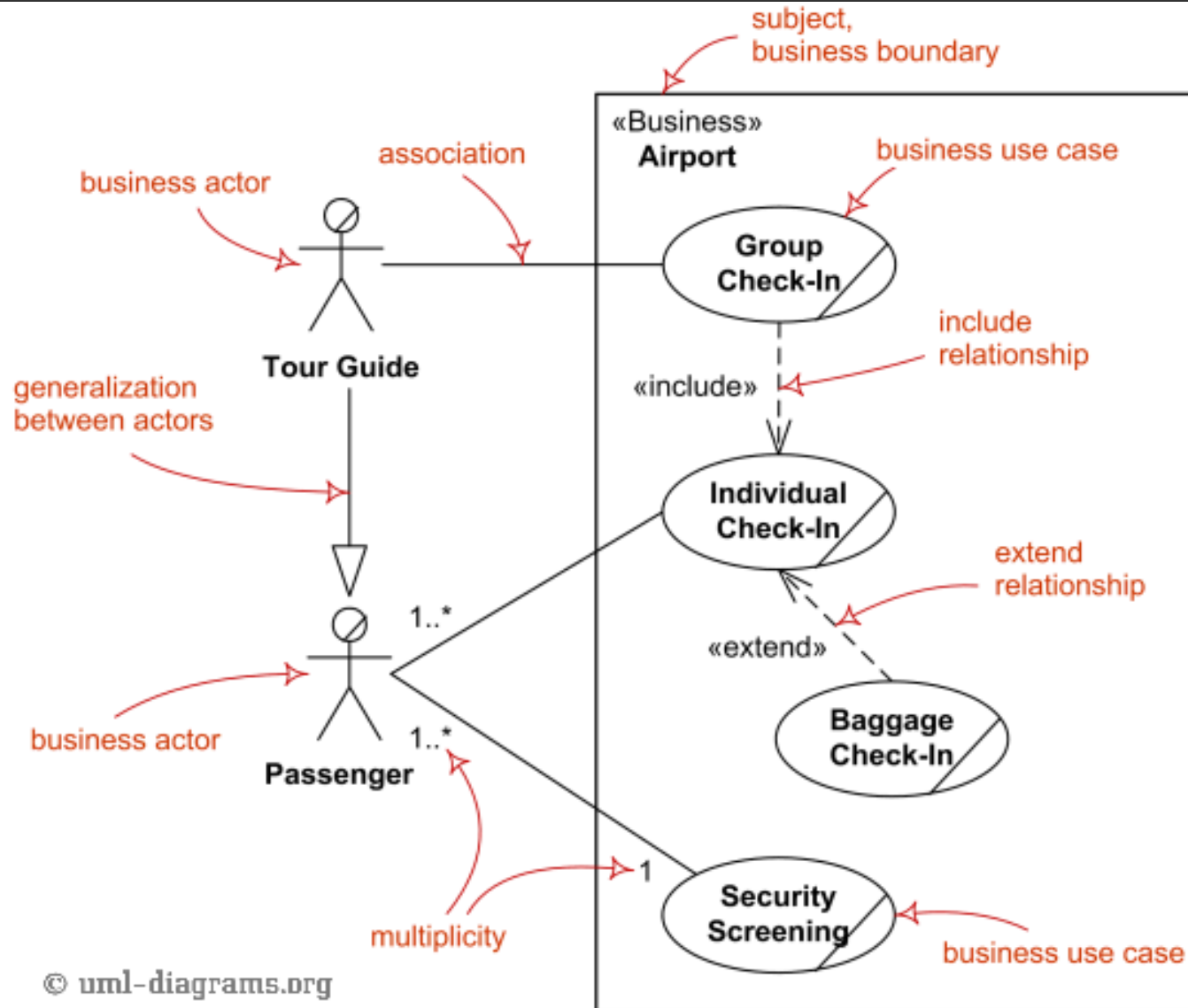
Information sources

- > System specs & problem formulation.
- > Relevant literature to the domain.
- > Interviews with domain experts.
- > Background:
 - >> Team expertise: own domain knowledge.
 - >> Previously developed systems.

Use Cases Diagrams

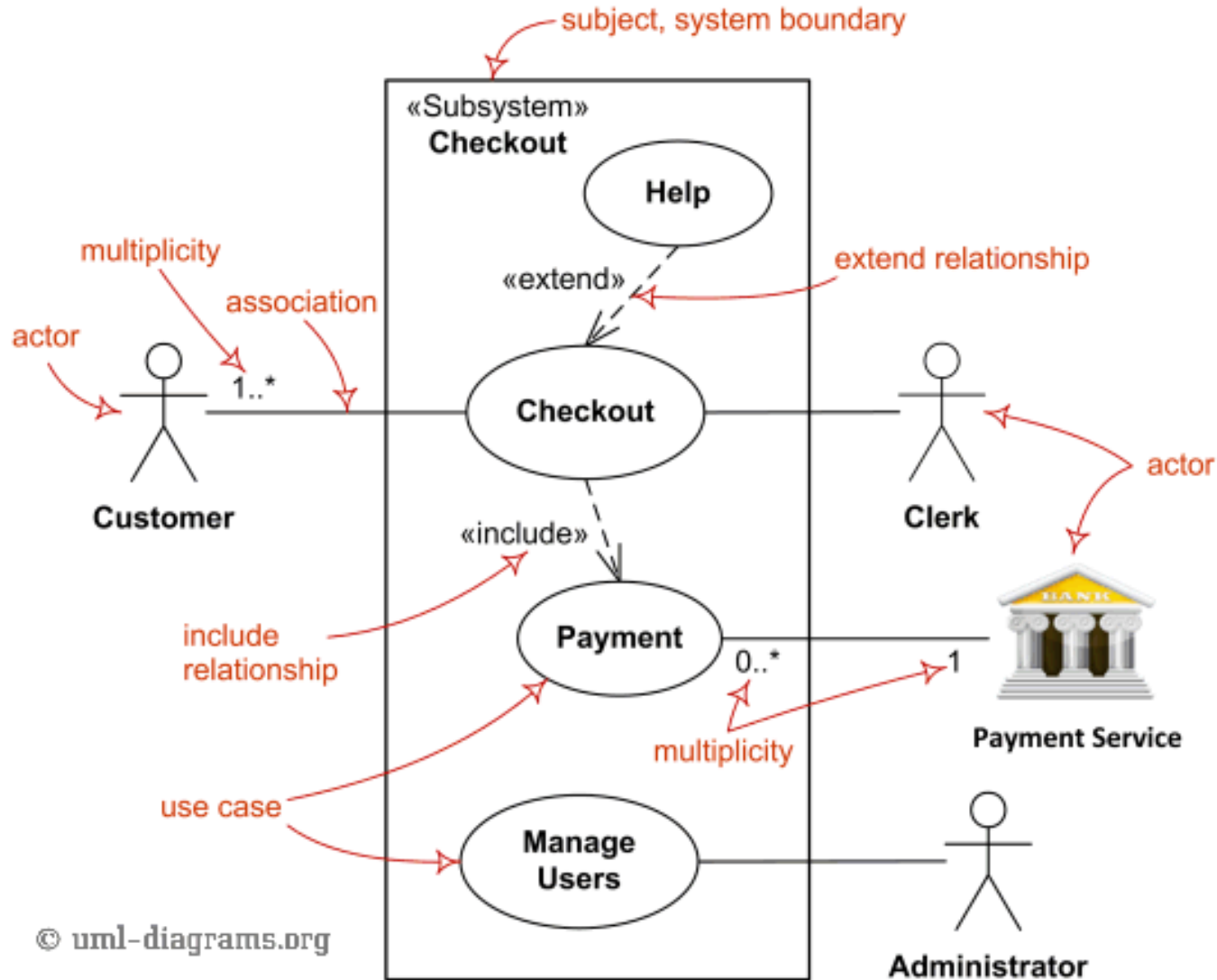


Business Use Cases Diagrams



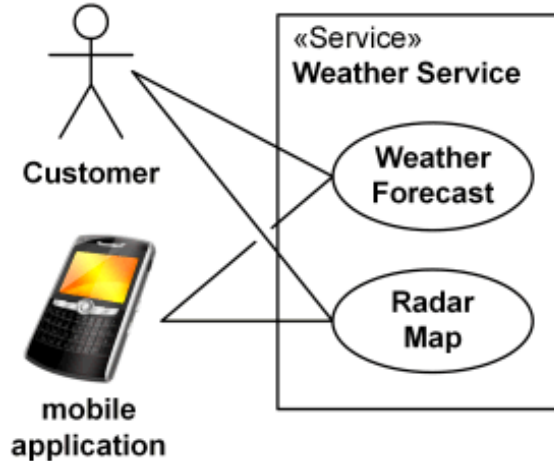
<<include>> is used to model shared functionality by multiple use cases, while **<<extend>>** is used to model optional or exceptional behaviour.

System Use Cases Diagrams



<<include>> is used to model shared functionality by multiple use cases, while **<<extend>>** is used to model optional or exceptional behaviour.

Use Cases Diagrams — Subjects



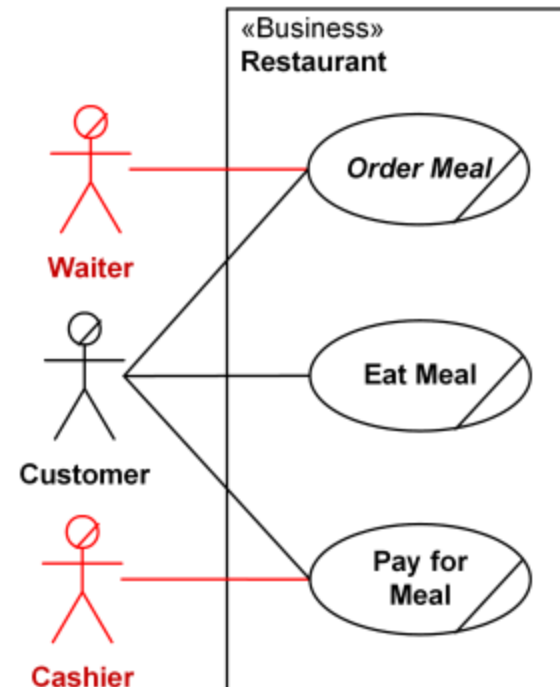
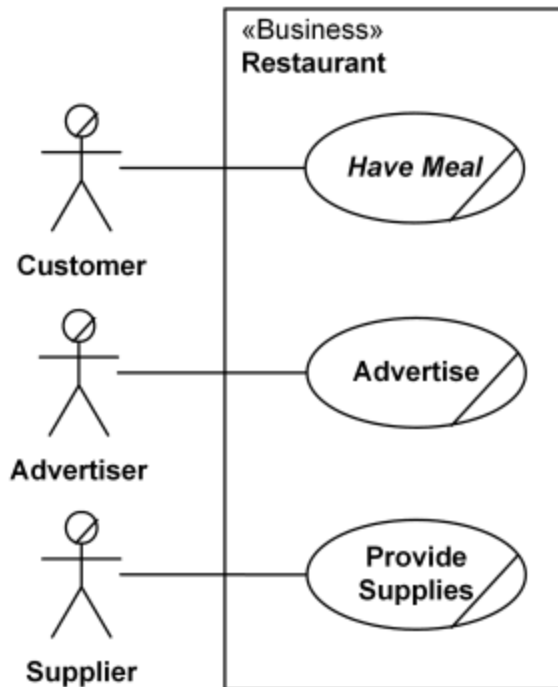
Weather Service subject stereotyped as «Service»

«Subsystem»

«Process»

«Service»

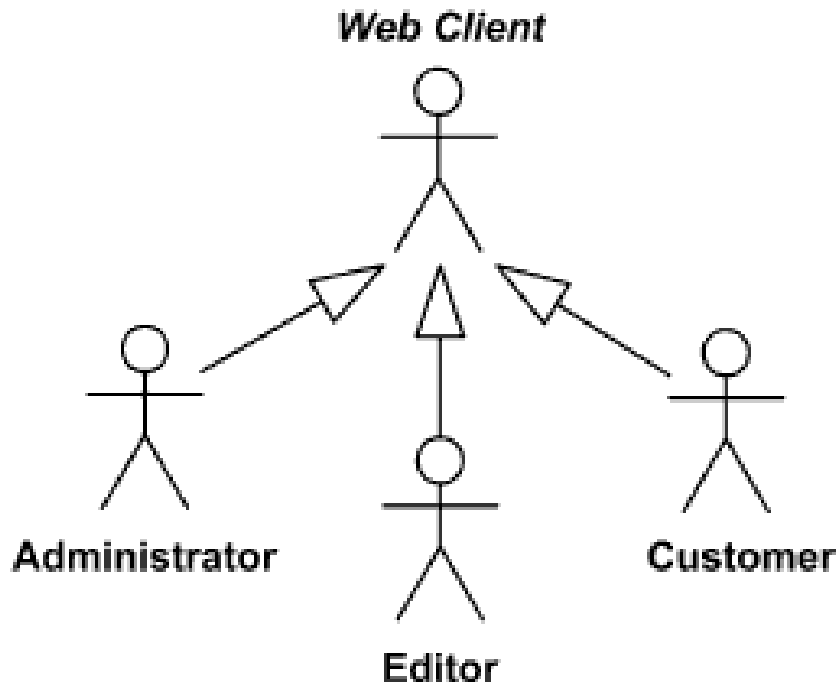
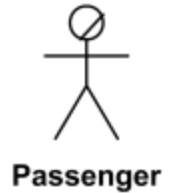
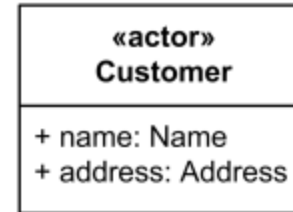
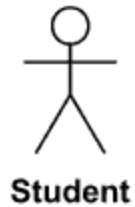
«Component»



Restaurant business subject with business actors and applicable use cases

Mistake: Restaurant business should not have Waiter and Cashier as actors

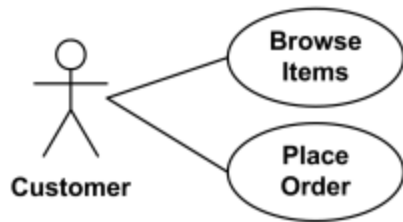
Use Cases Diagrams — Actors



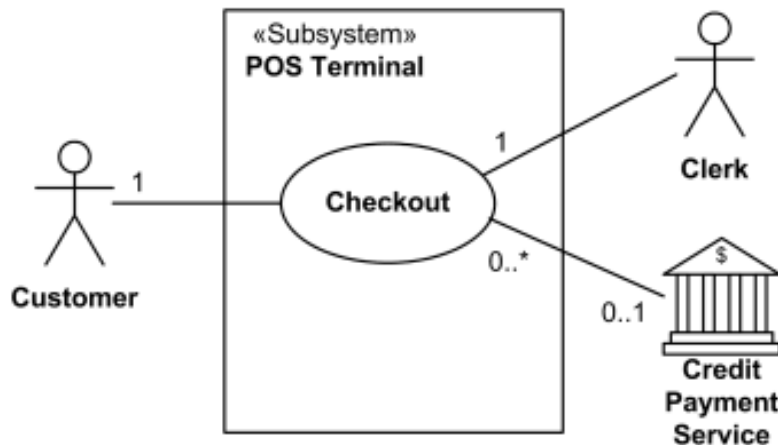
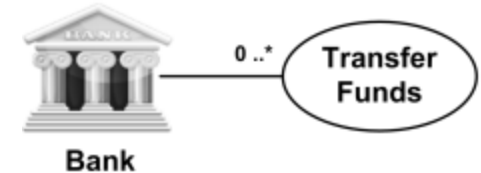
We can define *abstract* or **concrete** actors and specialise them using **generalisation** relationships.

Web Client actor is an abstract superclass for Administrator, Editor, and Customer.

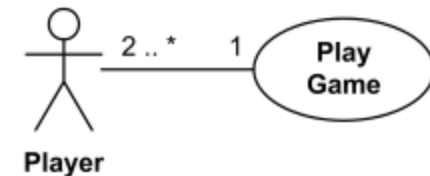
Use Cases Diagrams — Associations



Manage Account use case is associated with **Customer** and **Bank** actors.



Checkout use case requires **Customer** actor, hence the 1 multiplicity of **Customer**. The use case may not need **Credit Payment Service** (for example, if payment is in cash), thus the 0..1 multiplicity.



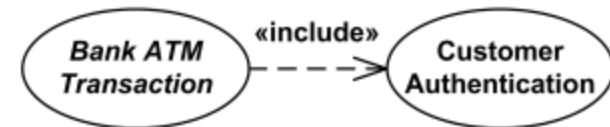
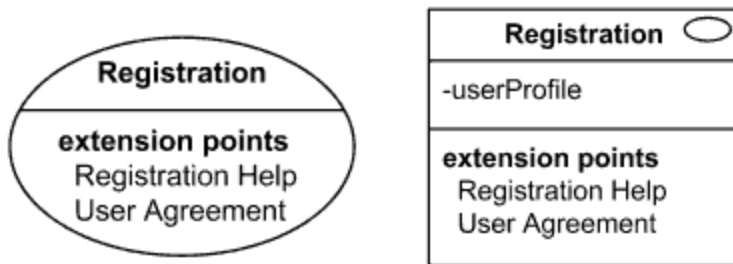
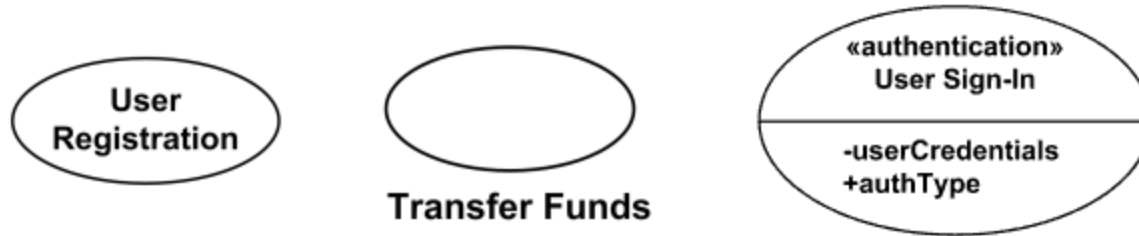
Two or more **Player** actors are involved in the **Play Game** use case. Each **Player** participates in one **Play Game**.

UML 2.x allows *any reasonable interpretation* of the interaction of "multiple actors" with a use case: it is "**undefined**".

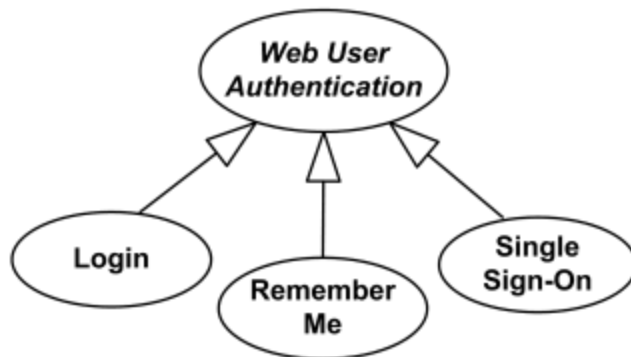
The **actor's multiplicity** could mean that an interaction of a particular use case with several separate actor instances might be:

- simultaneous (concurrent), or
- overlapping at different points in time, or
- mutually exclusive (sequential, random, etc.).

Use Cases Diagrams — Use Cases

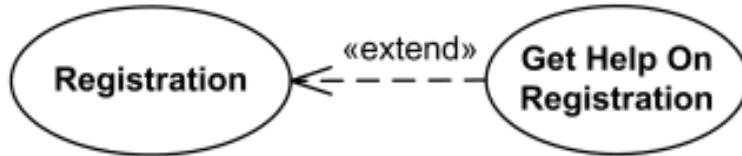


Bank ATM Transaction use case becomes abstract use case due to including Customer Authentication use case.

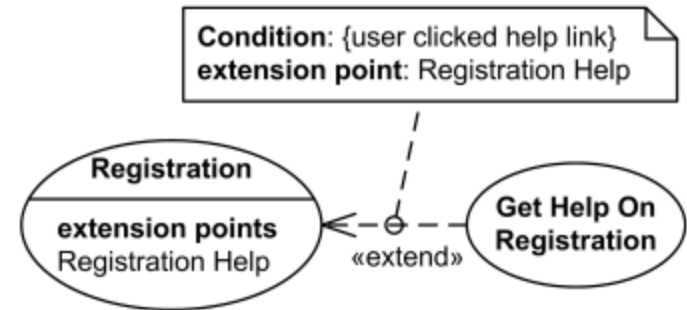


Web User Authentication use case is an **abstract use case** specialised by Login, Remember Me and Single Sign-On use cases.

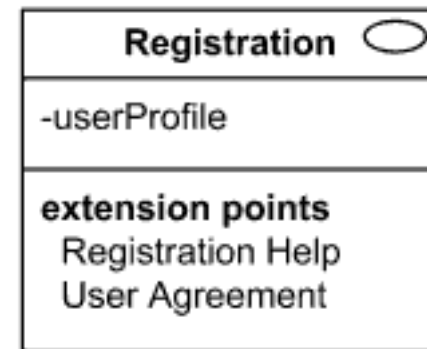
Use Cases Diagrams — Extend



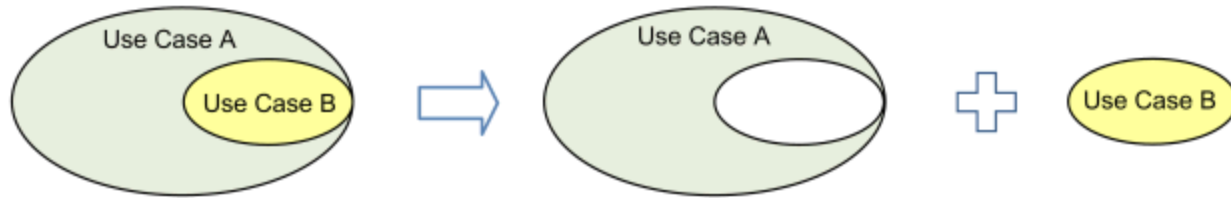
Registration use case is complete and meaningful on its own. It could be *extended* with *optional* **Get Help On Registration** use case.



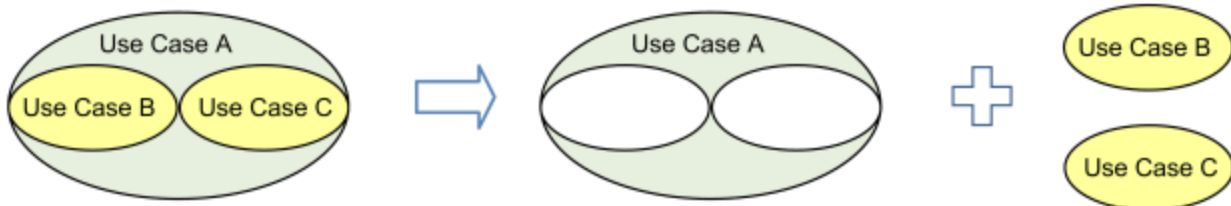
Registration use case is *conditionally extended* by **Get Help On Registration** use case in extension point **Registration Help**.



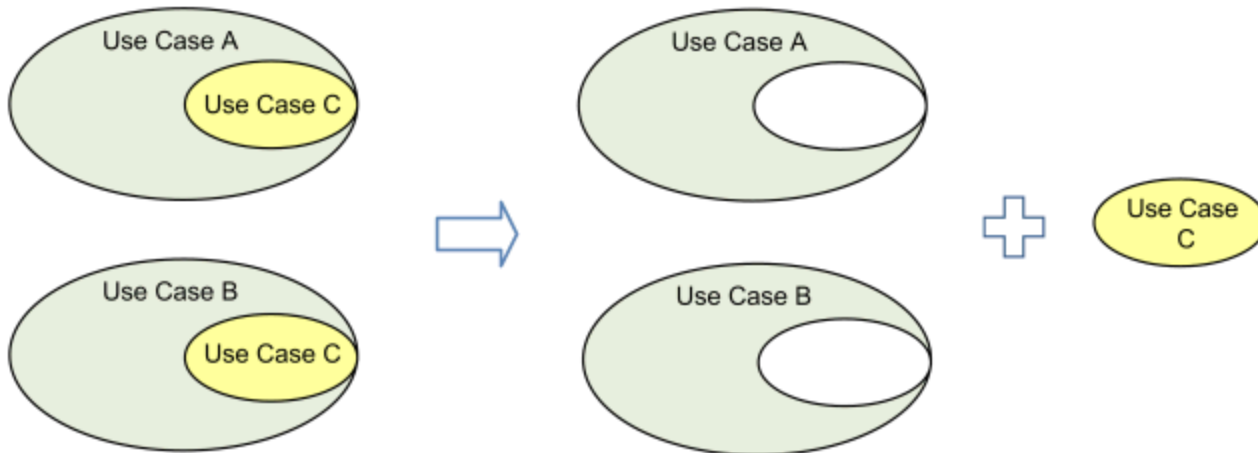
Use Cases Diagrams — Include



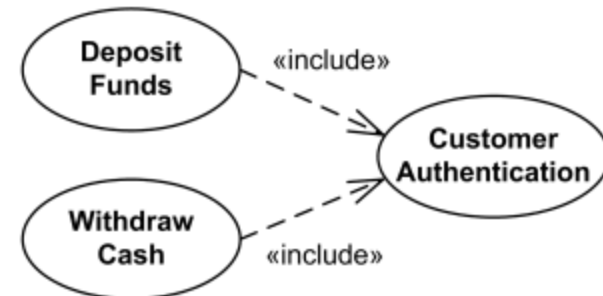
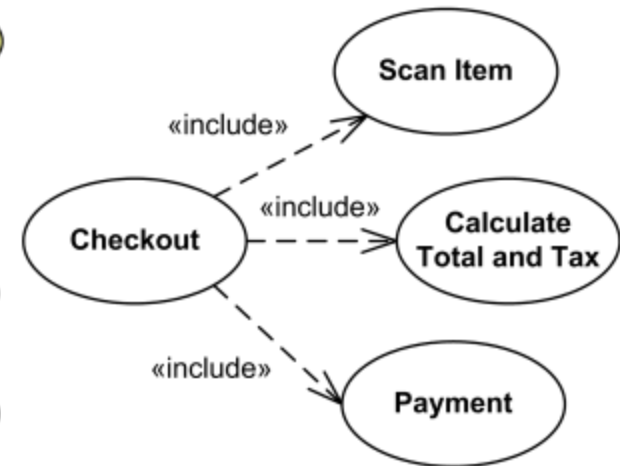
Use case B is extracted from larger use case A into a separate use case.



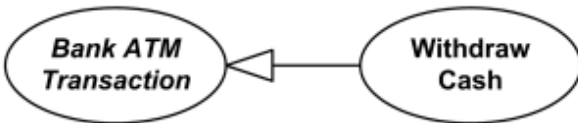
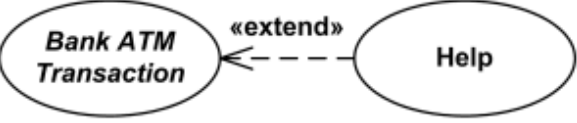
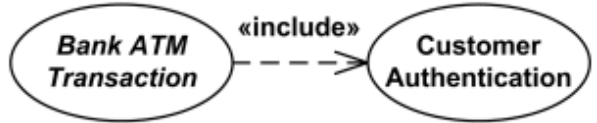
Use cases B and C are extracted from larger use case A into separate use cases.



Use case C is extracted from use cases A and B to be reused by both use cases using UML include relationship.



Use Cases Diagrams — Relationships

Generalization	Extend	Include
		
Base use case could be abstract use case (incomplete) or concrete (complete).	Base use case is complete (concrete) by itself, defined independently.	Base use case is incomplete (abstract use case).
Specialized use case is required, not optional, if base use case is abstract.	Extending use case is optional, supplementary.	Included use case required, not optional.
No explicit location to use specialization.	Has at least one explicit extension location.	No explicit inclusion location but is included at some location.
No explicit condition to use specialization.	Could have optional extension condition.	No explicit inclusion condition.

<https://www.uml-diagrams.org/use-case-diagrams-examples.html>

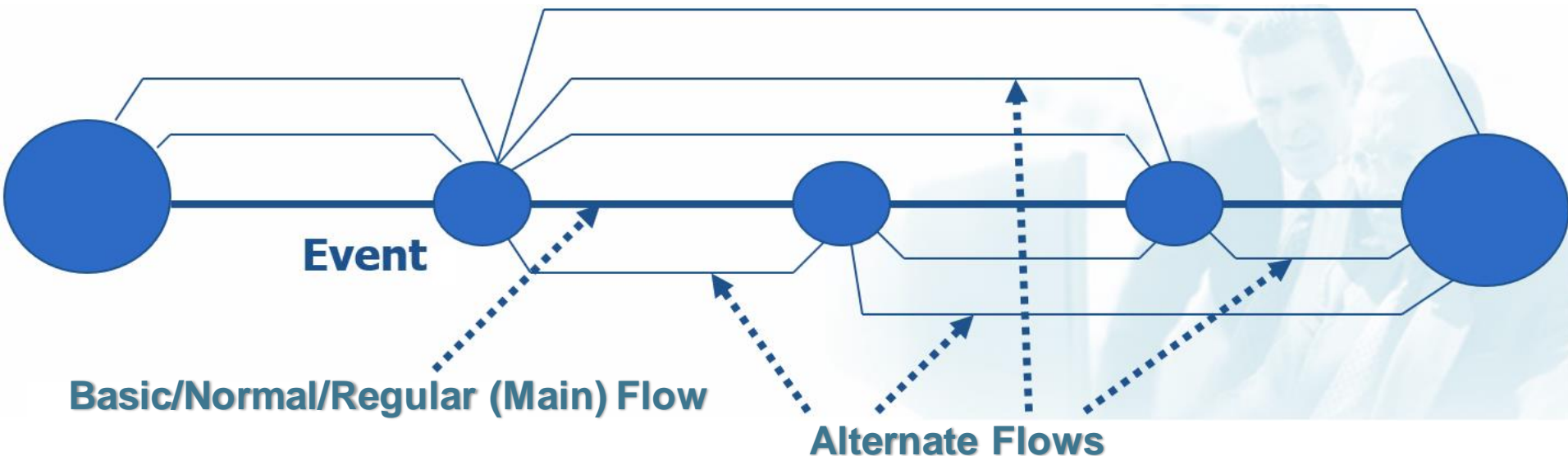
Use Case Documentation

- > Essential documentation and structure:
 - >> Brief description: its purpose in a few lines.
 - >> Detailed event flows: description of the primary and alternate event flows that happen when the use case starts.
 - >> The documentation must be read as a conversation between the actor and the use case.
- > The documentation is redacted in simple terms so that all major project stakeholders can easily understand it.
- > It is recommended to build a glossary of specific terms.

Event Flow

> Each use case:

- >> Has a basic/normal/regular (main) transaction sequence.
- >> May have several alternate transaction sequences.
- >> Usually, it has various **Exception** transaction sequences to handle errors and incorrect situations.
- >> May have well-defined pre- and post-conditions.

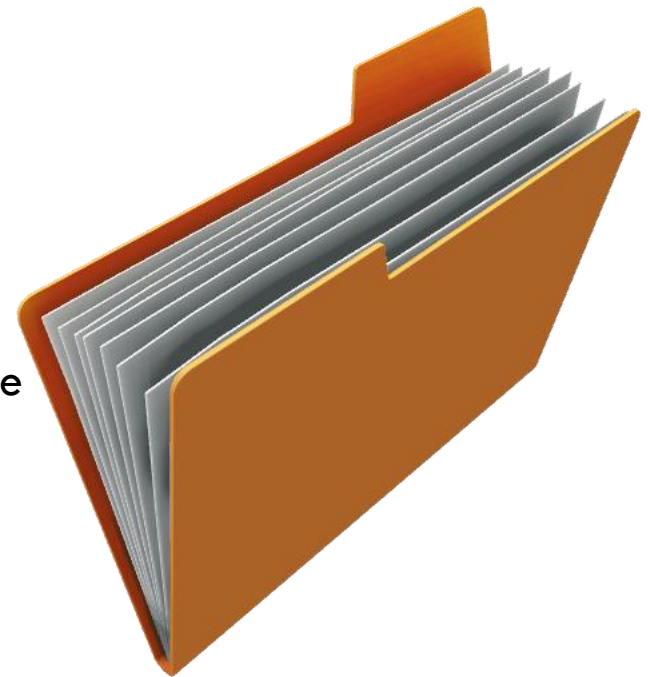


Event Flow

- > Must describe only the events that belong to the use case, not what happens in other use cases.
- > Avoid vague terms such as “delete information”, “read data”, etc.
- > The event flow must describe:
 - >> How and when the use case begins and ends.
 - >> When the use case interacts with the actors (dialogue).
 - >> The exchanged information between the actor and the use case. (**Do not describe the details of the user interface**).
 - >> The basic/normal/regular event flow.
 - >> Any other alternate flow and exceptions.

Who read the use cases?

- > **Clients/Customers:** they approve what the system must do.
- > **Users:** increase their understanding of the system.
- > **Developers:** document the system's behaviour.
- > **Reviewers/Auditors:** examine the event flows.
- > **System Analysts/Designers:** the use cases provide the basis for the analysis and design.
- > **System Testers:** the use cases are used as the basis for the testing scenarios.
- > **Project Leader:** the use cases provide insights for future project planning.
- > **Technical Writer:** the use cases provide the basis for writing the user manual.



1. Title.
2. General description.
3. Event flow.
 - a) **Preconditions:** list of mandatory conditions before the use case starts.
 - b) **Basic (Main) flow.**
 - Functionality of the beginning.
 - **Sequence of functional steps.**
 - Access point to alternate and exception flows.
 - Identify the alternate and exception flows through a standard naming convention: AF-01, EF-01.
 - c) **Alternate flows (AFs).**
 - **Sequence of functional steps.**
 - Connection with other alternate flows and/or exception flows.
 - d) **Exception flows (EFs).**
 - **Sequence of functional steps.**
 - List possible recovery options or normalisation for the use case's continuity.
 - It might be a finalisation event that ends the use case.
 - e) **Post-conditions:** list of expected system conditions after the use case finishes.

Use Case Content Structure — Template

Name:

Identifier (*A unique identifier for this use case, e.g. UC10*)

Description (*A couple of sentences or a paragraph describing the basic idea of the use case*)

Goal (*The business goal of the initiating actor*)

Preconditions (*List the state(s) the system can be in before this use case starts*)

1.

Assumptions (*Optional, List all assumptions that have been made*)

1.

Frequency (*Approximately how often this use case is realized, e.g., once a week, 500 times a day, etc.*)

Basic Course (*Describe the “normal” processing path, aka, the Happy Path*)

1. Use case begins when ...
- 2.
3. Use case ends when ...

Alternate Course A: Description of the alternate course

Condition: Indicate what happened

A.6 List the steps

Post conditions (*List the state(s) the system can be in when this use case ends*)

1.

Actors (*List of actors that participate in the use case*)

Included Use Cases (*Optional, List of use cases that this use case includes*)

1.

Extended Use Case (*Optional, The use case, if any, that this use case extends*)

Notes

List any “to dos”, concerns to be addressed, important decisions made during the development of this use case, ...

- > A scenario is a use case instance (an example): A flow through a use case.
- > Each use case has a network of scenarios.
 - >> Primary scenarios (*happy day scenarios*).
 - >>> Normal flow — the way the system must function.
 - >>> Alternate flow — alternate functional way.
 - >> Secondary scenarios.
 - >>> Exception flows — exceptions in the normal and alternate flows.
- > How many scenarios are necessary? As many as needed to understand the system!
- > Recommendations:
 - >> Primary scenarios: elaborate around 80% of these scenarios.
 - >> Secondary scenarios: elaborate on the most interesting and emblematic ones, such as the ones that depict a high-impact result deemed high risk to the organisation.
 - >> Identify the most frequent, the least frequent, and the scenarios that will never occur (edge cases).

Functional and Non-Functional Requirements

> **Functional requirements:** describe what the software system should do, or the actions it should perform. They specify the functionality that the system should provide to its users (essentially Use Cases). Examples:

- User authentication: The system should provide a secure login and authentication mechanism for users.
- Data processing: The system should process user data and generate reports based on that data.
- Payment processing: The system should be able to process user payments through multiple payment gateways.
- Search functionality: The system should provide a search feature that allows users to search for specific data or content within the application.
- Chat functionality: The system should provide chat functionality for users to communicate with each other.

> **Non-functional requirements:** describe how well the system should perform, or the quality of the system. They specify constraints and characteristics that the system should possess, such as performance, scalability, security, usability, maintainability, reliability, portability, flexibility, and reusability.

- *Performance:* The system should be able to handle a large number of users without any performance degradation.
- *Security:* The system should have robust security mechanisms to protect user data from unauthorized access.
- *Usability:* The system should be easy to use, with a user-friendly interface and intuitive navigation.
- *Scalability:* The system should be able to scale up or down based on the changing needs of the business.
- *Maintainability:* The system should be easy to maintain, with clear and concise documentation, and easy-to-understand code.

Functional and Non-Functional Requirements

Functional Requirements	Non-Functional Requirements
A functional requirement defines a system or its component.	A non-functional requirement defines the quality attribute of a software system.
It specifies “What should the software system do?”	It places constraints on “How should the software system fulfil the functional requirements?”
Functional requirement is specified by User.	Non-functional requirement is specified by technical peoples e.g. Architect, Technical leaders and software developers.
It is mandatory.	It is not mandatory.
It is captured in use case.	It is captured as a quality attribute.
Defined at a component level.	Applied to a system as a whole.
Helps you verify the functionality of the software.	Helps you to verify the performance of the software.
Functional Testing like System, Integration, End to End, API testing, etc are done.	Non-Functional Testing like Performance, Stress, Usability, Security testing, etc are done.
Usually easy to define.	Usually more difficult to define.
<ol style="list-style-type: none"> 1) Authentication of user whenever he/she logs into the system. 2) System shutdown in case of a cyber attack. 3) A Verification email is sent to user whenever he/she registers for the first time on some software system. 	<ol style="list-style-type: none"> 1) Emails should be sent with a latency of no greater than 12 hours from such an activity. 2) The processing of each request should be done within 10 seconds 3) The site should load in 3 seconds when the number of simultaneous users are > 10000

Exercise

- > Choose your favourite social network platform:
@tw, @fb, @ig, @dc, @sc, @wc, etc.
- > Imagine a specific module's functionality of the system.
 - >> Elaborate on the proper use case diagrams.
 - >> Write a definition for each identified actor.
 - >> For each use case:
 - >>> Write the content structure.
 - >>> List some of the possible scenarios.

Summary

- > The system's behaviour is HOW the system acts and reacts.
- > The system's behaviour can be characterised by a use case set.
- > A use case is “some functionality” the system executes in response to an external actor's stimulus.
 - >> It provides the vehicle to capture the system's requirements from a “user's perspective”.
- > An actor is an entity that must be “interfaced with” the system.
- > A user case diagram is a graphic description that depicts the system's actors and identified use cases.
- > The documentation consists of a brief description and the event flows.



Life with UML: It's Still Work

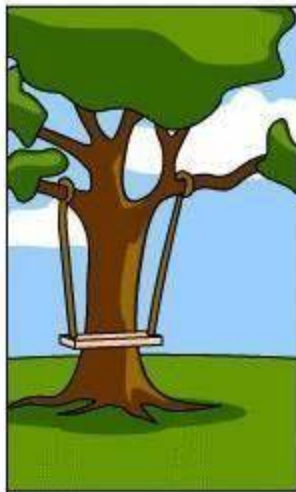
PHILIPPE KRUCHTEN,
SOFTWARE ARCHITECT

Many of the fevers identified in Alex's "Death by UML Fever" are related to the software process, absence of a software process, or to fundamental misunderstanding of what a process is for. I hear comments such as: "Oh, we ran all the activities described by RUP (Rational Unified Process) and created all the UML (Unified Modeling Language) diagrams it prescribes..." or "There is this widget in UML, and I can't find how RUP says to use it." UML is a notation that should be used in most cases simply to illustrate your design and to serve as a general roadmap for the corresponding implementation. Unfortunately, some users of UML leave their brains in the lobby, get settled behind their keyboards, and get busy drawing UML diagrams because doing so is a much easier alternative than doing difficult software engineering work.

Meme for today's lecture! Keep on practising!



How the customer explained it



How the Project Leader understood it



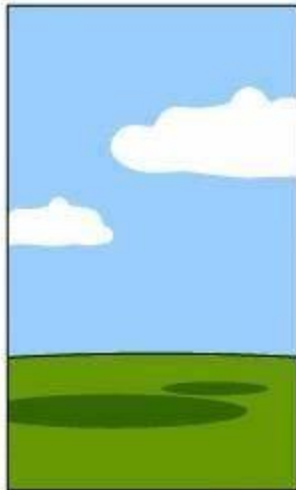
How the Analyst designed it



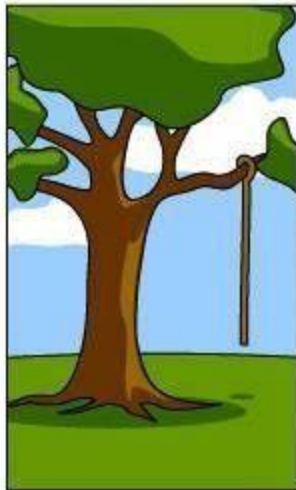
How the Programmer wrote it



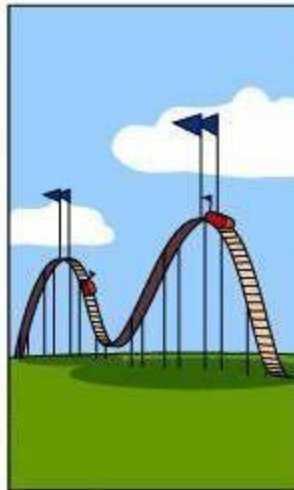
How the Business Consultant described it



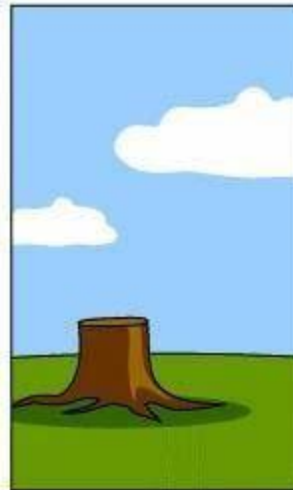
How the project was documented



What operations installed



How the customer was billed



How it was supported



What the customer really needed

Main References

- [1] USAC/MTIC/AS06-System_Architectures, Lecture notes. (2011).
- [2] Tata Consulting Services (TCS). Object-Oriented Design and Analysis (OOAD) Course. Chapter #3. (2000).
- [3] UML Diagrams / UML Use Case Diagrams. <https://www.uml-diagrams.org/>