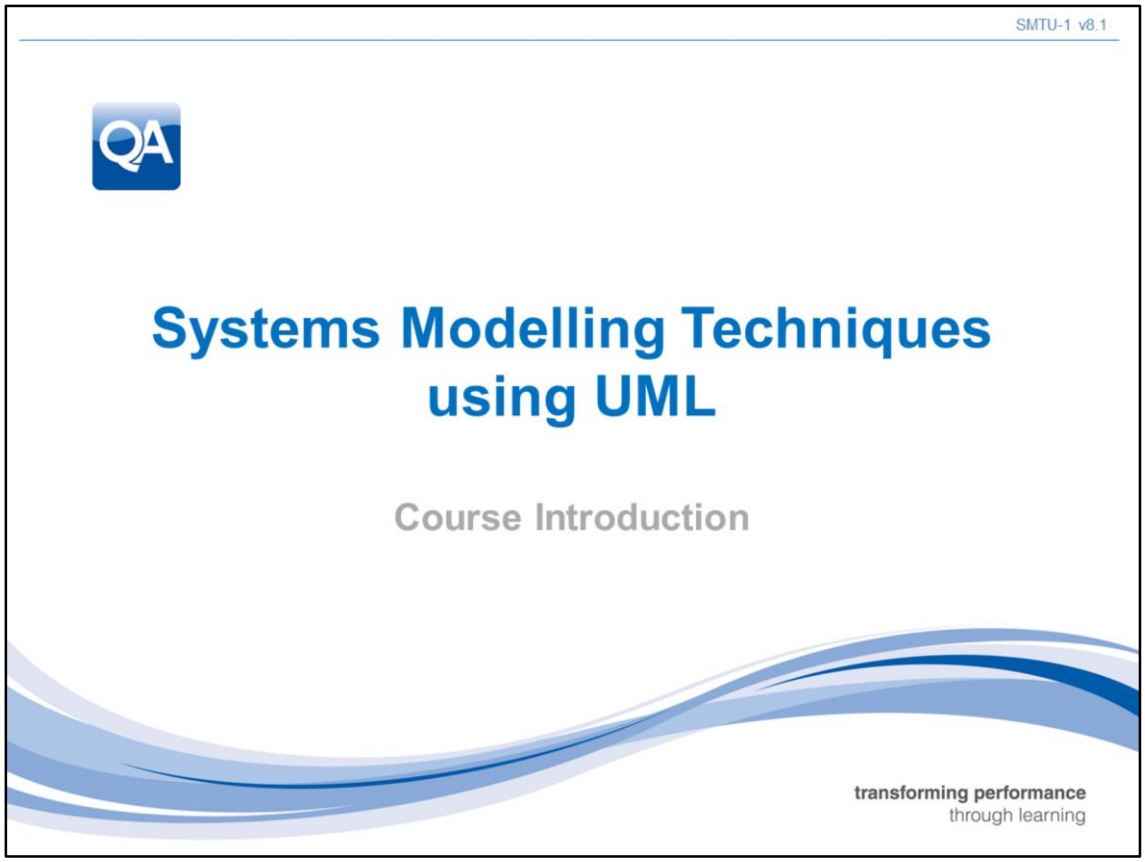
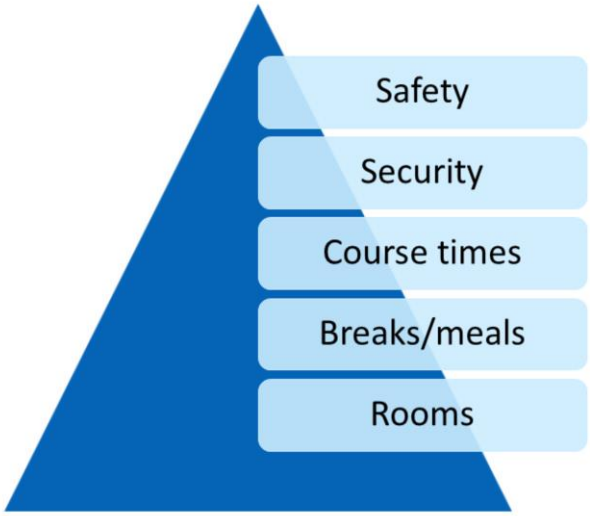
Systems Modelling Techniques using UML

Delegate Guide


SMTU-1 v8.1
Systems Modelling Techniques using UML
Course Introduction
transforming performance
through learning

SMTU-1 v8.1

Administration



SMTU-1 v8.1

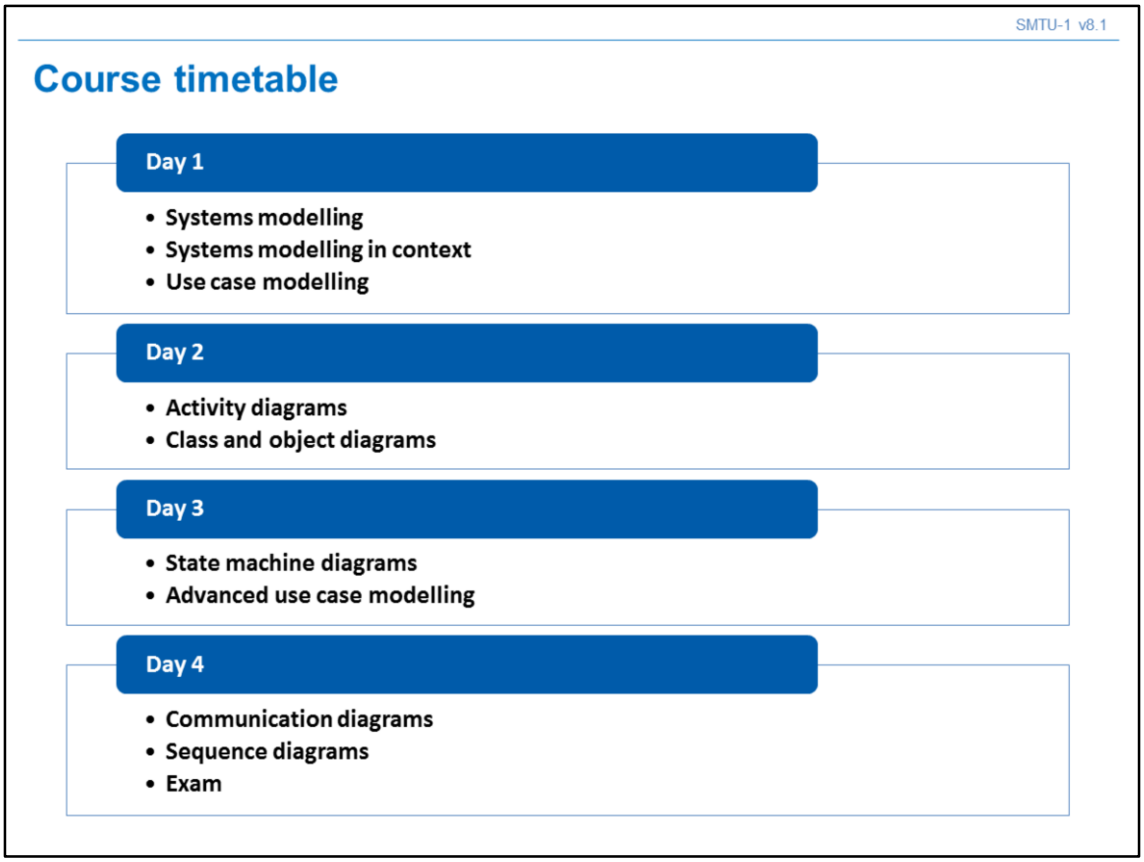
BCS certificate objectives

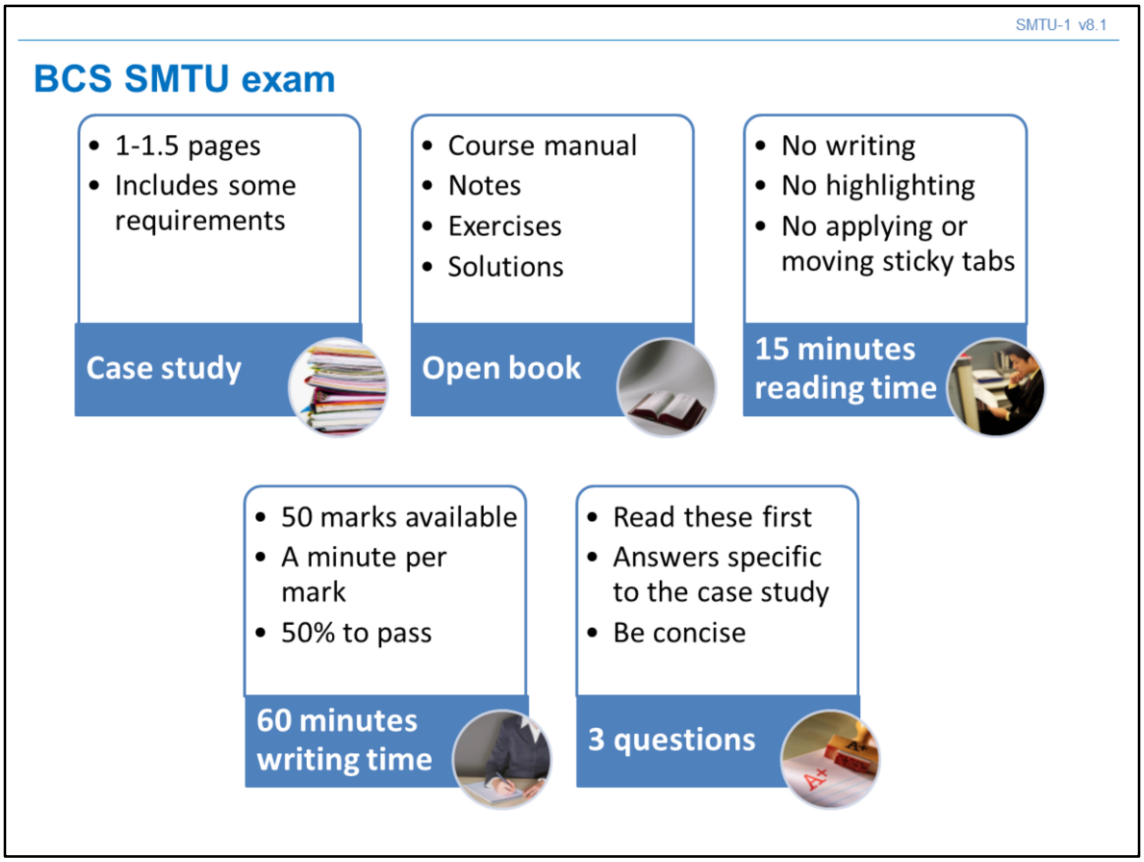
- Justify the need for IT system modelling and modelling techniques
- Explain why it is important to model IT system requirements from different perspectives
- Develop models of system functionality as use case diagrams with supporting use case descriptions
- Develop models of system data as class models
- Develop dynamic models as state machines and sequence diagrams showing the realisation of a use case
- Evaluate selected models against business objectives and system requirements
- Appreciate how the selected models inter-relate with each other
- Describe how the products of analysis feed into the design and development of a system

SMTU-1 v8.1

About you

- **Name**
- **Company**
- **Role and key responsibilities**
- **Knowledge of business and systems analysis**
- **Your objectives for attending this course**
- **Other courses**
  - BA or SD Diploma?


SMTU-1 v8.1
Course timetable
Day 1
• Systems modelling
• Systems modelling in context
• Use case modelling
Day 2
• Activity diagrams
• Class and object diagrams
Day 3
• State machine diagrams
• Advanced use case modelling
Day 4
• Communication diagrams
• Sequence diagrams
• Exam


SMTU-1 v8.1
BCS SMTU exam
1-1.5 pages
Includes some requirements
Case study
Course manual
Notes
Exercises
Solutions
Open book
No writing
No highlighting
No applying or moving sticky tabs
15 minutes reading time
50 marks available
A minute per mark
50% to pass
60 minutes writing time
Read these first
Answers specific to the case study
Be concise
3 questions

SMTU-1 v8.1

BCS International Diploma in Business Analysis

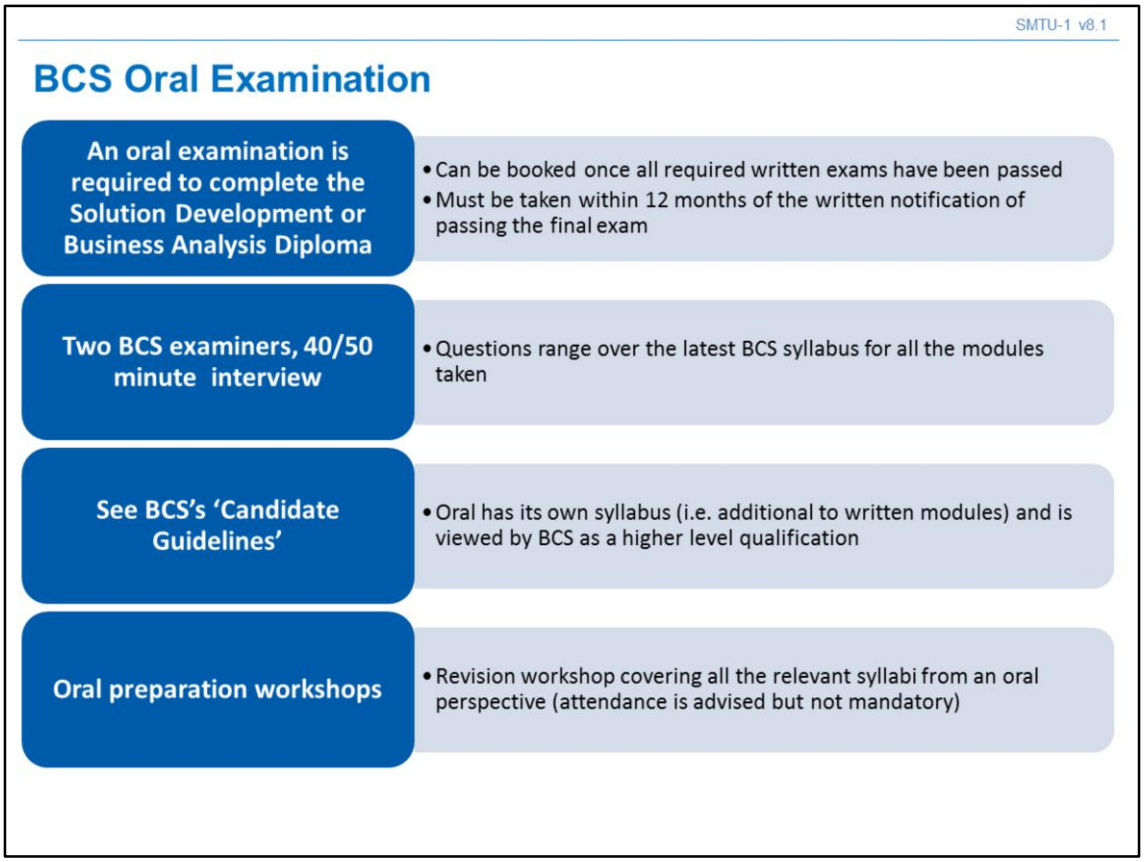
Core	Knowledge-based Specialism	Practitioner Specialism
Business Analysis Practice	Commercial Awareness	Modelling Business Processes
Requirements Engineering	Foundation Certificate in IS Project Management	**Systems Modelling Techniques (UML)**
	Foundation Certificate in Business Analysis	Systems Development Essentials
	Foundation Certificate in Business Change	Certificate in Benefits Management and Business Acceptance
Both	**One of the above**	**One of the above**

SMTU-1 v8.1

BCS International Diploma in Solution Development

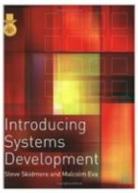
Core	Knowledge-based Specialism	Practitioner Specialism
Systems Development Essentials	Foundation Certificate in Systems Development	Business Analysis Practice
Systems Modelling Techniques (UML)	Intermediate Certificate in Enterprise and Solution Architecture	Systems Design Techniques(*)
	ISTQB-BCS Certified Tester Foundation Level	Practitioner Certificate in Enterprise and Solution Architecture
	Foundation Certificate in IT Service Management (V3)	Integrating Off-the-shelf Software Solutions
Both	**One of the above**	**One of the above**

* SDT complements SMTU. SMTU concentrates on Analysis, while SDT concentrates on Design.

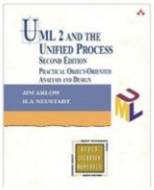

SMTU-1 v8.1
BCS Oral Examination
An oral examination is required to complete the Solution Development or Business Analysis Diploma
• Can be booked once all required written exams have been passed
• Must be taken within 12 months of the written notification of passing the final exam
Two BCS examiners, 40/50 minute interview
• Questions range over the latest BCS syllabus for all the modules taken
See BCS's 'Candidate Guidelines'
• Oral has its own syllabus (i.e. additional to written modules) and is viewed by BCS as a higher level qualification
Oral preparation workshops
• Revision workshop covering all the relevant syllabi from an oral perspective (attendance is advised but not mandatory)

Additional course resources

- **Introducing Systems Development**
  - Author: Steve Skidmore and Malcolm Eva
  - Publisher: Palgrave Macmillan
  - Publication Date: 2003
  - ISBN: 0333973690



- **UML and the Unified Process**
  - Author: Jim Arlow and Ila Neustadt
  - Publisher: Addison Wesley
  - Publication Date: 2005
  - ISBN: 978-0321321275



- **Object – Oriented Systems Analysis and Design Using UML**
  - Author: Simon Bennett, Steve McRobb and Ray Farmer
  - Publisher: McGraw Hill
  - Publication Date: 2005
  - ISBN: 0077092444

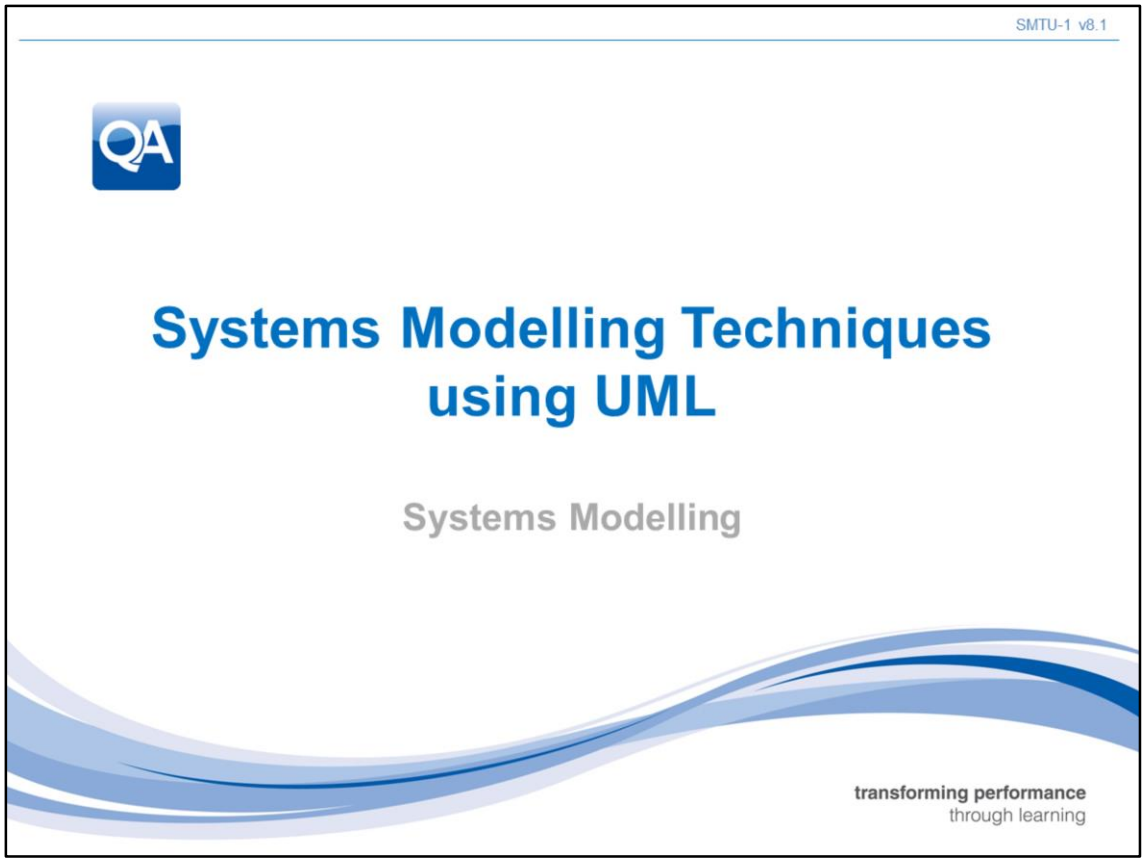
- **BCS Web Site for SMTU http://certifications.bcs.org/content/ConTab/38**

SMTU-1 v8.1

Additional Reading

- **UML Distilled**
  - Author: Martin Fowler
  - Publisher: Addison Wesley
  - Publication Date: 2004
  - ISBN: 0-321-19368-7

- **The Rational Unified Process Made Easy**
  - Author: Kroll, Kruchten
  - Publisher: Addison Wesley
  - Publication Date: 2005
  - ISBN: 0-321-16609-4


SMTU-1 v8.1
Systems Modelling Techniques using UML
Systems Modelling
transforming performance
through learning

SMTU-1 v8.1

Topics

- **Systems Modelling in Context**
  - Monitoring analysis against
    - Business objectives
    - System requirements
  - The bridge to:
    - Design
    - Software package selection
    - Development
- **Systems Modelling**
  - The need for modelling and modelling standards
  - Rationale for the selected approach
  - The approach and a Systems Development Lifecycle
  - Place of models within the Systems Development Lifecycle
  - Modelling the system from different perspectives
  - Interaction of the models
  - Validating and verifying models

SMTU-1 v8.1

Systems Modelling in Context

- **Software is essential in supporting modern businesses**
  - Businesses have their Mission, Goals and Objectives, including delivering Value to their Customers
  - Business Processes deliver the Value
  - Software supports the Business Processes; the analysis and design of Business processes drives the derivation of software System requirements

- **This course focusses on UML *Analysis* models and techniques**
  - These models are a bridge to Design activity, including the selection of Packages and/or the development of Bespoke software

SMTU-1 v8.1

Systems Modelling - Workshop

Why are IT projects often:

- Late
- Over budget
- Incomplete
- Unfit for purpose

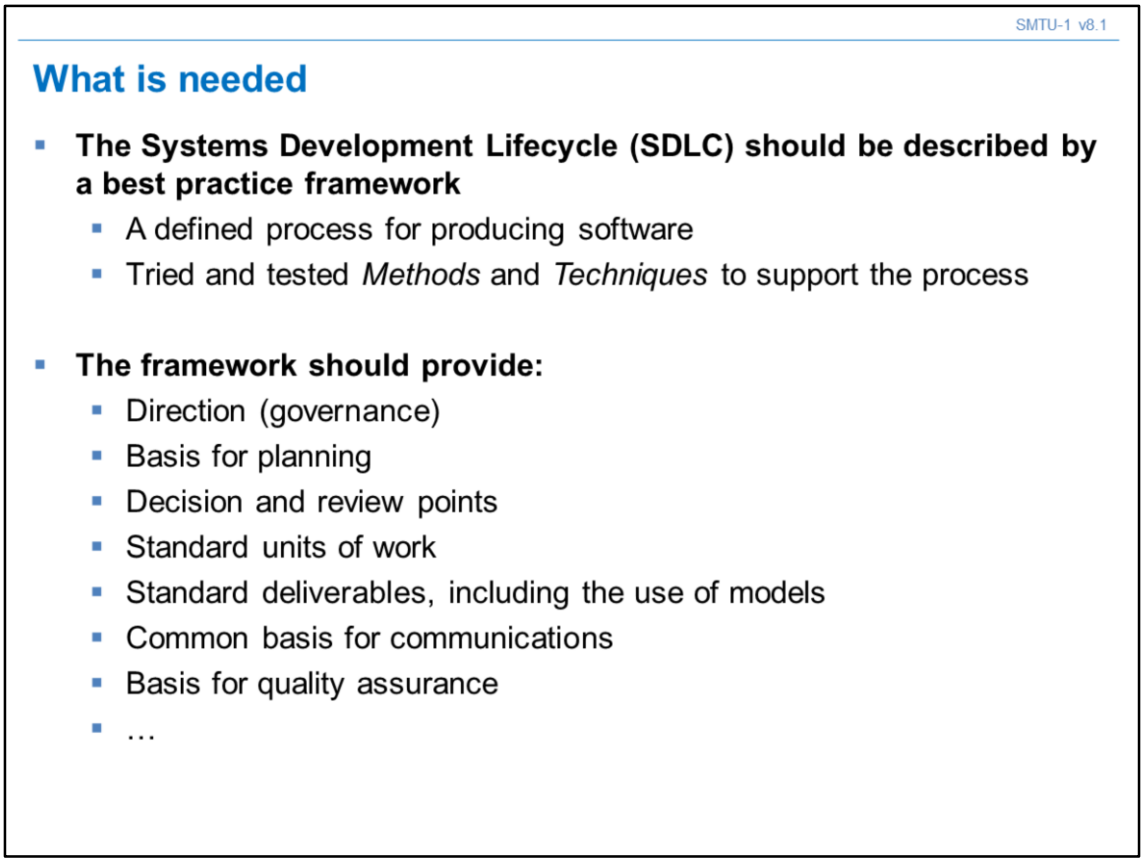
The purpose of this workshop is to suggest a list of factors that cause problems in systems projects from initiation to completion.

SMTU-1 v8.1

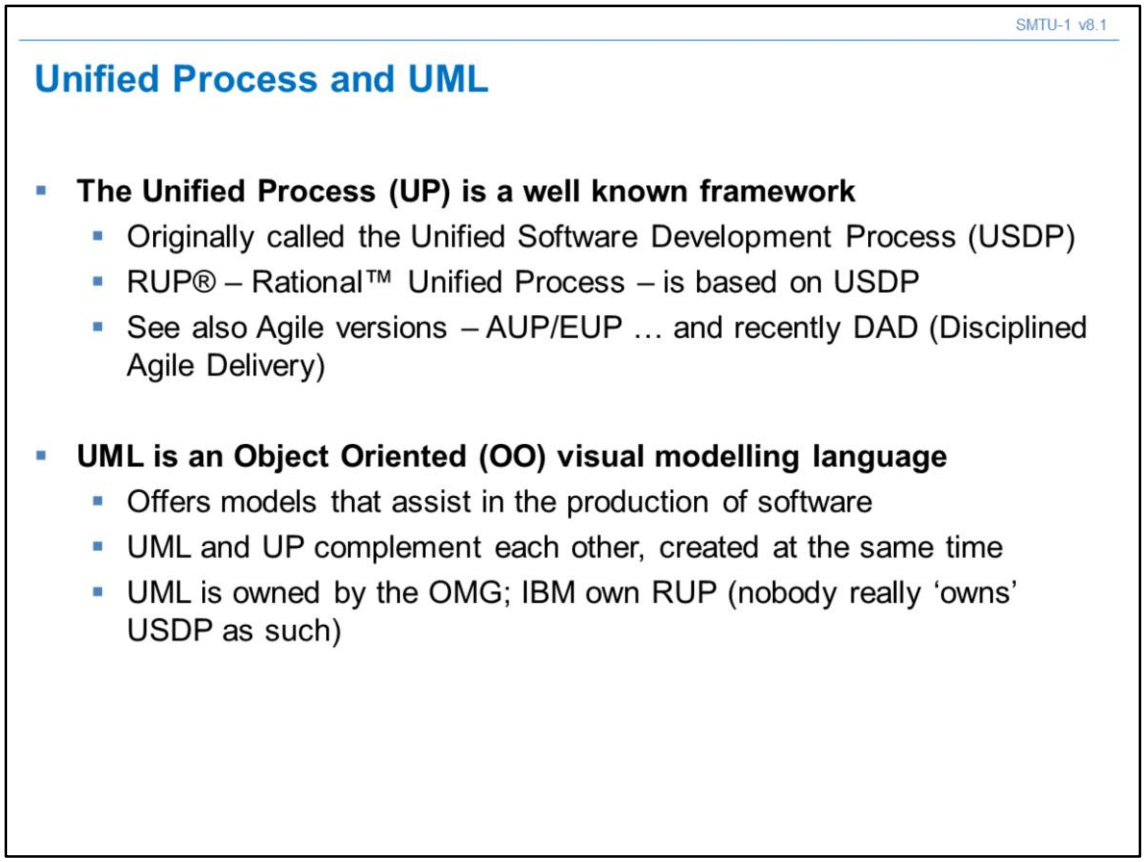
Some reasons

- Poor planning
- Scope unclear
- Requirements unclear
- Lack of direction
- Design not understood
- Poor communications
- Haphazard processes
- Lack of governance
- Poor documentation

Here are just a few examples.



Without a defined process for developing software, the results will fall short of expectations in terms of fitness for purpose and internal quality.

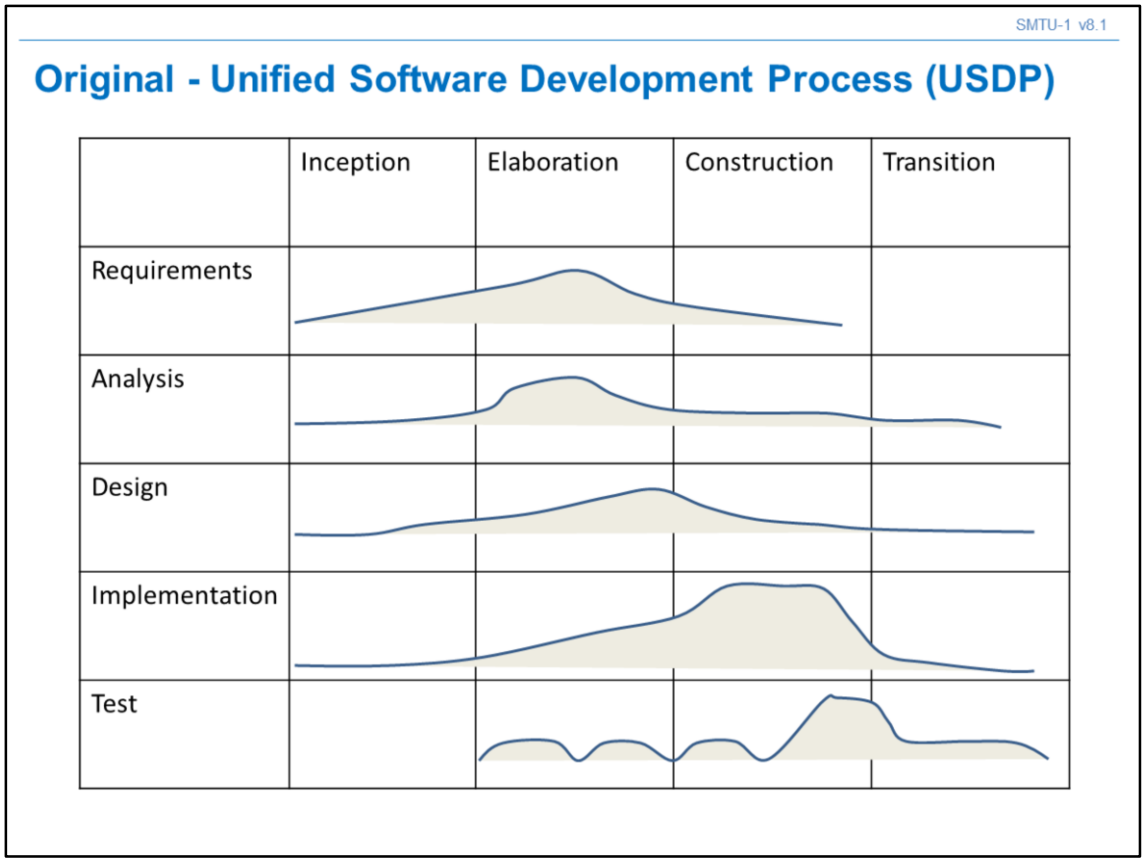


The Unified Process is a popular iterative and incremental software development process framework.

The best-known and extensively documented refinement of the Unified Process is the Rational Unified Process (RUP).

The Unified Process is not simply a process, but rather an extensible framework which should be customised for specific organisations or projects. The Rational Unified Process is, similarly, a customisable framework. As a result, it is often impossible to say whether a refinement of the process was derived from UP or from RUP, and so the names tend to be used interchangeably.

The name Unified Process as opposed to Rational Unified Process is generally used to describe the generic process, including those elements which are common to most refinements. The use of the Unified Process name is also to avoid potential issues of trademark infringement since Rational Unified Process and RUP are trademarks of IBM.



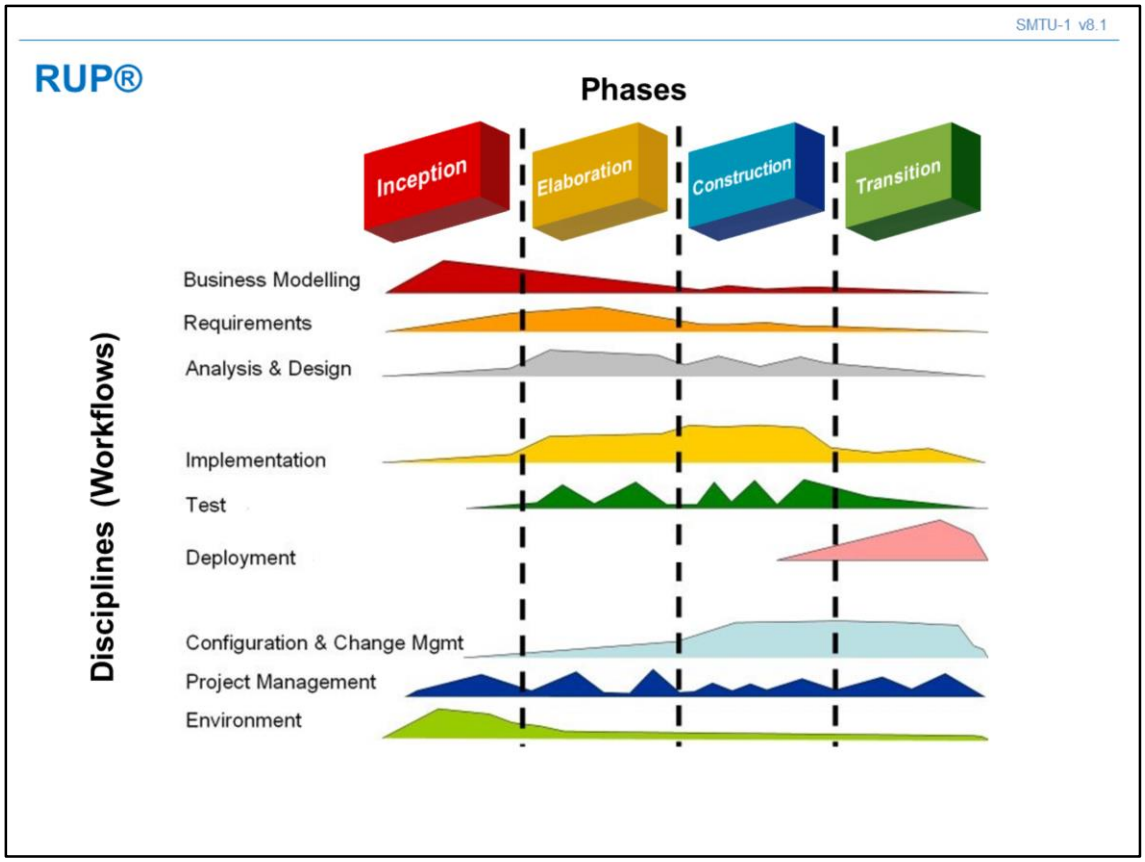
The graphic shows the basic structure of the original UP (USDP) into *Phases* and *Workflows*.

The line graph areas alongside each Workflow represent an indication of the amount of effort within each Phase; this obviously varies with the nature of the project in hand, and shouldn't be taken too literally.

Our course focusses mainly on the first 2 Workflows shown here, with emphasis on the models produced in the Inception Phase and early iterations of the Elaboration Phase.

The BCS course, Systems Design Techniques, SDT, covers mainly the Design workflow in the latter iterations of the Elaboration Phase.

In the BCS course Systems Development Essentials, SDE, the whole lifecycle is covered, at a relatively shallow depth of detail.



RUP® extends the basic UP to include a *Business Modelling* workflow and workflows that cover deployment into the production environment and management activities. BRADIT is a handy acronym for remembering the main workflows in RUP.

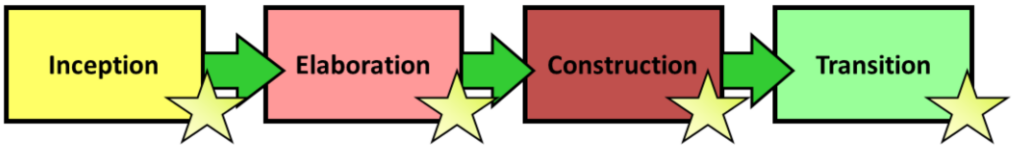
Workflows are called Disciplines in RUP, which is a commercial product owned by IBM (included now in the *Rational Method Composer* – RMC).

For the sake of simplicity we will refer to all versions as simply Unified Process (UP). We will retain the use of the word Workflow, but incline toward the RUP interpretation of UP.

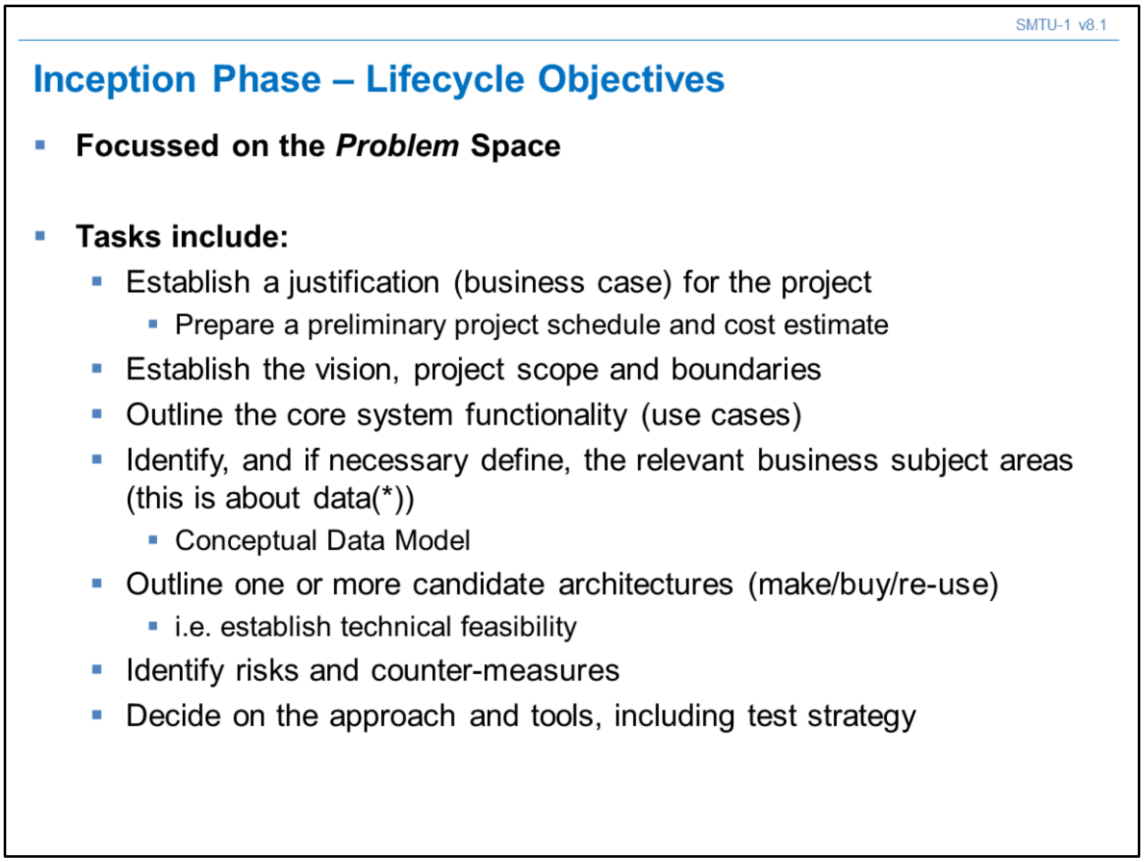
SMTU-1 v8.1

UP: 4 Phases, 4 Milestones

- In UP each development cycle is composed of four Phases
- There is a major milestone achievement at the end of each Phase:
  - **Inception** ➲ ☆Lifecycle Objectives
  - **Elaboration** ➲ ☆ Product Architecture
  - **Construction** ➲ ☆ Initial Operational Capability
  - **Transition** ➲ ☆ Product Release



We will examine the details of each of these phases in a little more detail, since that helps to situate the use of the models we will cover in the course.



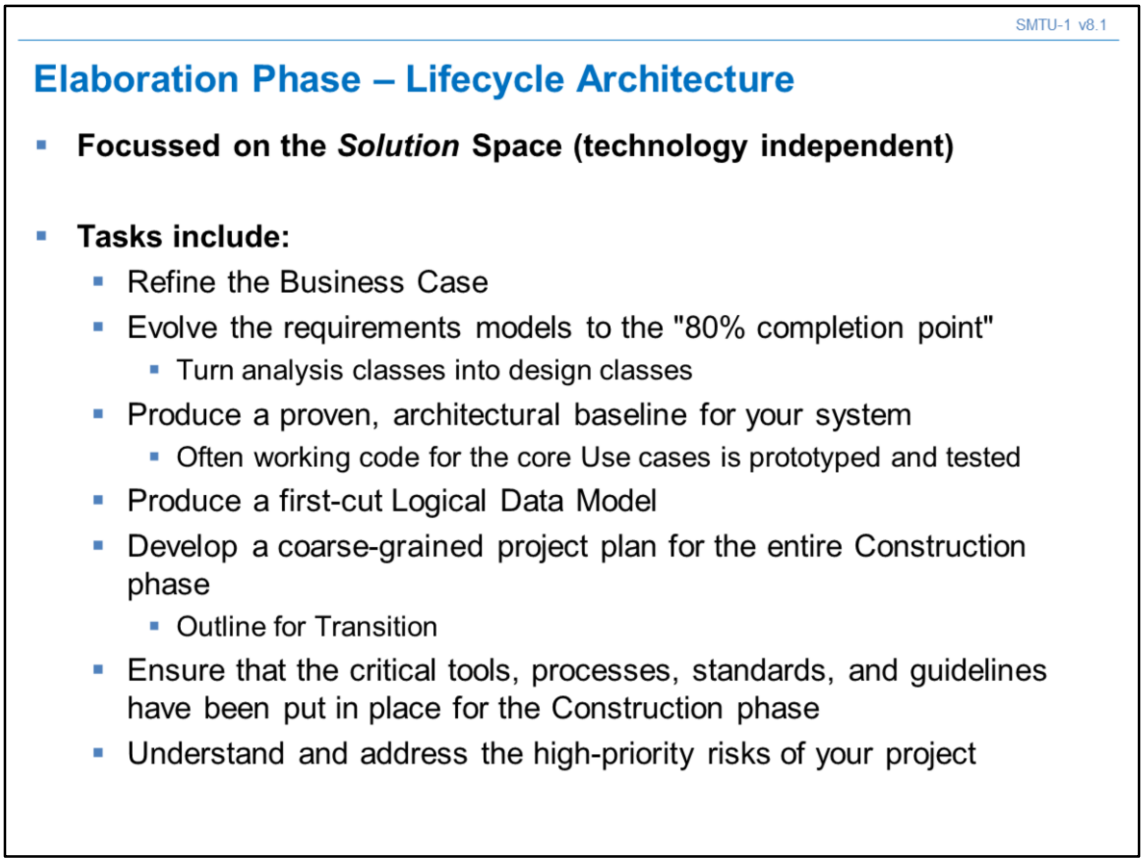
Inception is the smallest phase in the project, and ideally it should be quite short. If the Inception phase is long then it may be an indication of excessive up-front specification, which is contrary to the spirit of the Unified Process.

The emphasis is on 'visioning', generating a 'mile wide, inch deep' view of the software product required. The focus is very much around 'what to build', and how much time and money is required to build it.

Typical UML models include a Use Case Diagram covering the critical Use Cases, and a Domain Class model identifying the main data and information concepts. Short informal descriptions of Use Cases are produced, with perhaps some Activity Diagrams to sketch the logic of Business Processes and Use Cases.

This course is centred around the models used in this phase and the early part of Elaboration.

* Data doesn't stand out as a major topic in UP/RUP. This is because of its historic roots; however our concern is with *enterprise* software, where extensive data analysis and design usually does matter.

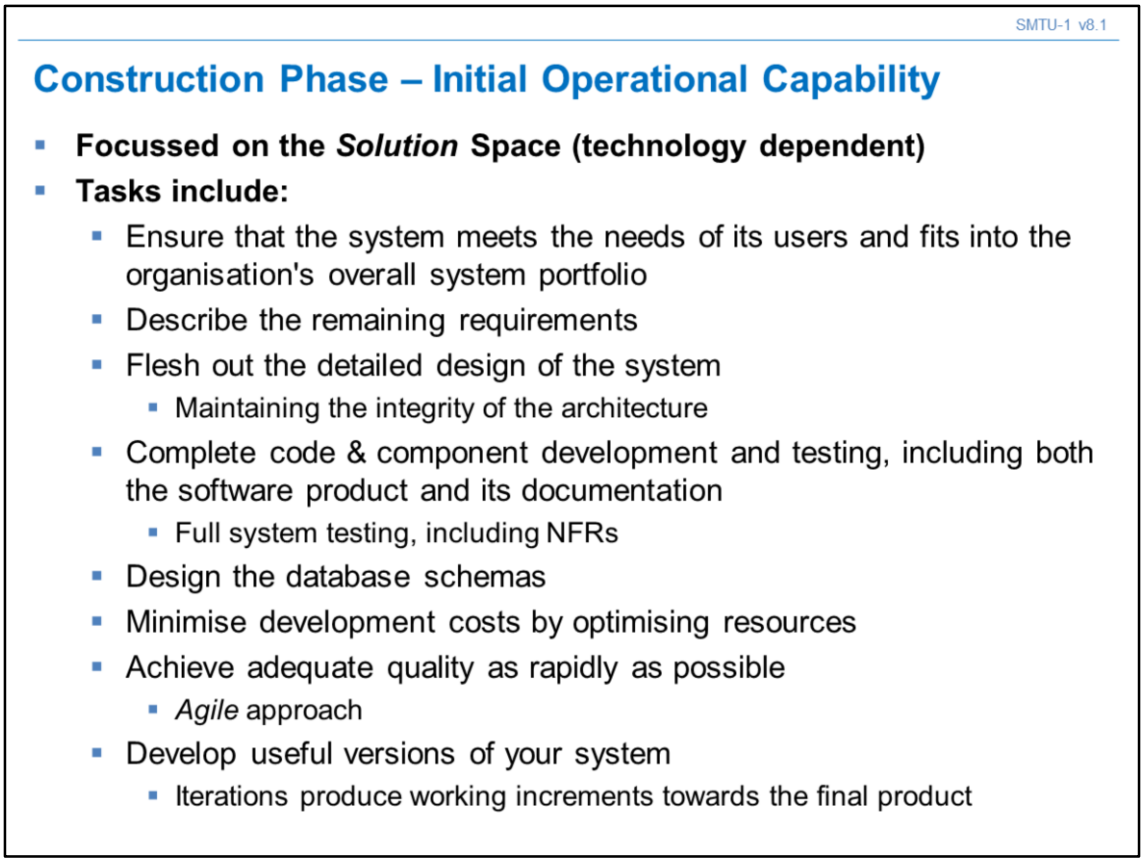


During the Elaboration phase the project team is expected to capture a healthy majority of the system requirements (80%). However, the primary goals of Elaboration are to address known risk factors and to establish and validate the system architecture. A 'first cut' of the core of the required system is often produced as a means of validation.

This phase is the most intensive in terms of modelling activity. Elaboration of (at least) the core Use Cases will be done here – this will include full narratives, and the use of UML Activity Diagrams and Interaction Diagrams. A Logical Data model, using a UML Class Diagram, is used to explore the structure of the persistent data required. State Machines are used to model the full lifecycle of persistent objects.

Architectural models are produced, to show the required distribution of objects, in particular Package, Interface and Component Diagrams.

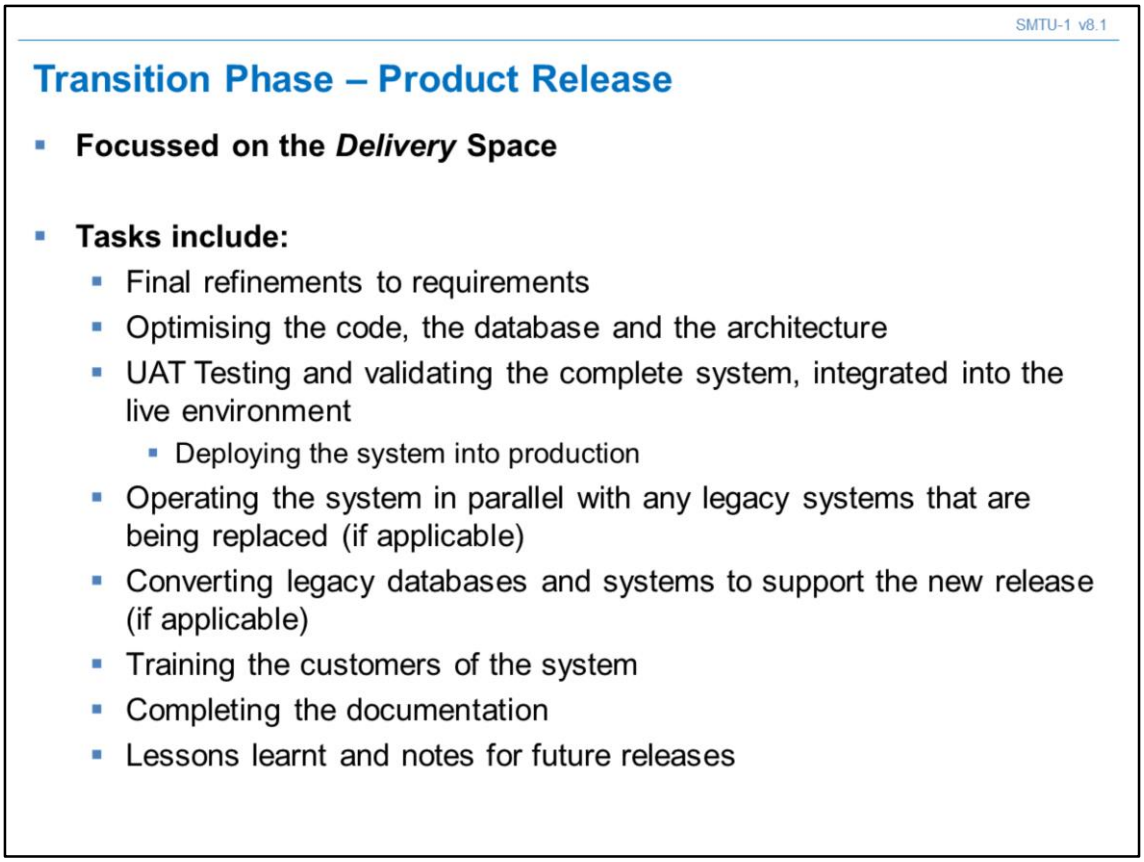
The combination of the SMTU and SDT courses cover the UML models used in Inception and Elaboration.



Construction is the largest phase in the project, in terms of effort. In this phase the remainder (around 80% of the code) of the system is built on the foundation laid in Elaboration. The target of this phase is to achieve *initial fully functional operational capability*, to a beta standard.

System features are implemented in a series of short, time boxed iterations. Each iteration results in an executable version of the software, which is evaluated for release. This phase is where an *Agile* approach can be most effective.

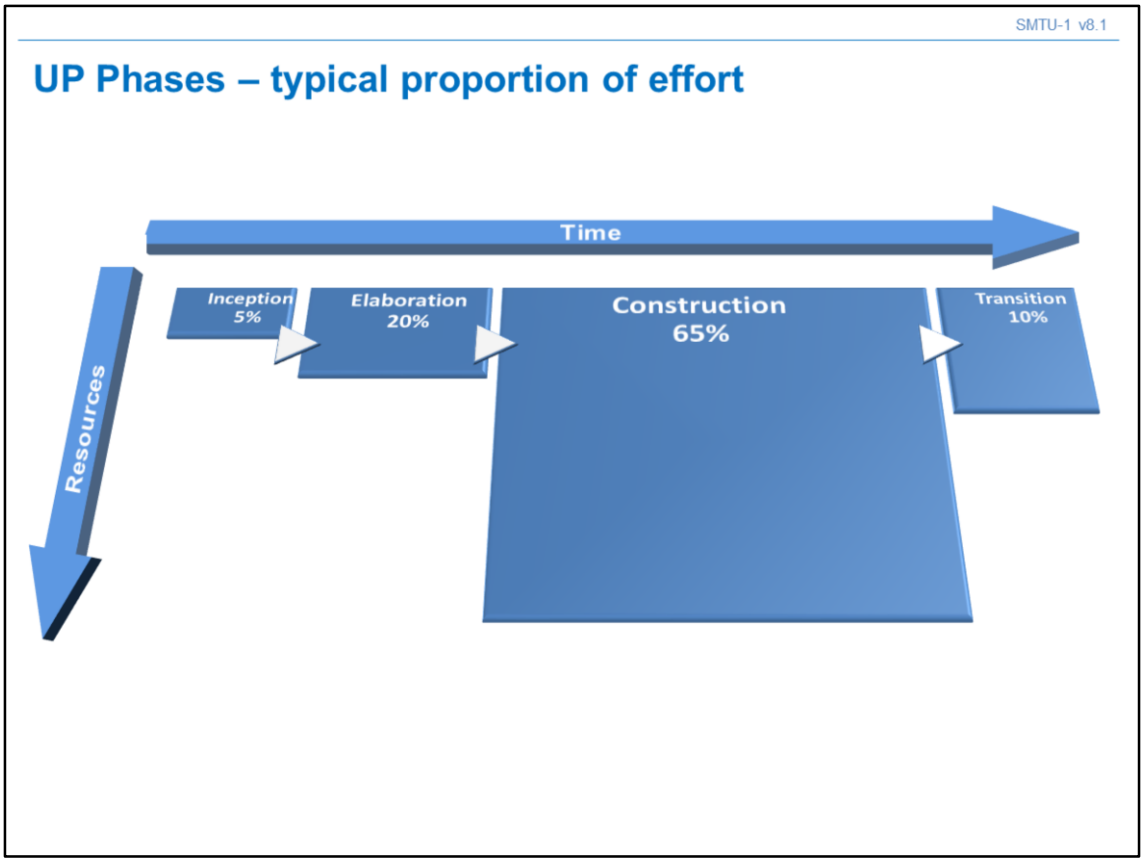
Any Use Cases not explored in Elaboration are investigated and documented, and indeed new requirements may well appear. The full range of UML diagrams may be used in this phase, many of which were produced in Elaboration (and may need updating).



The final project phase is Transition. In this phase the system is deployed to the target users and BaU is stabilised. The software is optimised, especially for its non-functional qualities.

Feedback is received from an initial release (or initial releases) and may result in further *refinements* to be incorporated over the course of a couple of Transition phase iterations. The Transition phase also includes system conversions and user training.

The UML diagrams most in use here are those concerned with configuration and deployment. These models are not covered in the BCS courses currently.

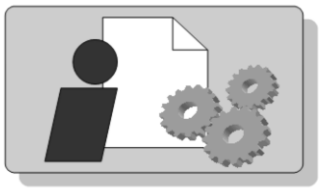


Profile of a typical project showing the relative sizes (man hours) of the four phases of the Unified Process.

SMTU-1 v8.1

UP: 6 Workflows

- Six Workflows are spread across the four phases
  - Business Modelling
  - Requirements
  - Analysis & Design
  - Implementation
  - Test
  - Deployment



- Each Workflow is a sub-process consisting of a set of related **Activities** that various project **Workers** perform to produce UP **Artefacts.** Together the Workflows create the Software Product

SMTU-1 v8.1

Workflows

Business Modelling	Modelling the context in which the software product will operate. One significant element here are business processes.
Requirements	Captures and documents the system requirements that address the context, via Use Cases.
Analysis and Design	Analysis and Design of the software that satisfies the requirements, using the prescribed architecture (*)
Implementation	Coding the software(*)
Testing	Testing, both static testing of documents etc. and dynamic testing of the code, including integration and acceptance testing (*)
Deployment	The specifications of the physical deployment of code and related resources into the live environment

(*) These workflows will require some modification if a package solution is acquired.

SMTU-1 v8.1

Characteristics of the Unified Process (UP)

- **UP is a…**
  - **Use Case driven, Risk focused**
    - UC express the Functional Requirements of the IT system
    - Concentration on early risk reduction
  - **Architecture-centric**
    - Early and continuing concentration on sound architecture
  - **Iterative**
    - Repeated cycle of steps, successively adding more and more detail or functionality
  - **and incremental**
    - Each iteration results in an increment, a step in the evolution of the final product

…approach to software development

SMTU-1 v8.1

Use Cases

- **Use Cases express the Functional requirements of the required system**
  - They are the services that the Product has got to provide, seen from the perspective of *all the users* of the Product
- **UP is Use Case driven**
  - The development approach is focused around Use Cases; analysis, design, implementation, testing…

SMTU-1 v8.1

Risk Focussed

"If you don't attack the risks they will attack you!"

- **Risks are events that could occur in the future that would jeopardise the success of the Project**
  - Risks should be identified and actions prescribed to deal with them
- **Risk focussed**
  - UP tries to de-risk the project as early as possible
    - Early analysis of critical Use Cases
    - Early exploration of feasible architecture
    - Focus on executable software

SMTU-1 v8.1

Architecture

- **Architecture describes the most significant elements in the software product, and how they relate to each other**
- **The 'Product' is *not* just the final code, it should include a number of other things too**
- **Architecture is represented by several *Views* in UP, each View deals with a sub-set of the whole Architecture**
- **Architecture should reflect Best Practice and often uses recognised Patterns**
  - Sound architecture will allow the Product to evolve in a low cost, low risk way

SMTU-1 v8.1

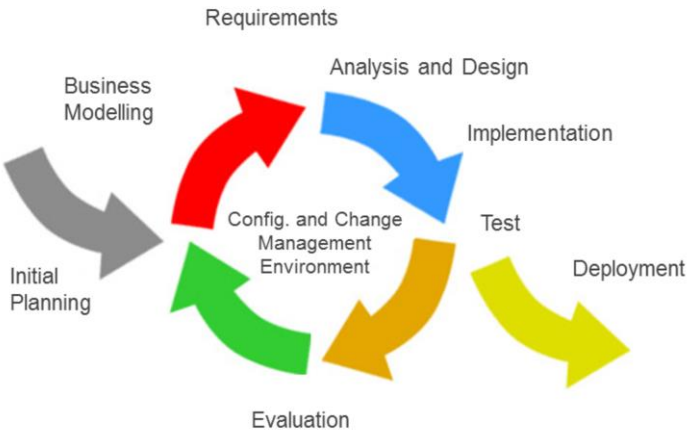
Form vs. Function

- **UC represent *function*, Architecture is the *form***
- **Intimately linked in a successful product**
  - UC are *realised* across the Architecture, but
  - Architecture must *support* the required functions (UC)
    - Now and in the Future!
  - Function, Best Practice and standard Patterns drive the architectural design
- **In UP, UC and Architecture evolve in parallel**

SMTU-1 v8.1

Iterative and incremental

- **Each UP phase consists of a number of *iterations***
- **Each iteration contains all the elements of a software project**



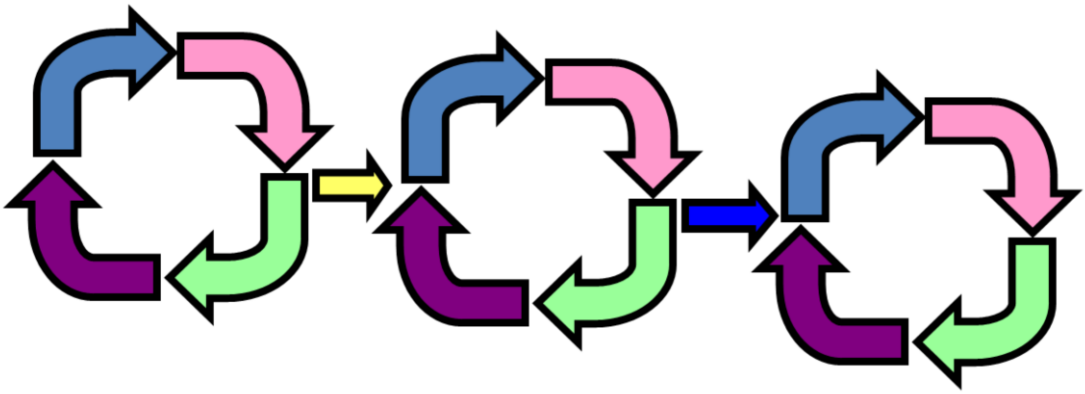
- **Each iteration develops an *increment* in the software product towards a *release***

Each iteration generates a baseline that comprises a partially complete version of the final system and associated project artefacts. Baselines build on each other over successive iterations until the finished system is produced. An increment is the difference between successive baselines, thus the Unified Process is both iterative and incremental. Phases may consist of several iterations.

SMTU-1 v8.1

Iterative

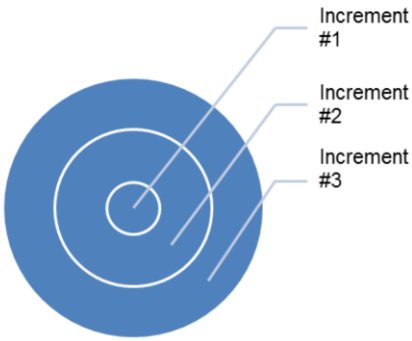
- **Repeated cycles through similar steps**
- **Improvement after each cycle:**
  - Improved quality and / or
  - Increased functionality / scope…

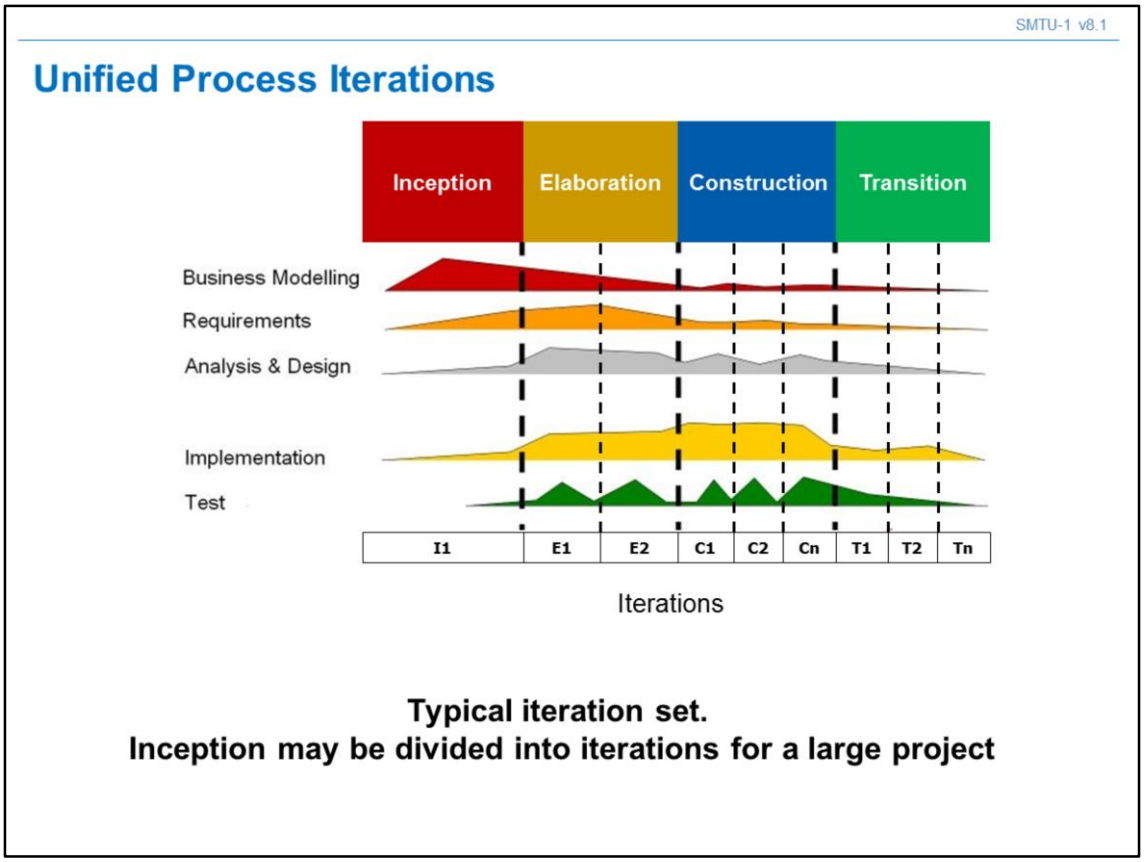


SMTU-1 v8.1

Incremental

- **Delivery of a whole by starting 'somewhere' and adding to it**
  - Each increment is of the *required* quality
  - Each increment is delivered by an iteration
  - Each increment is a baseline for internal or external release





Typical iterations are 1,2,3,2 – the number of iterations is planned at the end of every Phase.

Inception may be divided into iterations for a large project.

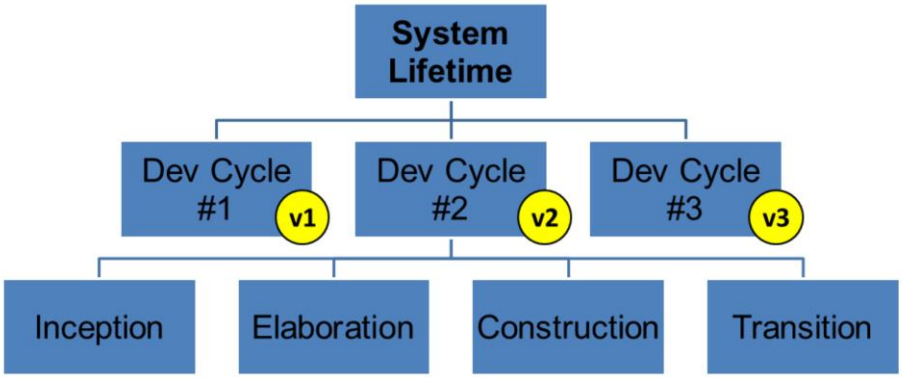
Some benefits of an iterative approach are:

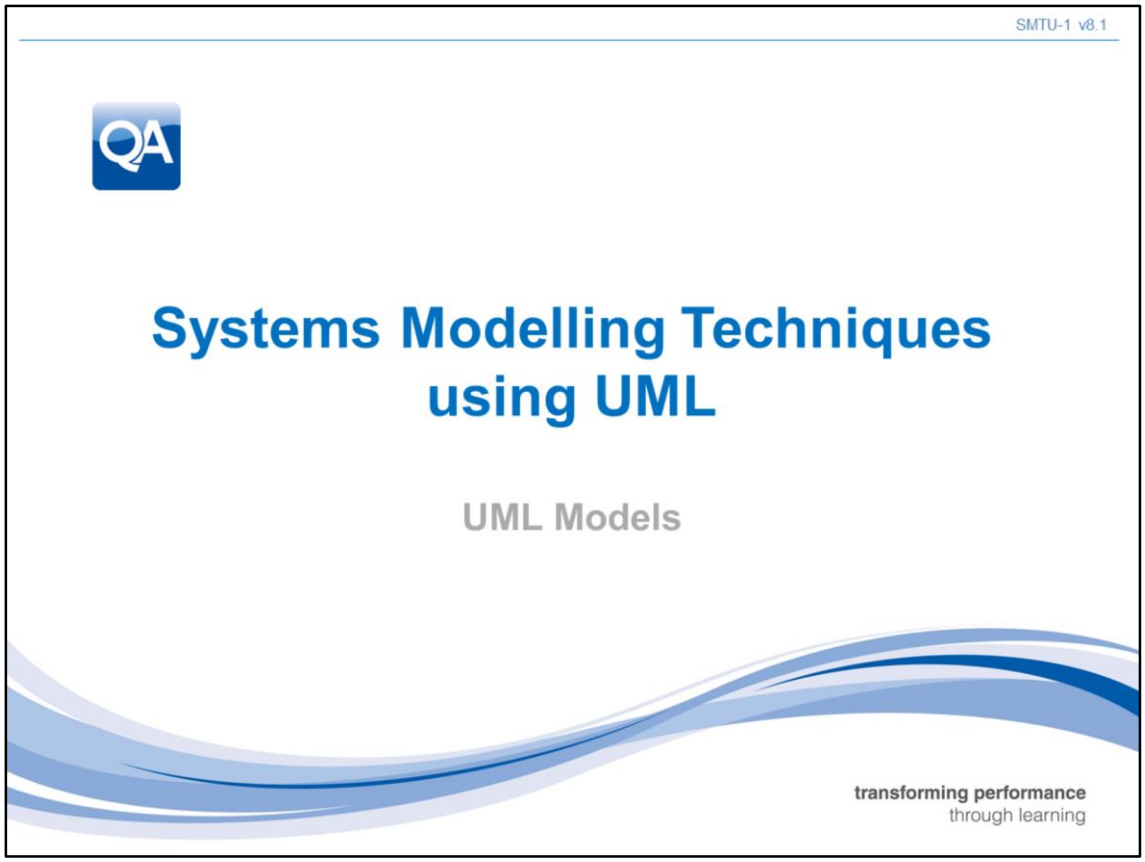
1. It tolerates changing requirements.
2. Elements are integrated progressively.
3. Risks are mitigated earlier.
4. It allows the organisation to learn and improve.
5. It facilitates reuse.
6. It results in a more robust product.
7. The process itself can be improved, and refined along the way.

SMTU-1 v8.1

UP: Lifecycle

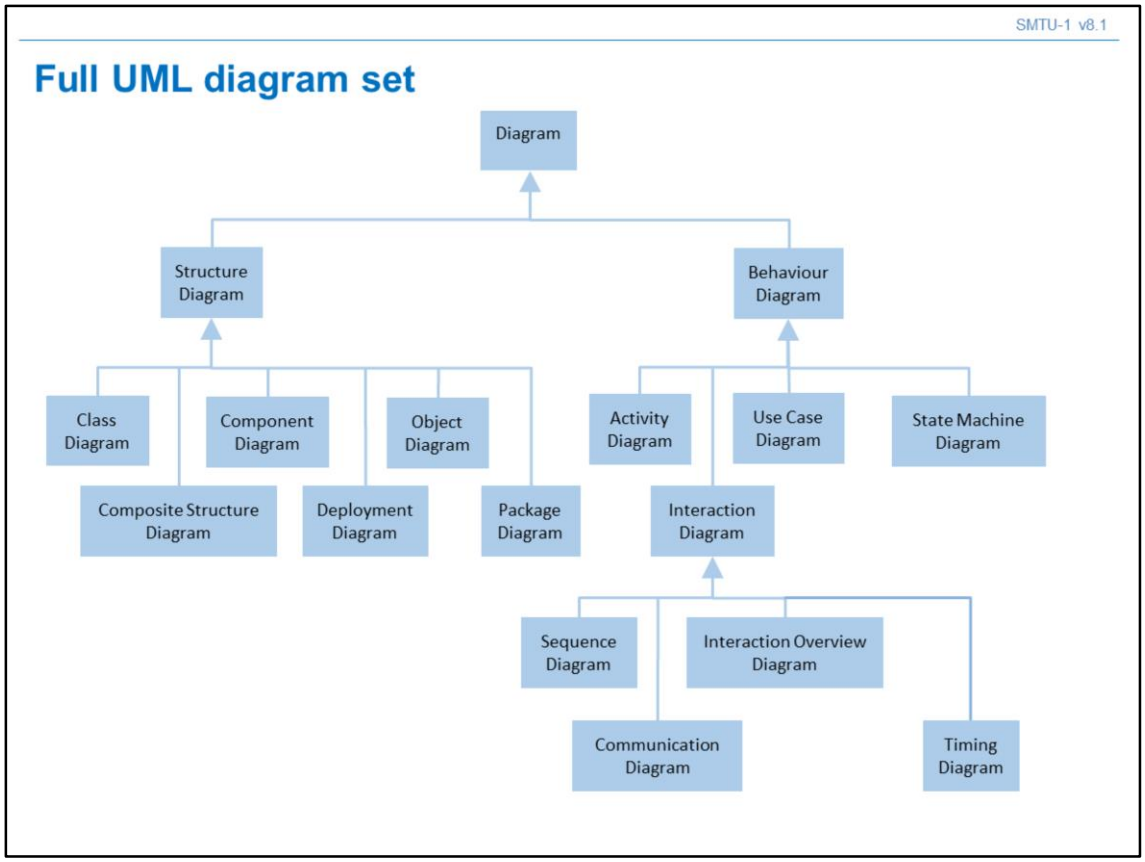
- **The Unified Process considers that the life of a software system can be represented as a series of development cycles**
  - A cycle ends with the (external) *release* of a version of the system
  - Each cycle consists of four phases




SMTU-1 v8.1
Systems Modelling Techniques using UML
UML Models
transforming performance
through learning

Role of UML

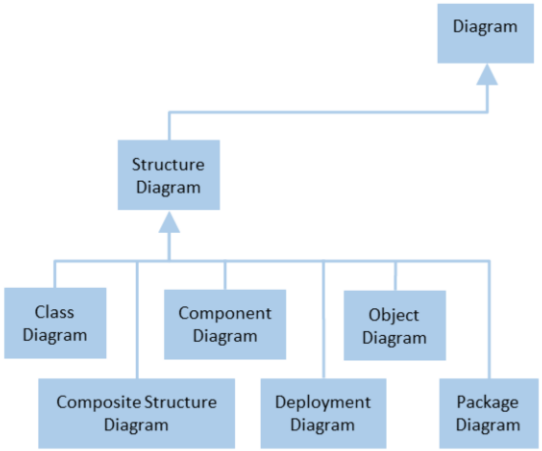
- **Models are useful abstractions about the software system**
- **Modelling a software system before building it is useful for many reasons …**
  - Captures and documents the problem and the requirements and the 'ideas' behind the software
  - Involves stakeholders early, facilitates communication and decision making
  - Reduces the risk of getting the wrong product
  - Saves time and effort during construction (less refactoring, blind alleys etc.)
  - …
- ***Visual modelling notations,* like UML, promote suitable levels of abstraction, and are easy for stakeholders to consume**
  - Models follow a syntax, reducing ambiguity
  - Models focus on different aspects of the software



The diagram shows the full set of models available with UML 2. Its purpose is to illustrate that this course does not cover the complete set, rather the sub-set concerning analysis and initial design.

SMTU-1 v8.1

Static diagram set

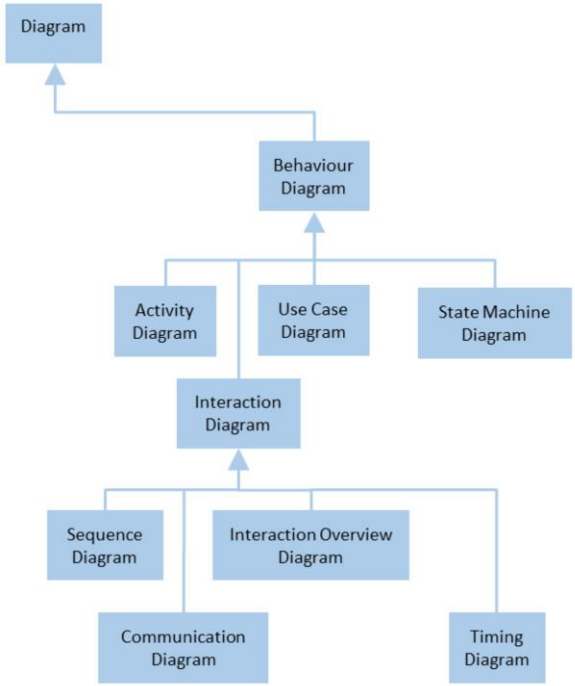


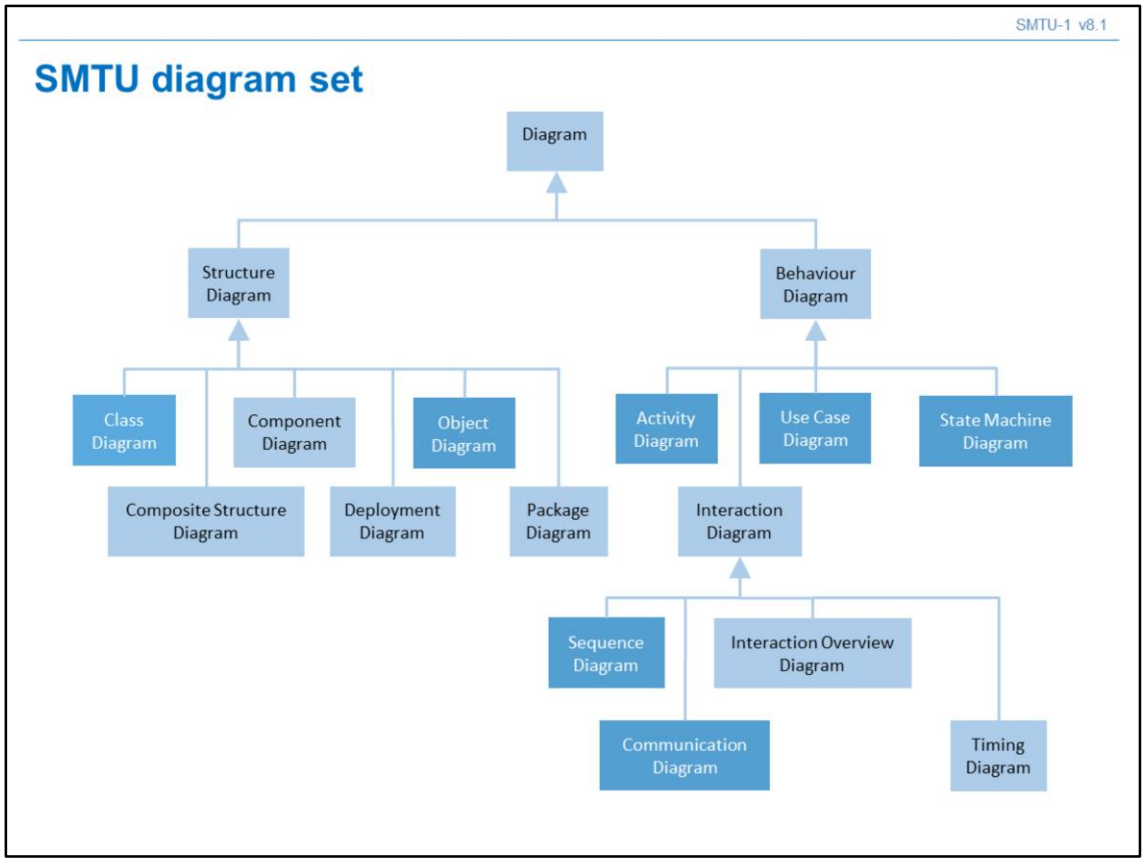
- **Models the static structure of the system's objects**
- **The Object specifications are fixed, irrespective of time – hence static**

SMTU-1 v8.1

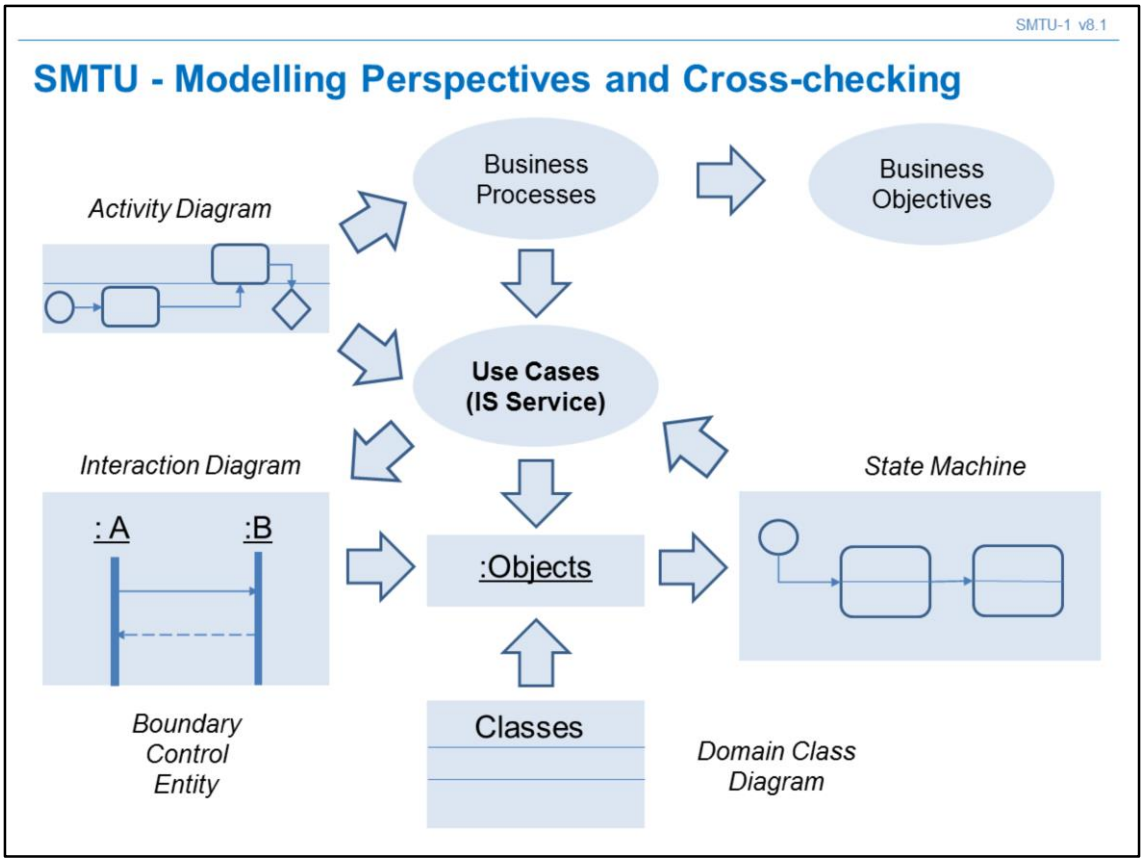
Dynamic diagram set

- **Models the dynamic behaviour of the system's objects**
- **Specifies the requirements and how the objects collaborate to realise them**





Diagrams within the scope of the course are highlighted.



All of these models, model the same software product, but from different perspectives.

Use Cases define the requirements for an automated information system in support of **Business Processes**. Business Processes are created to achieve **Business Objectives**. The logic of both Business Processes and Use Cases can be modelled using Activity Diagrams.

Use Cases manipulate **Objects**. Objects collaborate to achieve the goals of a Use Case. Objects are defined by their **Class**, which specifies their attributes (what they 'know') and their operations (what they can 'do').

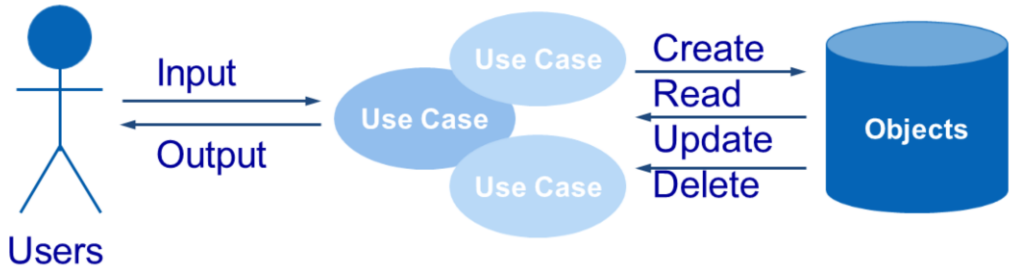
Objects that realise a Use Case may be modelled using an **Interaction Diagram (Sequence or Communication**). The distribution of objects and their roles should conform to a standard architectural pattern, and might therefore be grouped in layers such as Boundary, Control and Entity objects.

The characteristics of Classes from which Entity objects are derived may be modelled in a **Domain Class Diagram**. This diagram may be subsequently used as the basis for persistence design, using a DBMS.

The lifecycle of Entity Objects may be modelled using a **State Machine**. The events on this model should be cross-referenced to the set of Use Cases to ensured comprehensive coverage.

SMTU-1 v8.1

Use Cases and Objects – the core of UML

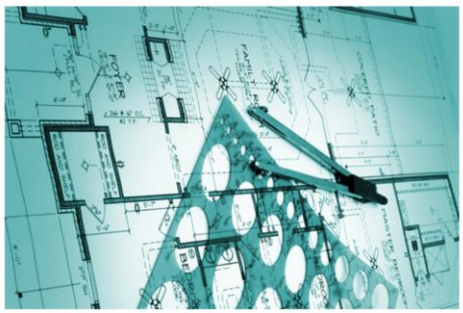


SMTU-1 v8.1

Example Scenario

- Travelling on business, employees have to pay bills (hotel bills, taxi fares, meals etc.) They keep their receipts as proof of payment to support their expense claims. Once a week they complete a claim form on a spread sheet and e-mail it to Accounts. They also print a hard copy of their claim form, attach the receipts to it, and post the form to Accounts. They mark their claim 'outstanding' and change this to 'paid' once payment is received.

- What can we model about this process and its software?



SMTU-1 v8.1

Modelling Dynamic Behaviour – Examples

- **Business Process**
  - Claim Expense Reimbursement
- **Events**
  - Expense incurred, End of the Week
- **Use cases**
  - Complete Claim, Send Claim, Print Claim
- **Changes in status of an object**
  - A claim is 'outstanding' and subsequently marked as 'paid'
- **Legal transitions between states**
  - It is legal for an 'outstanding' claim to become 'paid' but is it legal for a paid claim to become outstanding again?
- **Responsible stakeholders – 'Actors'**
  - Employee, Accounts

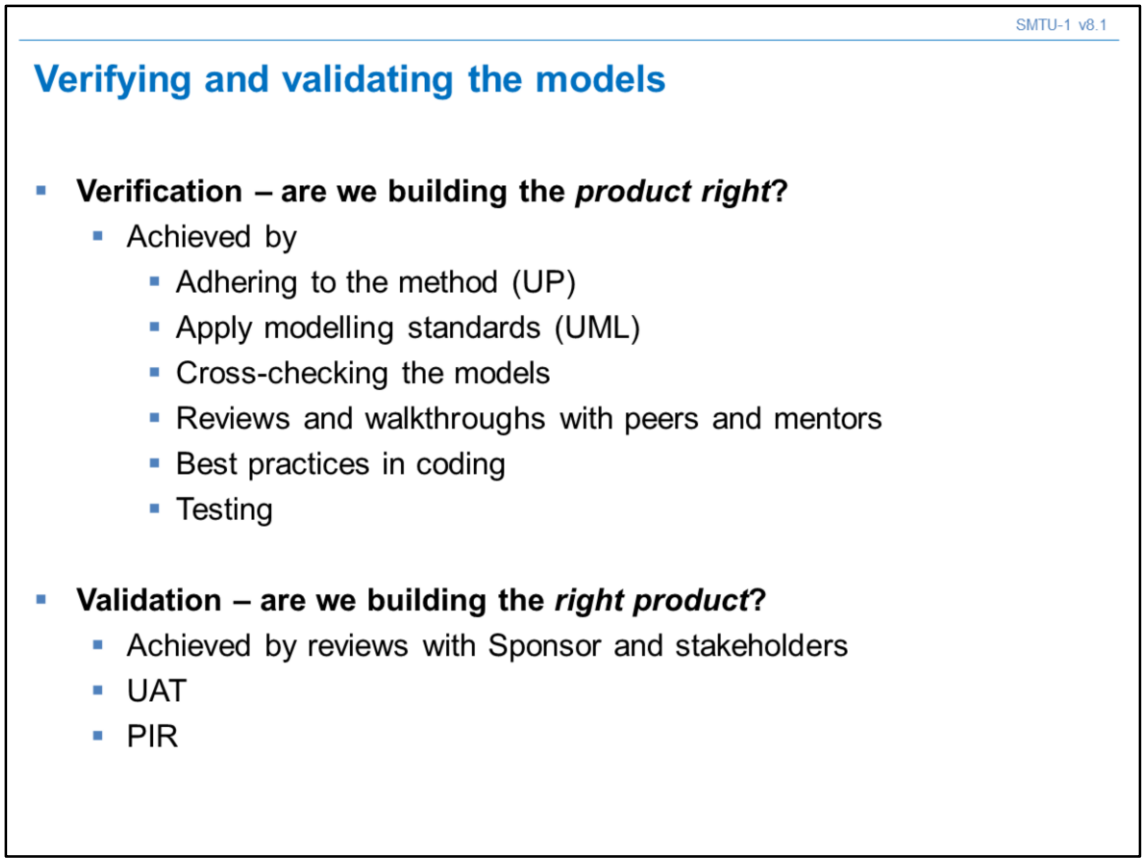
SMTU-1 v8.1

Modelling Static Structure - Examples

- **Objects -> Classes**
  - Receipt, Claim
- **Attributes of the Objects**
  - Date claim raised, total claim value…
- **Associations between Classes**
  - A Claim is supported by many Receipts

Typical use of SMT Models in UP

Phase / Model	Inception	Elaboration	Construction	Transition
Use Case	Core UC	80% UC	Remainder	Adjustments
Domain Classes	Core Business Concepts	Logical Data Design	DB Schemas	Adjustments
Activity Diagram	Business Process Logic	UC Logic	Operation Logic	Adjustments
Interaction Diagram	UC 'Robustness' checks. Storyboards.	Core UC Realisation. Architecture Layers. Component Realisation.	Remainder	Adjustments
State Machine	n/a	Entity Lifecycles	Adjustments	Adjustments



Many people find pictures easier to understand than words. It is easier to see the whole system in context on a diagram than pages of words. If there are rules, standards and best practice it means other people can understand and appraise the models we produce. This makes them easier to understand and easier to verify. Validation and verification are forms of testing. During Inception and Elaboration it takes the form of static testing – testing by inspection. In Transition it will take the form of UAT and Post Implementation Reviews – benefits realisation.

SMTU-1 v8.1

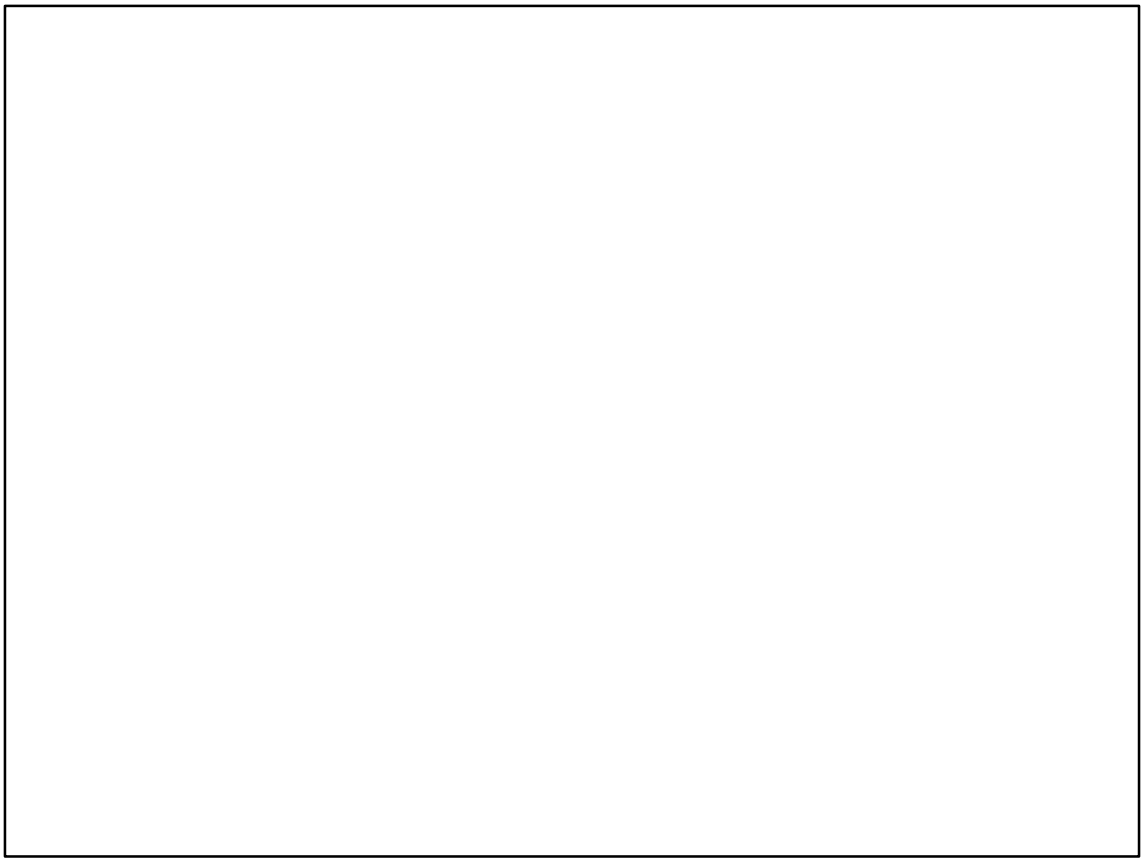
Summary

- Adopting a defined SDLC contributes to getting a quality software product
- Unified Process is a well known SDLC framework.
- UP makes use of UML models – visual modelling. There are 13 models available
- We can model the software system, using UML, from different perspectives
- The different models can be cross-checked, which aids validation and verification

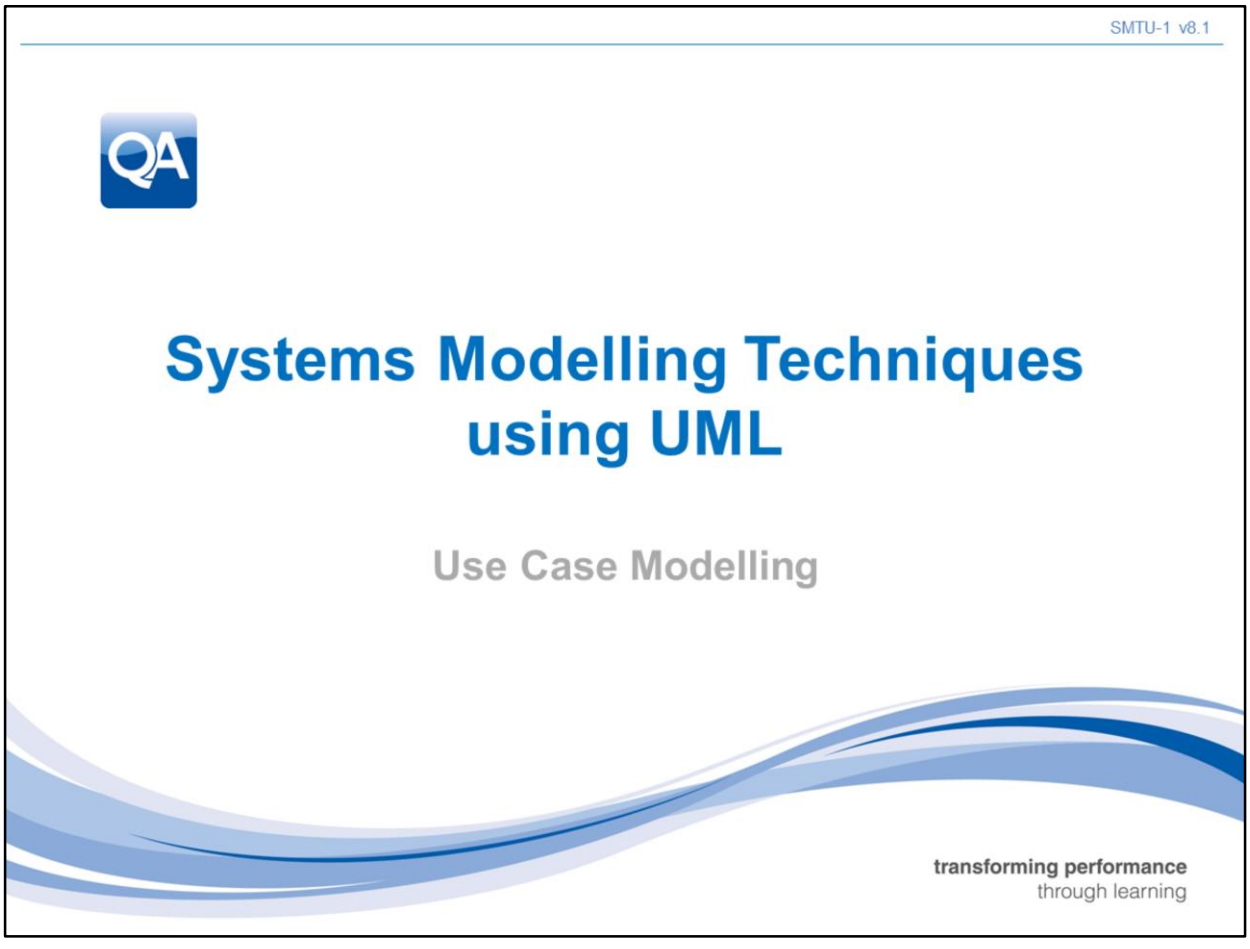
SMTU-1 v8.1

Activity

Read through the document describing the Library scenario. We will use this scenario to illustrate some of the models in the course.



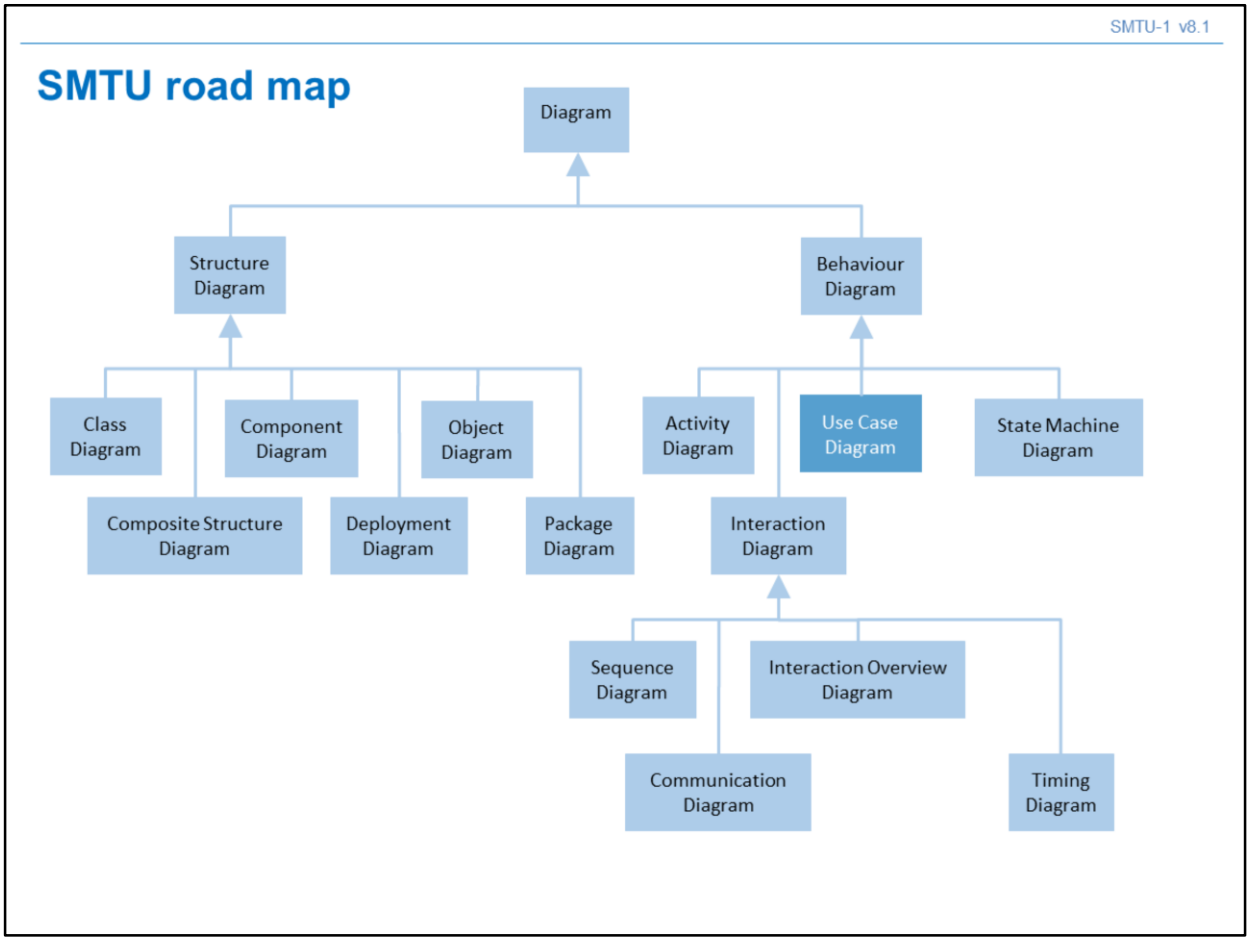
Intentionally blank.

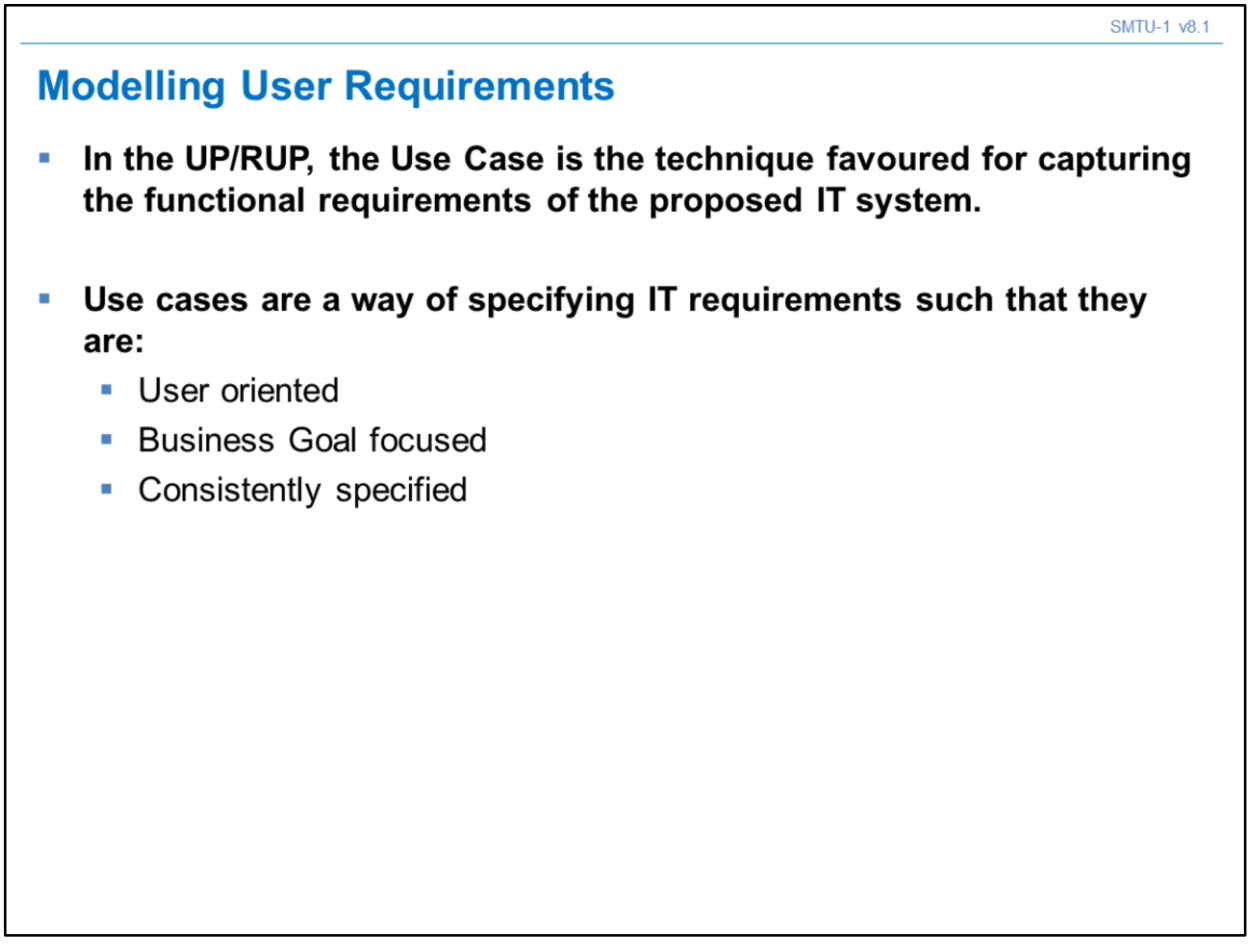

SMTU-1 v8.1
Systems Modelling Techniques using UML
Use Case Modelling
transforming performance
through learning

SMTU-1 v8.1

Topics

- Modelling user requirements
- Use cases
- Actors and the system boundary
- Use case diagrams
- Use case descriptions


SMTU-1 v8.1
SMTU road map
Diagram
Structure Diagram
Behaviour Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
Use Case Diagram
State Machine Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Interaction Diagram
Sequence Diagram
Interaction Overview Diagram
Communication Diagram
Timing Diagram



Remember, the UP is ***use case focused***; most of a UML-based specification is driven by the identification, analysis and design of use cases. The focus is on how ***Actors*** will want to interact with the software system while carrying out their part in business processes.

SMTU-1 v8.1

Use Cases

- **'Structured *functional* requirements' that describe 'What' the system must do but not 'How' it will be achieved**
- **A description of the dialogue between the user and the system to enable the system to meet the user's business goal**
- **Expected to deliver value to the user to further the aims of a business process**
- **The sum of the use cases is 'The System' (as seen by its end users)**
- **Must be named using a verb-noun phrase**

A use case is a 'structured functional requirement' that describes a piece of system functionality.

During analysis, use cases are descriptions of the dialogue between an actor and the system needed to meet the actor's goal.

In design, the use case description evolves into models (communication and sequence diagrams) that describe the internal operations of the components which are needed to realise the use case.

The use case is expected to deliver a result of value to an actor who is executing the use case in order to further the aims of some business process in which they are a participant.

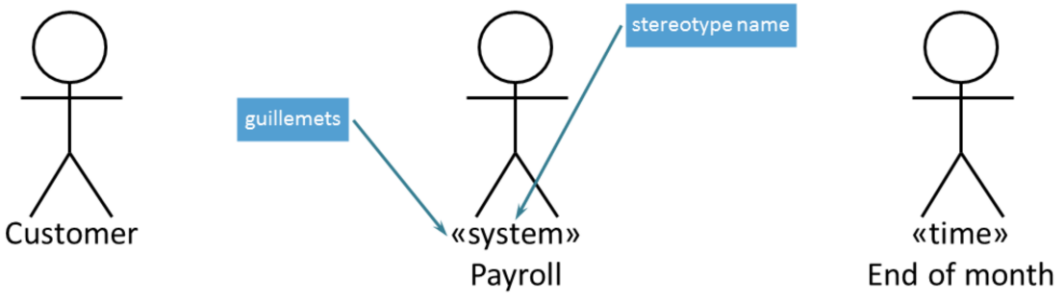
All use cases are, by definition, within the scope of the system. Indeed, the sum of the use cases is "The System".

The naming of use cases is important; use cases should be named using a verb-noun phrase such as 'Create Purchase Order', 'Accept Delivery' etc. because they are essentially *functional requirements*

SMTU-1 v8.1

Actors and the System Boundary

- **Actors represent named user roles that interface with the system**
- **Could be:**
  - Human
  - Non-human, another system, or time
- **Actors have some 'responsibility' with respect to the Use Case**
  - E.g. provide input to the system
- **Actors are *external* to the IT system under study and therefore define the system boundary**



An actor is an external entity performing a named role with respect to the system:

The role requires participation in one or more use cases

An actor may trigger a use case, provide input to the system and/or receive output from it

An actor is often a role played by a human

However, an actor can be any mechanism interacting with the system including other computer systems or devices or simply time passing

An actor interacts with the system, providing input such as:

Starting a use case

Confirming an action

Selecting an option

Starting a communications session

Passing sensor data

Tripping a timeout.

SMTU-1 v8.1

Use Case Diagram

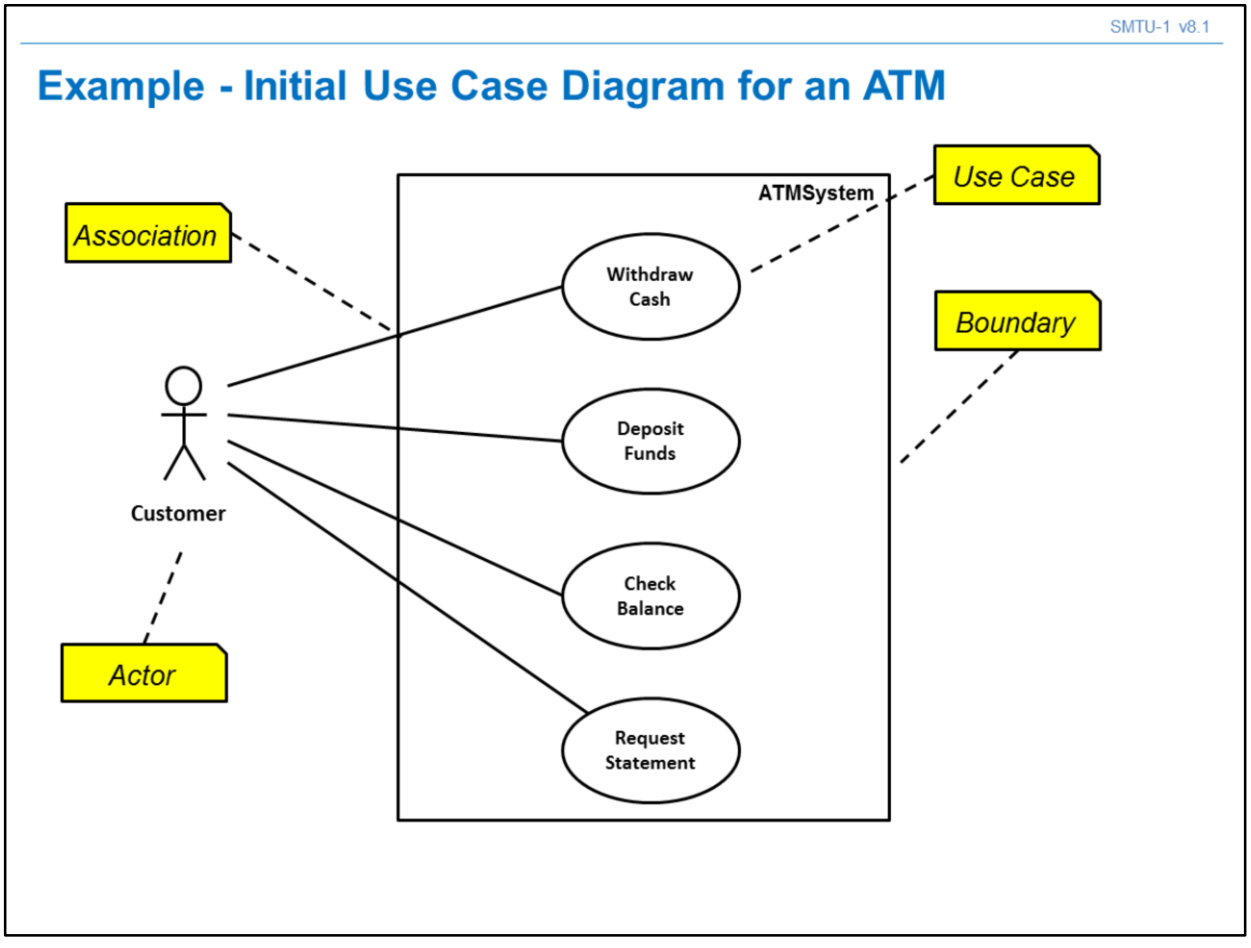
- **Clearly and unambiguously defines the scope of the project**
- **The three main symbols are Actor, Use Case and the Association that says:**

"This actor is *associated* with this use case"

Use case diagrams (UCD) are very simple documents that define the scope of the IT system visually.

Note that a UCD is ***not***:

- A Data Flow Diagram
- A Flow Chart



In this example of a banking system, use cases define the interaction that takes place between customers and automated teller machines (ATMs).

The Customer actor represents all customers who will use the ATM. When you use your local ATM to withdraw cash, you are an instance of Customer using a particular instance of the use case Withdraw Cash. The person standing in line behind you is another instance of Customer, who will use a different instance of the use case Withdraw Cash. Someone else may use an instance of the use case Check Balance or Request Statement. You may successfully withdraw cash from the machine, but the person behind you may find that he or she does not have enough money deposited, and the use case instance will proceed along a different course from yours, rejecting the request.

The term *scenario* is often used to refer to the different possible courses that different instances of the same use case might take. The use case diagram names only the use cases, additional documents or diagrams will show the alternative scenarios that could occur within a given use case.

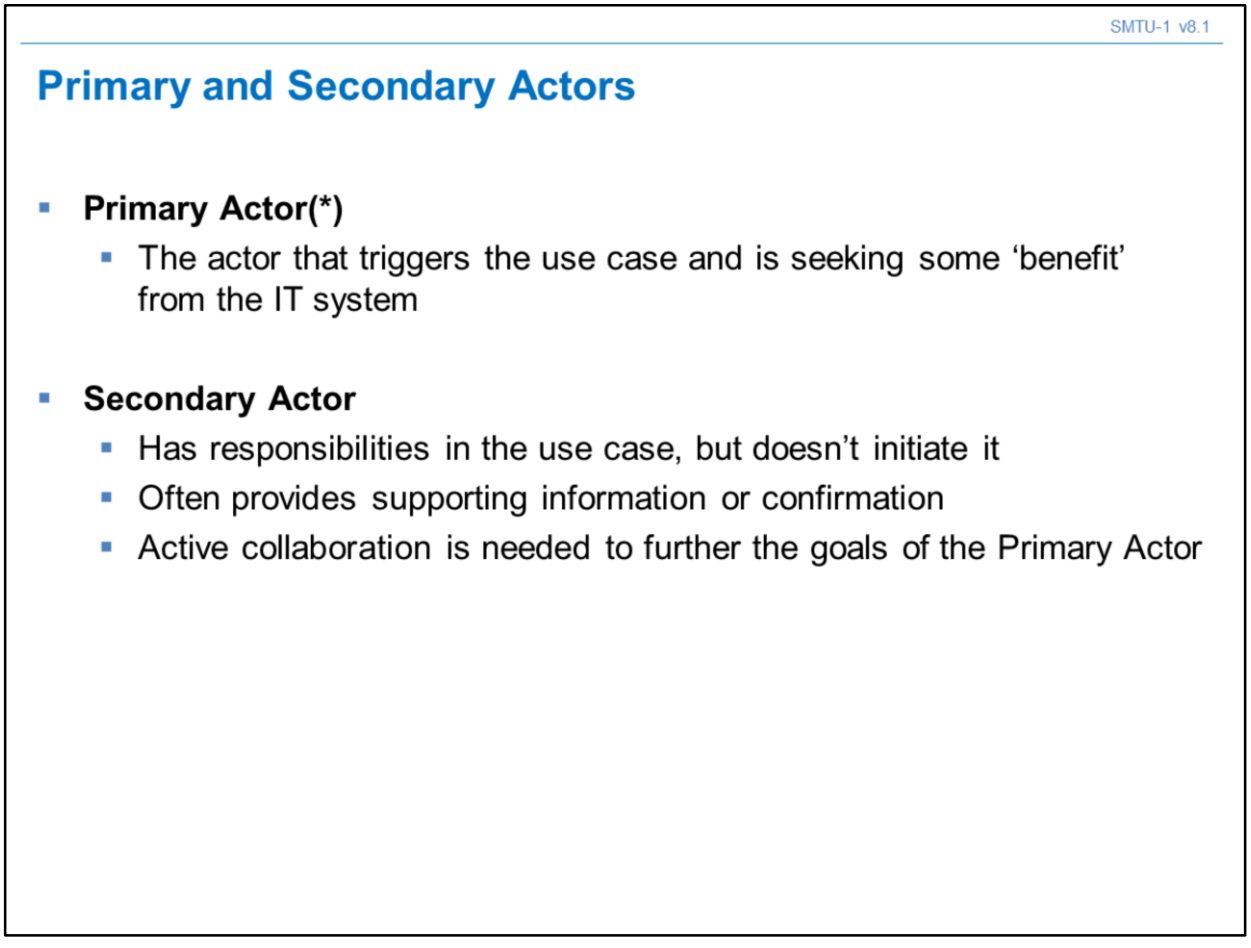
Further analysis will uncover other actors such as the bank staff that replenish the cash, the gateway to the LINK system…

SMTU-1 v8.1

Focus of Use Cases

- **Always described from the actor's point of view**
- **Yield a result of value for the actor**
  - "in a single sitting"
- **Support activities in business processes**
- **Describe only interactions with the IT system and not between actors**
- **Should not be confused with business processes**
  - The use case is a piece of IT functionality
  - A business process is an end-to-end set of business activities, some of which may be supported by use cases

Use cases describe *only* the interactions required with the IT system and *not* between actors like talking to a customer, writing a report etc.



When performing use case modelling, one of the first tasks is to identify the Primary actors – what 'entities' will want to interact with the new IT system?

Secondary actors are actors that participate in the use case but are not the initiators of the use case's execution. They often provide supporting information or confirmation. It may or may not be convenient to include them, depending on the diagram's audience.

(*) The distinction between primary and secondary actors may be useful, but it is not mandated in UML.

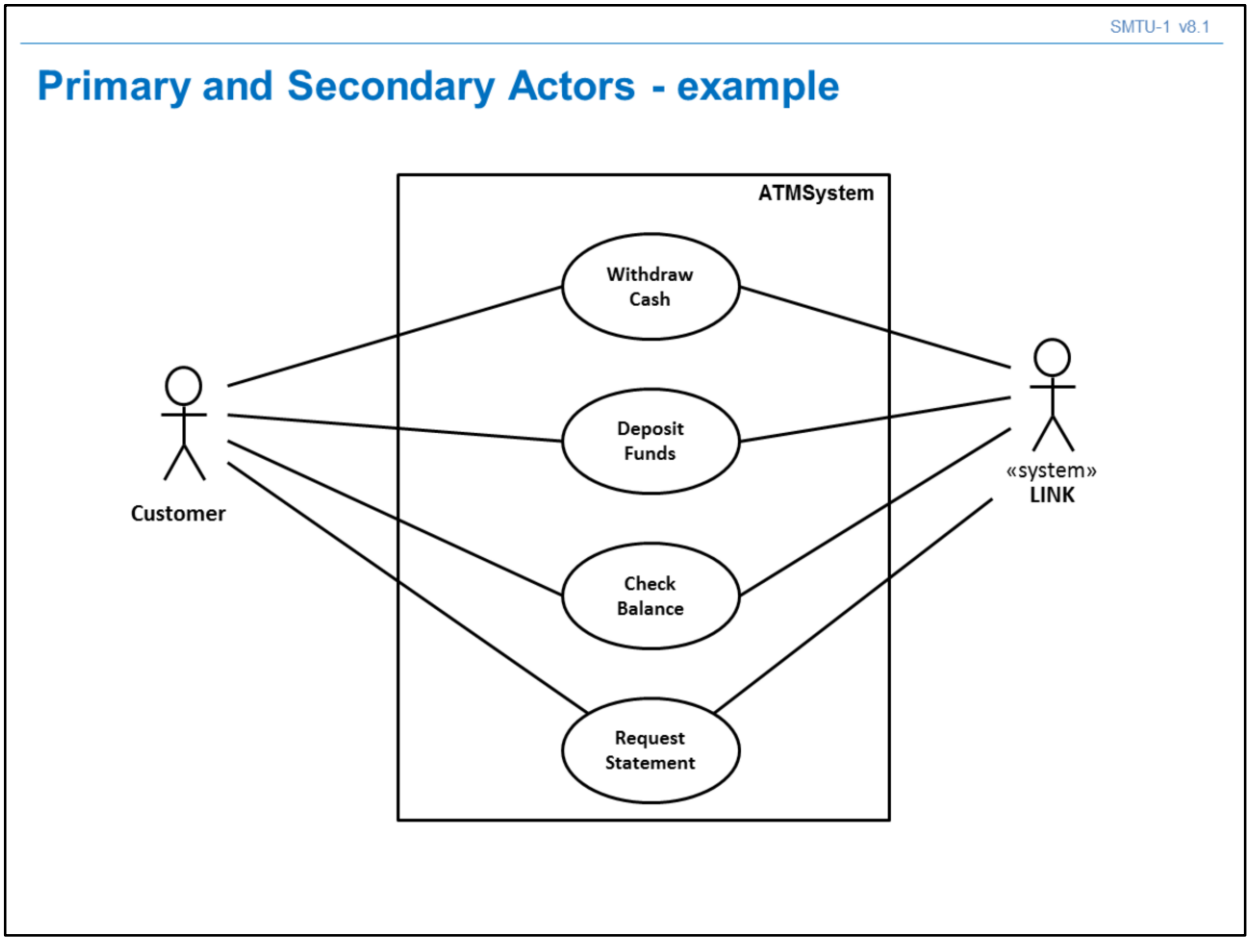
In a 2006 article Cockburn defined Primary and Secondary Actors:

Primary Actors: The Actor(s) using the system to achieve a goal. The Use Case documents the interactions between the system and the actors to achieve the goal of the primary actor.

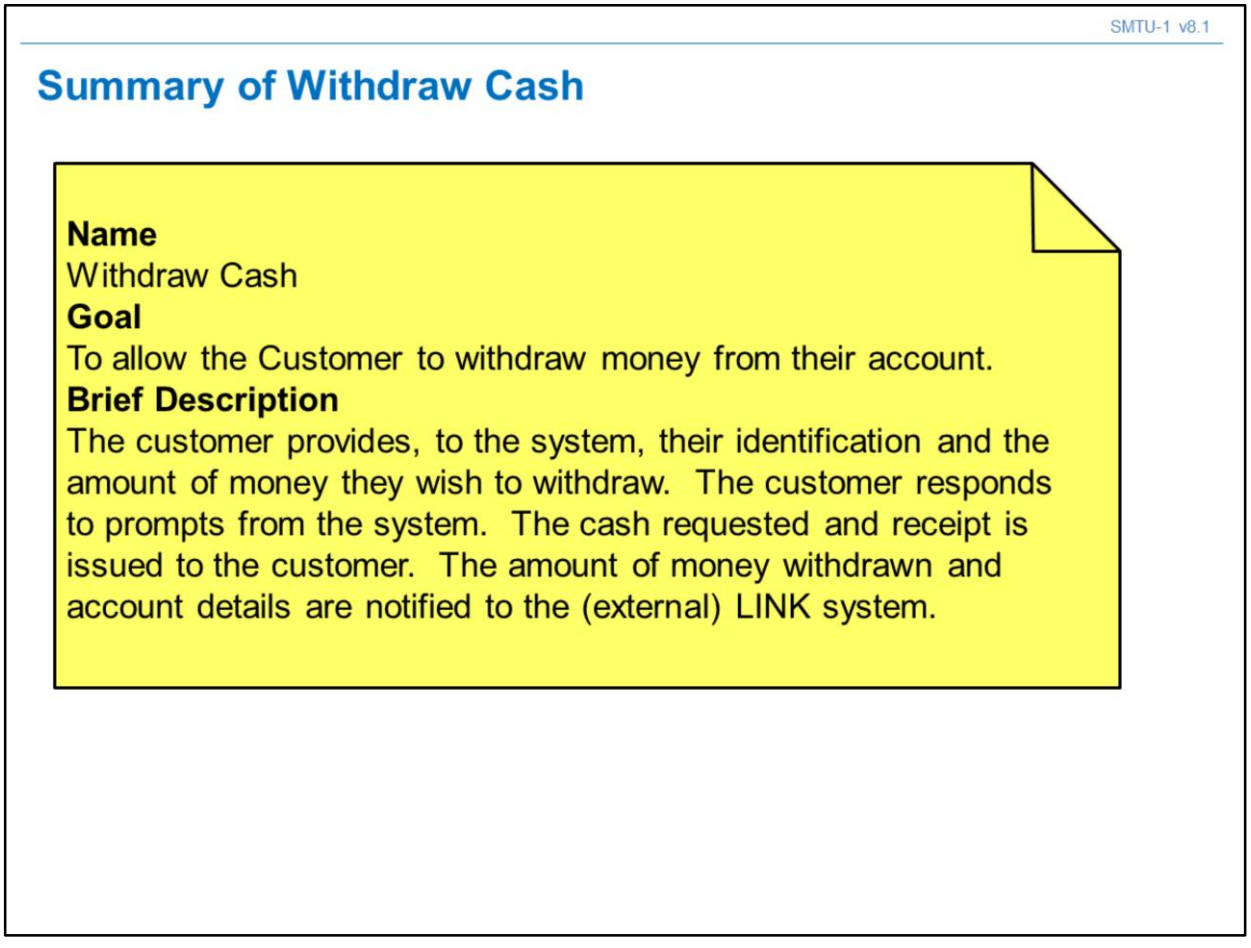
Secondary Actors: Actors that the system needs assistance from to achieve the primary actor's goal.

The designation of a secondary actor will be obvious in some cases (e.g. a credit check agency). Others may be less obvious or maybe irrelevant; for example an actor simply

receiving an email or report seems like a very passive involvement in achieving a goal!



UCD with a Secondary Actor included.



A UC diagram shows a summary of the use cases required, the scope of the required IT system. However simply having a set of verb-noun phrases isn't really enough for stakeholders to understand what is being specified. It is customary therefore to have *at least* a simple text like the one above available to describe each function.

This level of documentation is above all associated with the Inception Phase. This text will provide the basis for developing a fuller description of each UC using a recognized template, mostly in the Elaboration Phase. We will look at examples of the fuller text later in the course.

SMTU-1 v8.1

Workshop

- **For a public lending library, potential…**
  - Actors?
  - Use Cases?
- **Focus only on the immediate goals of the actors at this stage**
  - What do the actors want to do with the IT system?

Using the Library Vision Statement, develop a simple use case diagram to scope the library system. The focus is on lending items (books, DVDs etc.) not the HR aspects of the library.

- **Intentionally blank**

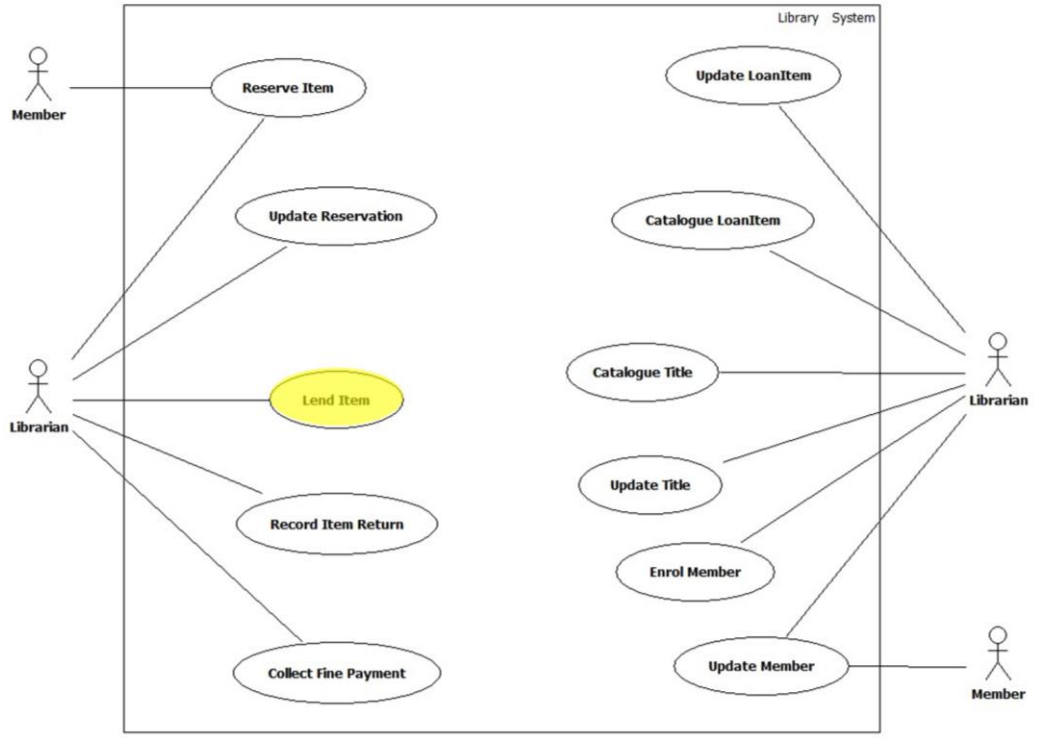
SMTU-1 v8.1

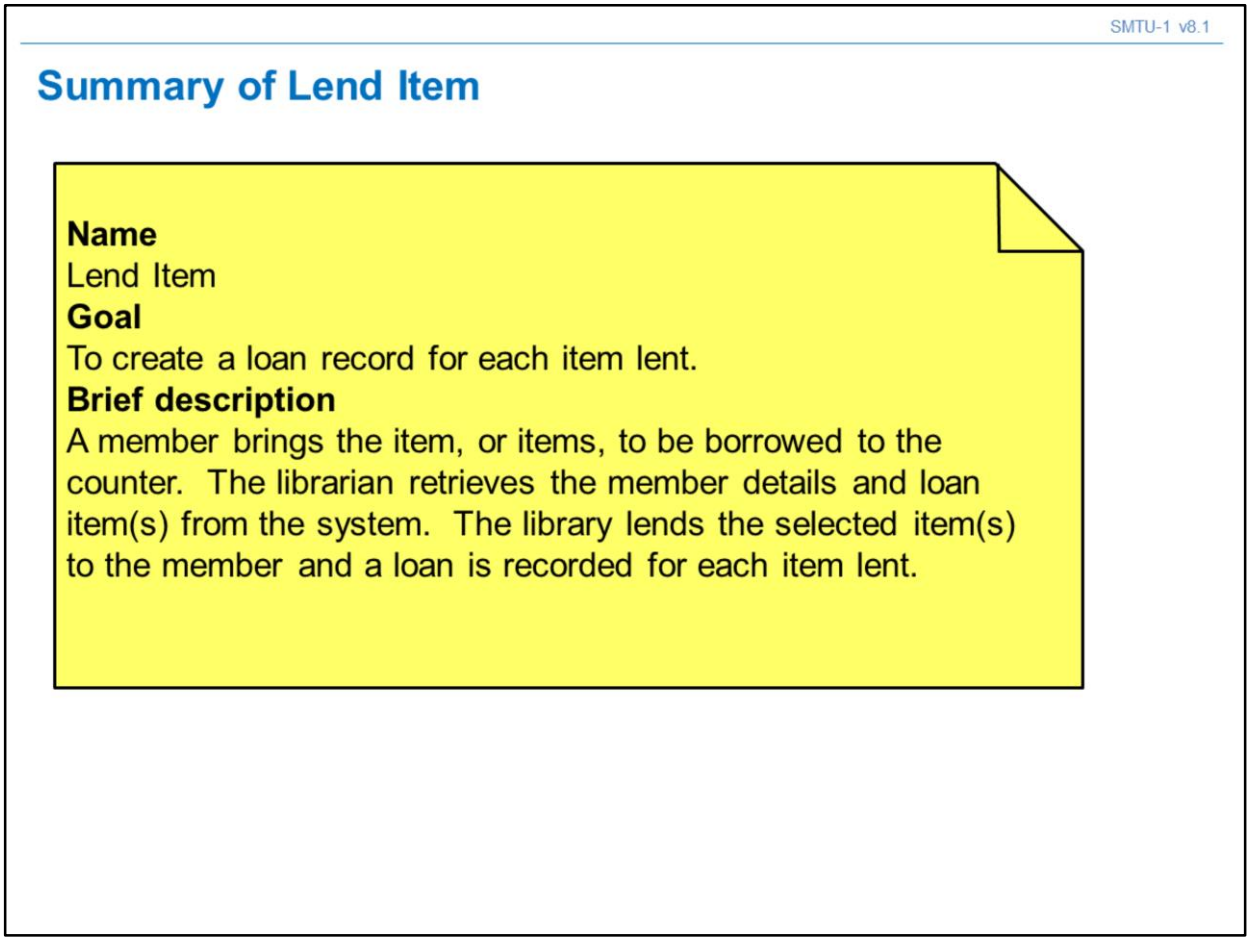
Use case diagram for a lending library

- **Potential actors**
  - Librarian
  - Member

- **Potential use cases**
  - Lend item
  - Record item return
  - Reserve item
  - Update reservation
  - Collect fine payment
  - Catalogue loan item
  - Catalogue title
  - Enrol member
  - Update loan item
  - Update title
  - Update member

Use case diagram for a lending library





This adds a little more information to the verb-noun construct of the use case name. It doesn't need to be much at this stage, just enough to demonstrate an understanding of how this bit of functionality should work.

This format has nothing to do with UML, which doesn't specify how to document a use case. However something along these lines is conventional.

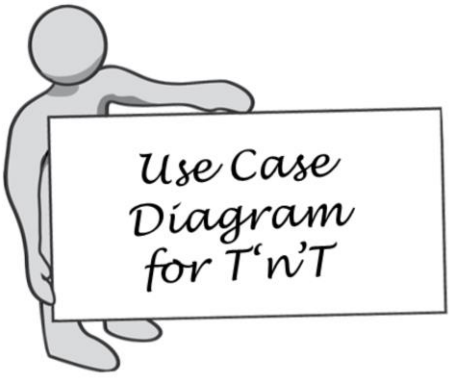
SMTU-1 v8.1

Activity

As guided by your instructor, produce a use case diagram from the Course Booking scenario.

SMTU-1 v8.1

Case study

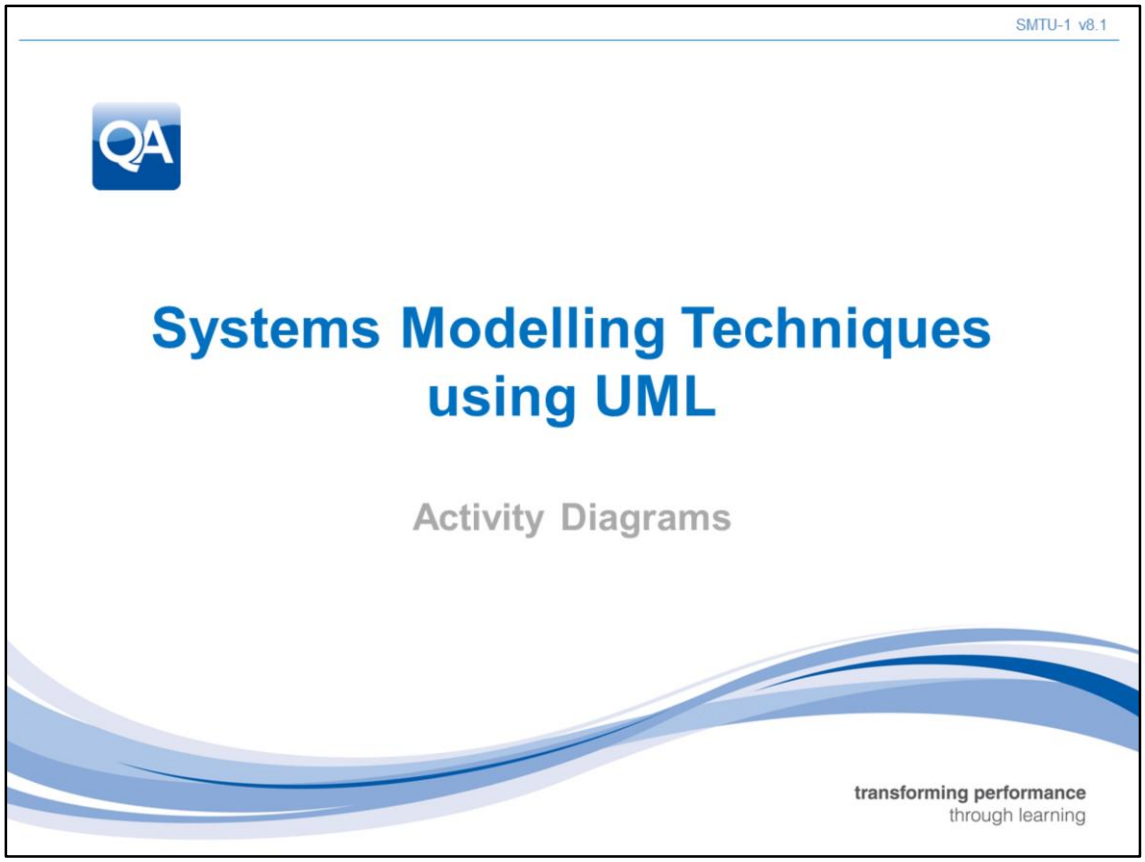


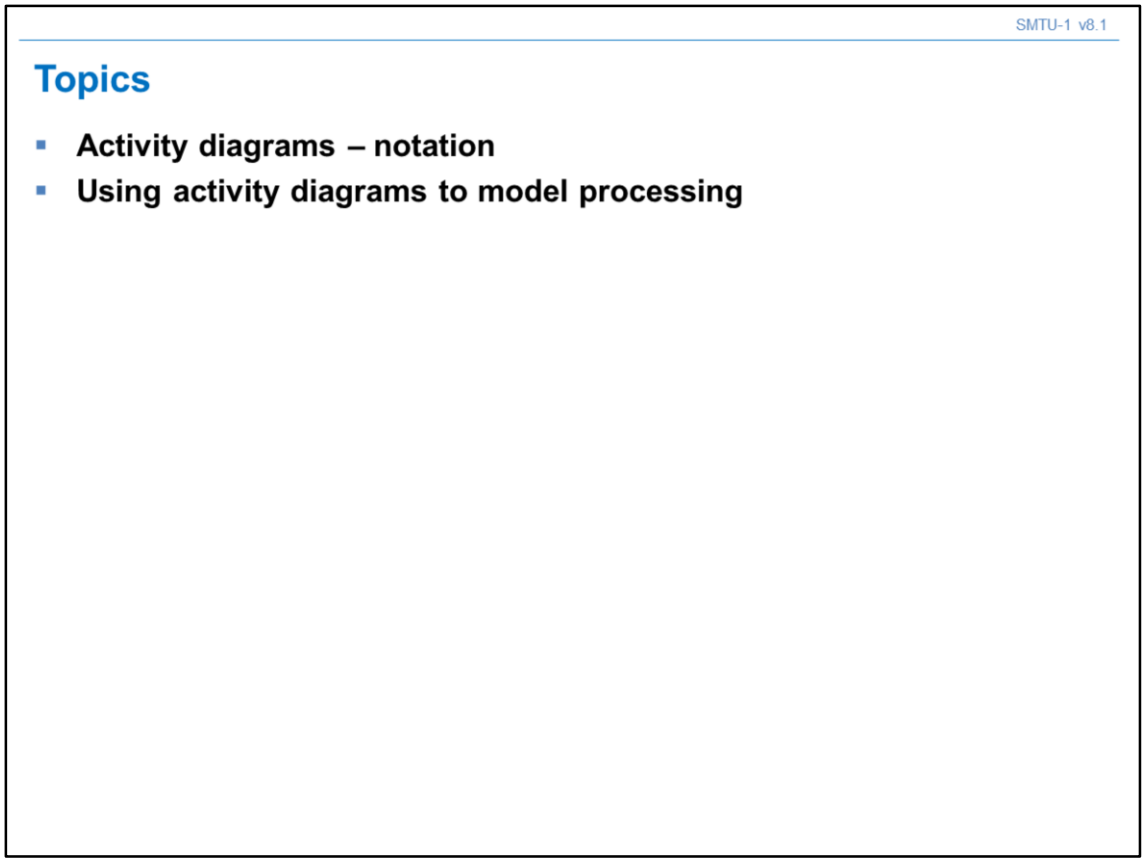
As guided by your instructor, produce a use case diagram for the Stores Project for Taps 'n' Traps from the vision statement.

SMTU-1 v8.1

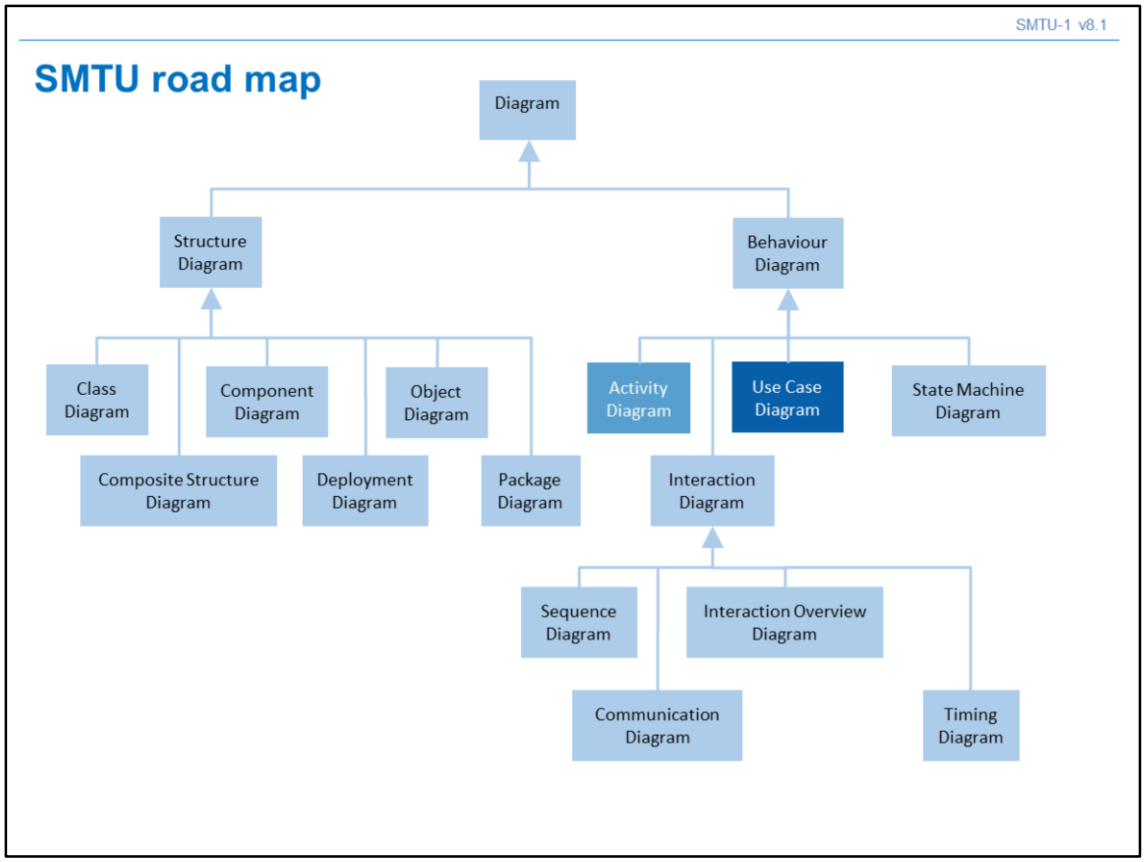
Summary

- Modelling system functional requirements as use cases
- Use case diagram elements
- Use case short specification for main scenarios


SMTU-1 v8.1
Systems Modelling Techniques using UML
Activity Diagrams
transforming performance
through learning



Activity diagrams model the flow of control of some procedure of interest. They are the UML equivalent of a flowchart but can show a lot more than just the sequence of actions. In this course we will look at how to use them to model business process flow and as support for the textual use case description.

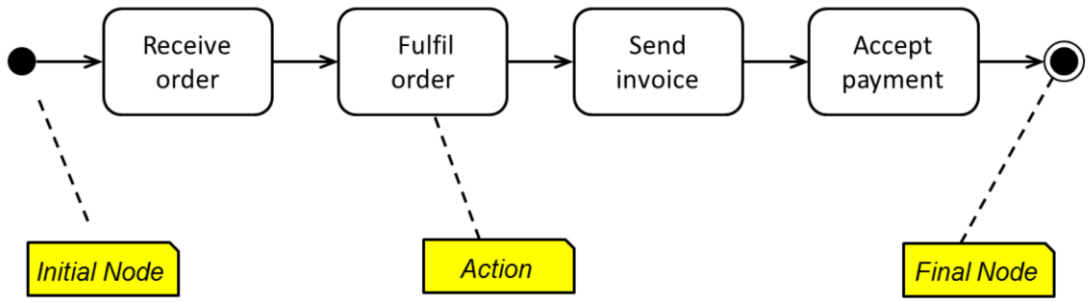


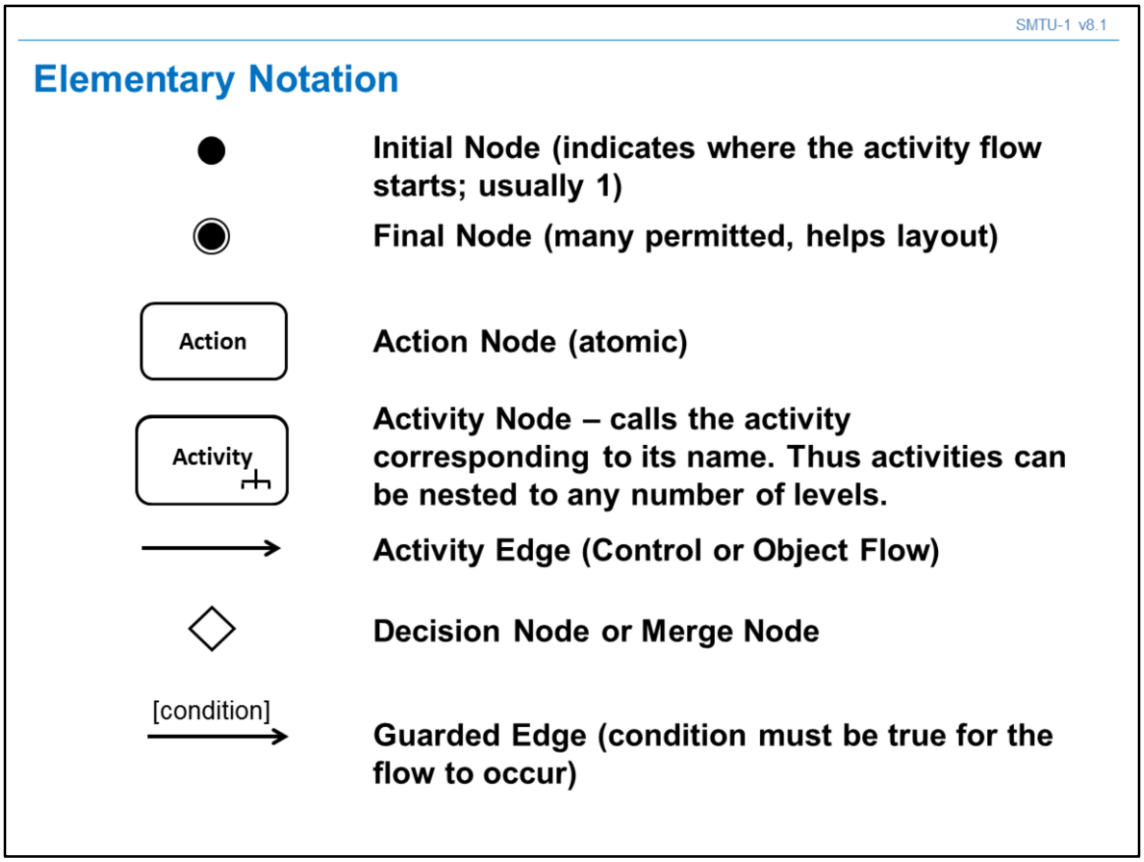
In this first session on activity diagrams the focus is on modelling business processes. In a later module we will examine how activity diagrams can be used to model the internal logic of system use cases.

SMTU-1 v8.1

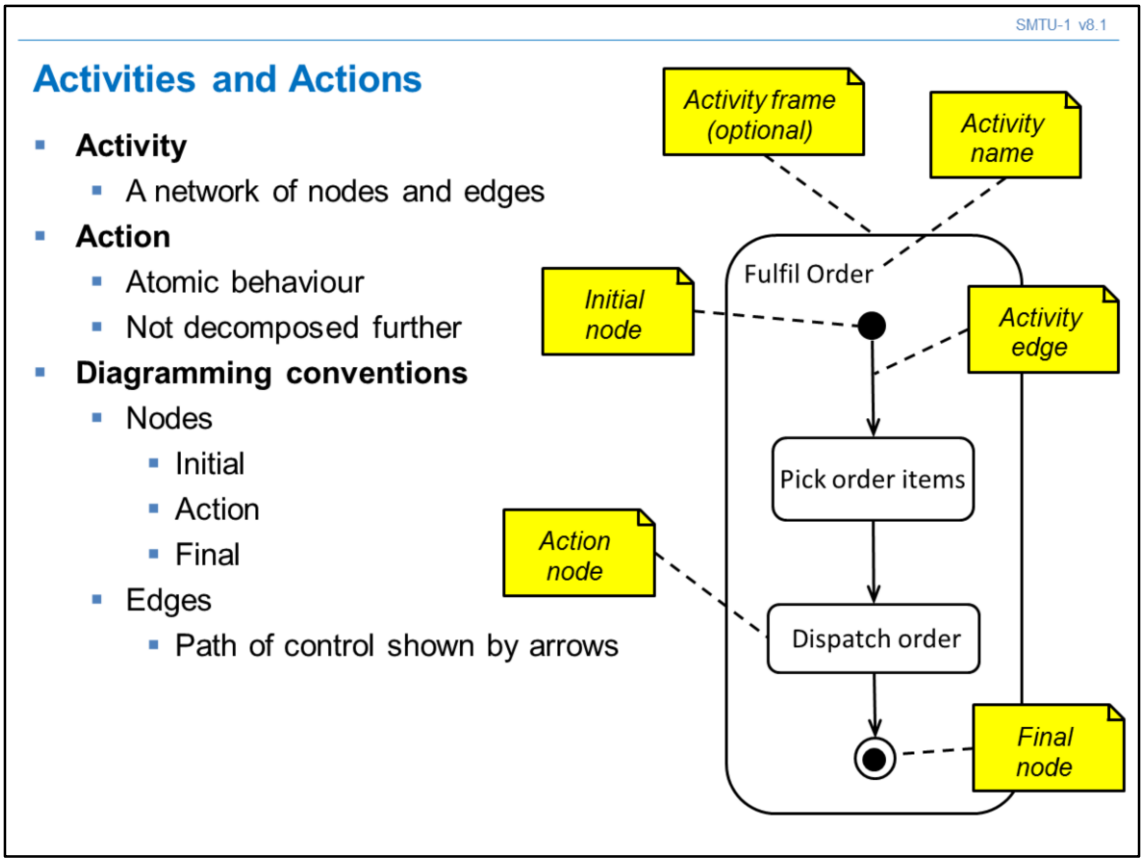
Activity diagram

- **Activity diagrams describe the flow logic of a *network of actions***
- **Can be used to describe workflows**
  e.g. Business process flows in varying levels of detail
- **Can also be used to support Use Case definitions**

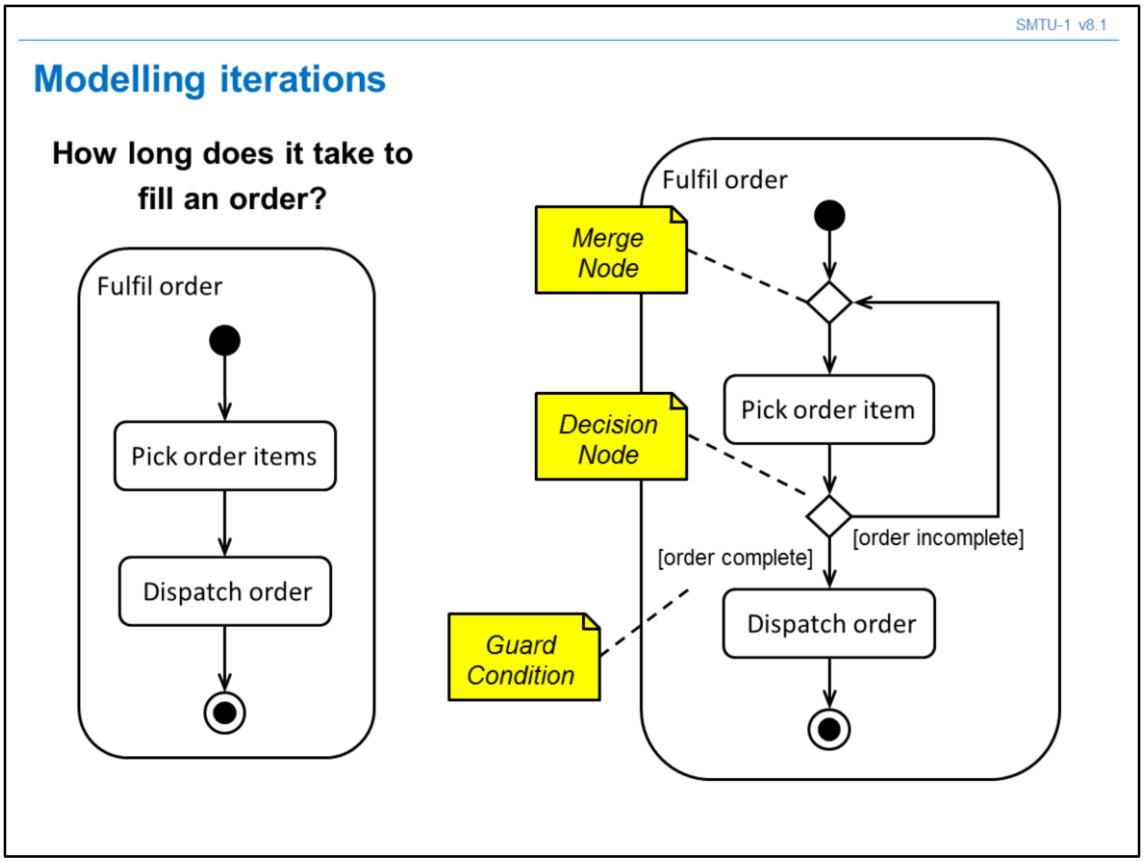




Illustrated are some of the basic notational elements.



Note the use of the UML ‘note’ to provide additional information. This can be used on any UML diagram, anywhere.



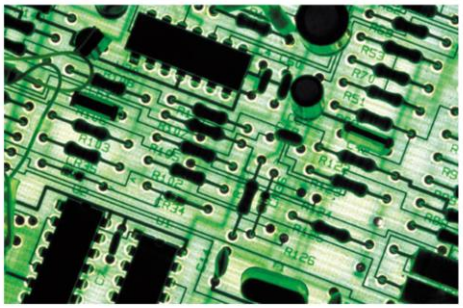
Question: How long does it take to fill an order?

Answer: It depends on how many items are being ordered! To clarify that “Pick order Items” may be repeated, it could be shown as an iterative action. On the other hand that detail could be hidden within the action – it is up to the modeller, taking into account the audience for the model.

Note the need for a ‘merge’. 2 or more flows directly *into* an action signifies a logical ‘and’ in UML – this is most inconvenient!

SMTU-1 v8.1

Network of Actions



UML sees an Activity as a 'network of Actions'.

The network is notionally traversed by 'Tokens', rather like designing a circuit board

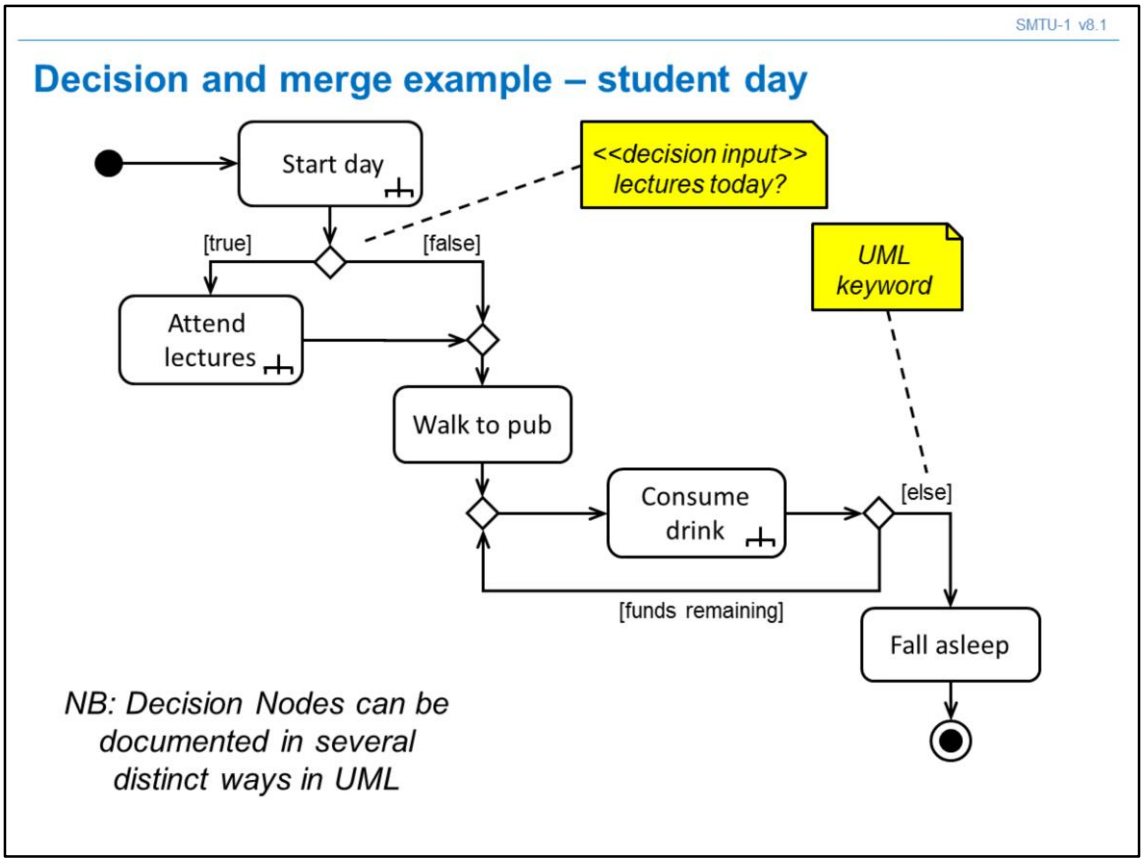
The modeller must be clear on what each instance of the activity being modelled represents.

For example imagine modelling the processing of Sales Orders. Does each Activity instance represent *each* Sales Order, all Sales Orders *in a month*, all Sales Orders *for ever*?

SMTU-1 v8.1

Idea of a 'Token'

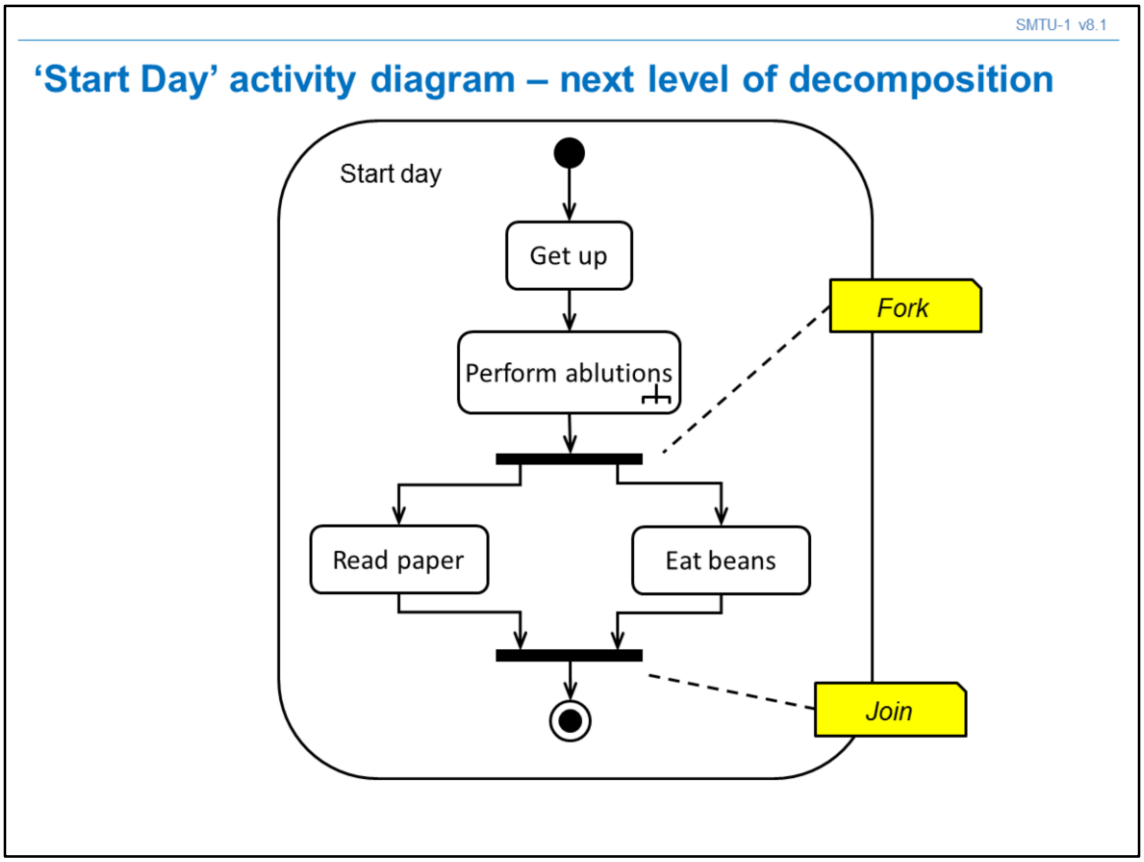
- **When an Activity is instantiated, a Token is place on all the nodes with no incoming edge, for example on the Initial Node**
- **Tokens then move through the Activity, tracing out the sequence pathway. When a Token reaches any Action, that Action becomes active**
- **Tokens are 'consumed' when they reach the Final Node and the Activity ends**
- **Complex activities may have several Tokens active at any one time**
  - Accounting for the Tokens helps get the models 'right'



Here's another illustration of some of this basic syntax.

Let us imagine our student is a young male living in a bed-sit. The activities undertaken during his day are shown in the activity diagram above. After finishing his 'start of day' activity, the student must decide whether or not he needs to attend lectures. If the decision is no, he drinks all day and falls asleep, which ends the process. If the decision is yes, he undertakes the sub-process (an activity) of attending lectures which will have its own activity diagram. When Attend Lectures is completed, the student drinks and finally sleeps as before.

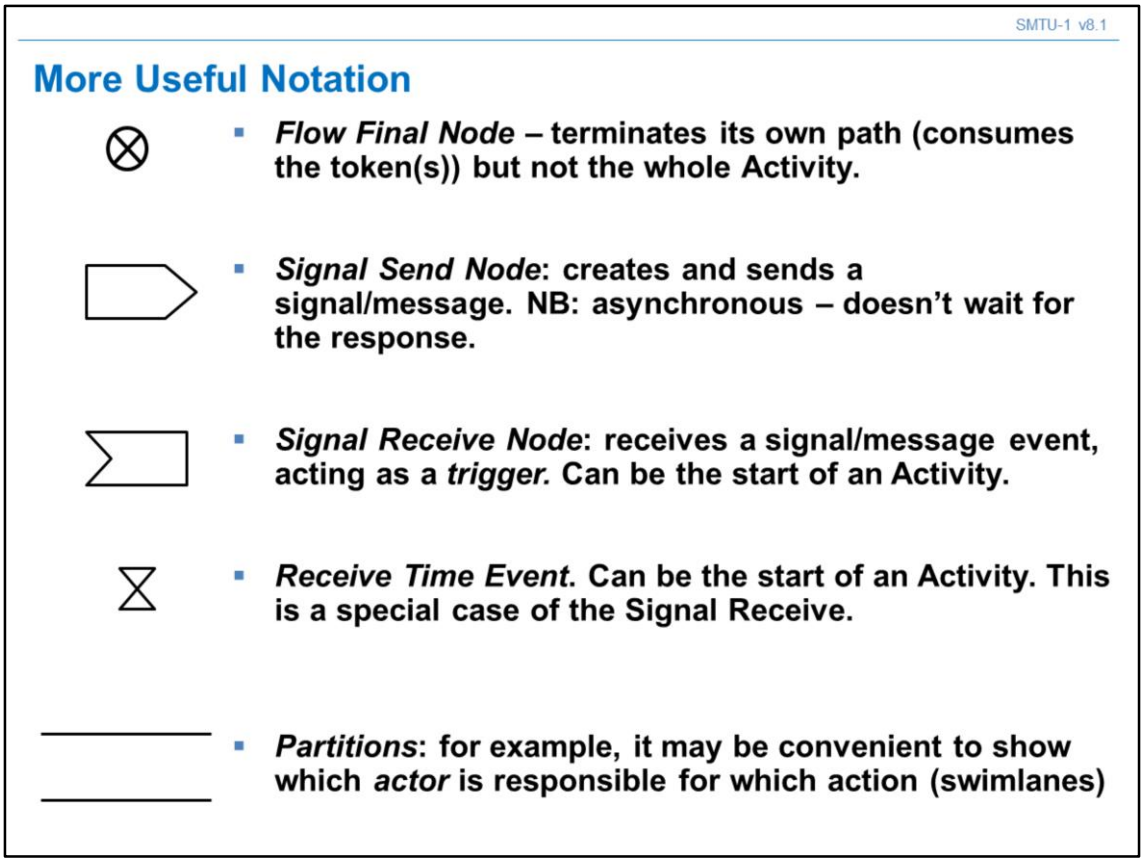
Note the 2 different ways of labelling control flows from a decision node (keyword *else*).



Here is the breakdown of the 'Start day' activity.

It starts with the 'Get up' action. Naturally this might be a protracted affair involving, scratching, yawning, groaning but we can use the 'description' box in the action's properties to describe the action in as much detail as we want. If the 'Get up' action is too complex to describe easily, then perhaps we should break it down into 'Wake up', 'Perform getting up rituals' and actually 'Get out of bed' actions. If its even more complex than that, then we might make 'Get up' an activity which will be described by its own activity diagram.

After performing his ablutions (which we will not break down further here), the student undertakes two actions at once, in that he reads the paper while eating his beans. The Fork and Join symbols allow us to show parallel paths. Note that they do not have to happen at exactly the same time, but the process cannot continue until both of them are complete.



Flow Finals are useful in terminating a specific flow, when it wouldn't be correct to terminate the whole Activity.

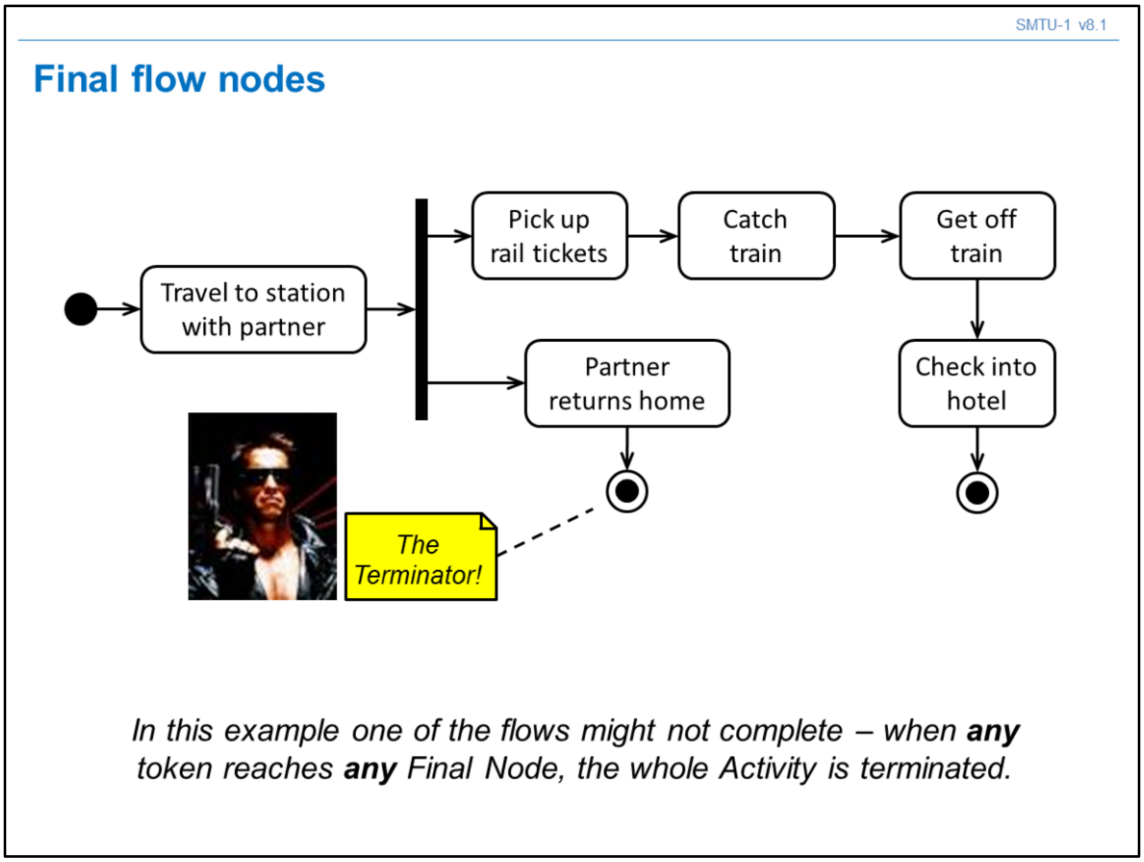
Send signals are signals/messages sent, normally to external participants. Signals are sent asynchronously, meaning the activity does not wait for the response but moves on to the next action after the signal is sent. The recipient will probably do something in response, but if that recipient is external, that action isn't modelled on your activity diagram.

A *receive signal* acts as a trigger in the activity diagram. It captures the signal / message from an external source and makes its contents available to the activity.

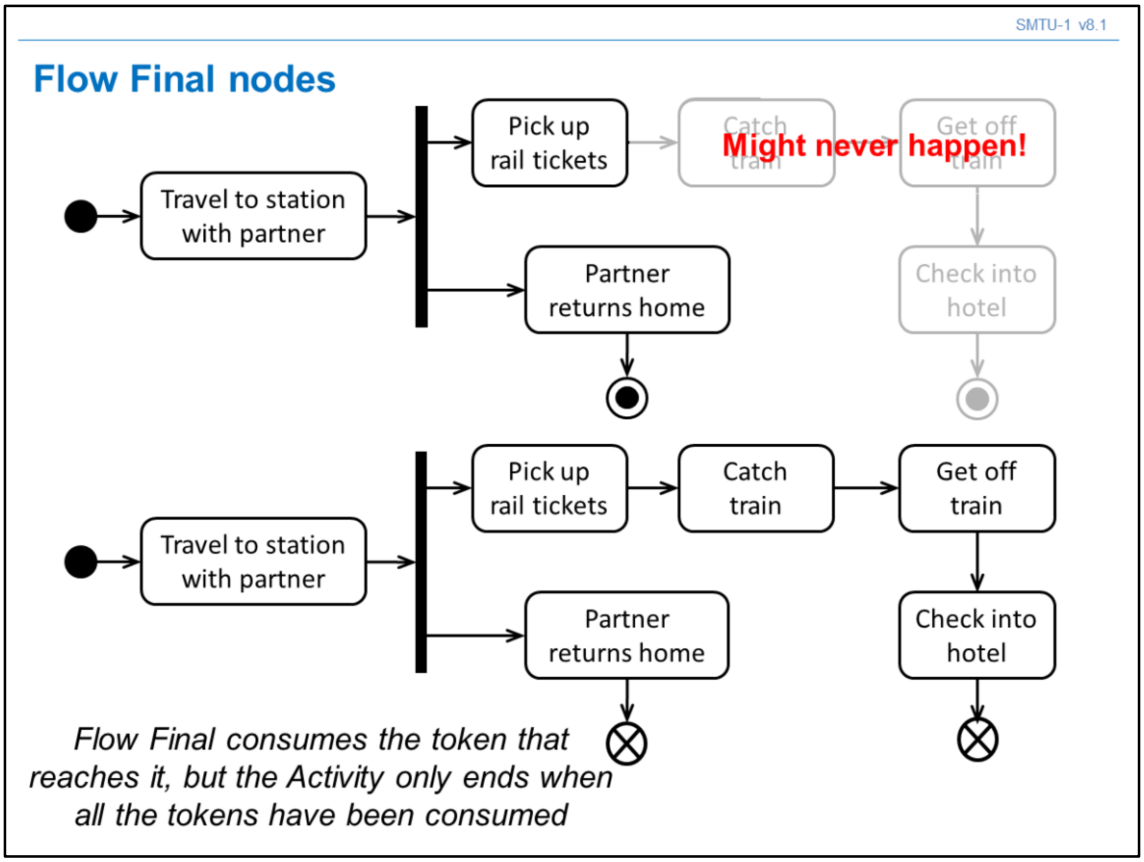
Sometimes *Time* is a factor in the activity. It may be necessary to model a wait period, such as waiting three days after shipping an order before sending the invoice or kick off an activity at a regular interval such as the monthly reading of a utility meter. Time events are drawn with an hour glass symbol – really they are a special case of a *receive signal.*

Finally actions may be categorised into *partitions*, which are normally used to denote the *roles* that perform the actions (swimlanes). Note this is purely cosmetic, although very useful, since it doesn't affect the logic of the activity being modelled.

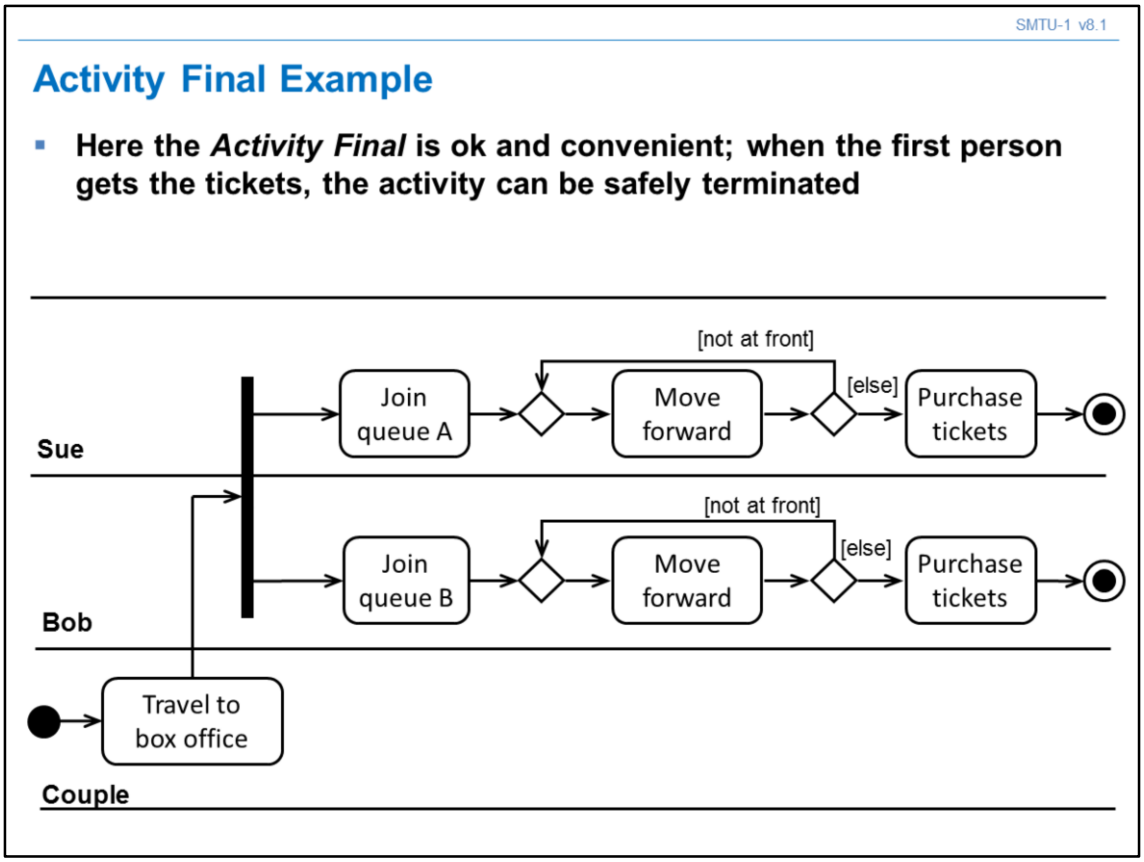
Let's see some examples of the use of these notational elements …



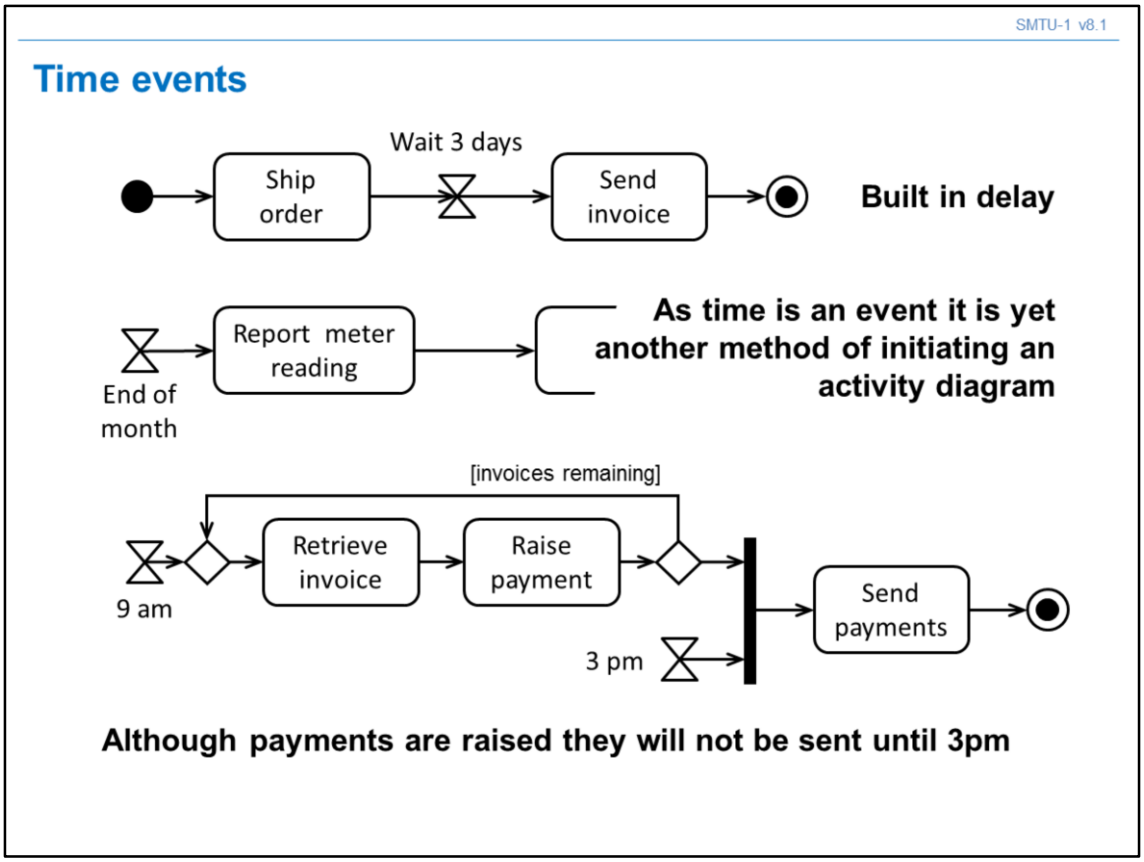
In this example, one of the flows might not complete.



Flow Finals only terminate their flow, consuming their tokens. The activity instance is extinguished once *all* generated tokens have been consumed.



In the example above, one of the parallel streams will finish before the other and terminate the activity; but here it doesn't matter, the tickets will have been purchased.

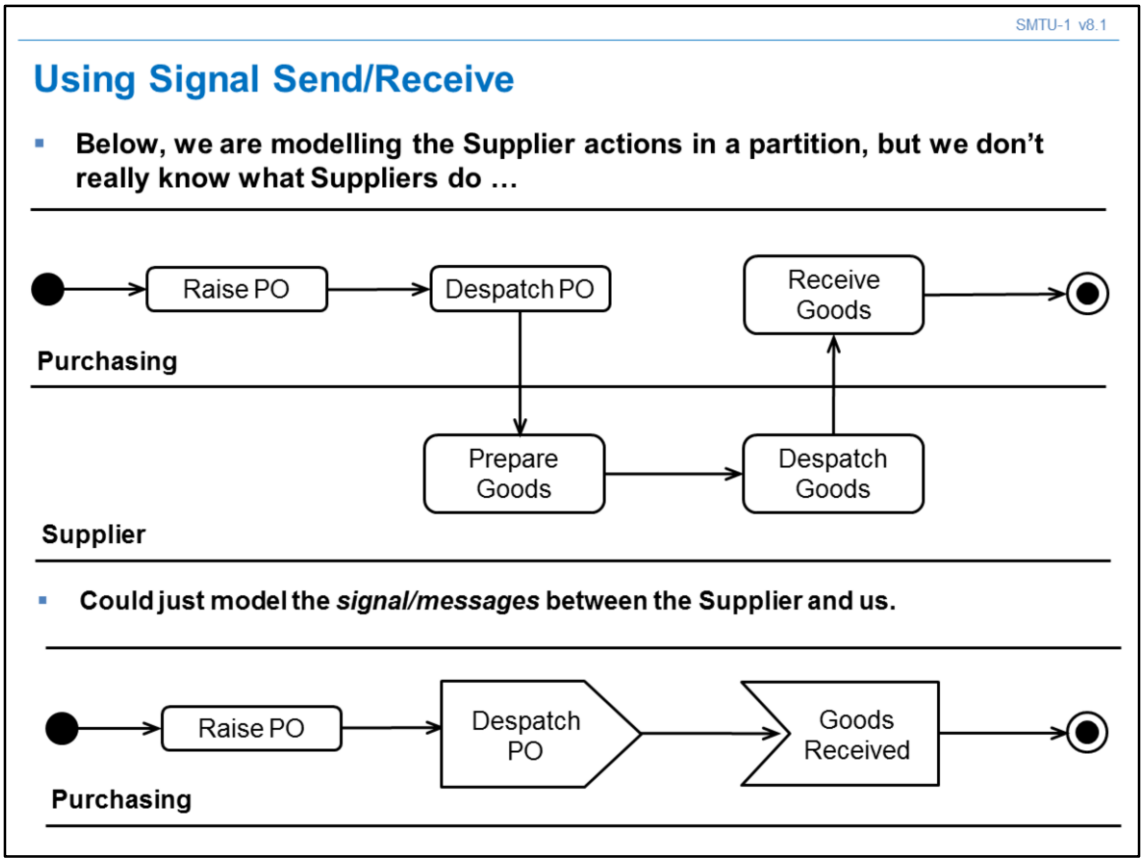


Here are some examples of the use of the ‘timer’ event.

1. The text next to the hour glass – ‘Wait 3 days’ – shows the amount of time delay once a token reaches it.

2. A time event with no incoming edge is potentially a recurring time event, meaning it’s active once the activity instance has been created, until the instance is extinguished. This event will thereafter be activated with the frequency of the text next to the hour glass, and a token is generated each time the event triggers. Some care is needed then when using this, to make sure the logic is accurate for the scope of the activity instance.

3. Time can be the trigger for the start of an activity, use of the timer can show this. Also the timer can be used together with a synchronisation ‘join’ bar; here, although payments are raised they will not be sent until 3pm. No matter which token arrives first, the join waits for the other one, before passing a combined token on.



In the lower diagram:

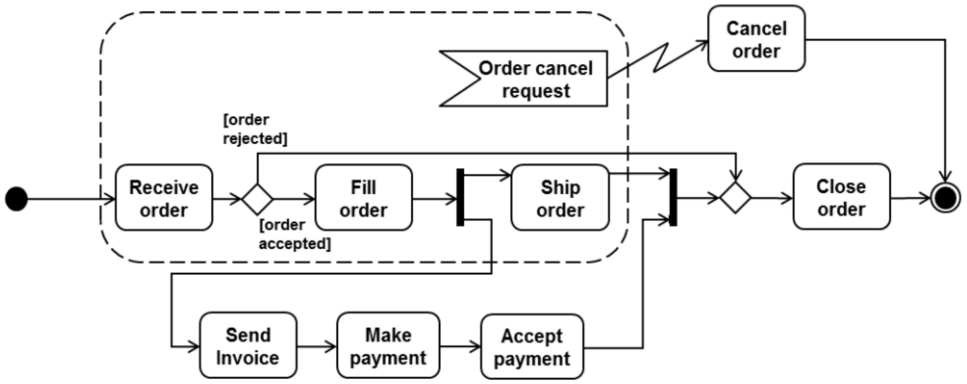
'Despatch PO' is a *signal send* and the action doesn't wait for a reply, the token moves straight on. *Signal sends* create and send the required signal or message; since they are a form of action, a verb-noun phrase is an appropriate descriptive label. NB: the 'signal' can be the despatch of physical goods or documents.

'Goods Received' is a *signal receive*. When the token arrives there, it waits for the signal to be captured, then moves on. This node is a trigger event, so should be labelled as something that *happens*. NB: If the signal *never* arrives, the token will wait there for ever! A way of dealing with this is to use an *interruptible region* …

SMTU-1 v8.1

Interruptible regions

Imagine the rule is: "A request to cancel an order will be accepted at any time after it was received up until the goods are shipped"



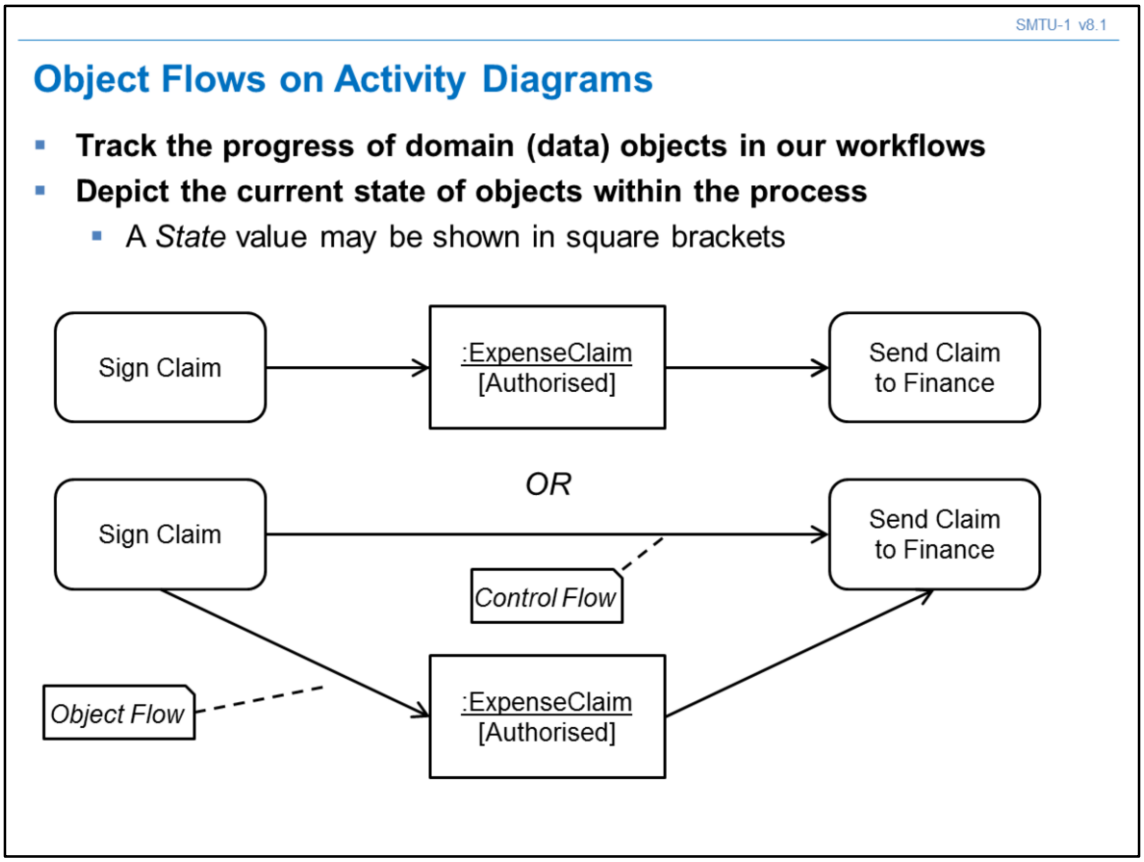
The dashed box indicates an *interruptible region*

The activity logic may mean that a set of actions on an activity diagram could be interrupted by an event that might occur at any time, or you may need to model a 'time-out' situation.

This can be shown by surrounding the actions to be interrupted with a round-cornered dashed-line rectangle. A zigzag activity edge, known as an interrupting edge, is used to show a move from the interruptible region to some action that is used to deal with the interrupt.

In the example, a request to cancel an order will be accepted at any time after it was received up until the point of shipping.

When the *first* of any active token leaves the region, any other active tokens are de-activated and lost.



A useful addition to the activity diagram is the Object Flow, as shown above. The colon/underline indicates that it is an object, followed by the class of the object.

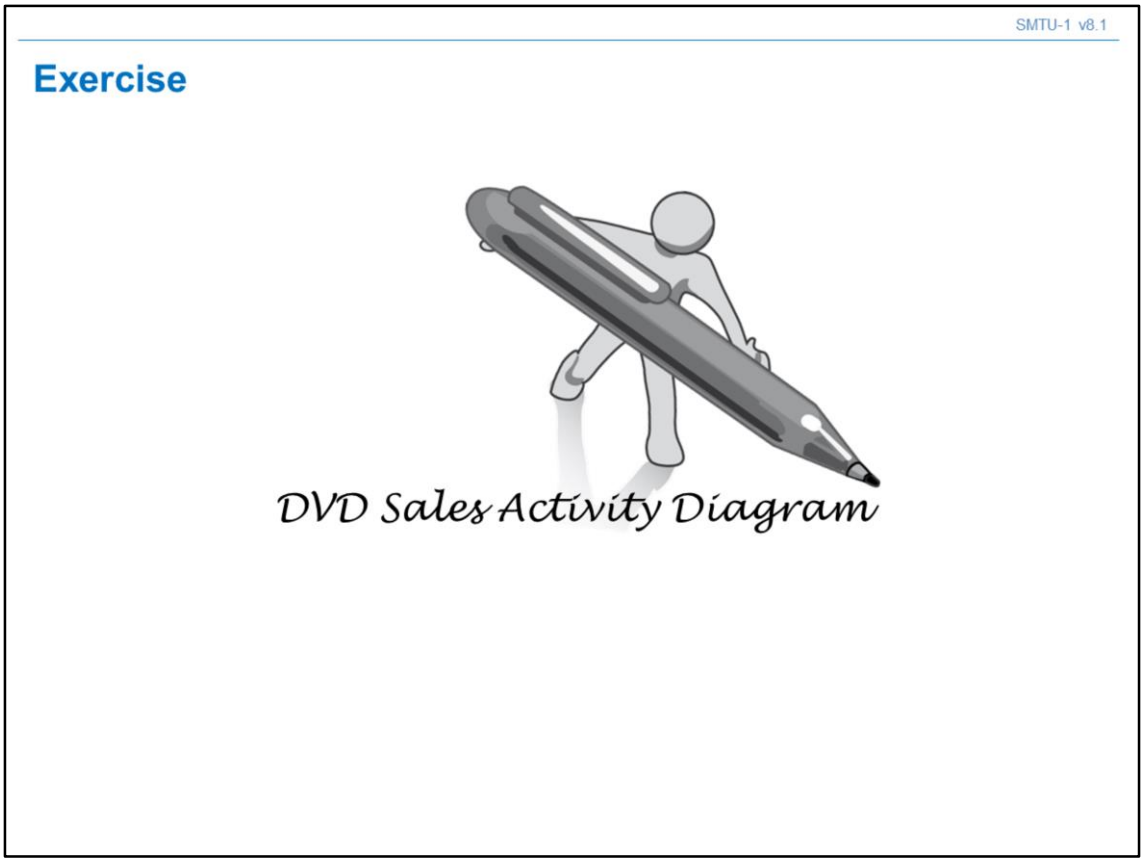
In the upper diagram, we see that the action ‘Sign claim’ causes the state of the Claim object to change to ‘Authorised’. The ‘arrival’ of this object triggers the Send Claim action.

In the lower diagram, the action ‘Sign Claim’ causes the state of the Claim object to change to ‘Authorised’. The start of Send Claim, triggered by the control flow, triggers a read of this object.

This type of documentation helps develop an understanding of the data requirements for the software, which can be checked against the Domain Class Model (see later session).

In this context you may see an object stereotyped as <<datastore>> in UML 2.x, meaning the object is a *copy* of persistent data. A flow *to* a <<datastore>> updates it, a flow *from* a <<datastore>> is a read.

To avoid clutter it may be convenient to only show update effects.



Follow the guidance from your instructor to produce an activity diagram from the DVD Sales scenario.

SMTU-1 v8.1

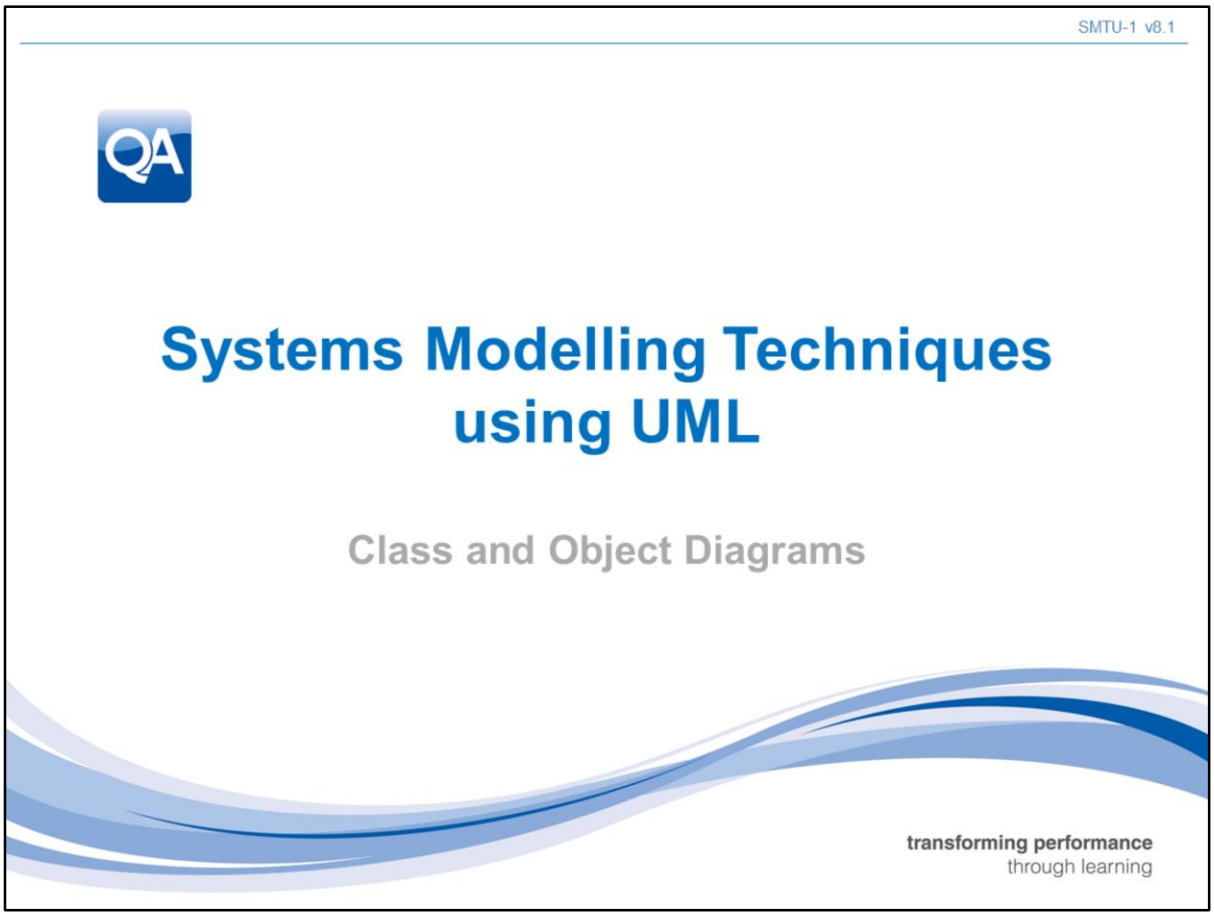
Case study

Tasks will be allocated to each syndicate group by the instructor. You will be asked to model one or more of Goods In, Issues or Stock Checking.

SMTU-1 v8.1

Summary

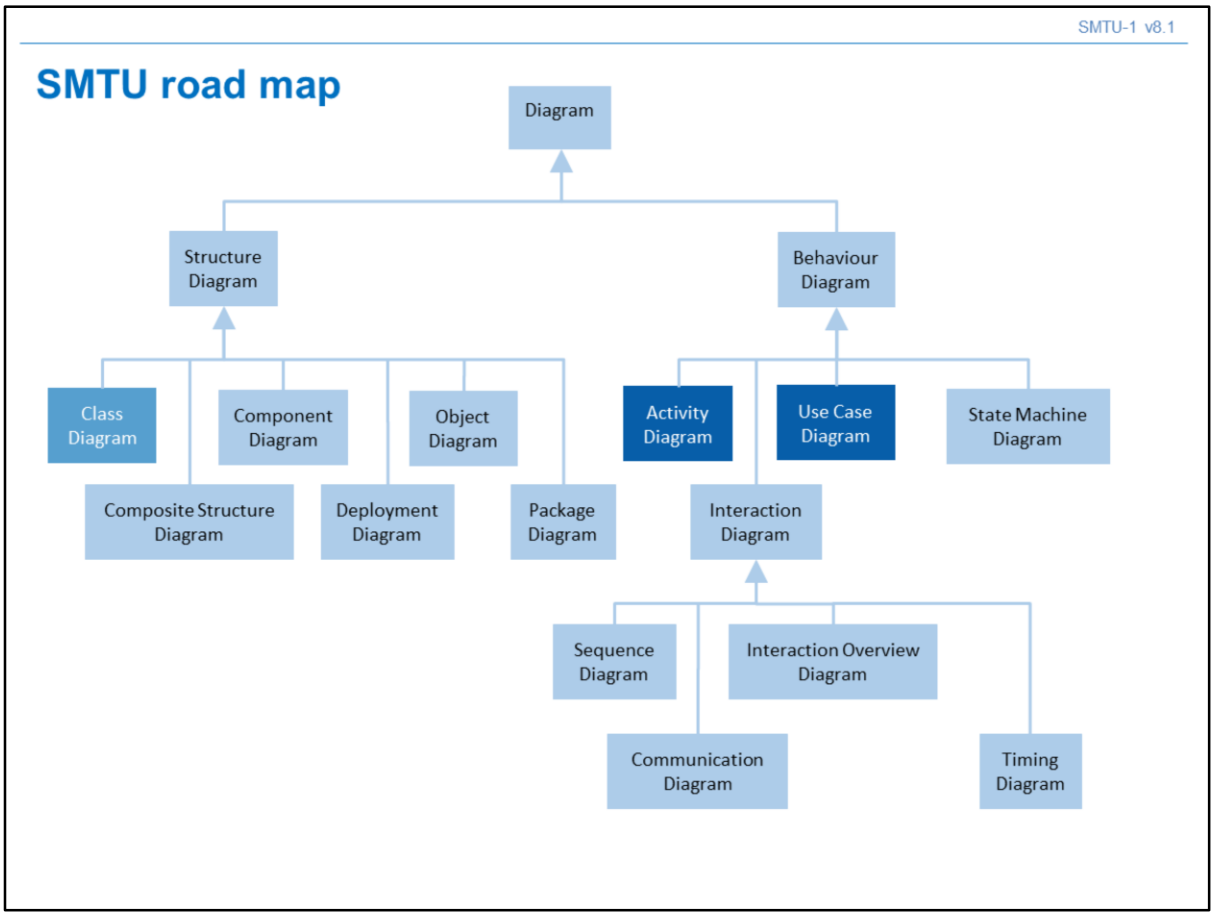
- Activity Diagrams may be used for modelling the logic of Business Processes and Use Case flows
- UML sees an Activity as a *network of actions*, traversed by *tokens*
- Object Flows can be usefully added to AD


SMTU-1 v8.1
Systems Modelling Techniques using UML
Class and Object Diagrams
transforming performance
through learning

SMTU-1 v8.1

Topics

- Objects and classes
- Analysis class modelling rationale
- Representing classes; name, attributes, operations
- Defining attributes
- Basic Associations and Multiplicity
- Domain Class Diagram
- Advanced Associations
- Abstraction, polymorphism and encapsulation
- Object diagrams


SMTU-1 v8.1
SMTU road map
Diagram
Structure Diagram
Behaviour Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
Use Case Diagram
State Machine Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Interaction Diagram
Sequence Diagram
Interaction Overview Diagram
Communication Diagram
Timing Diagram

SMTU-1 v8.1

Objects and Classes

- **UML is an Object Oriented (OO) visual modelling language**
  - Objects are based on Class definitions
- **In Analysis, a *Domain (or Entity)* Class represents a concept, an abstraction of things interesting to the business – e.g. 'Customer'**
  - Defines the common set of features shared by all its objects
  - Associations relate classes and allow objects to communicate
- **An Object represents a real thing – e.g. *Class* 'Customer'– *Object* 'Smith'**
  - An object is an *instance* of a class
  - An object is a cohesive packet of data (attributes) and functions (operations)
    - defined by the class
- **To meet a use case goal, a use case will perhaps manipulate several objects' data**
    - This is achieved by calling operations (sending a 'message')

Classes are templates that describe the different types of objects the system should have. Class Diagrams show these classes and their relationships.

Use Cases describe system functionality, whilst classes describe the objects needed within the system to support the functionality. In OO software, functionality is realised by a *collaboration of objects*, based on class definitions.

The class diagram shows the static building blocks of the system; the classes of object that define the system. The potential for cooperation among those objects, through message passing, is shown in the associations (relationships) between these classes.

SMTU-1 v8.1

Purpose of Domain Class Modelling

- **Class diagrams are used throughout the UML development process**
- **The main purposes of producing *Domain Class Diagrams* in the analysis phase are to:**
  - Identify the key business concepts; what are they, what do they mean?
  - Show the main features of these concepts
    - Attributes (data), Operations (behaviour), Associations between them
  - Understand and define some of the business rules
  - Gain a deeper insight into the way the business operates
  - Establish a common vocabulary within the project and, hopefully, across the enterprise
  - Prepare the ground for persistent data design
- **Class diagrams are *static* models, so they do *not*:**
  - Show how classes interact with each other dynamically
  - Show the functional requirements of the system

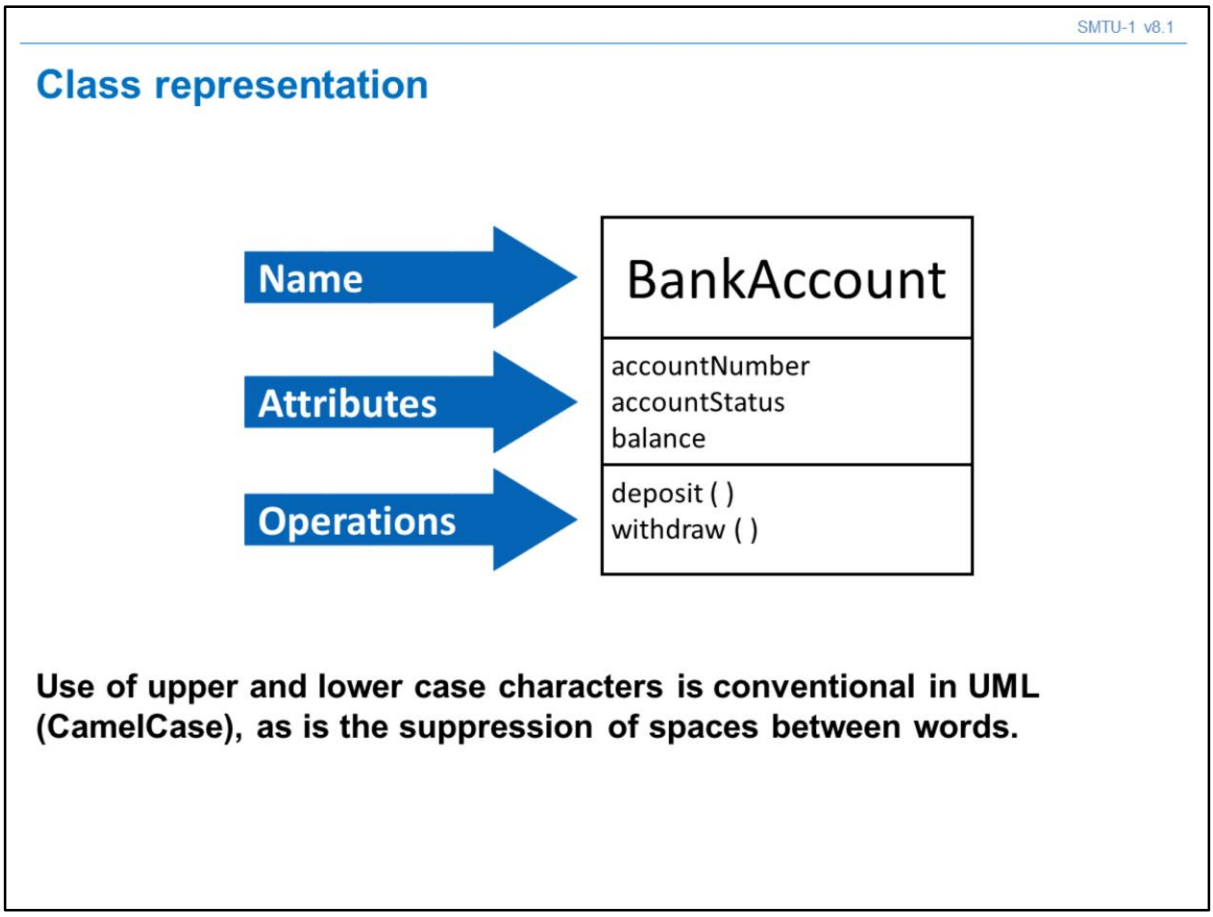
Classes are used throughout the development life-cycle. The classes encountered at the Inception/Elaboration Phase are «entity» classes. We will distinguish between various class types when we cover use case realisations.

SMTU-1 v8.1

Guidelines for finding Domain Classes

- **Existing data models or class diagrams may be available**
  - May be used as the start point
  - Concentrate solely on business domain classes
- **Look for 'nouns'. Class names appear on business documents, process descriptions and the nouns in use case names**
- **Test the validity of the class:**
  - Is it easily described?
  - Has it got obvious attributes (information about it)?
  - Does the business recognise it?
  - Has it got more than one uniquely identifiable occurrence? (How is one instance distinguished from another?)

Looking for entity classes is not trivial and will require several iterations. Basically the question to ask is "what would the business need to keep data and information *on*?" in the domain where the software will be applied.



The basic building block for class diagrams is the class. The class symbol has three sections: Name, Attribute list and Operation list.

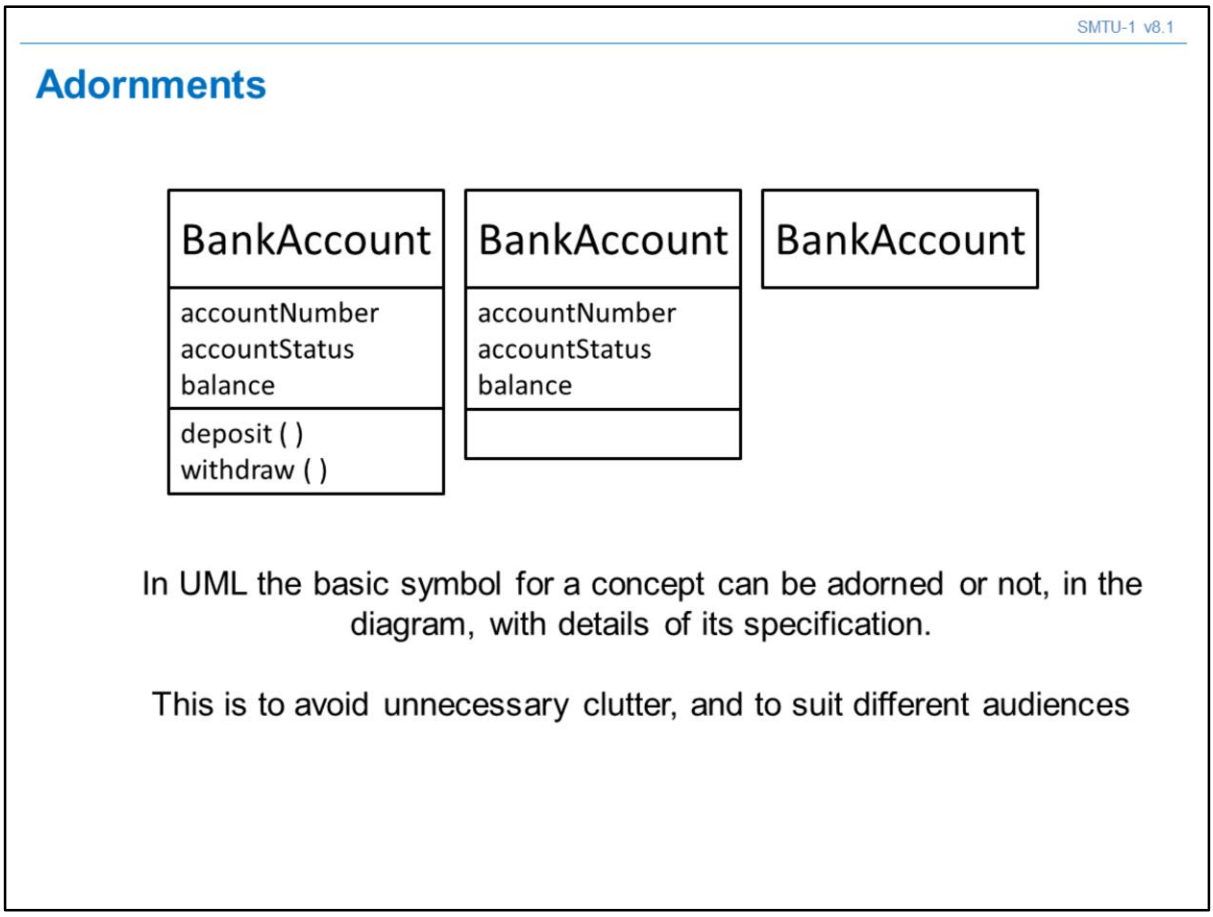
Classes are shown as rectangles, with the name of the class centred in the rectangle. The class name should be a singular noun phrase and start with a capital letter.

By convention a class's name should have no spaces between multiple words in the name, but should start each subsequent word with a capital letter. This is not significant in Analysis, but more important moving into Design – languages like Java don't support spaces in names. So for example, BankAccount … not Bankaccount, Bank account, Bank_account nor Bank Account.

Also by convention, naming for Attributes and Operations uses *camel case* – the first word starts with a lower case letter and then each subsequent word starts with an upper case letter. Again no spaces.

Attributes are the 'pieces' of data which are held by objects belonging to the class. They specify what the object *knows.*

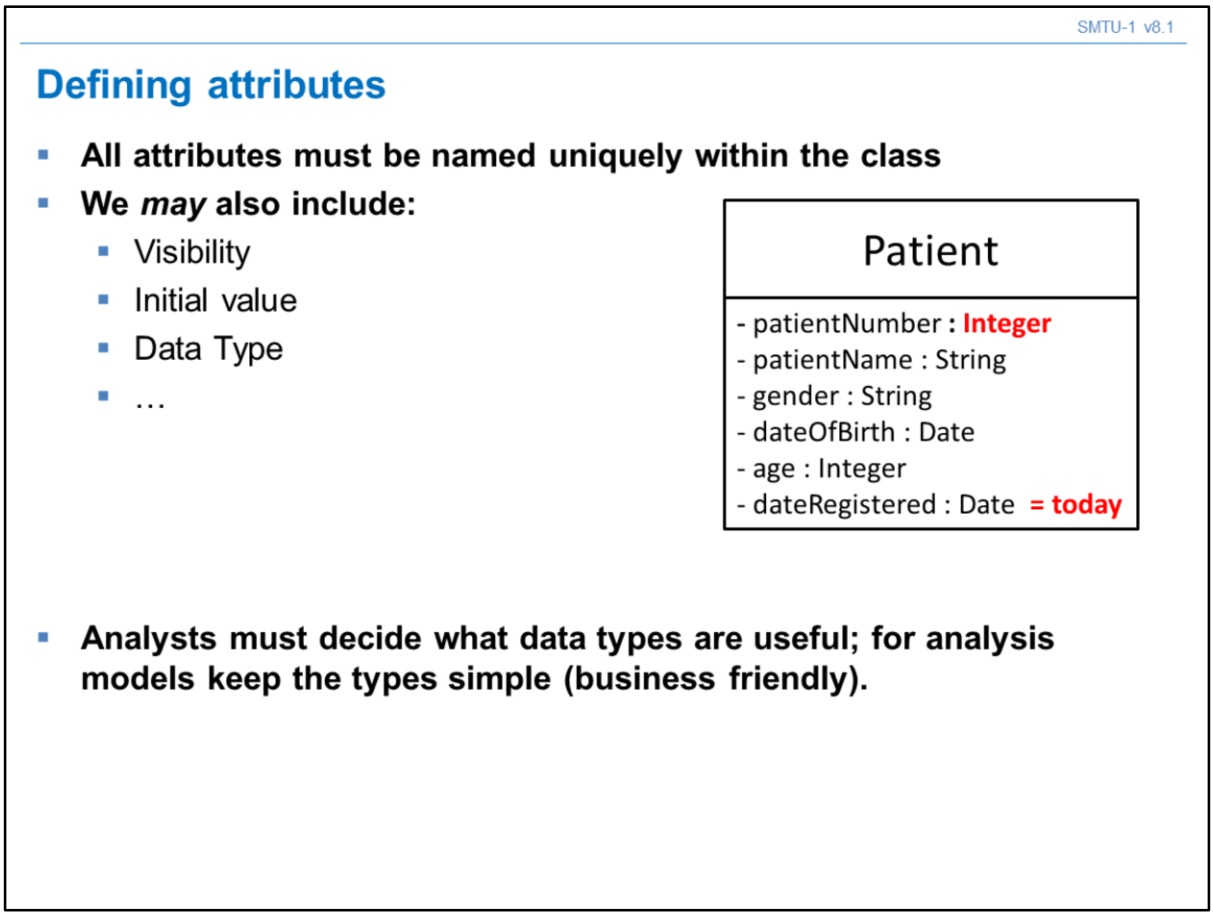
Operations specify what you can ask an object of a class to *do*. In the brackets following the operation name you can show any parameters that the operation needs to do its work. We will talk more about operations in the session on dynamic modelling.



The list compartments for attributes and operations can be independently omitted. In other words it would be acceptable to show a class as:

- class name, a list of attributes and a list of operations
- class name and a list of attributes
- class name and a list of operations
- class name only

The UML features that specify each part of a class are known as *adornments*. What adornments are included will depend on the purpose of the diagram, and the audience.



The notational elements shown are usually sufficient for the analysis view.

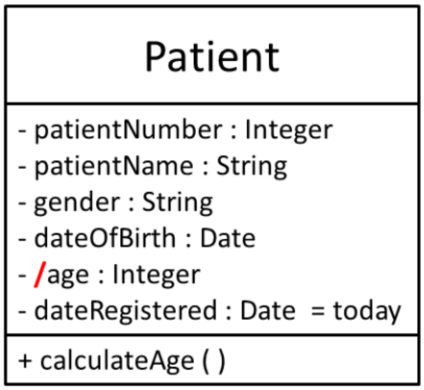
In the UML, all attributes belong to a type (although we may not know which type initially). UML does not specify which set of types to use, so we could use an implementation-independent set or use the types from our target programming language. At this stage it is best to keep them simple, e.g. string, integer, date, currency etc.

The minus sign indicates visibility, in this case *private visibility.* Attributes are usually best 'hidden' from direct access from outside the class, which means we can only get at their contents via an operation. This is one example of an implementation of the OO feature known as *encapsulation*.

SMTU-1 v8.1

Derived attributes

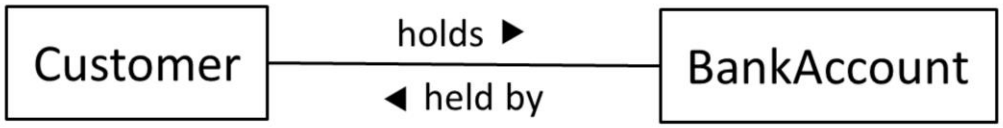
- We can indicate that the value of an attribute is derived from other information by using the / notation
- Here, 'age' is derived by comparing the 'date of birth' with today's date
- If the designers or DBA choose to derive the patients age then expect to see an operation such as calculateAge to be defined later
- This is more a *design* concern than an *analysis* one



SMTU-1 v8.1

Associations

- **An *Association* is a relationship between two classes that has meaning to the business**
- **Association naming:**
  - name and show "direction" for clarity
  - convention is to use **present tense** verb-phrase names which allow the association to be discussed
    - A customer *holds* a bank account
    - A bank account *is held by* a customer
  - Usually shown in 1 direction only; the reverse phrase is inferred



NB: the association must be true at all moments in time

We are looking for direct, business phrases to name associations and asking the most important question: "Would the business recognise this phrase, in the context of objects from the 2 Classes?"

Associations give us the business context in which objects from one Class are related to another, which helps us understand the structure and meaning of the data needed by the software.

Note that the *association labels* can be included in the diagram in both directions, but usually only one of them is, to reduce clutter.

SMTU-1 v8.1

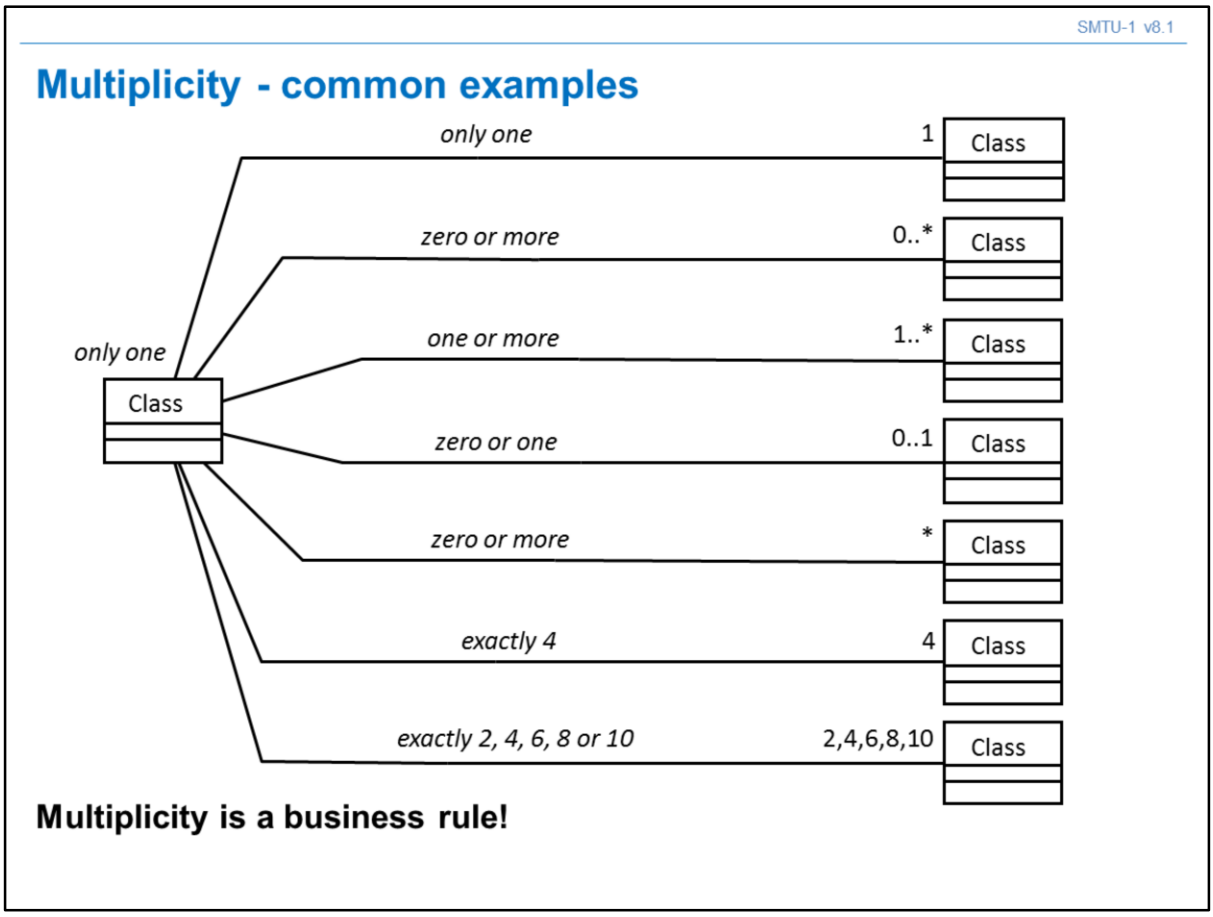
Association multiplicities

- **Association multiplicity is a key mechanism for capturing business rules**
- **They state how many instances of a class are associated with a single instance of another class**
  - Each customer can hold many accounts – there is no limit (*)
  - A customer does not have to hold an account (0) – this caters for potential customers that have yet to open an account
  - Each bank account must be held by at least one customer (1)
  - The bank account can be held by up to four customers (4) – this caters for joint accounts

  - Note where the multiplicity is written in relation to the class to which it applies

Association multiplicity is a key mechanism for documenting business rules. Given there is an association, it indicates how many instances of one class are associated with a single instance of another class.

In the example above, we are stating the business rule that each Customer *holds* zero,1, or many Accounts. Each Bank Account can be *held by* 1 to 4 Customers.



In UML we can be very specific about the multiplicity required. Some examples are shown above.

SMTU-1 v8.1

Workshop

For a public lending library, potential…

- Classes?
- Associations?

Read through the Library Vision Statement and identify the domain classes. Look for the nouns, these are the potential classes or attributes of classes.

A valid class will have several attributes; i.e. it is a concept about which we want to hold information. An attribute on the other hand is a single piece of data *about* something. Whether something should be a class or an attribute may not be obvious initially, although eventually we would be able to determine that.

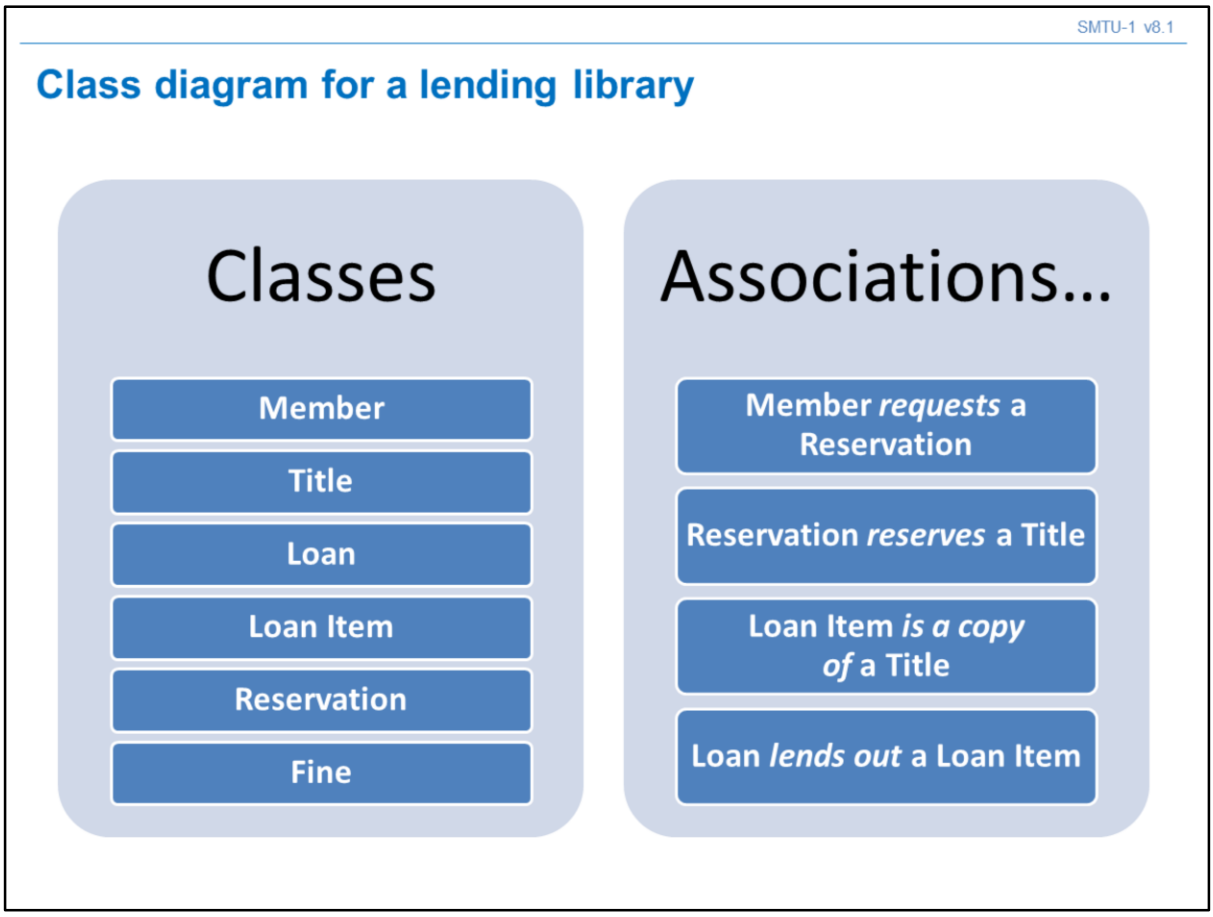
Also bear in mind that something that is a class in one business may simply be an attribute in another – think of *Post Code* for example. Post Code would undoubtedly be a class for Royal Mail, but an attribute for most other businesses.

Classes can emerge from several sources, including discussion with the business. For example a Use Case diagram can be a source of inspiration for classes. Use cases are named verb-noun and the nouns may well be classes.

Once you have a few classes; can you describe a meaningful direct relationship between them without mentioning a third class? If you can the word, or words, describing the nature of the relationship is the association. Avoid ‘has’ or ‘associated’ as an association name, these are not useful phrases!

SMTU-1 v8.1

- **Intentionally blank**



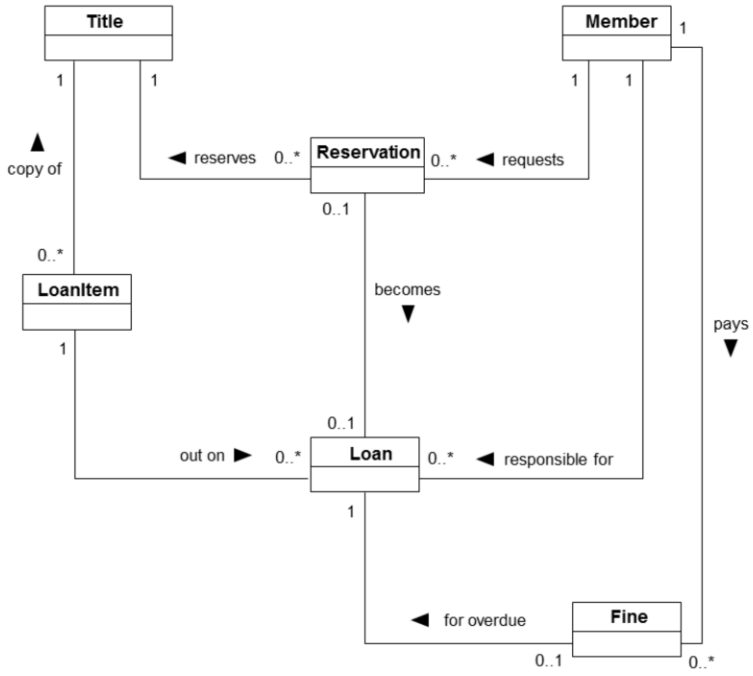
Loan item represents the physical “thing” that is removed from and returned to the library. Title might not emerge until the concept of a reservation is considered. A member is not concerned with reserving the physical item, rather the name of the item – title!

Librarian is a noun but in the context of the scope do we need to hold any information about a librarian? This is not an HR system. We need to know who the loan item is being lent to but is there a need to record who issued the loan item? If the answer is no then librarian will not be a class.

The issue of whether a concept should be modelled as a Class or as an Attribute always arises at some point. Guidance here is: if it is a concept *about which* information is held, then it is a class. If it is information *about* a concept then it is an attribute. This may not be obvious in Analysis and may need adjusting later. The best strategy is “if in doubt model it as a class”.

SMTU-1 v8.1

Class diagram for a lending library



Not all members will pay fines but when a fine is levied it will be associated to the member who will have to pay it. However, at the moment, we have no way of showing the business rule that child members will not be fined. Additional notation is required to show this kind of rule.

SMTU-1 v8.1

Activity

Produce an initial class diagram for a hospital, in accordance with the narrative in the exercise.

The first thing to do is to make a list of the *domain classes (*business concepts*)* that seem relevant to the business situation.

Then examine each class in turn and ask “what *direct* association(s) (i.e. business connections) does this class have with any other?”

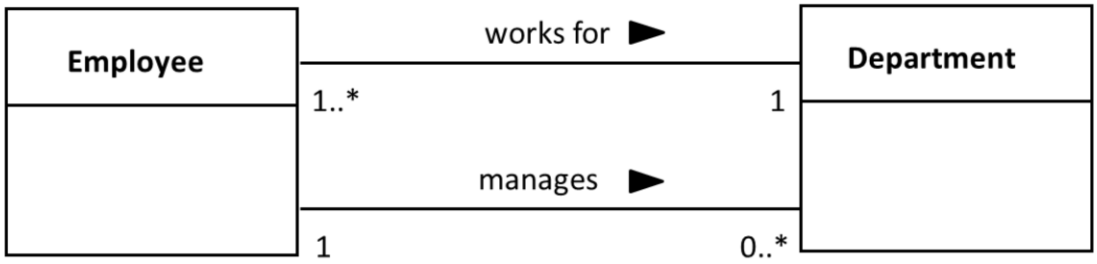
Draw the diagram accordingly, classes and associations.

Finally work out the multiplicity, according to your understanding of the business rules, i.e. for *each* object of class A, how many objects of class B could be associated with it – minimum and maximum.

SMTU-1 v8.1

Multiple associations

- Each Employee works for one and only one Department
- Each Department has assigned to it one or more Employee(s) and
- Each Employee manages zero or more Department(s)
- Each Department is managed by one, and only one, Employee

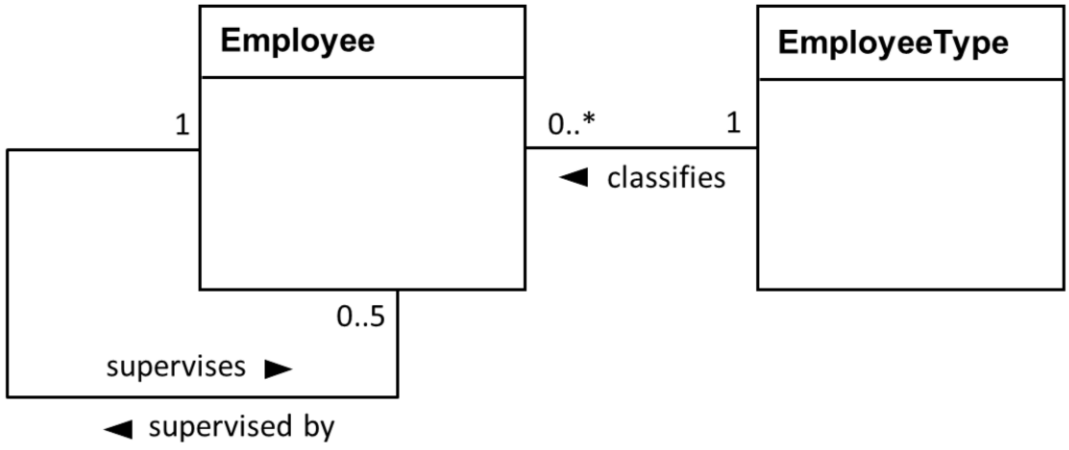


Note that there can be multiple associations between classes, as shown above. Each association has a different business meaning and is independent of any others.

SMTU-1 v8.1

Reflexive associations

- A class may have an association with itself if different objects of that class fill different roles
- The syntax for such an association is the same as for any other form of association

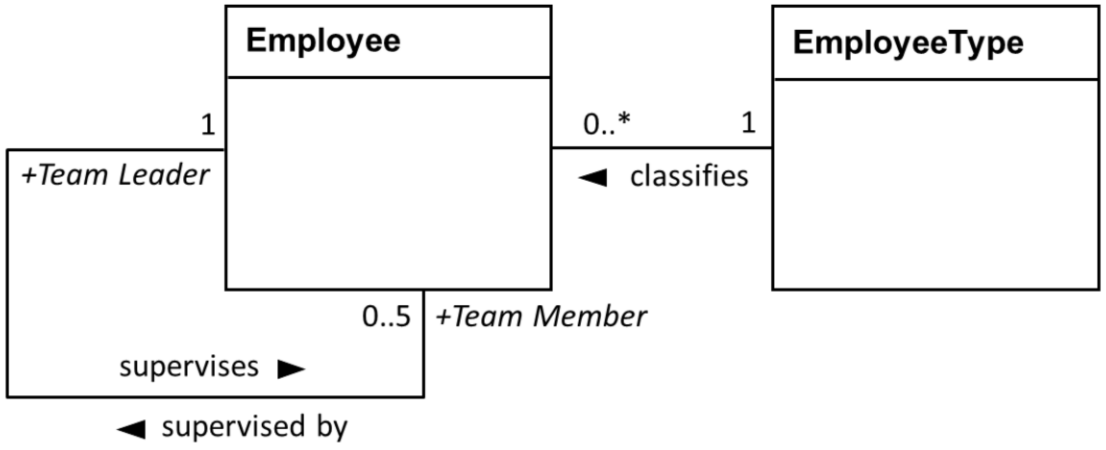


It is worth pointing out that the association between EmployeeType and Employee is a pattern for all ‘Type’ defines…

SMTU-1 v8.1

Role names

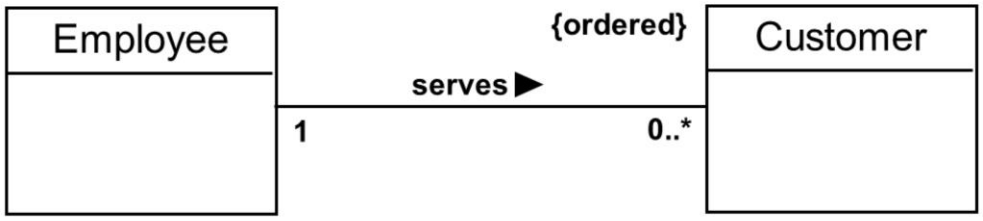
- **May be useful in some cases to indicate the respective *roles* on the association (NB: *roles* can be used on any form of association)**



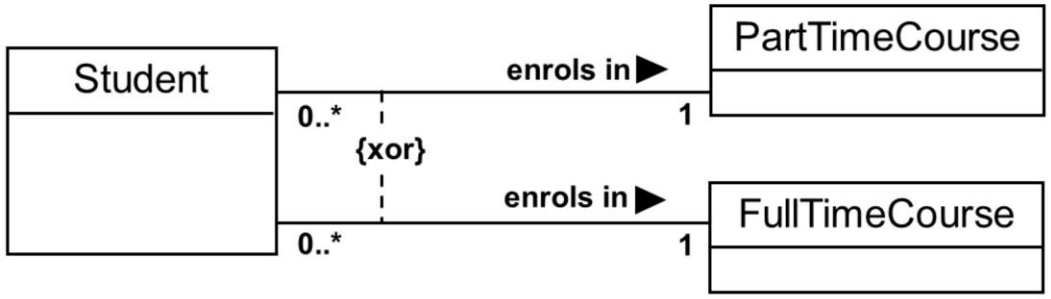
Role names may be used as well as, or instead of, association labelling.

SMTU-1 v8.1

Constraints on associations



Although the employee serves many customers they are served one at a time in an ordered queue



Full time course or part time course but NOT both

Sometimes an association has to follow a rule and this is shown by putting the constraint in curly brackets near the association line by the class that the constraint impacts on. In the example above, the Employee serves the Customers in the order in which they are in the queue.

Another type of constraint is where objects of one of the classes can only participate in one of the linked associations. In the example above a student may either enrol in a part-time or full-time course, but not both.

SMTU-1 v8.1

Many to many associations

- Many to many associations are perfectly acceptable on Analysis Class Diagrams
- They may however hide attributes that have been overlooked
- For this Patient, treated by this Doctor, where do we keep the information on the treatment received, who received it, who dispensed it and when it was given?

Many to many associations are common in domain class diagrams. Consider the association above:

- Each Doctor can treat zero or many Patients
- Each Patient can be treated by zero or many Doctors.

We should always scrutinise a m:n association to see if it is concealing missing concept and attributes. If it is not, then that is fine and we leave well alone. If it is then we need to flush the hidden attributes out.

We ask ourselves the following question: “For a Doctor (Dr Smith say), when did they treat this Patient (Ms Brown say), and is there any information that we need to capture?”

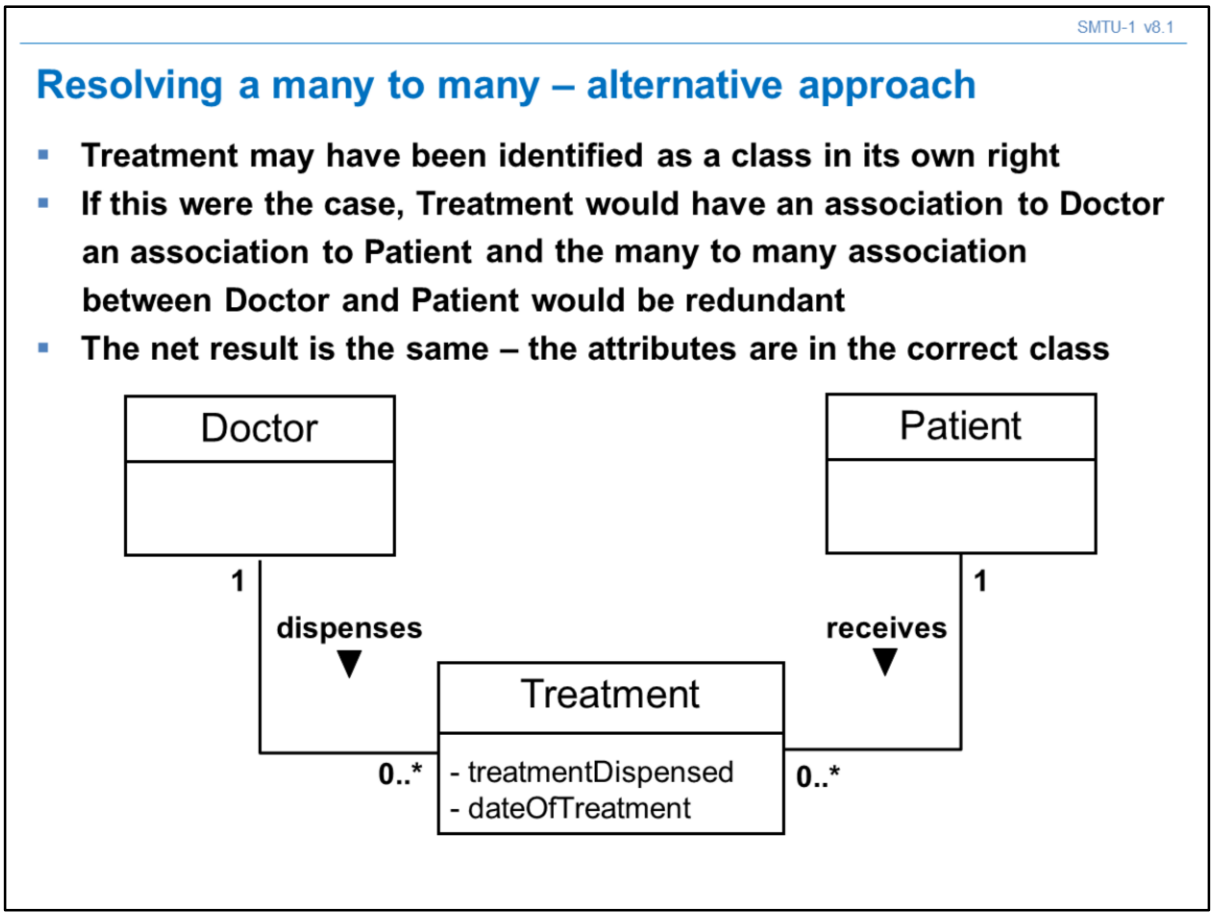
Again, this is purely a business question and in this case our user tells us that we need to log what treatment Ms Brown received from Dr Smith and when she received it. Where do these attributes belong? They do not belong in the Doctor class nor in the Patient class because there may be many instances and the attributes should be atomic. They belong in a missing class of Treatment!

SMTU-1 v8.1

Resolving a many to many association

- ***Treatment*** **is an association class that resolves the dilemma**
- **The Treatment class holds the treatment dispensed and the date of treatment**
- **For each instance of the association – one object of the association class will exist**

Association classes are useful UML constructs and are an elegant way of leaving the main m:n association intact while showing the new class we have uncovered. Think of a Treatment instance connected to each link between an instance of Doctor (in this case Dr Smith) and Patient (Ms Brown) showing the treatment dispensed, the date of treatment and thus the treatment dispensed to Ms Brown by Dr Smith.



Treatment may have been identified as a class in its own right. If this were the case Treatment would have:

- an association to Doctor
- an association to Patient

and the many to many association between Doctor and Patient would be redundant.

The net result is the same – the attributes are in the correct class. In fact, the resulting class structure shown above, is how to interpret the nature of the association class.

SMTU-1 v8.1

Activity

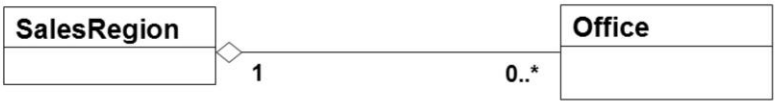
Enhance your original class diagram from the additional information provided.

SMTU-1 v8.1

Aggregation notation

- **Aggregation: a whole-part structure (APO)**
- **Aggregation may be used when a class is a collection or container of other classes**
- **In this case, if the container is destroyed, its contents are not**
- **The aggregation diamond is always at the whole end of the aggregation – the class that contains the parts**

Part exists independently of the *Whole*



- **The business can restructure its Sales Regions, an Office can be transferred to a different Sales Region**

A common use of aggregation is to indicate that an object from the *'whole' class*, as a concept, indicates a *collection* of objects of the '*part' classes*. The 'whole' class will usually have its own attributes.

Note that the use of aggregation does not override the need for multiplicity specifications, although often the multiplicity at the 'whole' end is not shown, and therefore assumed to be 1.

The white diamond is always connected to the class that is the aggregate; the class that is the collection. The term whole-part ("a part of" or APO) is sometimes used to refer to aggregation associations.

The defining characteristic of aggregation is that the relationship Whole-Part may be broken. That is, the lifecycle of objects from the Part are independent from the lifecycle of objects from the Whole. In the example, above, the business can restructure its Sales Regions, so an Office could be transferred to a different Sales Region.

SMTU-1 v8.1

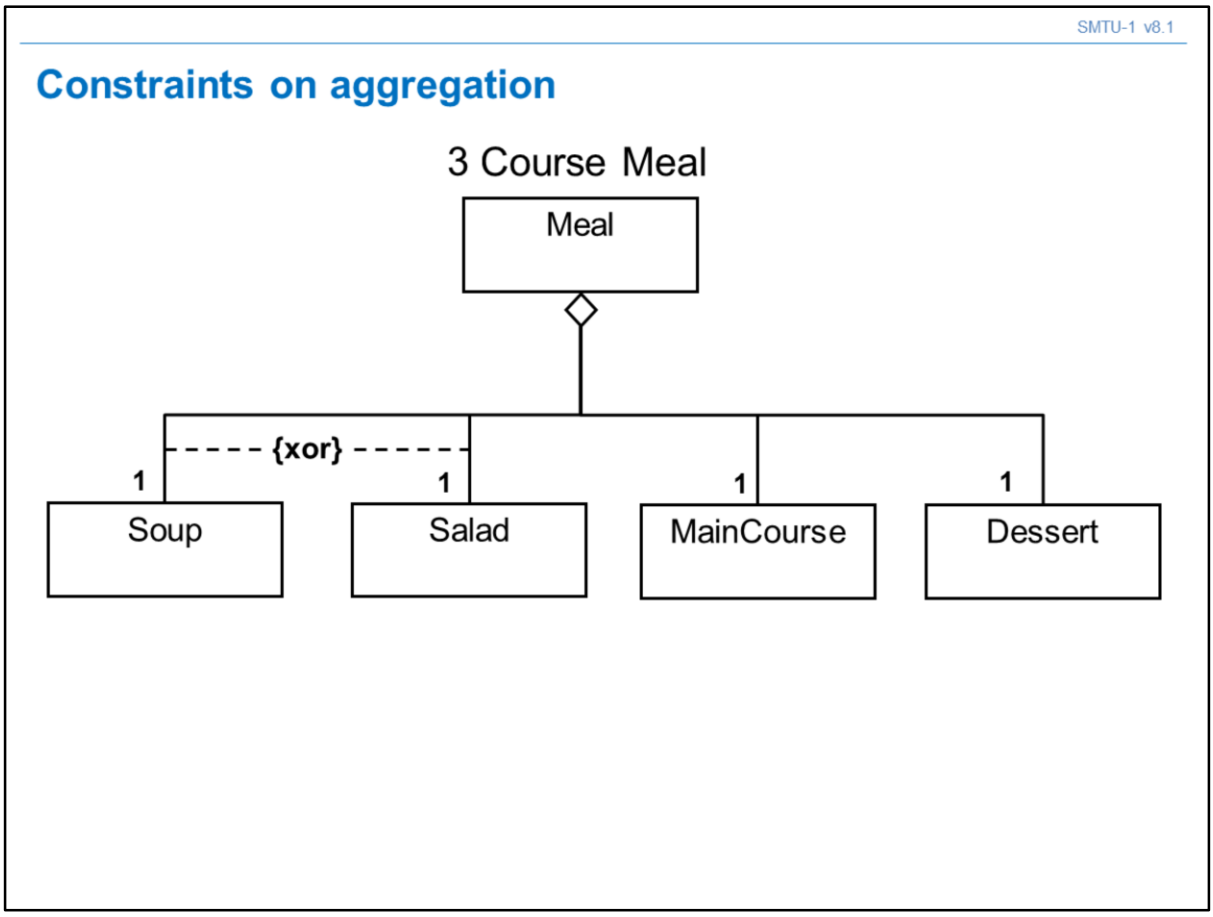
Composition notation

- A composition association implies that the life-cycle of the 'part' cannot extend beyond the life-cycle of the ‘whole’ (APO)
- In composition, a part cannot exist without being part of a whole
- If the container is destroyed, its contents are destroyed as well
- Typically lists of things – order, invoice, receipt, delivery…
- When a list – content name typically line or item

Part existence depends on the *Whole*

A composition association implies that the life-cycle of the 'part' cannot extend beyond the life-cycle of the ‘whole’. The black diamond means the Whole also has Lifetime-Control responsibility of the Parts.

In this association, you can’t have the *part* without the *whole*; an Order consists of Order Lines which are part of that Order. When the Order is created it is created with at least one OrderLine. If the Order were to be deleted all its corresponding Order Lines should also be deleted. Also it makes no sense to think of transferring an OrderLine to another Order.

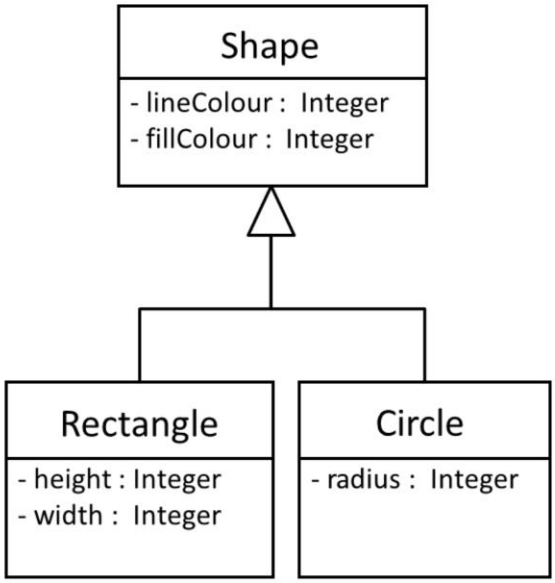


Note that constraints can be used on aggregations too. Without the constraint it's a four course meal. With the constraint it's a three course meal, the first course is either soup or salad.

SMTU-1 v8.1

Abstraction

- ***A kind of* association (AKO)**
- **Generalise out all of the common attributes and operations to a superclass**
- **Put all of the specifics in the subclasses**
- **Use a Generalisation association to relate the subclasses to the superclass (aka *base* class)**
- **The subclasses inherit all of the attributes, operations and associations of the superclass**



If we can identify common attributes and/or operations of *Rectangle* and *Circle*, we can abstract these to a *Shape* class. Then we can place Rectangle and Circle in a *generalisation association* with Shape. In UML, the generalisation relationship is shown with a hollow arrow head.

Shape is said to be the superclass or base class of Rectangle and Circle; Rectangle and Circle are Shape's subclasses. The relationship is often verbalised as 'A Rectangle is a kind of Shape' and 'A Circle is a kind of Shape' ("a kind of" - AKO). In OO terms the base class Shape is *extended by* Rectangle and Circle.

The key thing to understand is that Rectangle and Circle inherit all of Shape's attributes, operations and associations and then extend or override Shape with their own specialisations.

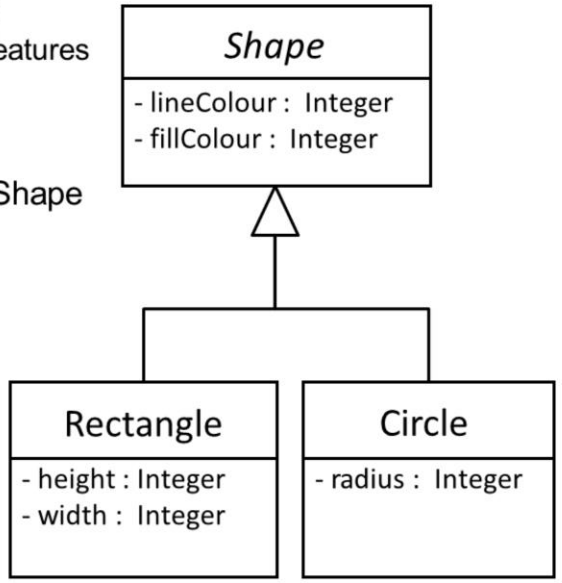
Note that, unlike other associations, generalisation/specialisation explores relationships *within one class*.

In practice this type of relationship is fundamental to the way OO software is built – we will return to this theme later.

SMTU-1 v8.1

Abstract Classes

- **This base class is abstract (*italics*)**
  - Acts as a 'placeholder' for common features
- **It is never instantiated**
  - There will be no objects of class Shape
- **This is common in OO in general**
  - *Inheritance* hierarchies may be a useful way to organise a class

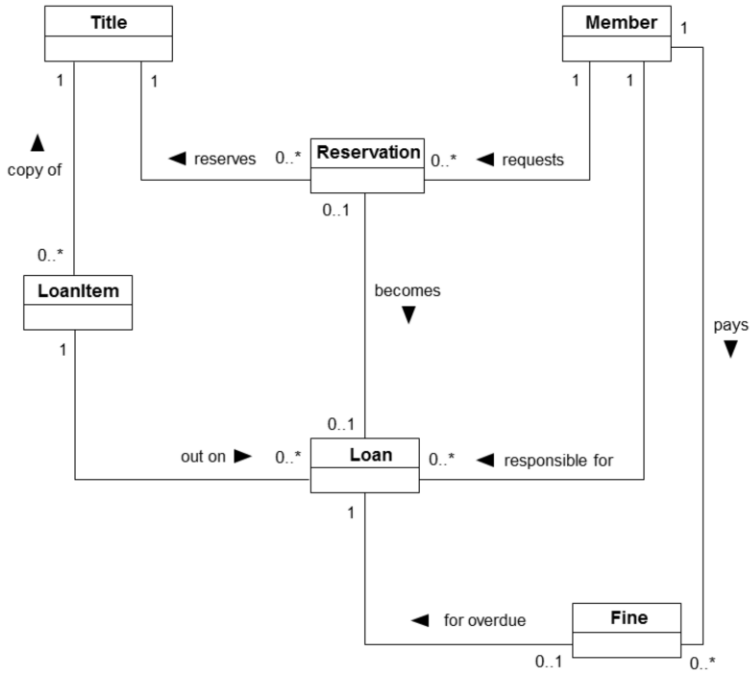


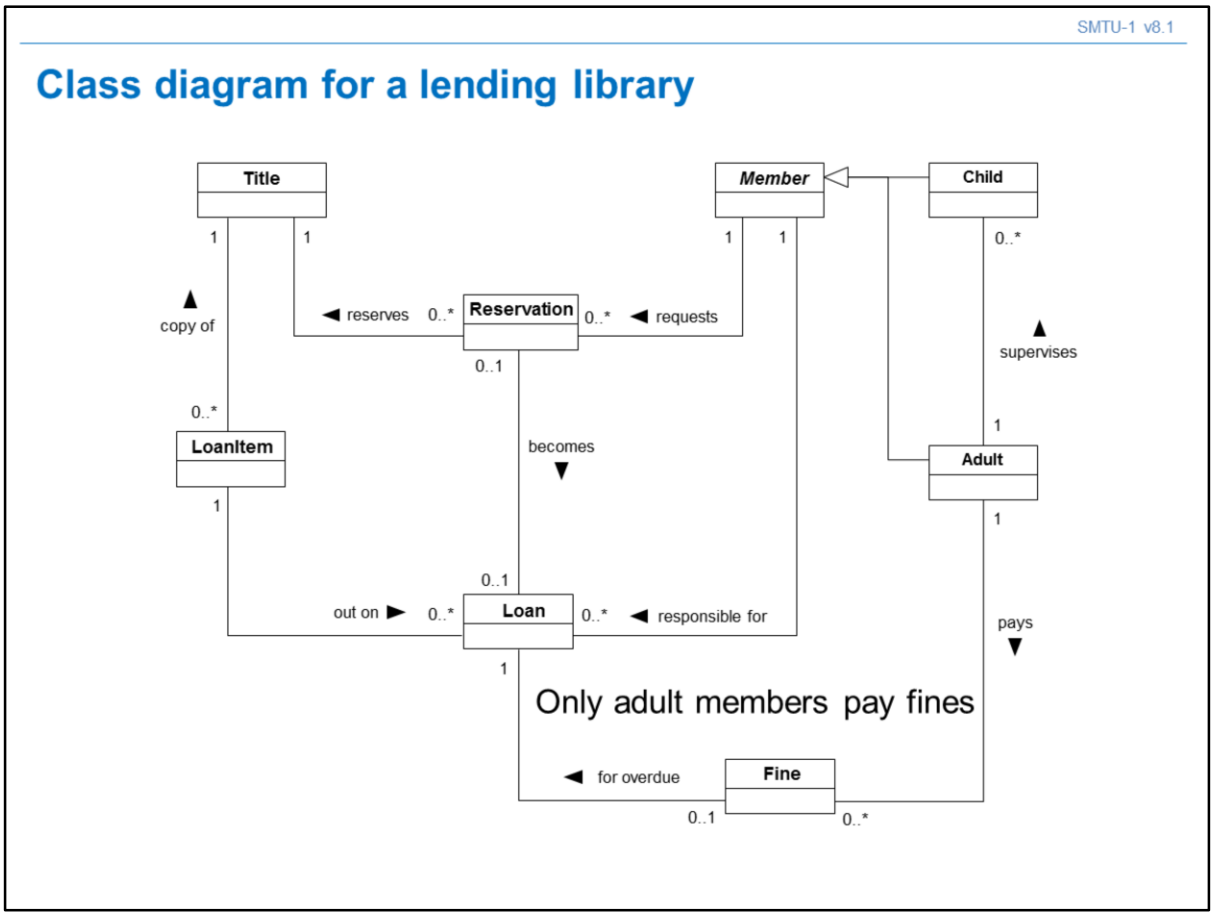
Note that the name of the superclass Shape is shown in italics. That is because it is an *abstract class*. This means it will never be instantiated; there will never be an object of class Shape.

Abstract classes are useful for holding all of the common attributes and operations of the sub-classes, but often a base class concept is only ever used through its specialisations.

SMTU-1 v8.1

Class diagram for a lending library





Specialisation clarifies that only Adult members pay fines and Child members must have an Adult to be responsible for them.

SMTU-1 v8.1

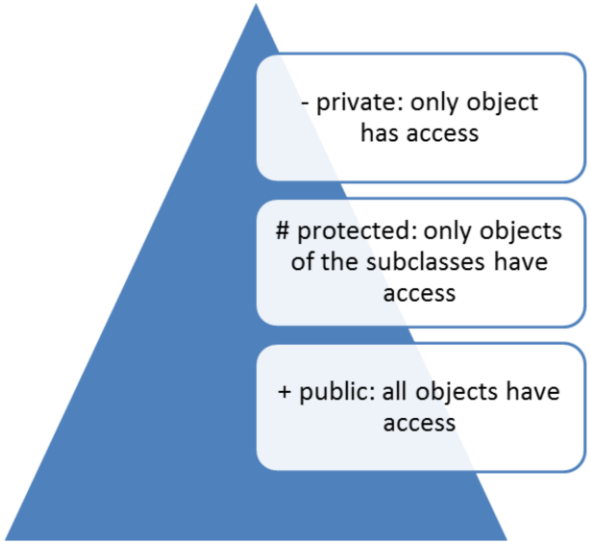
Activity

Complete the next hospital class diagram exercise.

SMTU-1 v8.1

Attribute and operation visibility

- **Visibility means 'accessibility by other objects'**

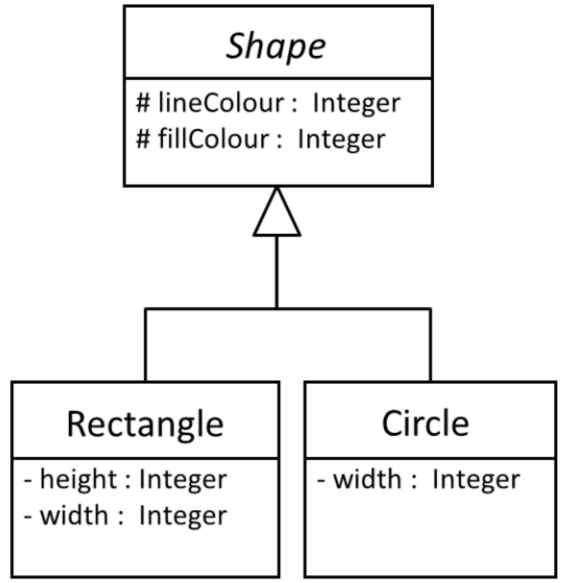


- **With most CASE tools, attributes are private by default and operations are public by default**

SMTU-1 v8.1

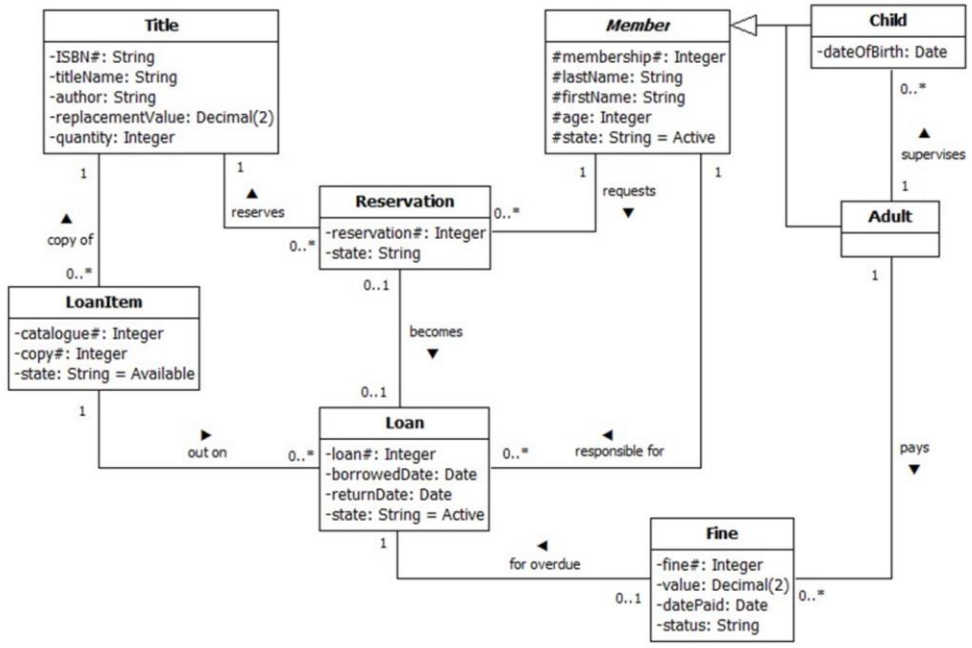
Attribute visibility for superclass

The subclasses can access these attributes but they remain private to any object from another class.



SMTU-1 v8.1

Lending library class diagram with attributes

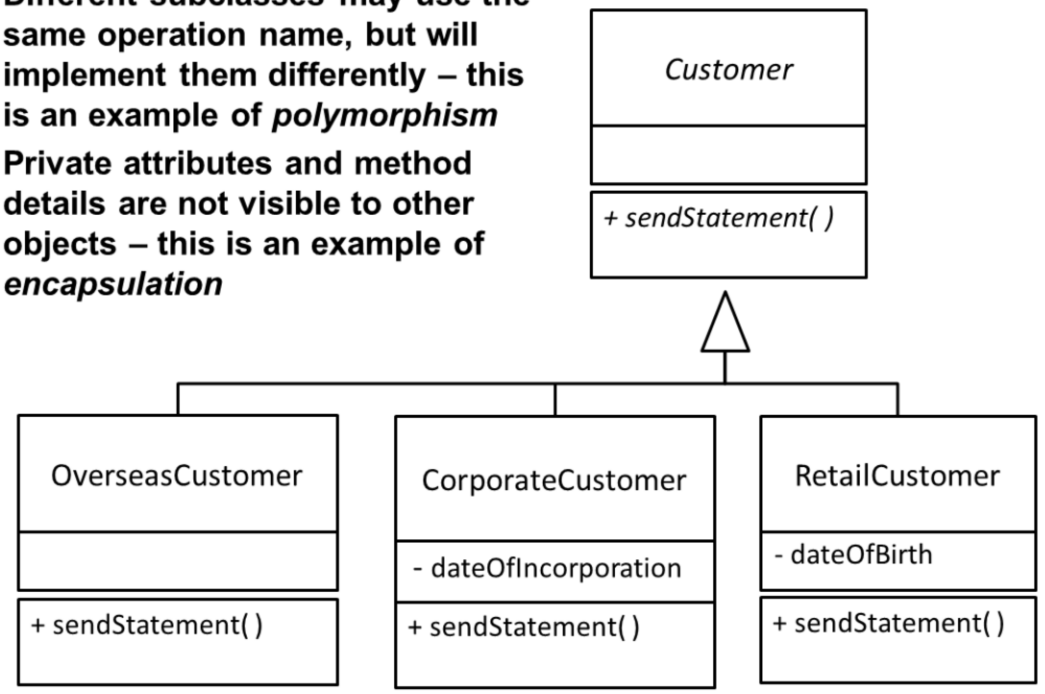


The Child class contains the attribute dateOfBirth. This is needed to support the business rule “A child member will become an adult member when they become 16”. The library is not interested in the birth date of Adult members.

SMTU-1 v8.1

Polymorphism and Encapsulation

- **Different subclasses may use the same operation name, but will implement them differently – this is an example of *polymorphism***
- **Private attributes and method details are not visible to other objects – this is an example of *encapsulation***



An operation may be defined as abstract on the base class, i.e. it is simply a 'placeholder' with no content. This will have the following effects:

- The base class itself must be defined as abstract.
- All the sub-classes must implement a concrete operation (i.e. one with content) with the same signature (or be abstract themselves).

Each sub-class might have different implementation details for this operation; in our example, different logic around sending a Statement. According to our example, any sub-class of Customer *must have* a concrete sendStatement operation. This is very useful for a *client object* in software that wants to use an object of type Customer to send a statement - the client object doesn't have to care *what* sub-class of Customer is involved. This is an example of *Polymorphism*. If we need to add in another type of Customer in the future, this is easy to do with no effect on clients that use this type of object.

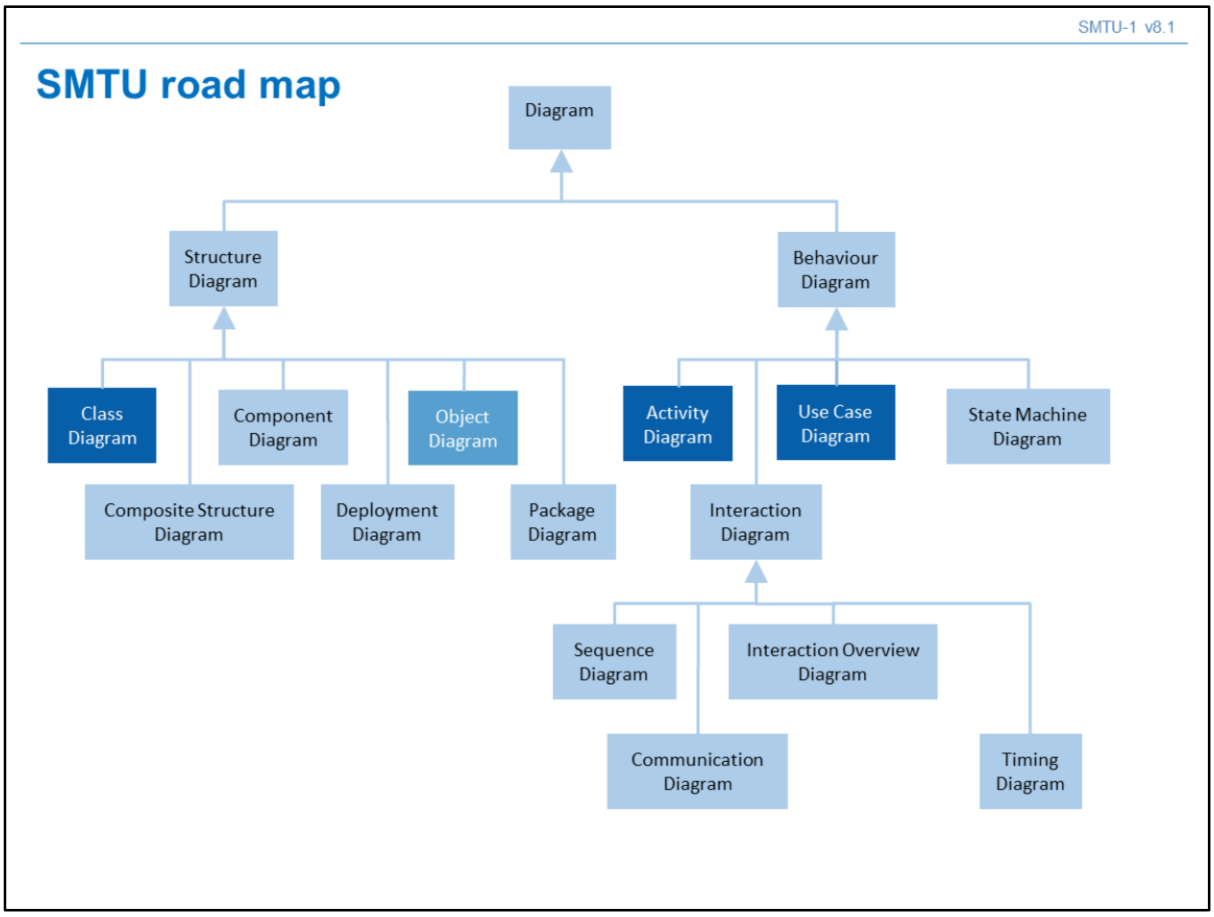
Also note that the operation's logic details are hidden from the client object using it, as are the private and protected data associated with the class. This 'hiding' of procedural logic and data and information is referred to as *Encapsulation*.

Promotion of Polymorphism and Encapsulation wherever practical are 2 basic principles of OO software architecture and development. They allow us to reduce dependencies within the software resulting in modular, flexible code, that is easy to maintain at low cost and low risk.

SMTU-1 v8.1

Activity

Under the guidance of your instructor complete this next exercise.


SMTU-1 v8.1
SMTU road map
Diagram
Structure Diagram
Behaviour Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
Use Case Diagram
State Machine Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Interaction Diagram
Sequence Diagram
Interaction Overview Diagram
Communication Diagram
Timing Diagram

SMTU-1 v8.1

Objects and classes

- **Objects can be considered as 'things'**
  - A person – Alan Turing 
  - A car – KS 51 YNE 
  - A company – BT
- **A Class is a template (blueprint) for Objects; an Object is an *instance* of a Class**
- **Objects can be described by their attributes; at any moment in time the attributes have values which overall determine the *state* of the object**

A class is simply a 'description of' or 'template for' a set of objects. The class defines, for example, the set of attributes (data) that all instances (objects) of that class will have. Each individual object holds its own values for its attributes.

SMTU-1 v8.1

Class diagrams and object diagrams

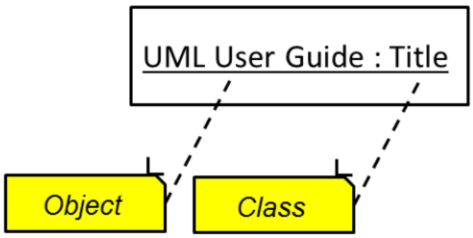
- **Class diagrams show all the classes for the area of business being analysed and how they relate to each other over time**
- **Object diagrams show how specific instances of the classes relate to each other at a specific instant in time**
- **Objects are written with their names underlined and representative instances of a class association are shown**
- **An object diagram may be useful to exemplify class relationships**
  - Sanity check!

So far we have looked at class diagrams. We will now have a look at object diagrams too.

SMTU-1 v8.1

Representing objects

- **Shown with *instance name* : *class name*, all underlined**



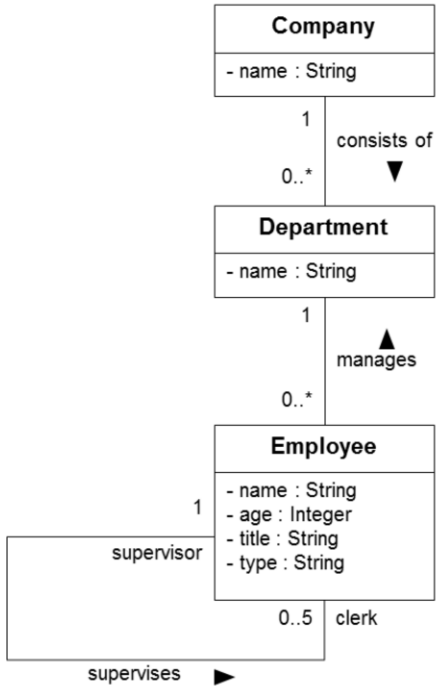
- **May have *anonymous* objects represented**

We represent the object in a rectangular box, the same as a class, but we now label it with the name of this particular instance followed by a colon and the class name. All of these are underlined.

Note that it is possible to have anonymous objects. This just means that we don't care *which* instance is being modelled, could be *any* one from the named class. This device for example is useful in *interaction diagrams* as we'll see later.

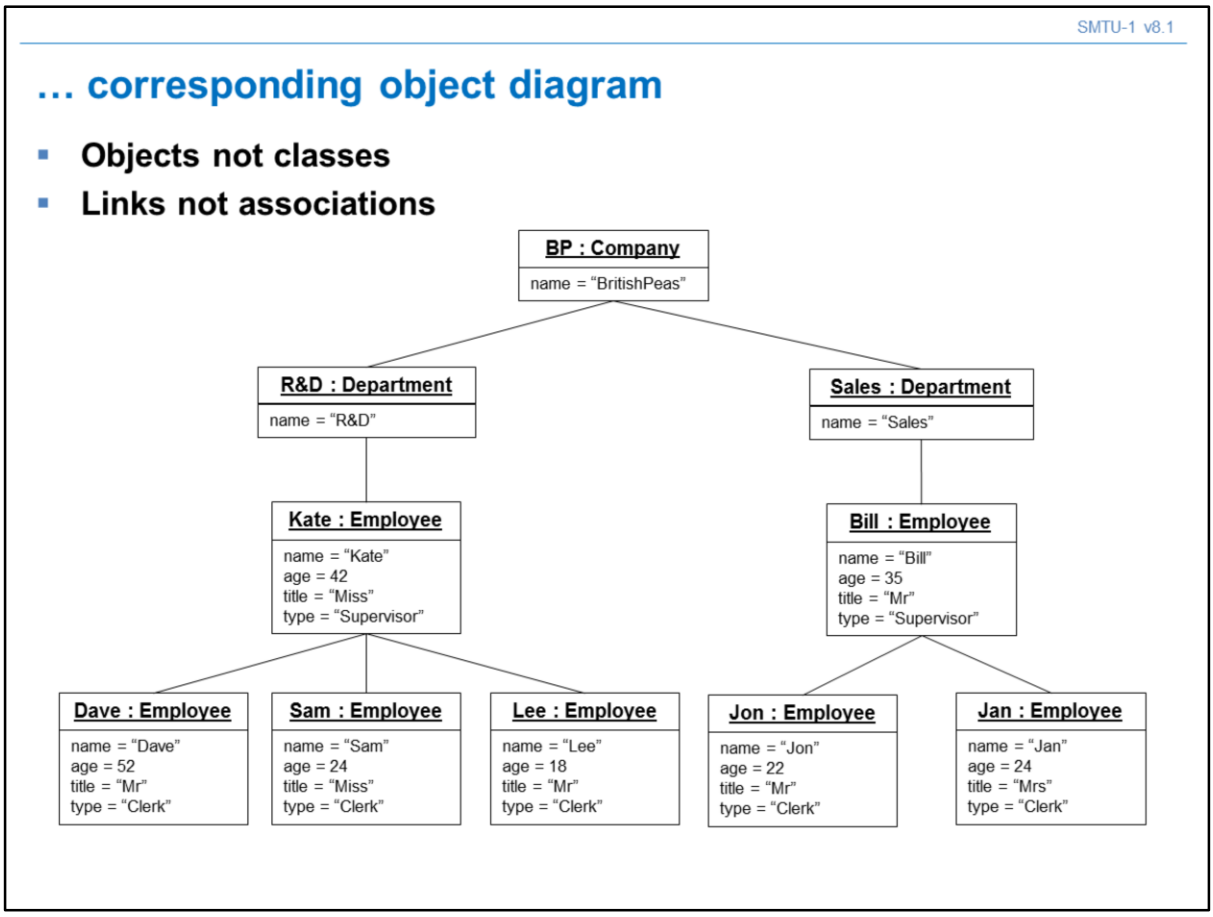
SMTU-1 v8.1

Class diagram …



This class diagram represents a familiar hierarchy with a reflexive association seen earlier.

Let's imagine that people new to class modelling might find the concept of a reflexive association hard to grasp. Representing the classes and the associations via an object diagram can make the associations clearer, and help to verify whether it all makes sense!



The object diagram describes what is happening to these particular employees at an instance in time. Note that on an object diagram the relationships are called *links* and are not named. Business people may find this type of diagram easier to validate than a class diagram.

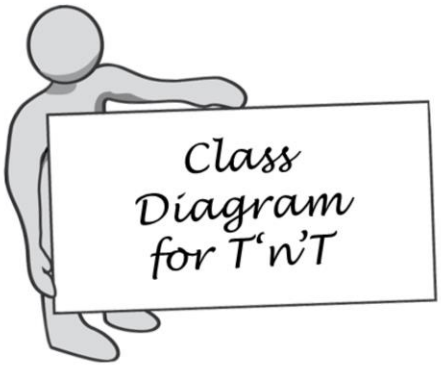
SMTU-1 v8.1

Activity

Produce a class diagram for the bakery.

SMTU-1 v8.1

Case study



Under the guidance of your instructor, complete the next exercise.

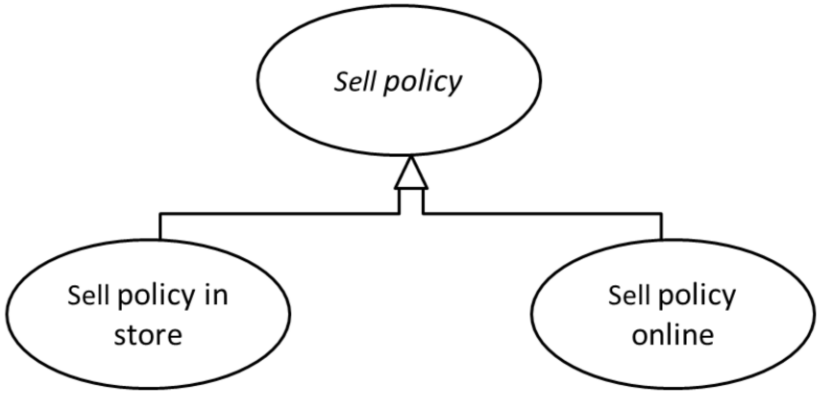
SMTU-1 v8.1

Generalisation and specialisation recap

- **Generalisation is the relationship between a general superclass and more specific subclasses or subclass**
- **Subclasses inherit all of the attributes and operations of their superclass but have their own specialisations**
- **The principal of generalisation can also be applied to Use Cases and Actors**
  - in UML these are examples of *classifiers* – i.e. Classes!

SMTU-1 v8.1

Generalising use cases

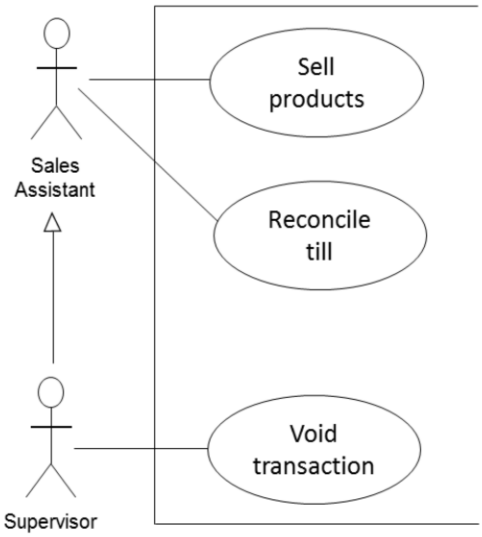


This form is not widely used in practice

SMTU-1 v8.1

Generalising actors

- **'Child' actors inherit all of the characteristics of 'Parent' actors, plus those specific to their role**



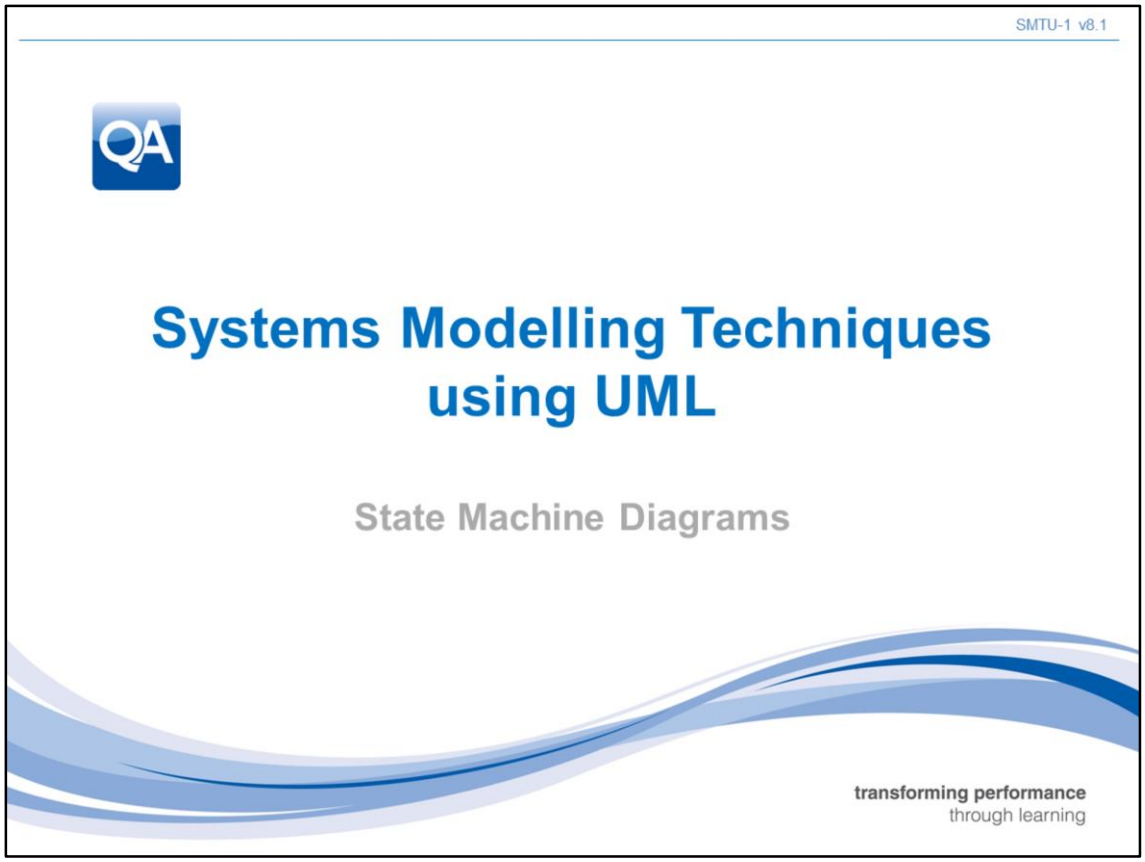
This form may be very useful for simplifying a UC diagram.

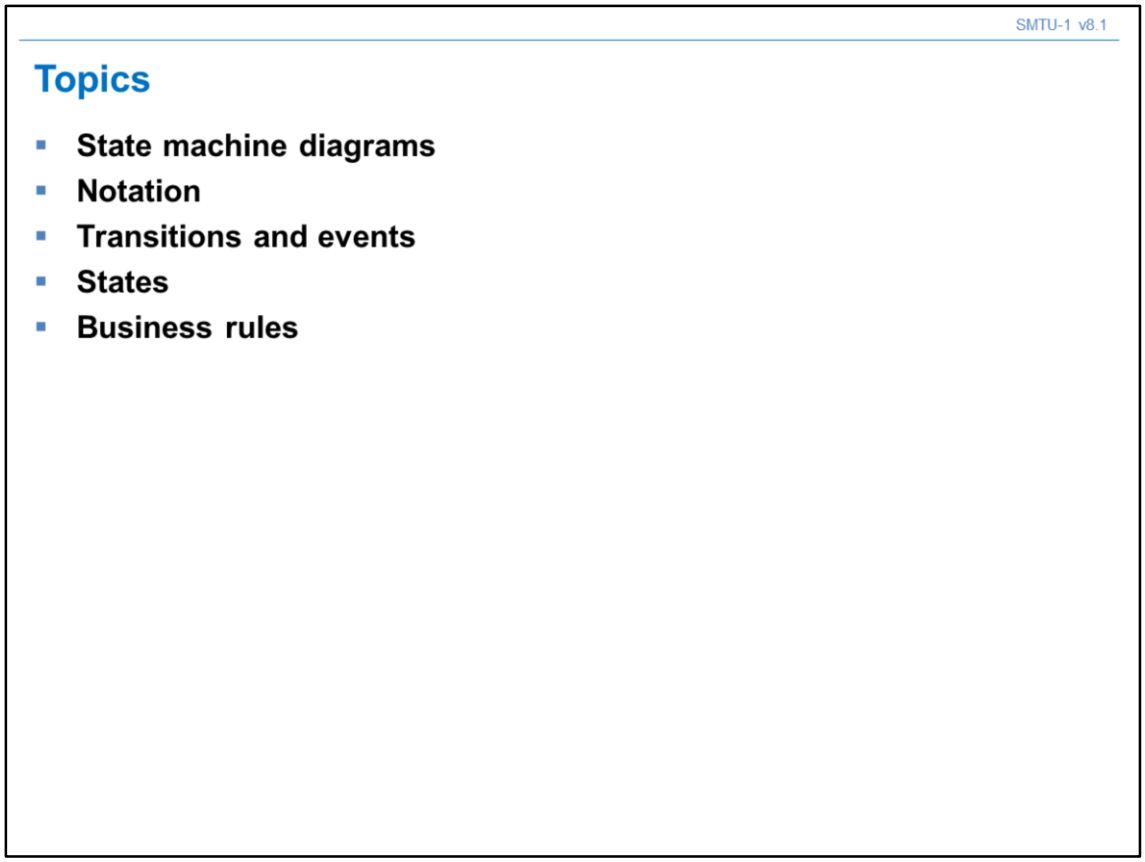
SMTU-1 v8.1

Summary

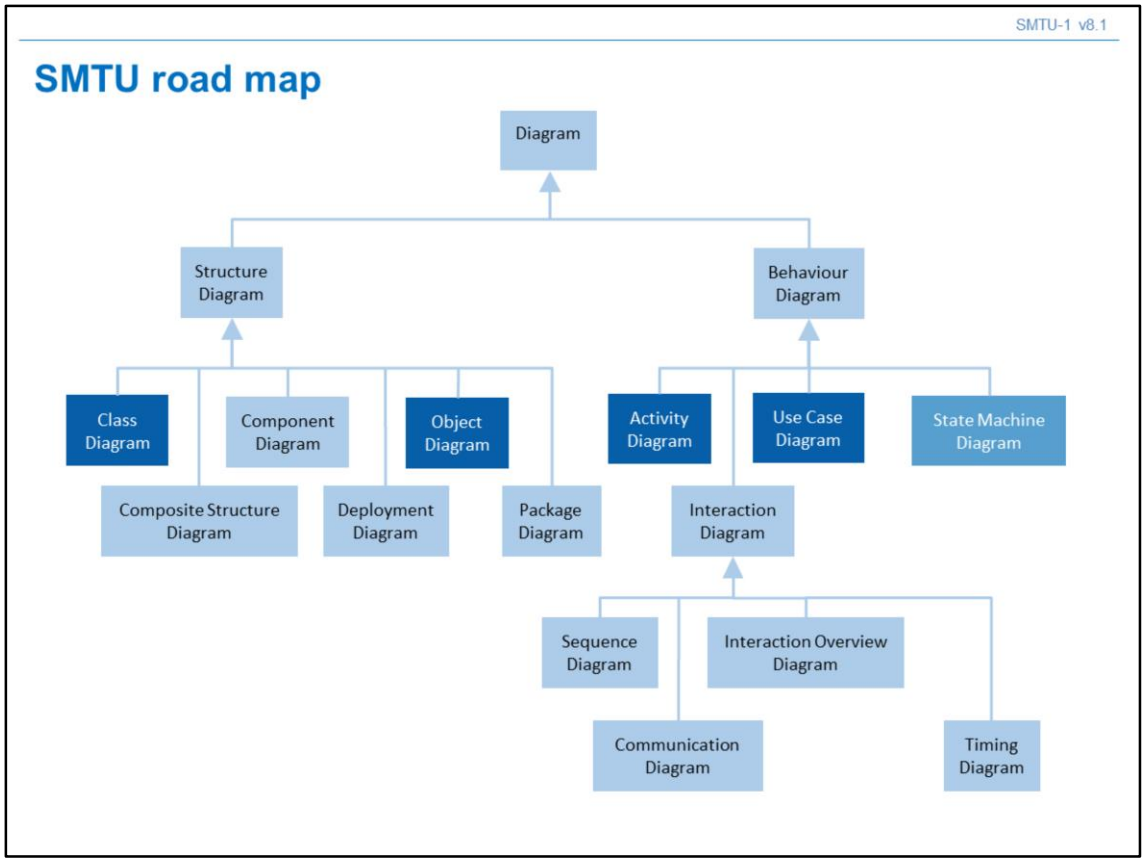
In this session we covered:

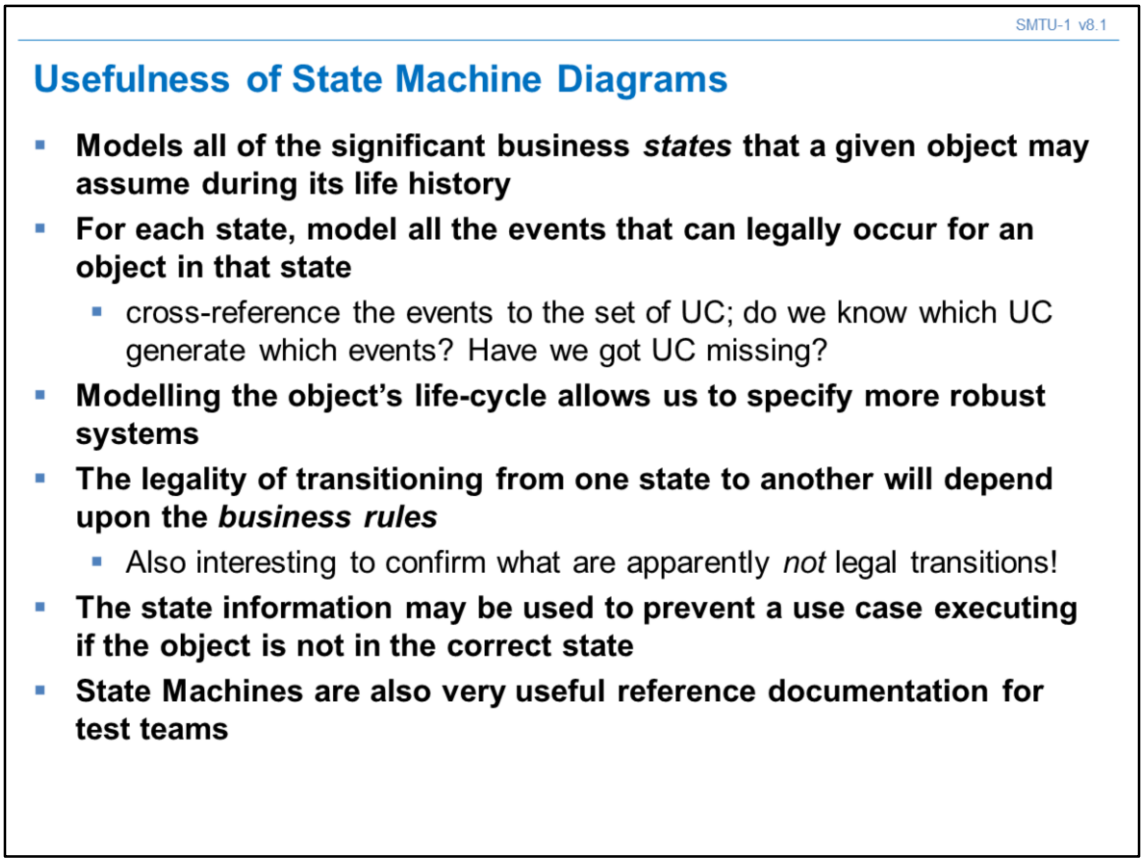
- **The purpose of analysis domain class models**
- **Classes, Attributes and Associations**
- **Objects are instances of Classes**
- **Exploring class hierarchies**
  - Generalisation and specialisation
- **Special associations between classes**
  - Composition and Aggregation
- **Object Diagrams can be helpful as a 'sanity check'**
- **Specialising Use Cases and Actors**


SMTU-1 v8.1
Systems Modelling Techniques using UML
State Machine Diagrams
transforming performance
through learning



We looked in the last session at how to produce a static model of the classes. Instances of the classes (objects) will be created, updated and deleted throughout their existence. Here we will look at how to model dynamically what happens to the objects.


SMTU-1 v8.1
SMTU road map
Diagram
Structure Diagram
Behaviour Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
Use Case Diagram
State Machine Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Interaction Diagram
Sequence Diagram
Interaction Overview Diagram
Communication Diagram
Timing Diagram

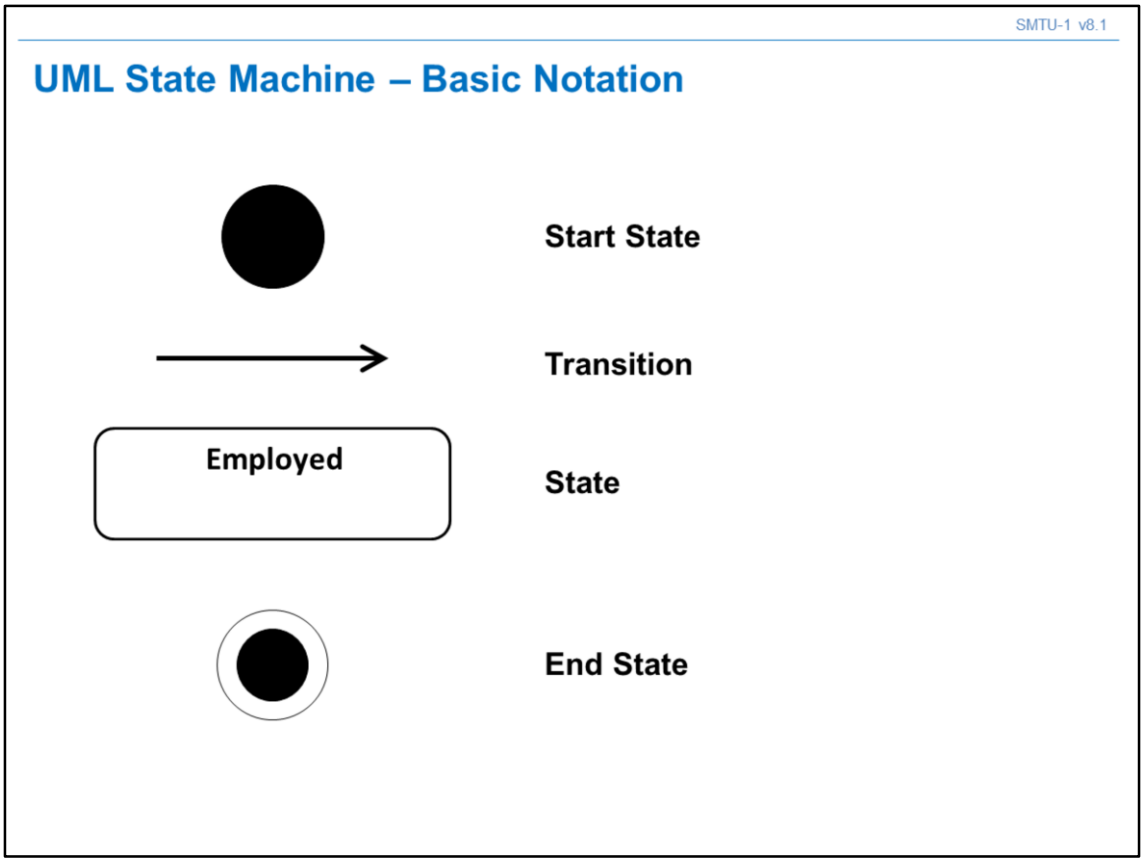


A State Machine can be used to model the lifecycle of an object from any class, but here we are interested only in the Domain classes.

Based on our Domain model, we model all of the states that any given object ('anonymous' object) from each Class may assume. For each state, we then model all the events that can legally occur for an object in that state. All objects have a lifecycle, (i.e. birth, mid-life and 'death'), and modelling the lifecycle allows us to specify more robust systems. This is another way we can use UML to capture business rules and cross-reference models.

In practice, given project constraints, we might only create a State Machine Diagram for certain key objects; those with 'interesting' lives, central to the application, that are subject to many events caused by the use cases that manipulate them.

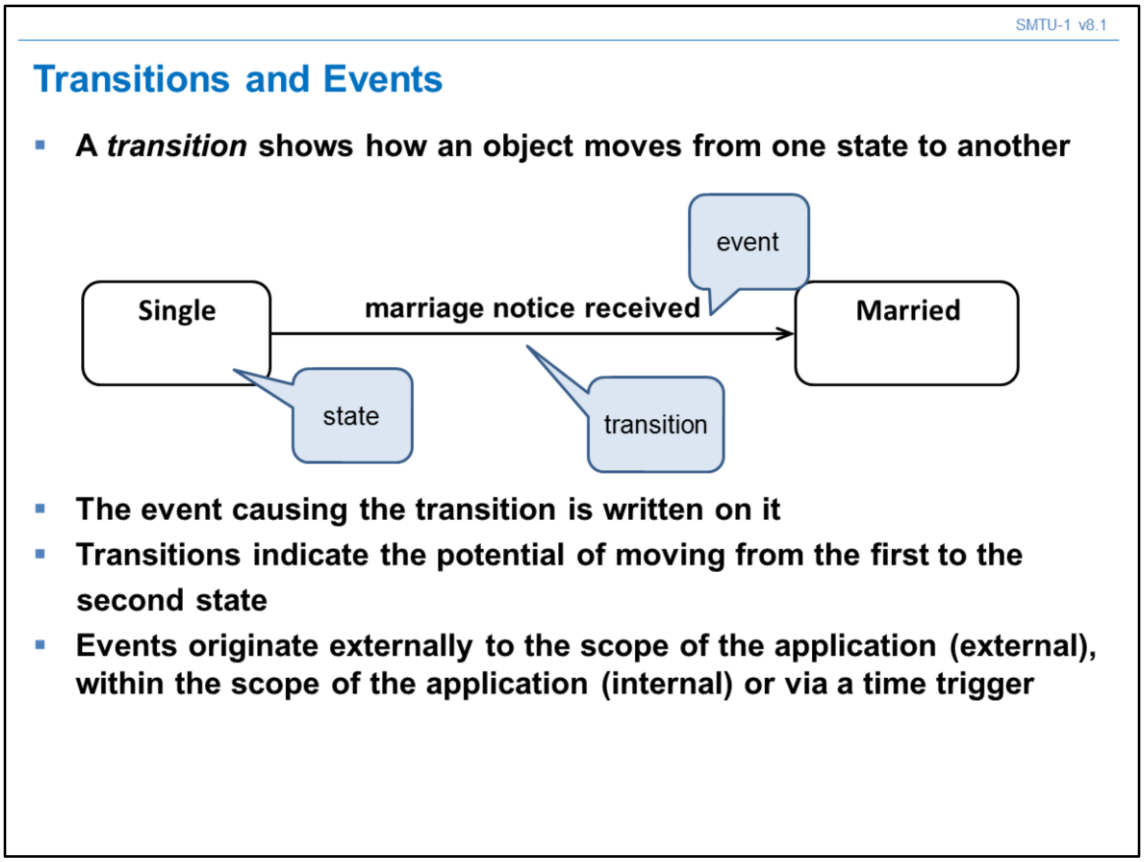
The systems we develop must recognise 'state' and cater for changes in the state of objects; if not erroneous processing will occur and the integrity of the system will be effected.



A UML state is shown as a rounded box with the state name enclosed. These states are captured because they are important to the business being modelled. All states, other than the end state, must have a transition to some other state.

A start state is shown as a filled circle. This is the state before the object of the class is created. For example, it is the state a person is in before they apply for a job; they are unknown to the Employer. There should always be at least one start state on a State Machine diagram.

An end state is shown as a ‘target’. It is the state after the object is destroyed; i.e. it is unknown to the system again. Normally there is a need for a period of time to elapse before the object is removed and thus becomes unknown. There is almost always an end state, depending on the business rules and sometimes influenced by regulation. NB: ‘purge’ is the word commonly used to describe the complete removal of an object. ‘delete’ may simply mean putting the object into a ‘deleted’ state.



A transition moves the object from one recognised state to another and is shown as an arrowed line between the states. The presence of a transition means it is possible to move from one state to another (here, single to married). NB: The *absence* of a transition means that it is *not possible* to move to that state. The event that causes the transition is shown as text on the transition.

Technically in UML any change in an object's data changes the state of the object. However in this context we are only interested in identifying 'state' that is significant in business terms. Hence some (permissible) events cause a change of (business) state while some do not. For example, a 'change of address' event may be permissible for any state within the life-cycle of a customer object, and it will not change the actual 'state' of the customer, from a business point of view.

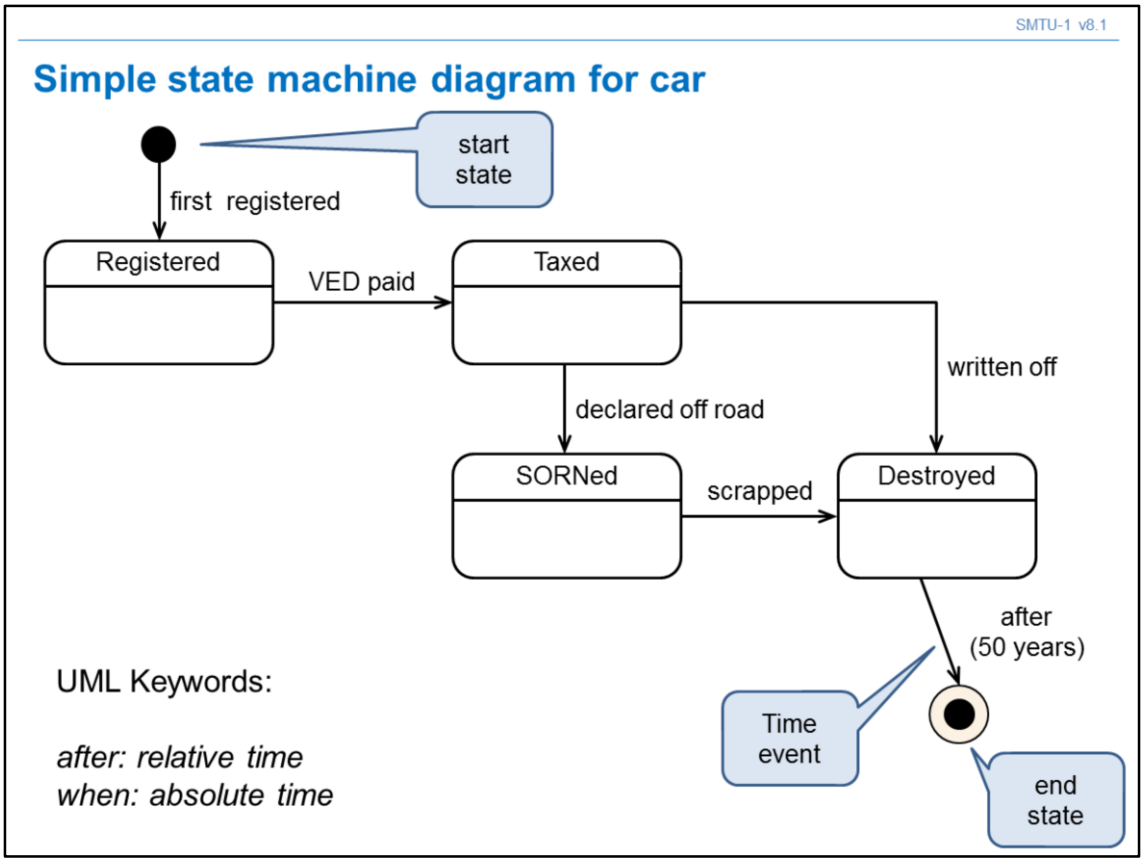
Events for a 'car'

- **The DVLA maintain records of all vehicles in the UK**
- **The events of interest for a 'Car' might be:**
  - First registered
  - VED paid
  - Taken off road
  - Written off

States of interest for car

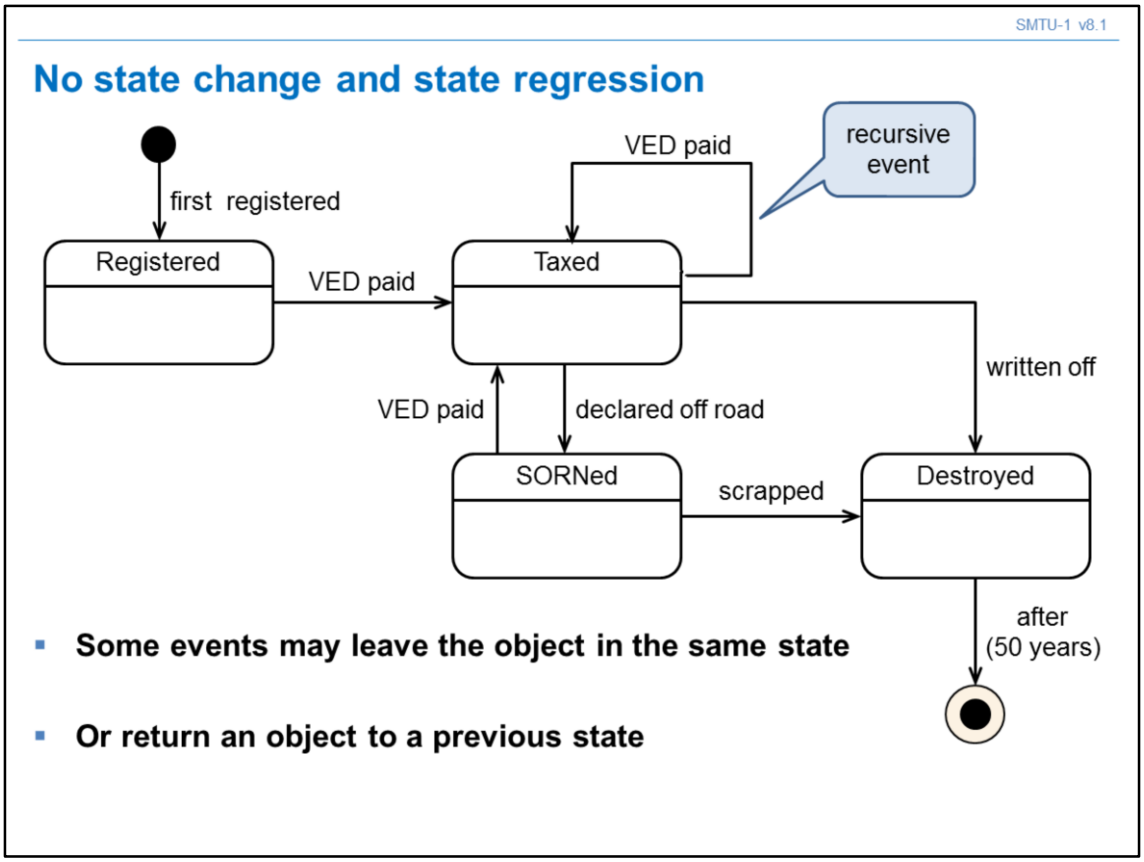
- **The resulting 'state' of a Car after undergoing each event:**

	Event		State
After	First registered	Car becomes	Registered
After	VED paid	Car becomes	Taxed
After	Taken off road	Car becomes	SORNed
After	Written off	Car becomes	Destroyed
After	Scrapped	Car becomes	Destroyed



It should be obvious that the state machine diagram captures important business rules. It determines the constraints on the modification of the object and (as we shall see) the actions that must be carried out as a result of attempts to modify the object. These rules must eventually be implemented in the software.

after is a UML keyword to denote the elapse of time (relative time). Also available is the keyword *when* for absolute time.

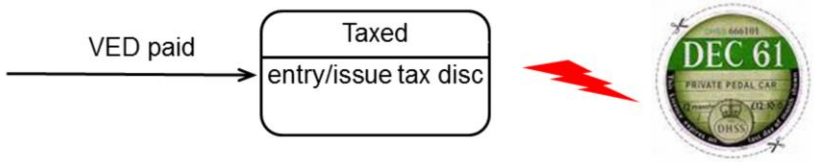


A recursive event can be noted which doesn't change the state of the object, but it will provoke any state exit/entry behaviour.

SMTU-1 v8.1

Action events

- **When a event occurs, it may initiate some action on entry to a state**

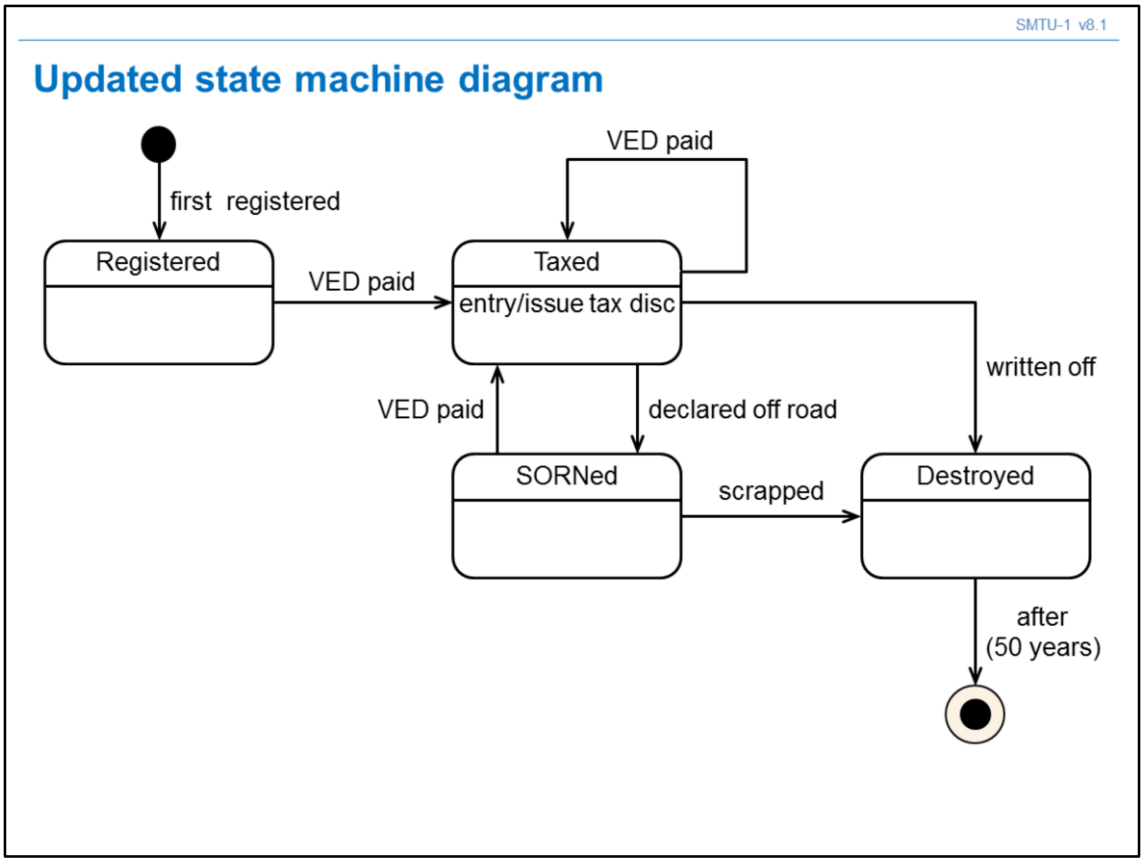


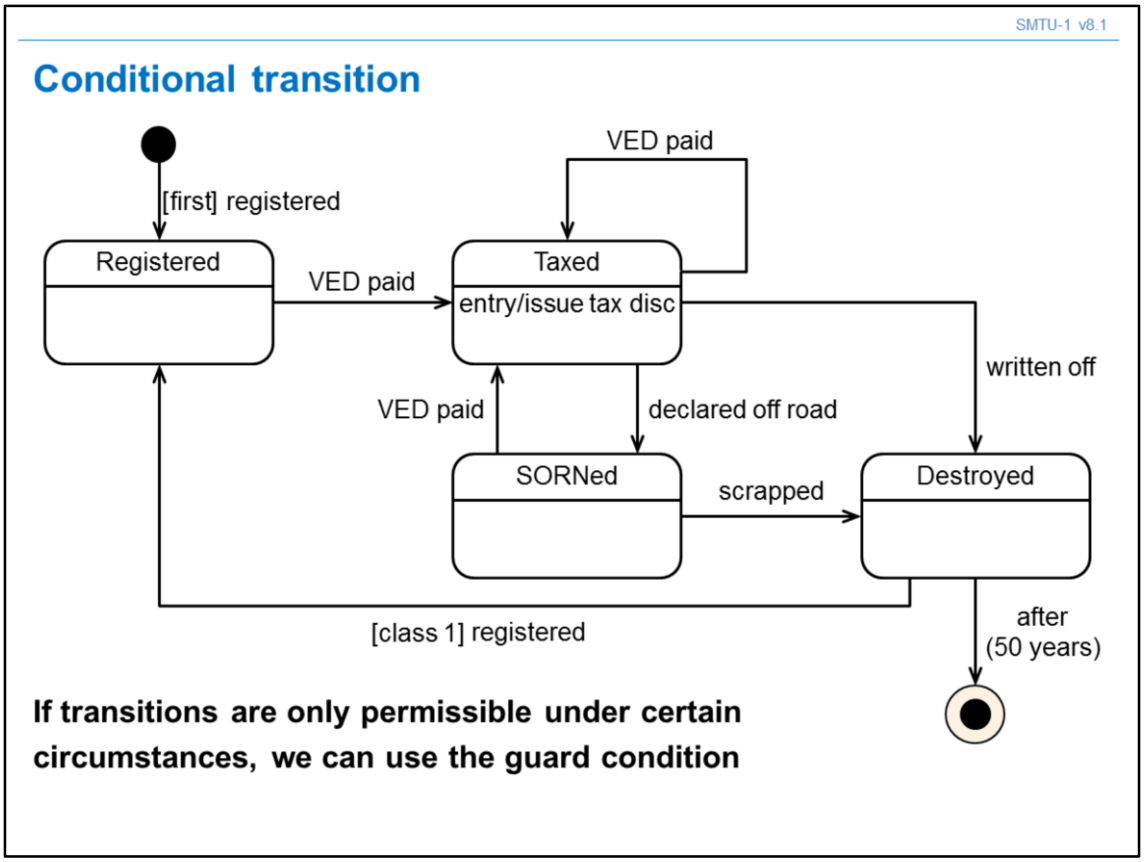
- **Other popular keywords:**
  - *Exit*: actions to execute on exit from the state
  - *Do*: actions to perform while in the state

When an event occurs on an object we may want to initiate an action There are several possibilities here:

- We might want to fire the action whenever we enter a state – an ‘Entry action’
- We might want to fire the action whenever we leave a state – an ‘Exit action’
- We might want to fire the action while in the state – a ‘Do action’
- We might want to fire the action only when we enter a state from some specified other state – an action on the transition.

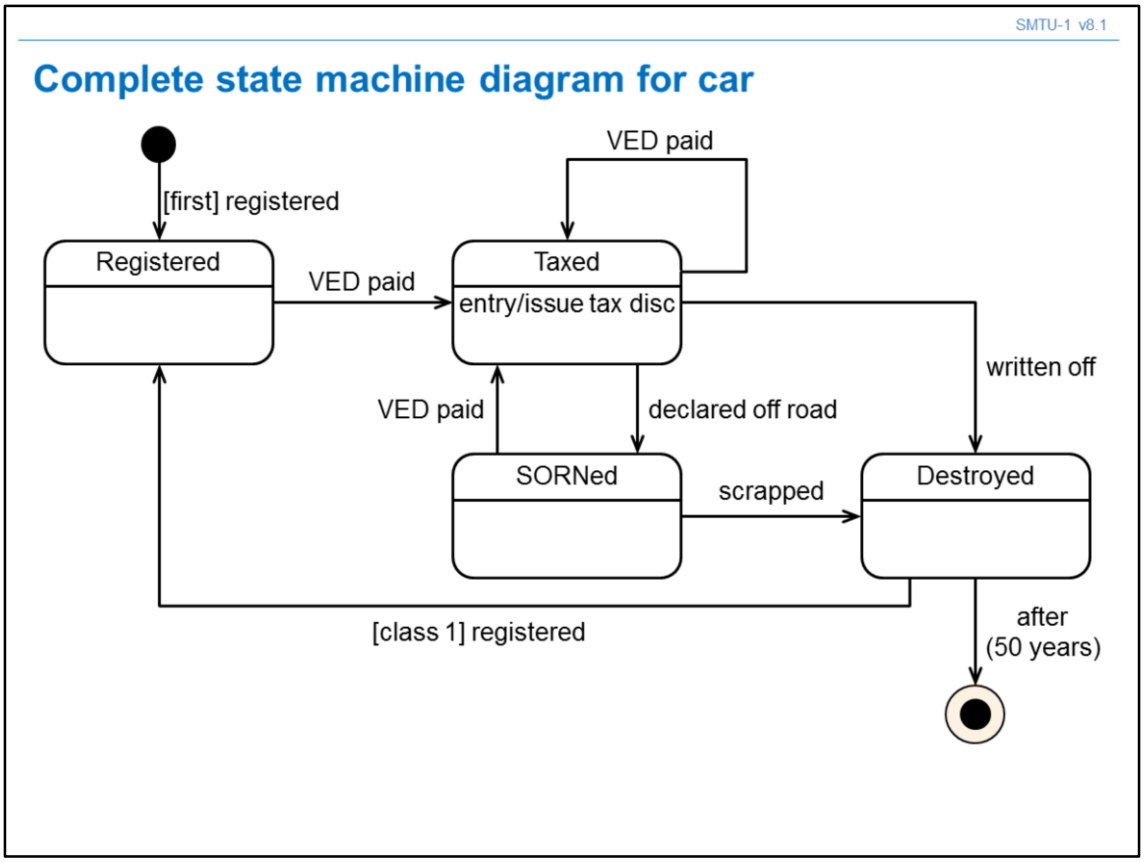
In the diagram, any event that causes a transition to the ‘Taxed’ state will cause a tax disc to be issued when the state is first entered. This is an example of why we would document a recursive event; each time a vehicle is taxed we want to ‘enter’ the state to provoke the issue of a license.


SMTU-1 v8.1
Updated state machine diagram
first registered
Registered
VED paid
Taxed
entry/issue tax disc
VED paid
written off
VED paid
declared off road
SORNed
scrapped
Destroyed
after
(50 years)



We can put guard conditions on transitions to indicate that the event will only be serviced if the condition is true. In the example, it is only possible to re-register class 1 write-offs.

If you have two events with the same name leaving the same state, then they must be labelled with different (mutually exclusive) guard conditions.


SMTU-1 v8.1
Complete state machine diagram for car
[first] registered
Registered
VED paid
Taxed
entry/issue tax disc
VED paid
written off
VED paid
declared off road
SORNed
scrapped
Destroyed
[class 1] registered
after
(50 years)

SMTU-1 v8.1

Workshop

- **For a public lending library, identify potential…**
  - States
  - Events
- **For a loan item**

SMTU-1 v8.1

- **Intentionally blank**

SMTU-1 v8.1

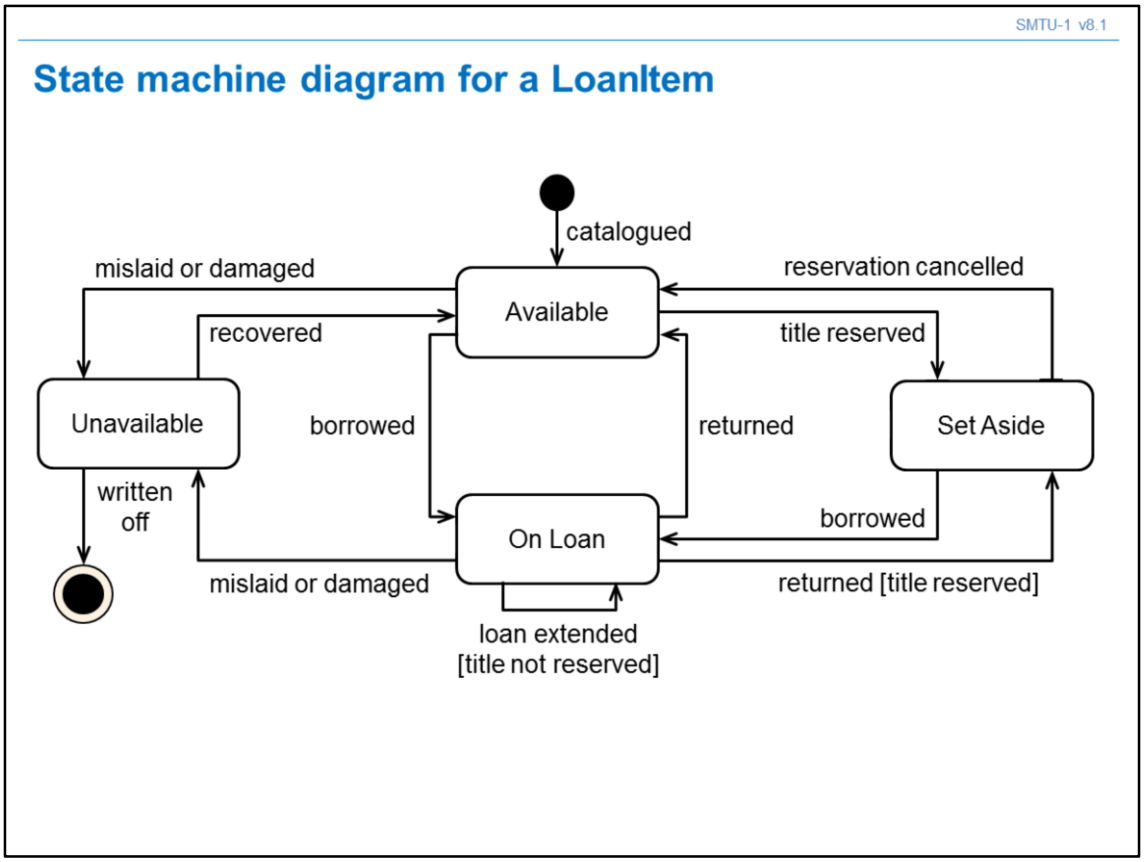
State machine diagram for a LoanItem

Potential Events	Potential States
borrowed	On Loan
returned	Available
loan extended	On Loan
catalogued	Available
title reserved	Set Aside
mislaid	Unavailable
recovered	Available
written-off	End state

Conventions for event labelling are not defined in UML. In practice there seems to be 2 variants:

- Events labelled as something that happened (past participle)
- Events labelled as an action (verb-noun)

It is of course important to create a standardised approach.



A good tip for drawing these diagrams is to decide on a list of relevant states and then map out the ‘normal’ route first; what would the ‘normal’ lifecycle look like?

The typical life cycle of the states for a loan item, once catalogued, toggles between ‘Available’ and ‘On Loan’. At some point it becomes unusable and is written off.

Further research reveals that:

A member could ask to extend the loan period and this would be allowed, providing no other member had requested a reservation.

If the loan item was available in the library when the title was reserved then a loan item copy of the title would be set aside. At some point the member will arrive at the library to borrow the loan item.

When a loan item is returned and the title has been reserved then the loan item will be set aside rather than made available to other members.

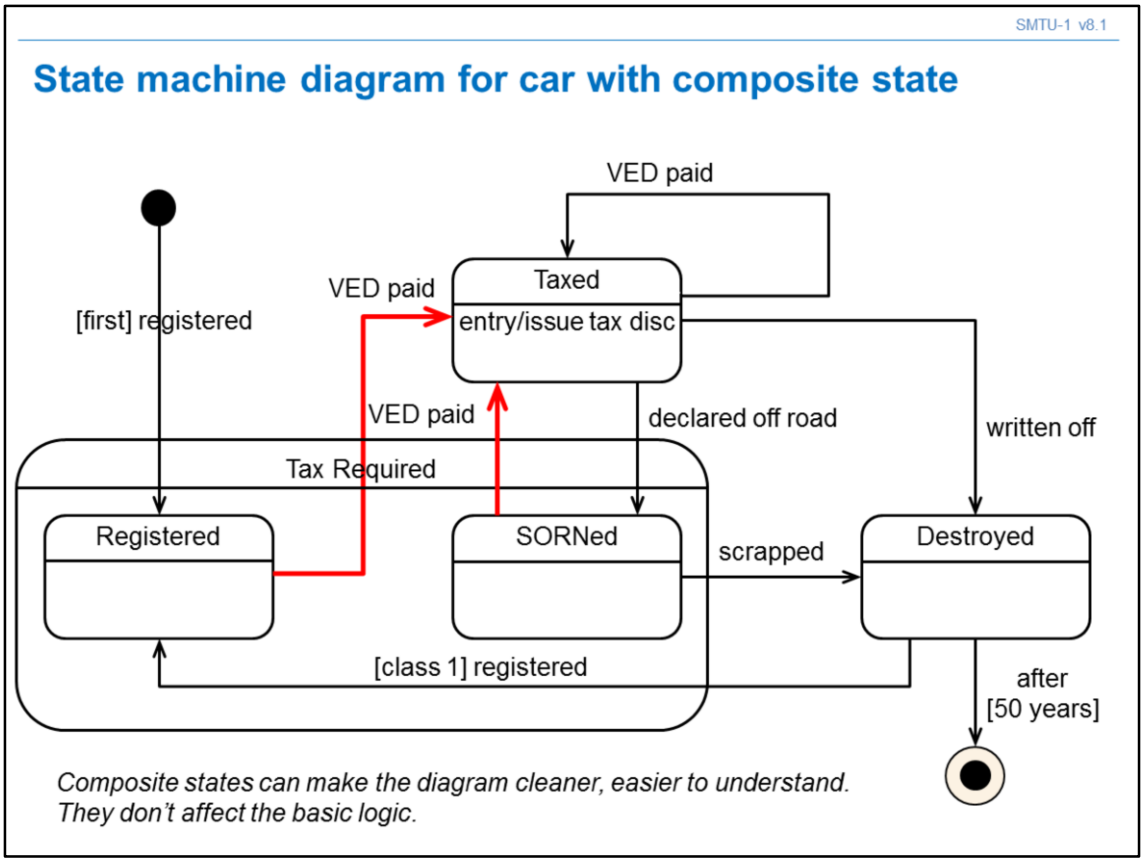
A member could cancel a reservation before the loan item is borrowed or the reservation could lapse after a set period. This would be determined by a business rule.

A loan item could be lost or stolen (mislaid) or damaged whilst in the library or on loan. Ideally the loan item will be recovered (which would include repair) or it will have to be written off, again determined by business rules.

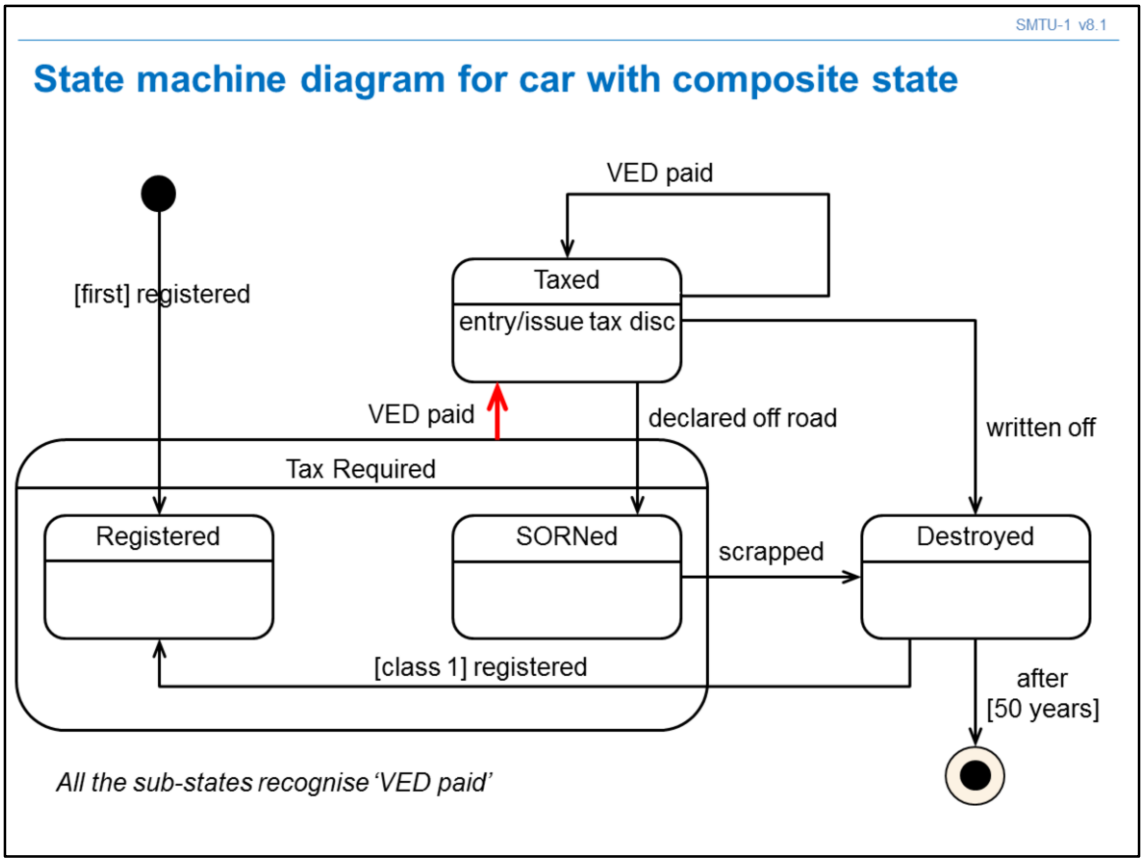
SMTU-1 v8.1

Composite states

- Two states in the 'Car' State Machine Diagram recognise the 'VED paid' event that transits to the single state 'Taxed'
- Could use a *composite state* that includes those states as sub-states



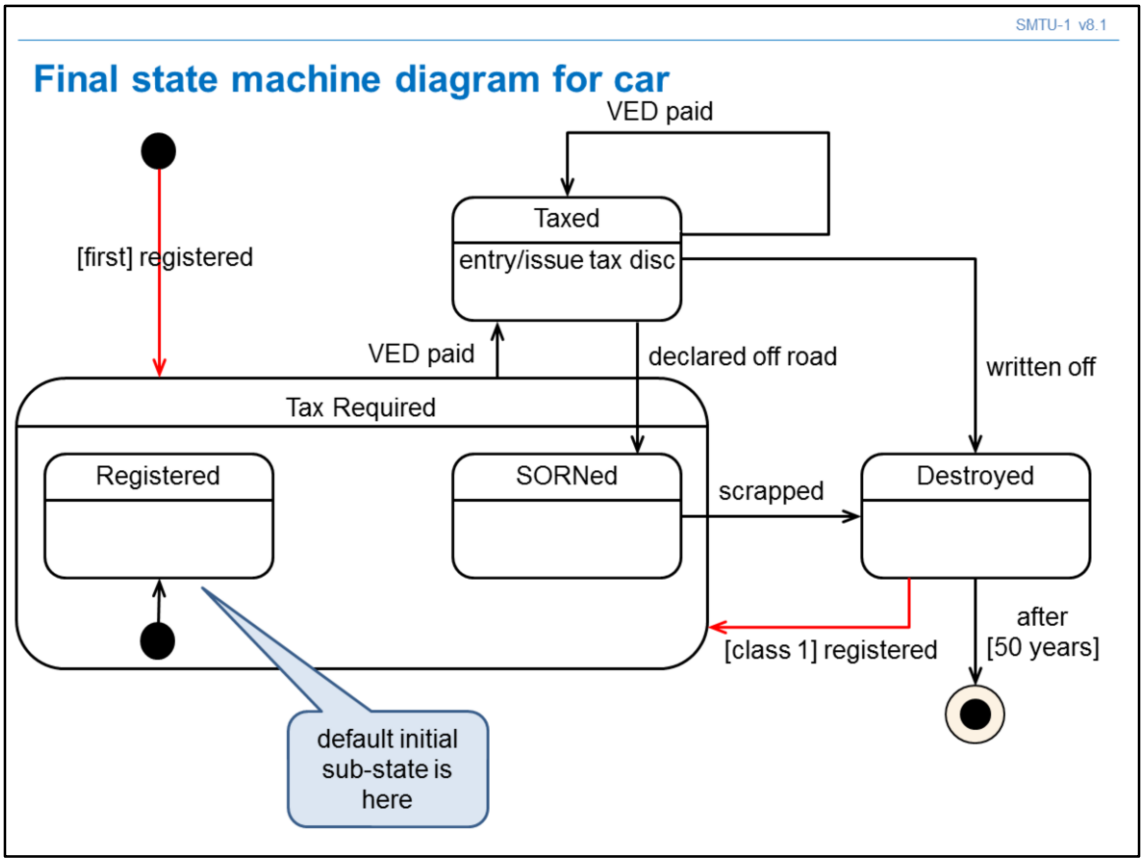
The composite state 'Tax Required' has 2 sub-states 'Registered' and 'SORNed'. A composite permits modelling transitions that all its sub-states can recognise. The sub-states may have their own transitions, which are not shared.


SMTU-1 v8.1
State machine diagram for car with composite state
VED paid
[first] registered
Taxed
entry/issue tax disc
VED paid
declared off road
written off
Tax Required
Registered
SORNed
scrapped
Destroyed
[class 1] registered
after
[50 years]
All the sub-states recognise 'VED paid'

SMTU-1 v8.1

Transitions to composite states

- More than one event in the 'Car' State Machine Diagram provokes a transition to the 'Registered' state in the 'TaxRequired' composite state
- Use the 'start state' notation to show the default starting sub-state when a composite state is entered

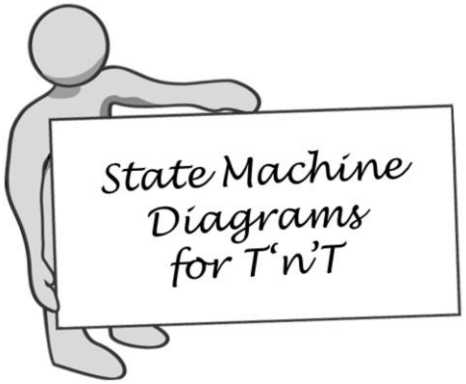

SMTU-1 v8.1
Final state machine diagram for car
VED paid
[first] registered
Taxed
entry/issue tax disc
VED paid
declared off road
written off
Tax Required
Registered
SORNed
scrapped
Destroyed
after
[50 years]
[class 1] registered
default initial
sub-state is
here

SMTU-1 v8.1

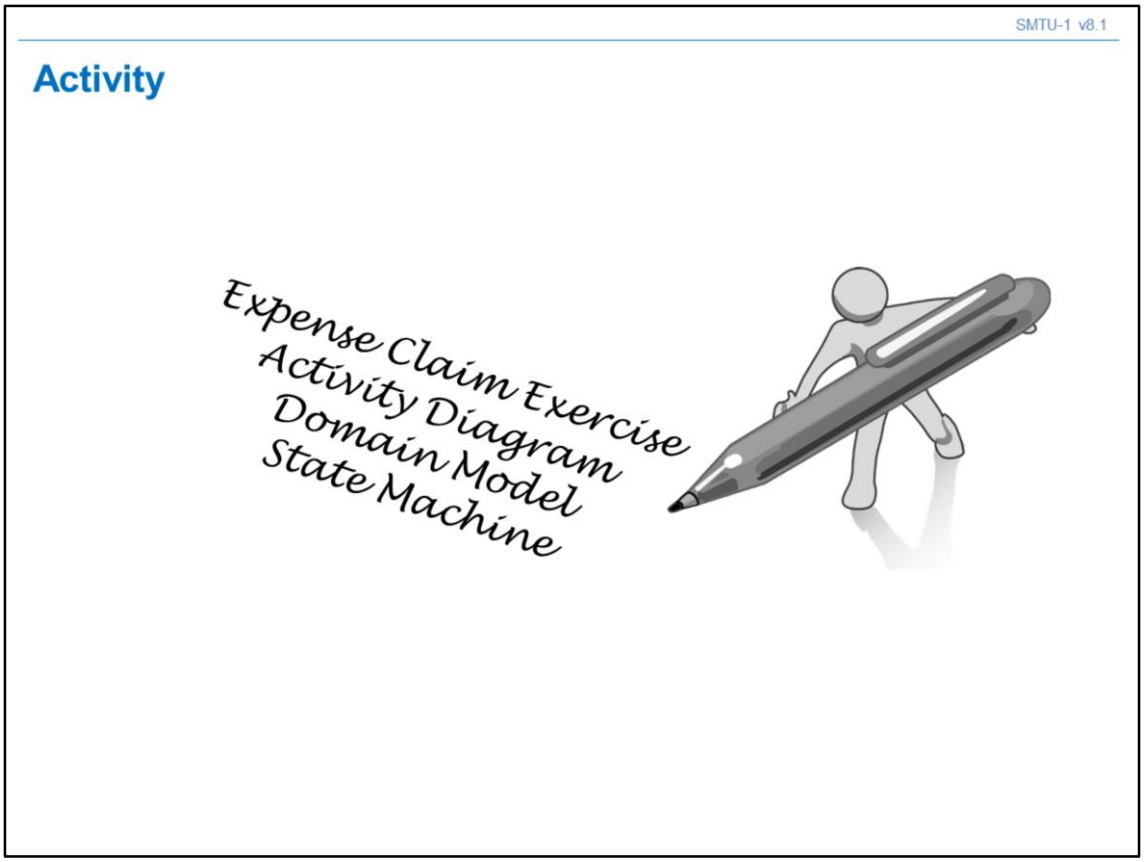
Exercise

SMTU-1 v8.1

Case study



Create State for this exercise under the guidance of your Instructor.

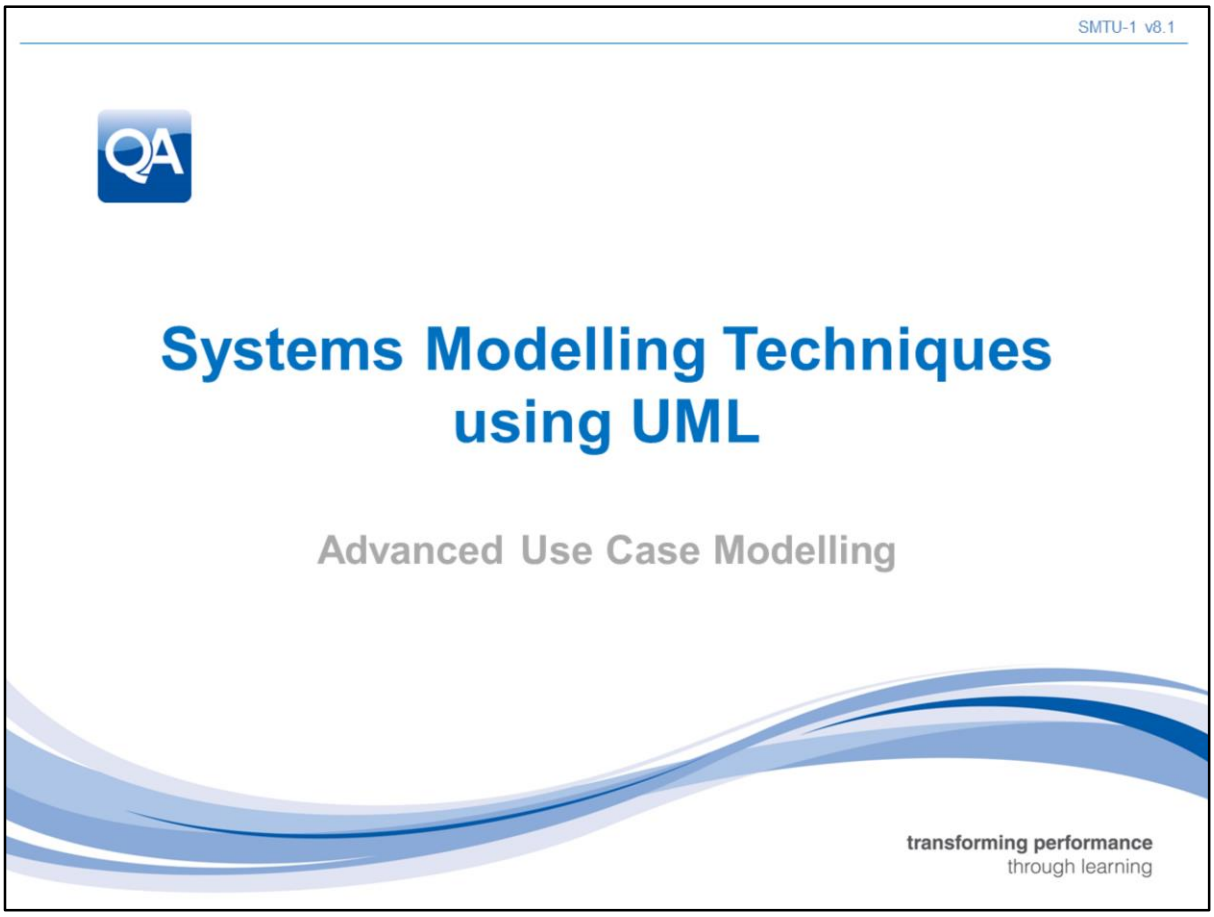


This exercise contains Activity Diagrams, Domain Class Models and State Machines, and so is similar to the BCS exam structure

SMTU-1 v8.1

Summary

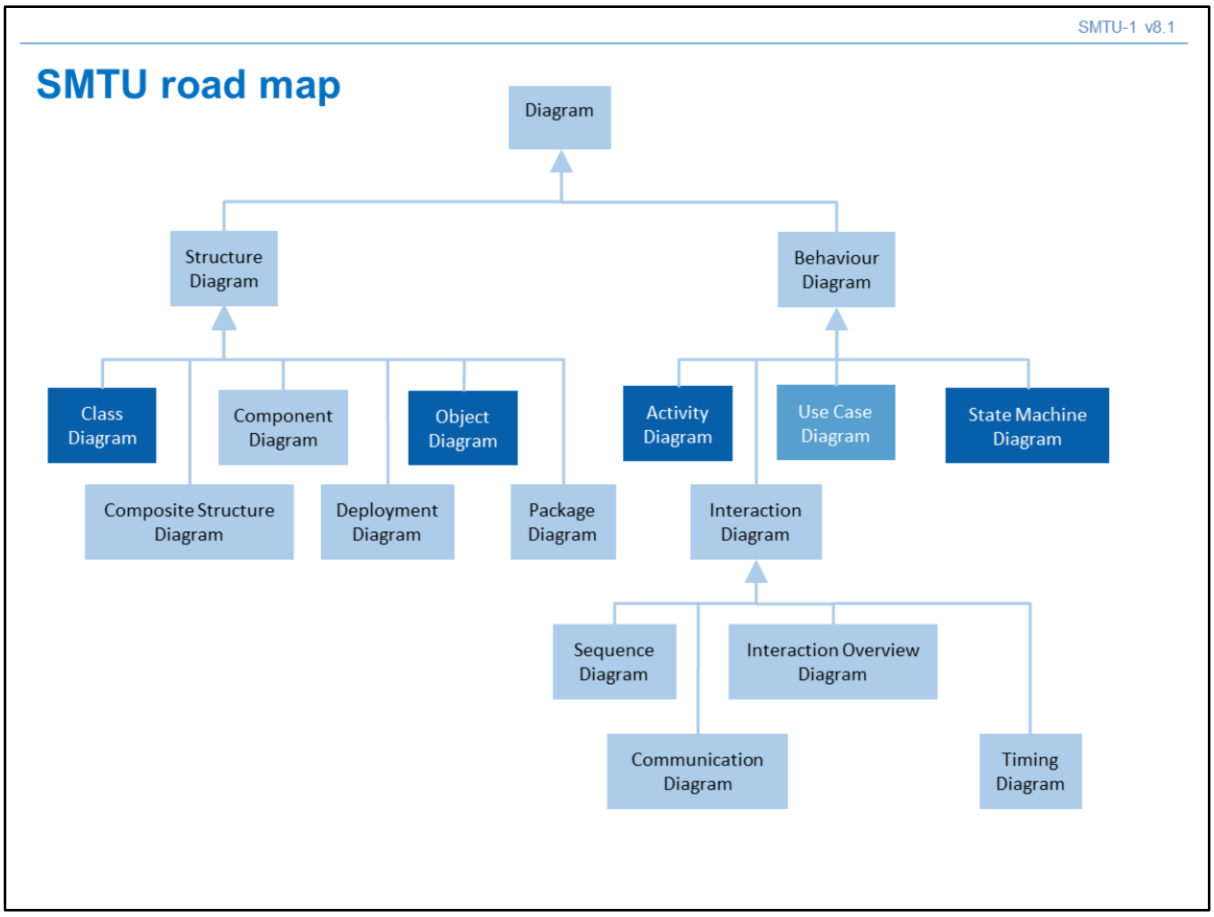
- **State Machine Diagrams model an object's life-cycle from creation, through various states, to deletion**
- **All of the states that an object may assume are modelled**
- **For each state, all the events that can legally transition to or from that state are modelled**
- **Events trigger use cases. For each event is there a matching use case?**
- **For each use case**
  - What was the state of the object before the event occurred?
  - What was the state of the object after the event occurred?
- **These states help define the pre-conditions and post-conditions of the complete use case specification (next session)**


SMTU-1 v8.1
Systems Modelling Techniques using UML
Advanced Use Case Modelling
transforming performance
through learning

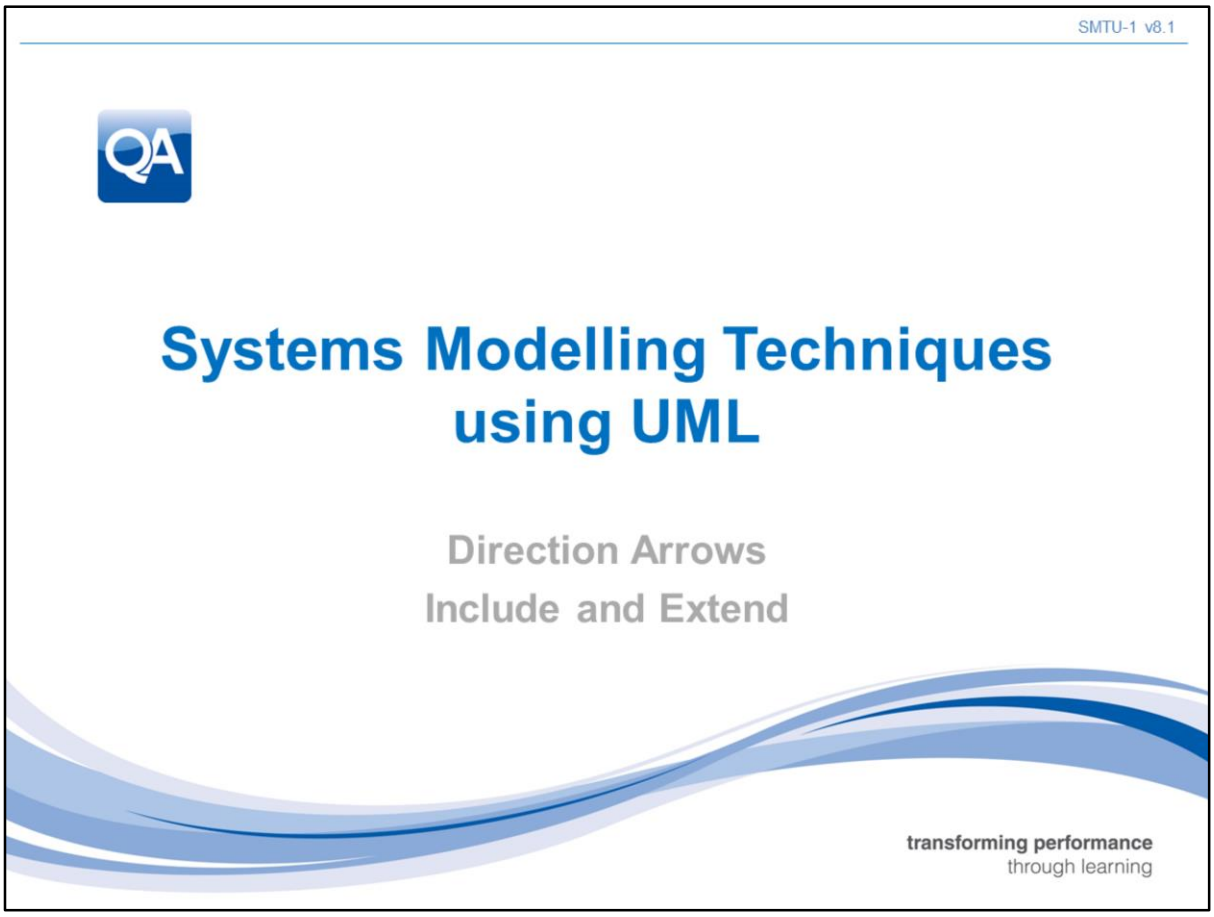
SMTU-1 v8.1

Topics

- Use case relationships
- Detailed use case descriptions
- Activity Diagrams to model detailed use case descriptions



We are revisiting Use Cases in the light of a much better understanding of the system and will cover more advanced aspects of use case modelling.


SMTU-1 v8.1
Systems Modelling Techniques using UML
Direction Arrows
Include and Extend
transforming performance
through learning

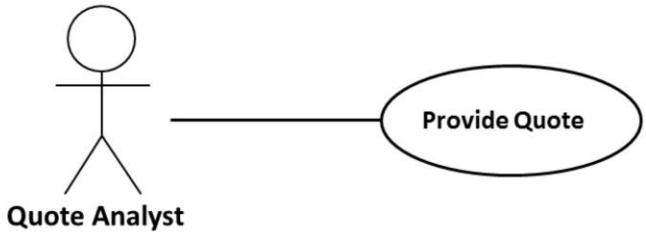
SMTU-1 v8.1

Actor to Use Case relationship

- Previously we explored associations between Actors and Use Cases
- The three main symbols Actor, Use Case and the Association that says:

This says "this Actor is involved with this Use Case"

Typically means a *dialogue* Actor/System

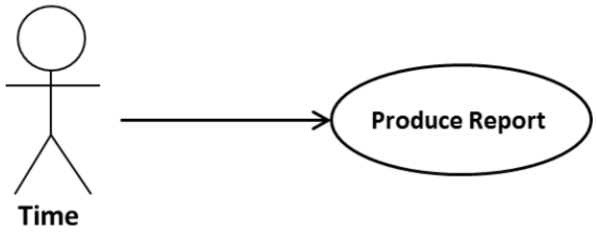


SMTU-1 v8.1

Direction Arrows

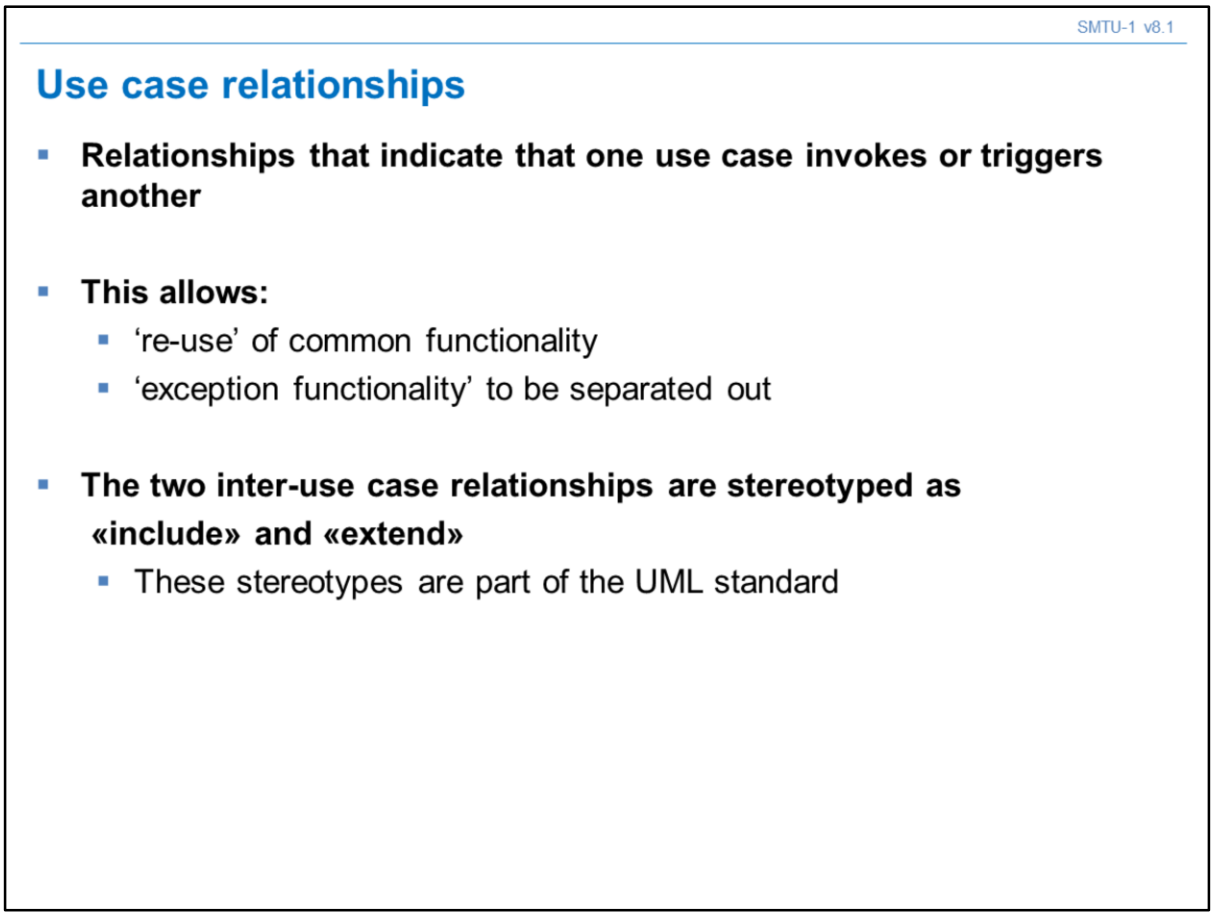
- If the association is *guaranteed* to be uni-directional, an arrow may be used
- May improve the clarity of the diagram
- An obvious example is Time as an Actor

This says "Time triggers this Use Case"



Could also apply to asynchronous communication with an external <<system>>

Adding direction arrows may help to clarify the diagram's meaning. But… leave directionality undefined (no arrows) if there is any doubt.



Relationships between Use Cases are not common, given the definition of what a Use Case is. This is particularly true in the Inception Phase, where the potential for common functionality has not yet been explored.

Relationships between Use Cases indicates that one Use Case invokes or triggers another.

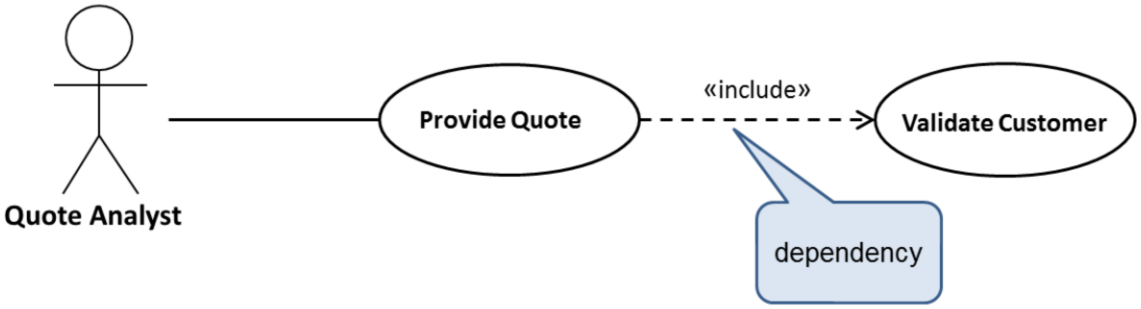
This allows 're-use' of common use case functionality across other use cases. It also allows 'exception functionality' or any other functionality to be separated out from the main functionality in order to reduce complexity.

There are two common inter-use-case relationships: «include» and «extend».

SMTU-1 v8.1

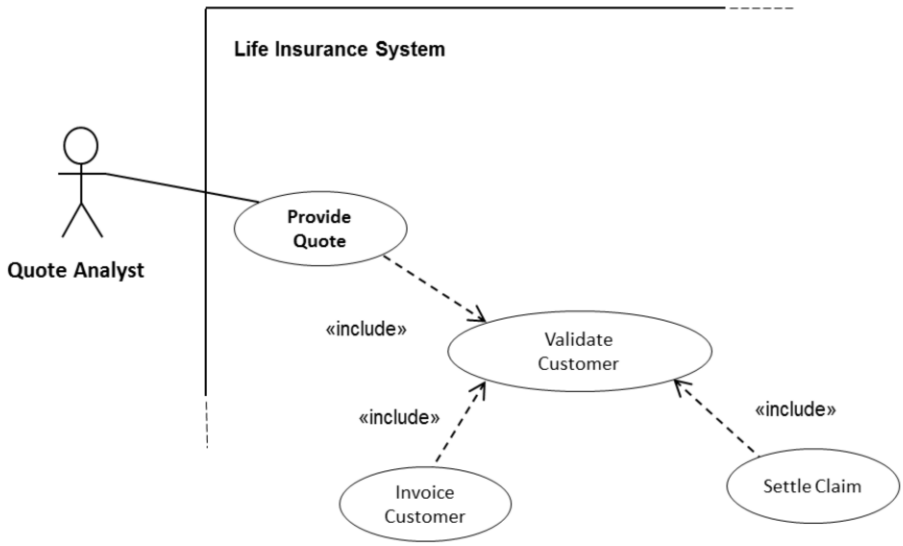
Include example

- **The «include» association indicates that the *Provide Quote* use case includes the common functionality of the *Validate Customer* use case**
  - *Validate Customer* is common to several other use cases such as *Invoice Customer* and *Settle Claim*
- **<<include>> has the sense that this functionality is always (unconditionally) included – indeed *Provide Quote* is incomplete without *Validate Customer***



The implication of separating out the functionality like this is that it is common to several Use Cases.

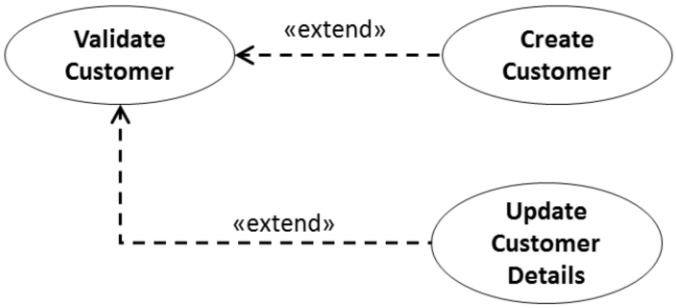
Use case diagram



SMTU-1 v8.1

Extend example

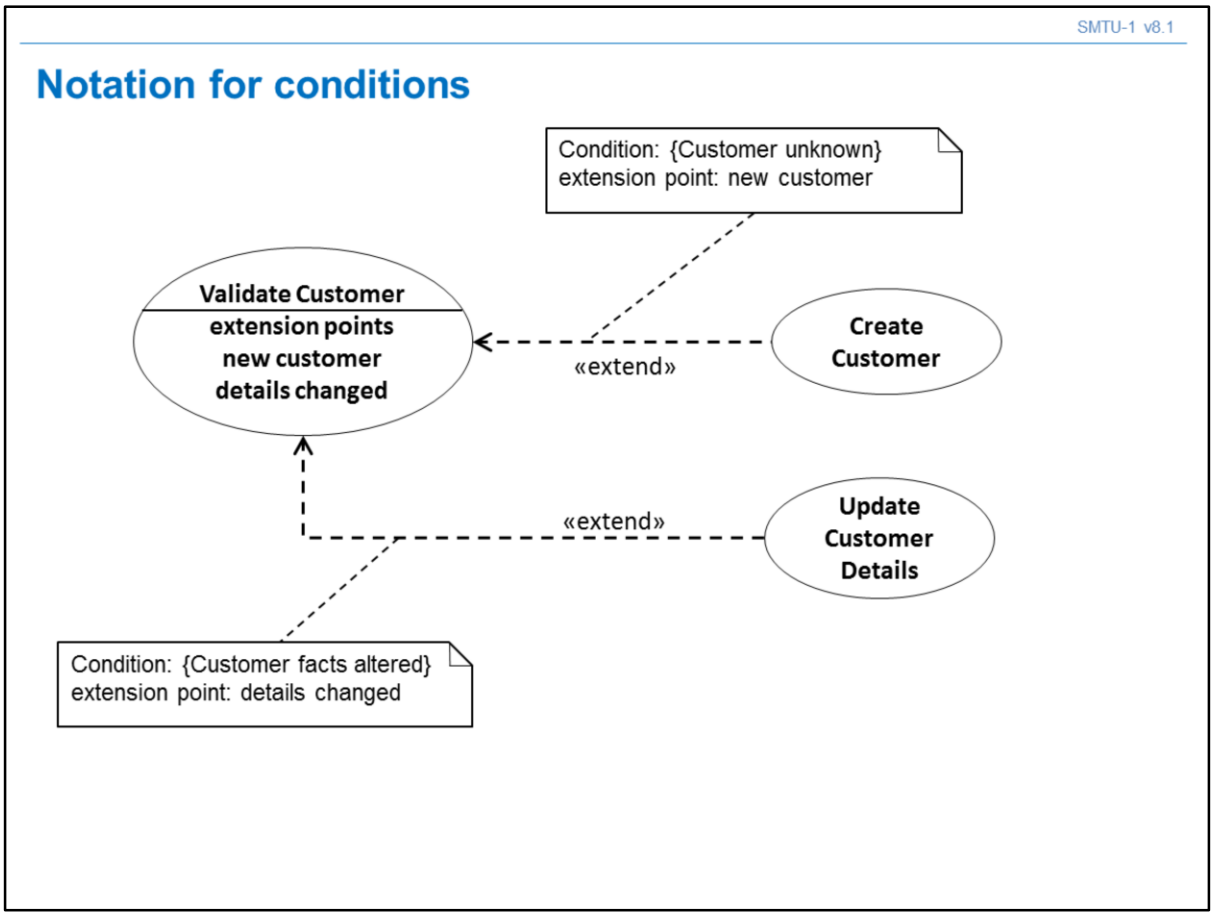
- **The «extend» relationship indicates that a use case has its functionality extended by the functionality of another**
- **Extensions occur under certain conditions:**
  - *Validate Customer* will be extended by the functionality of *Create Customer* if the customer is unknown to the system or extended by *Update Customer Details* if the customer details have changed



The extension takes place at one or more specific extension points defined in the extended use case. The behaviour of the extending use case is considered to be inserted into the extended use case.

The extended use case is defined independently of the extending use case and is meaningful independently of the extending use case. On the other hand, the extending use case typically defines behaviour that may not necessarily be meaningful by itself. Instead, the extending use case defines a set of modular behaviour increments that augment an execution of the extended use case under specific conditions.

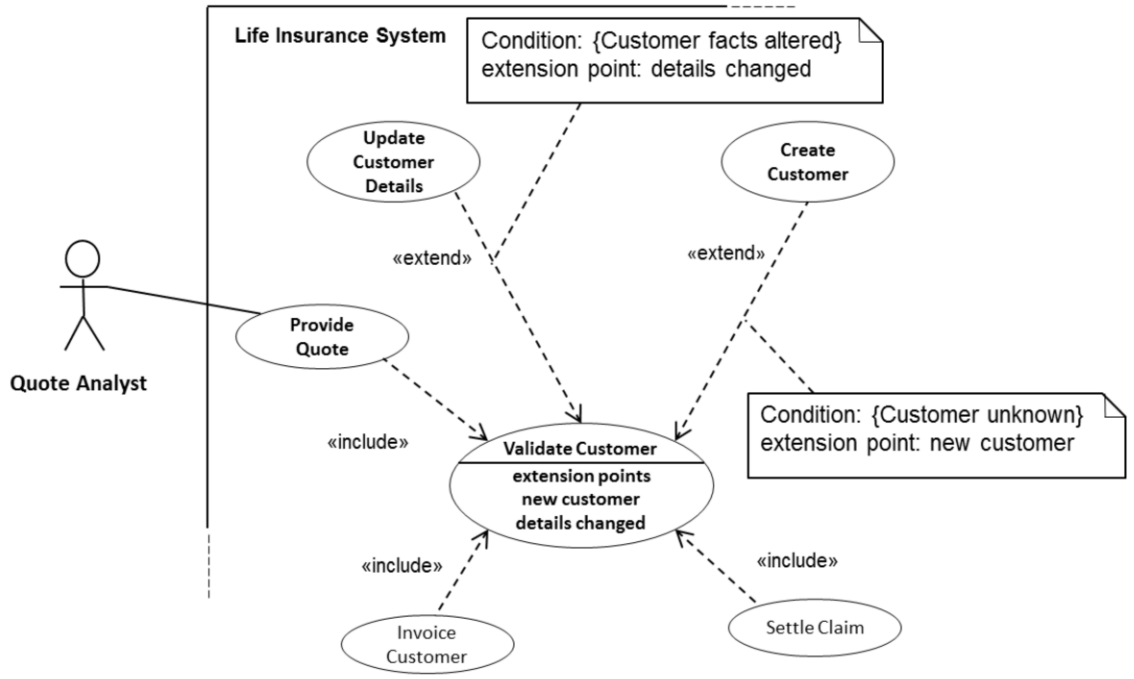
Note that the same extending use case can extend more than one use case. Furthermore, an extending use case may itself be extended. The extend relationship itself is owned by the extending use case.

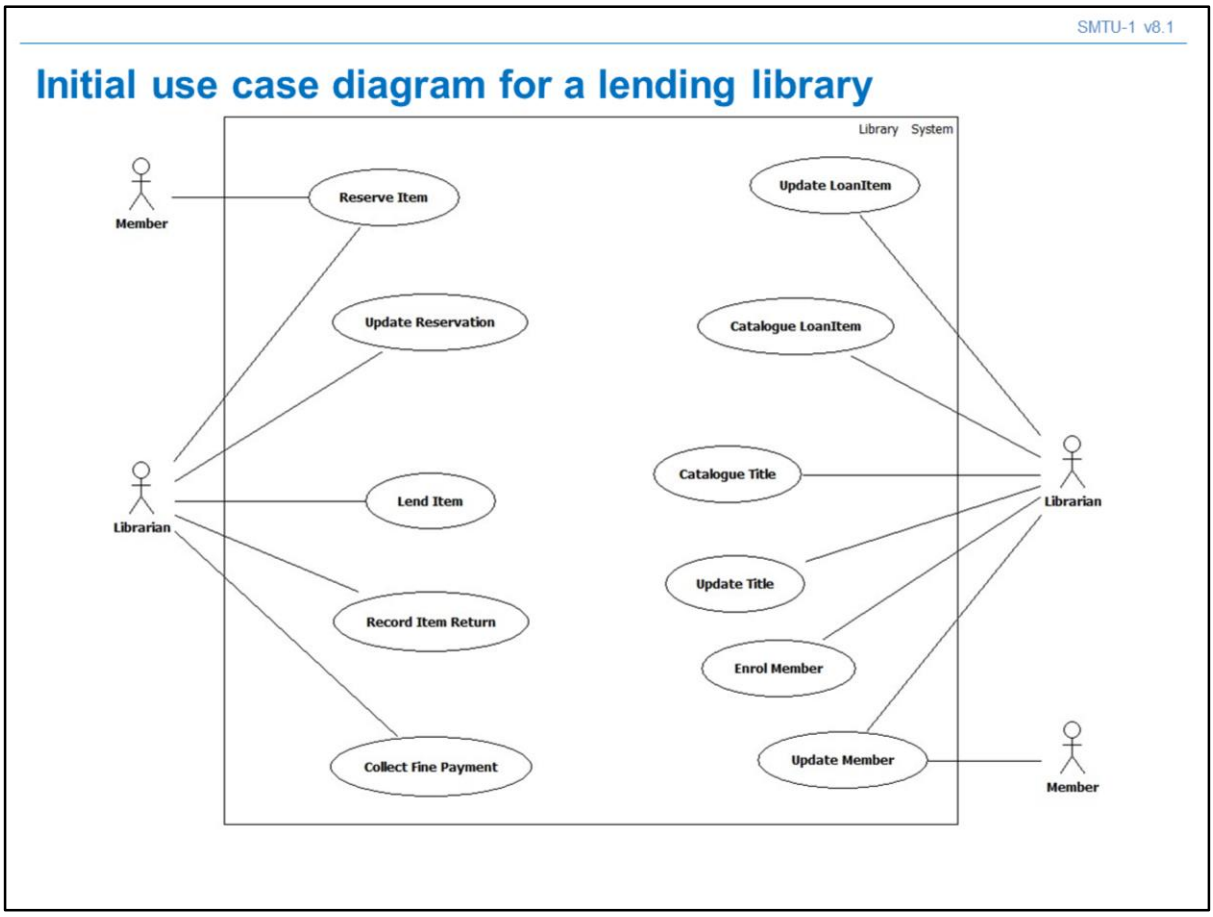


If the condition of the extension is true at the time the first extension point is reached during the execution of the extended use case, then all of the appropriate behaviour fragments of the extending use case will also be executed. If the condition is false, the extension does not occur.

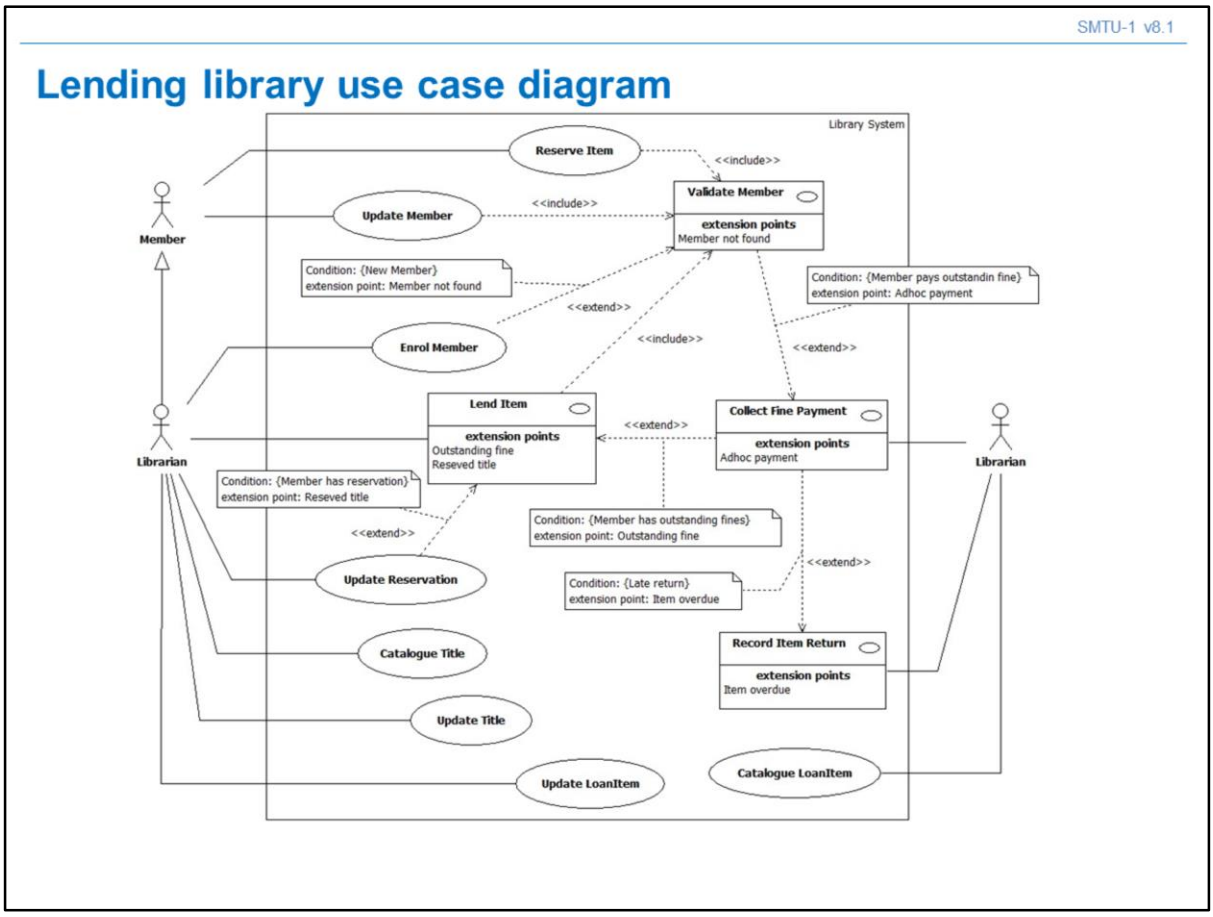
SMTU-1 v8.1

Use case diagram





Consider the impact of includes and extends for Lend Item on the original use case diagram.



As we gain a better understanding of the system, the use case diagram will evolve to reflect this understanding. It is best practice to acknowledge that this will happen and try not to get everything in the very first model. It will inevitably need modifying as stakeholders' understanding of the requirements evolve. Note the generalisation of actors to simplify the model.

SMTU-1 v8.1

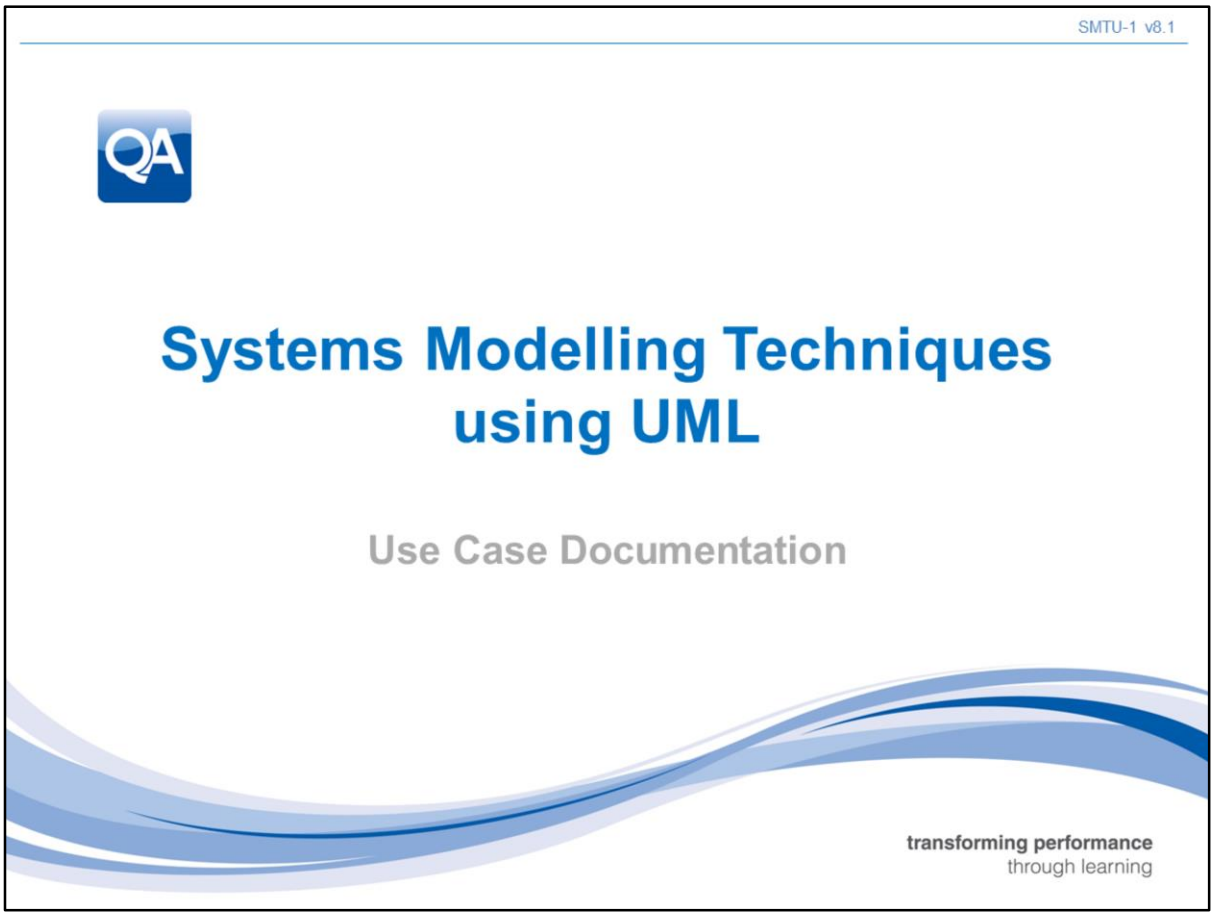
Use with care!

- **Be careful identifying <<include>> and <<extend>>**
- **There are dangers:**
  - Avoid functional decomposition – a Use Case is a single *atomic* business transaction with an IT system
  - Avoid *flow* – a Use Case diagram is not a flow chart! Nor is it a Data Flow Diagram
  - Avoid 'empty' Use Cases that are not about business transactions – for example selecting from a menu or list is *not* a Use Case by itself
  - Avoid designing the application – Use Cases are requirements, not design specs

Although <<include>> and <<extend>> are legitimate parts of the UML spec, use them with care – they are often misused.

Indeed some authorities recommend not to use them at all, especially <<extend>> which has very confusing semantics.

In any case these constructs are more important to Design than to Analysis.


SMTU-1 v8.1
Systems Modelling Techniques using UML
Use Case Documentation
transforming performance
through learning

SMTU-1 v8.1

Complete use case specification

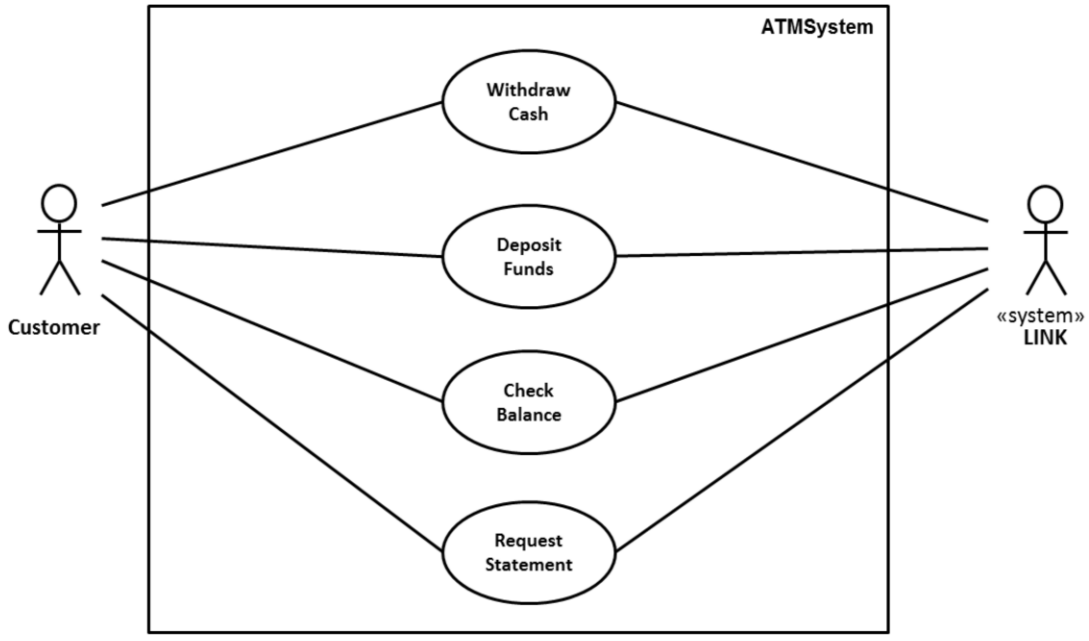
- **A complete use case specification**
  - Main scenario (aka Happy Path, Sunny Day etc.)
  - All the alternate scenarios
- **Alternate scenarios**
  - Legitimate alternate paths, which still achieve the goal of the UC
  - May have own post-conditions
- **Exception scenarios**
  - Exception that occur, which frustrate the users efforts to reach their goal
- **For each use case Main scenario**
  - Brainstorm all the potential alternate and exception scenarios with stakeholders
  - Some scenarios may cause the use case to end prematurely
- **System must always be left in a stable state at the end of the UC**

A complete use case specification consists of more detail than the main flow specification – it specifies not only the main scenario but all the alternate scenarios. Alternate scenarios are legitimate paths through the use case and they may have their own post-conditions. Once the interaction between the system models is better understood the complete use case specification can be developed.

Take each use case main scenario and brainstorm, with domain experts and users, to identify all the alternate scenarios with which the system will have to cope. Some alternate scenarios may cause the use case to end prematurely and the goal is not achieved. These are called Exception Flows. If this happens the system must be left in a stable state.

SMTU-1 v8.1

Reminder - Use case diagram for an ATM



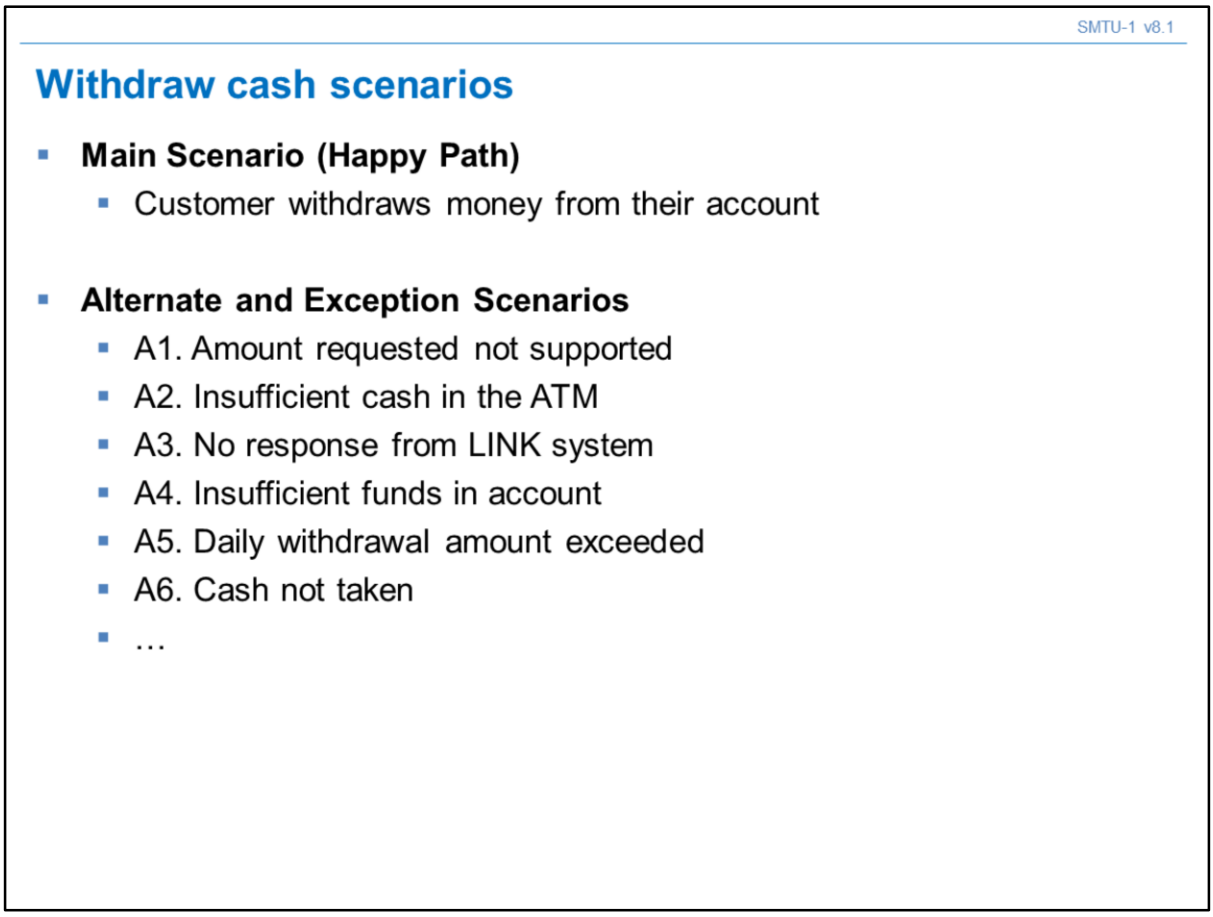
SMTU-1 v8.1

Initial main flow description for 'Withdraw Cash'

	Main Flow	
1.	*Customer* indicates that they wish to withdraw cash.	
2.	*System* prompts for an amount.	
3.	*Customer* specifies an amount.	
4.	*System* sends transaction details to the LINK system.	
5.	*System* receives confirmation of sufficient funds.	
6.	*System* dispenses amount.	
7.	*System* prompts "Take notes."	
8.	*Customer* takes notes.	
9.	*System* produces receipt.	
10.	*Customer* retrieves receipt.	
11.	Use case ends.	

Best practice (Cockburn) suggests that a UC should be documented as a dialogue between the Primary Actor and the IT system. In other words a UC description describes a series of Actor/System 'round trips'.

This format is easy for business users to understand, especially when accompanied by Storyboards, prototypes etc.



The above list is representative of alternate scenarios. Once alternate scenarios have been revealed consider the goal/focus of the use case. Other use cases that support the alternate scenarios may already exist. You may discover that common functionality is required to handle the secondary scenarios «include», or new use case need to be created for complicated secondary scenarios «extend».

SMTU-1 v8.1

Complete main flow description for 'Withdraw Cash'

	Main Flow	
1.	*Customer* indicates that they wish to withdraw cash.	
2.	*System* prompts for an amount.	
3.	*Customer* specifies an amount.	A1, A2
4.	*System* sends transaction details to the LINK system.	A3
5.	*System* receives confirmation of sufficient funds.	A4, A5
6.	*System* dispenses amount.	
7.	*System* prompts "Take cash"	
8.	*Customer* takes cash	A6
9.	*System* produces receipt.	
10.	*Customer* retrieves receipt.	
11.	Use Case Ends	

Alternate and exceptions can be referenced in another column, then elaborated elsewhere.

SMTU-1 v8.1

Alternate flows for 'Withdraw Cash'

	Alternate Flows	
A1	Amount not supported	
1.	System prompts that the amount must be a multiple of the bills on hand.	
2.	Use case resumes @ "System prompts for an amount."	
A2	Insufficient cash in ATM	
1.	System prompts "The amount requested exceeds the amount available. Please re-enter amount."	
2.	Use case resumes @ "System prompts for an amount."	
A3	No Response from LINK system	
1.	Use case ends.	

SMTU-1 v8.1

Alternate flows for 'Withdraw Cash'

	Alternate Flows	
A4	Insufficient funds in account	
1.	System prompts "The amount requested exceeds the funds available. Please re-enter amount."	
2.	Use case resumes @ "System prompts for an amount."	
A5	Daily withdrawal amount exceed	
1.	System prompts "The amount requested exceeds the daily amount. Please re-enter amount."	
2.	Use case resumes @ "System prompts for an amount."	
A6	Cash not taken after time out	
1.	System prompts "Please remove cash."	
2.	Customer takes cash	
3.	Use case resumes @ "System produces receipt."	

SMTU-1 v8.1

Use case specification template

- **Use case descriptions are not part of the UML standard as defined by the Object Management Group**
- **Best practice(*) indicates that a use case specification should include headings like the following:**
  - Unique Reference ID (might also state the *system* that the UC is part of)
  - Name (Verb-noun)
  - Priority (MoSCoW)
  - Brief Description (Active-verb goal phrase that names the goal of the primary actor; *user story* format could be used here)
  - Trigger (user does *what* to initiate the UC)
  - Primary and Secondary Actors
  - Pre-conditions
  - Main Flow
  - Alternative flows
  - Post-conditions
  - Extension points
  - Open issues

(*) for example Alistair Cockburn's book 'Writing Effective Use Cases'. The most important point though is that every organisation adopts its own standard format.

SMTU-1 v8.1

Pre-conditions and post-conditions

- **Pre-Conditions**
  - What the system will ensure is true before letting the use case start
  - For example:
    - The Librarian is logged on
    - The loan item status is “Available”
- **Post-Conditions**
  - Declarations about the state of the system on completion of the use case
  - Details how the system changed in respect of what data changes were made
  - Note that post-conditions describe WHAT was changed but not how the change was achieved
  - For example:
    - A loan record is created for each item borrowed
    - The loan record is linked to the member borrowing the item

The pre-conditions of the use case announces what the system will ensure is true before letting the use case start. Since it is enforced by the system and known to be true, it will not be checked again during the use case execution. A common example is, **“The user is already logged on and has been validated.”** (Cockburn).

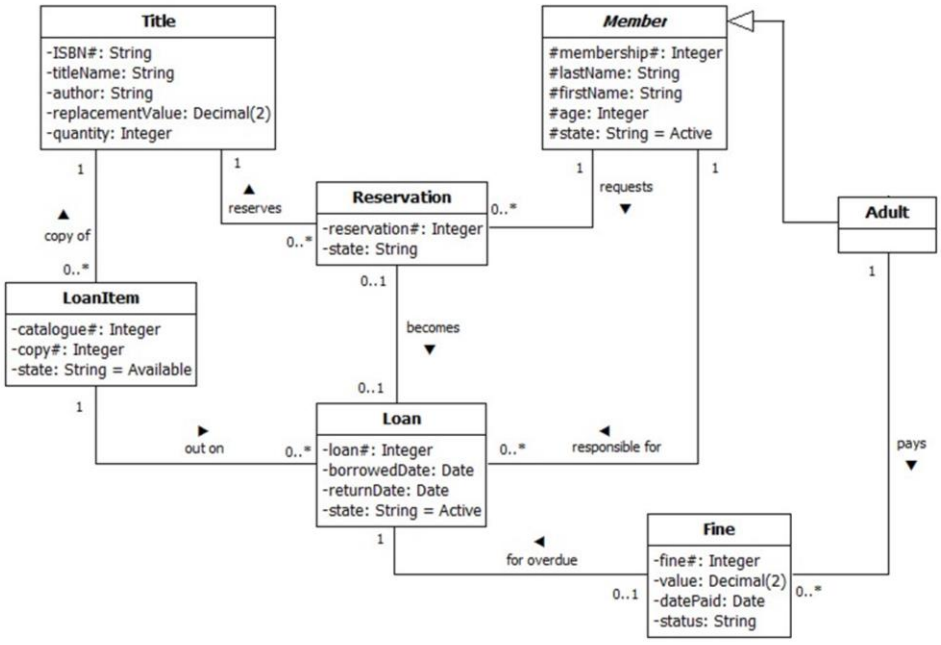
Note that alternate flows and exceptions can have their own post-conditions.

SMTU-1 v8.1

View of participating classes (VOPC)

- **A diagram of all the classes whose instances work together to implement the use case**
  - Main and alternate flows
- **This documentation is a subset of the Domain Class Model**
- **This information will be useful when we come to specify the collaboration of objects across the software architecture; in particular the intervention of data-related objects**

Lend Item Use Case - VOPC



Our understanding from the analysis of the UC is that objects from these classes will be involved in realising the UC.

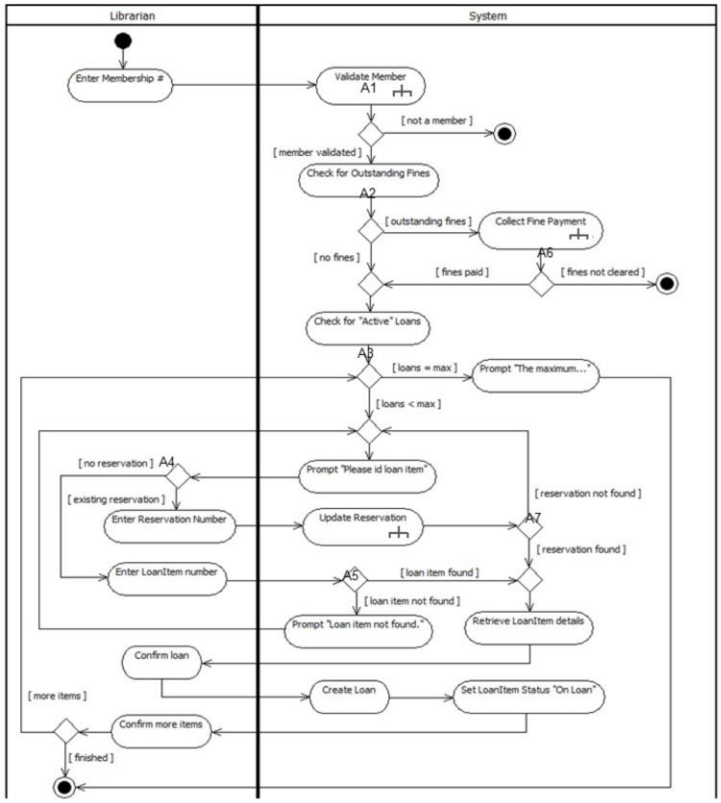
SMTU-1 v8.1

Use Case descriptions supported by Activity Diagrams

- **Use Case descriptions describe the dialogue between the actor and system**
- **Activity Diagrams can represent this dialogue**
  - Especially useful for complex cases
  - Can include secondary actors as partitions
- **Diagrams are less ambiguous than text so the production of the description as an Activity Diagram can help verify the use case**
- **Once the Activity Diagram is produced testers can calculate the number of paths required to provide full test coverage**
- **Object Flows, if present, will help establish the VOPC**
  - Validates the Domain Class model

While many people find the textual descriptions of use case flows perfectly acceptable, there are those who prefer to see things graphically. This is particularly true for testers who can calculate the number of test paths from the decisions on the activity diagram and thus how to set about testing the use case!

The number of paths can be calculated from the number of decision points plus one. (Providing the decisions are binary.)



Note the correlation between the alternate flows and decisions.

SMTU-1 v8.1

Case study

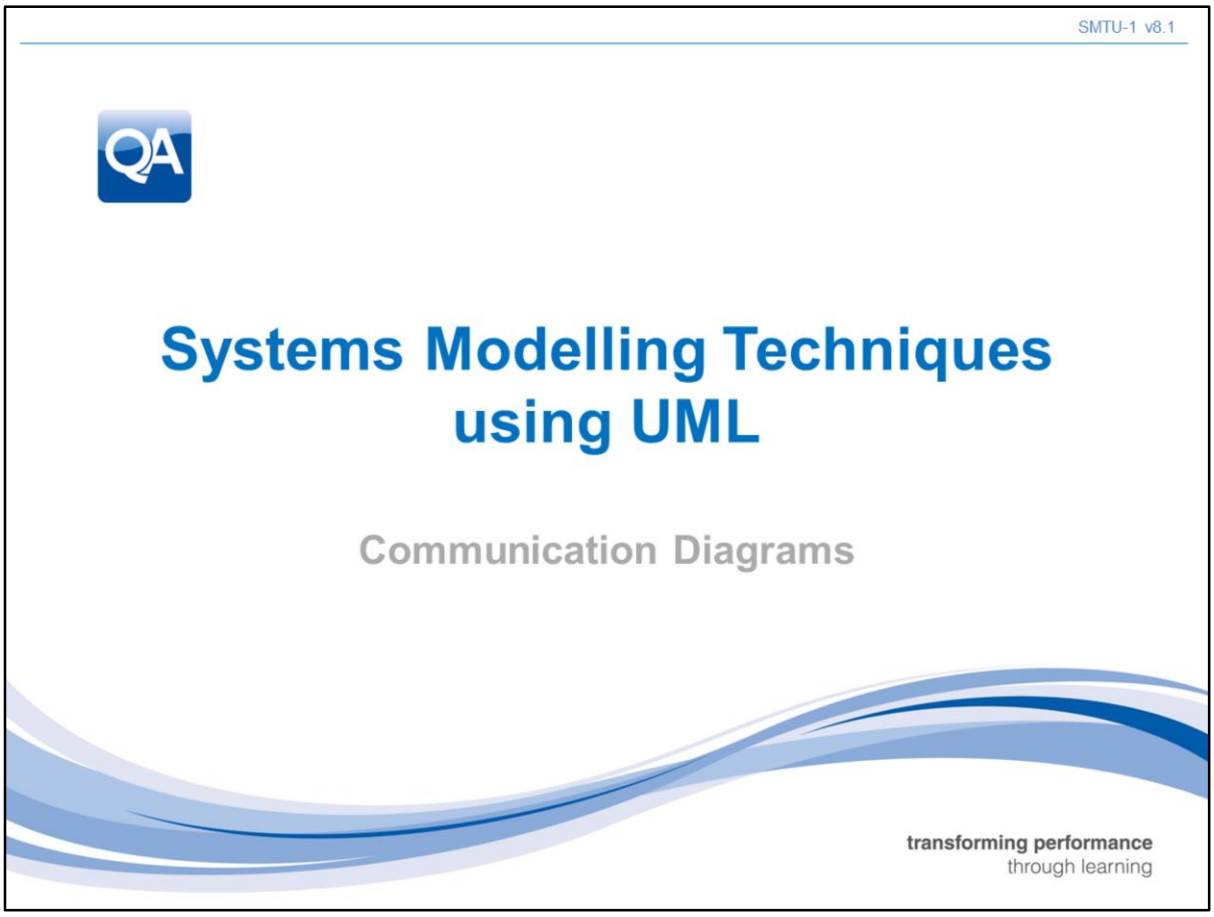
a. Create the ‘main flow’ of a use case description for “Record Product Delivery” (TnT), using the template suggested here.

b. Expand upon the main flow to include the secondary scenarios.

SMTU-1 v8.1

Summary

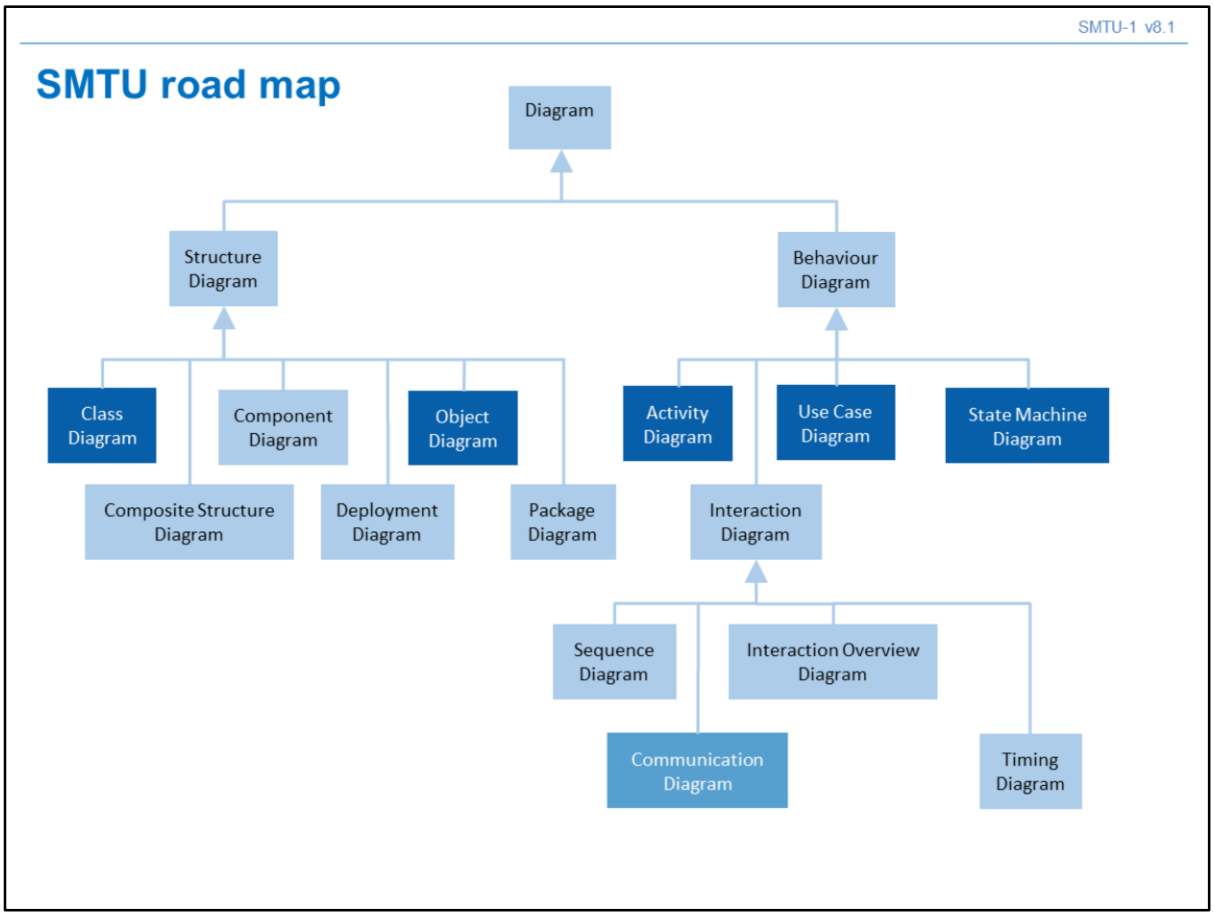
- **UC relationships**
  - «include» common UC functionality
  - «extend» for major alternate paths that complement the UC
- **Full template documentation of a UC – main and alternate flows**
- **Pre and post conditions to include states of objects in business terms**
- **Use VOPC documentation to highlight the necessary data items included in the system use case description**
- **Use activity diagrams to support the UC description**


SMTU-1 v8.1
Systems Modelling Techniques using UML
Communication Diagrams
transforming performance
through learning

SMTU-1 v8.1

Topics

- Use case realisations
- MVC pattern
- UML messages
- Communication diagrams



Communication diagrams are a good starting point for the introduction of use case realisations. They build on the VOPC pattern and object diagrams.

SMTU-1 v8.1

Use case realisation

- **A sequence of *events* and *object interactions* that form part of a use case (i.e. one scenario)**
- **A temporal specification of actions carried out by the system objects to meet the use case description**
- **May be several for each use case, one at least for the main (or basic) flow and others if alternate or exception flows are complex**
- **The purpose of a UC realisation in analysis is to 'prove' that the UC 'works' as far as we have analysed it, e.g.:**
  - Logic makes sense
  - Domain model supports it
- **The UC realisation models will be helpful when we come to do the detailed design**

The distinction between a use case and a use case realisation is as follows. A use case is a description of the external sequence of events between an actor and the system needed to meet a requirement; a use case realisation maps out the internal effects of this dialogue as a sequence of events and interactions between relevant objects .

Each use case realisation gives us a temporal specification of events and actions carried out, and for each use case there will be several realisations. The impact of these events and actions on relevant objects can be modelled, and these are modelled in UML using *Interaction Diagrams.*

SMTU-1 v8.1

MVC pattern

- **The format for UC realisation is not specified in UML but in general some form of crude Interaction Diagram is used:**
  - Communication or Sequence Diagram
  - Used in a fairly informal way
  - Sometimes referred to as 'robustness diagrams'
- **A typical way to structure this diagram follows the *MVC pattern:***
  - *Model, View, Controller*
- ***View*: This layer groups objects that interact with the actors using the IT system.**
- ***Control:* This layer groups objects that execute the flow control within the application.**
- ***Model:* this layer contains objects representing the business knowledge accumulated and stored in the system**

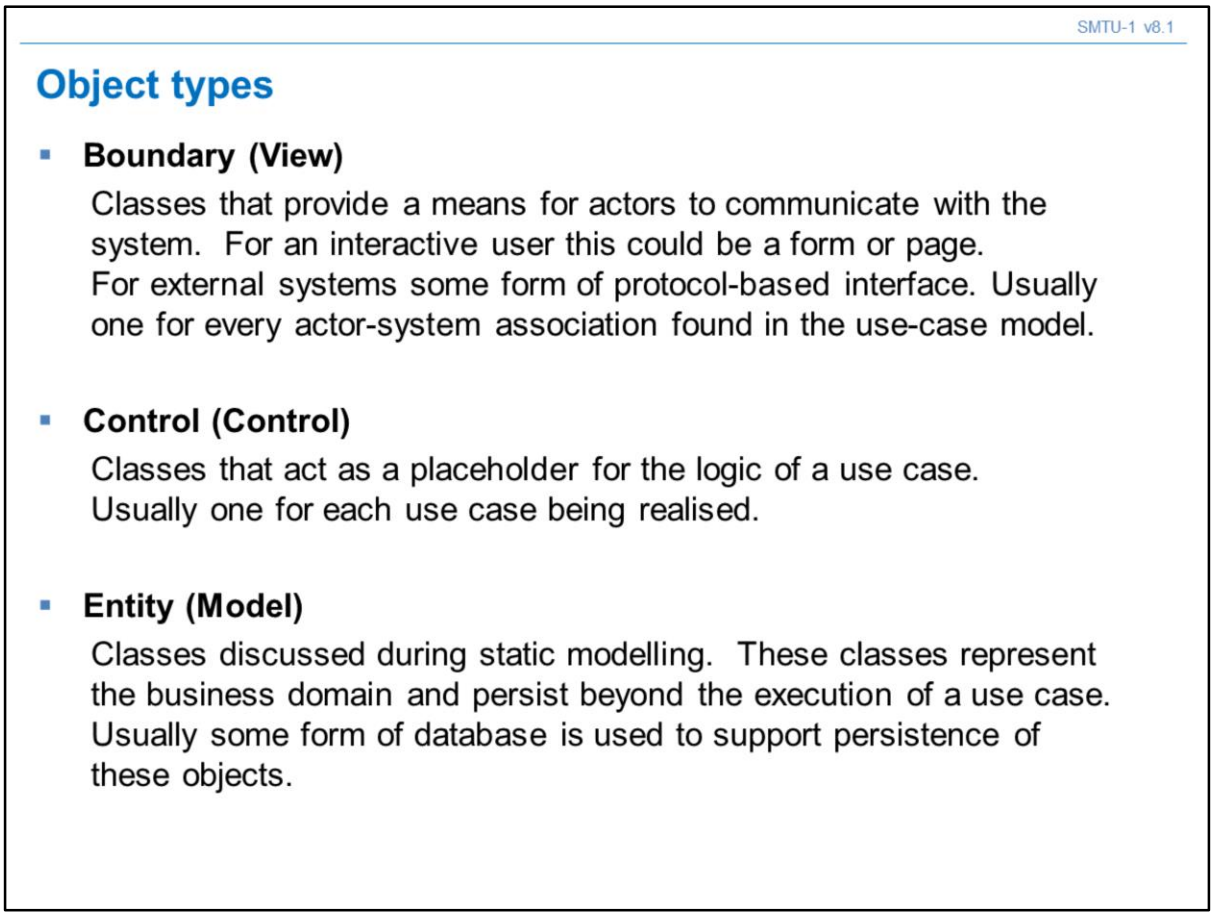
The MVC pattern is widely used as the basis for structuring the internal architecture of modern applications. It concerns the distribution of objects in *layers* within the system according to their role in realising the requirements.

View: This layer groups objects that interact with the actors using the IT system.

Control: This layer groups objects that execute the flow control within the application.

Model: this layer contains objects representing the business knowledge accumulated and stored in the system.

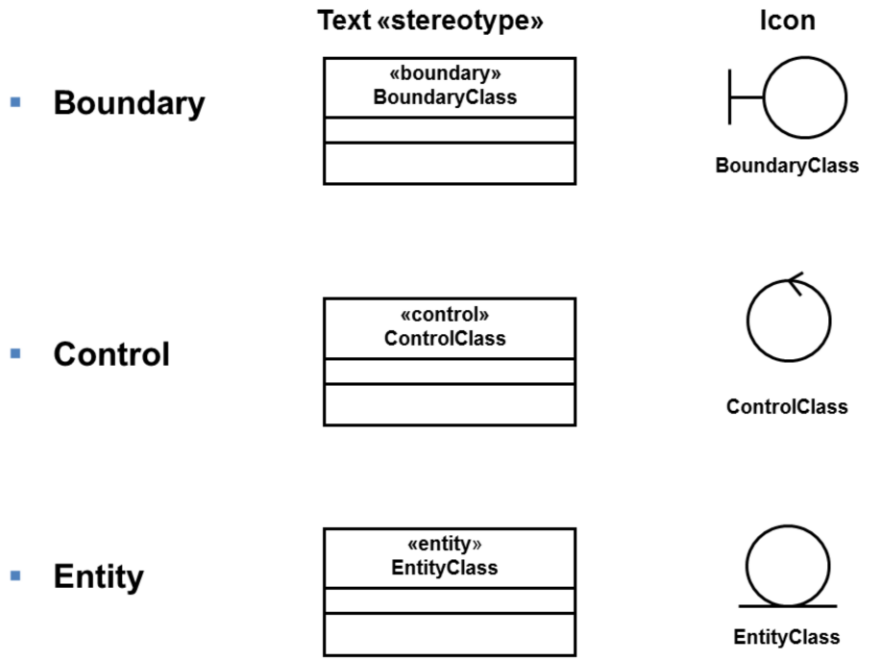
The pattern requires that layers only communicate with each other through defined interfaces. Furthermore *Boundary* and *Entity* can only talk to each other via *Control*. The basic benefit of the pattern is to reduce the number of direct dependencies between objects in the system design, so that change can be done easily and cheaply in the future.



The MVC pattern is recognised in UML via these standard stereotypes. Objects can be stereotyped as control, boundary, and entity. These stereotypes are used to increase the semantic meaning of the classes and their usage in modelling situations.

- The *boundary* stereotype specialises the class for presentation and limited data manipulation, like local validations. Boundary classes present and communicate the information in a system to another system, such as a human or a machine (that is, actors). Boundary classes are typically aspects of the UI: windows, buttons, dialog boxes, etc.
- The *control* stereotype is used for classes that deal with the flow of control within the UC. The objects from these classes generally respond to events detected at the boundary, they handle the logical flow of the UC and may send messages to entity objects to carry out the required data manipulation. These classes are responsible for formatting and sending information back to the boundary.
- The *entity* stereotype is used to model the business concepts that were documented in the Domain Class model. They typically represent persistent data that is stored in the system. Included here too, for convenience, may be classes that ‘know’ the business rules and provide interfaces to business services like email.

Object types representations



SMTU-1 v8.1

Communication diagrams in UC realisation

- **Communication diagrams show interactions between participant objects involved in a use case.**
  - Participant objects are parts of the system that interact to realise the use case
- **Communication diagrams show the communicating links between participants and the messages passed along those communication links**
  - Objects cannot communicate unless an association exists between the classes
  - Communication diagrams therefore help validate the Class diagram
- **Sequence is inferred from the numbering scheme employed on the diagram**

Communication diagrams show interactions between participants. They show the participants along with the messages that travel between them.

Some may ask why we need both communication and sequence diagrams since they are semantically equivalent, and one can easily be turned into the other. The answer is that while a sequence diagram emphasises the time ordering of interactions, a communication diagram emphasises the context and overall organisation of the participants that are interacting.

SMTU-1 v8.1

Example - Use case for Retrieve Reservation

- **Description**

  Librarian wishes to check on the status of a reservation

- **Actor**

  Librarian

- **Pre-condition**

  Librarian logged onto system

  Reservation exists on system

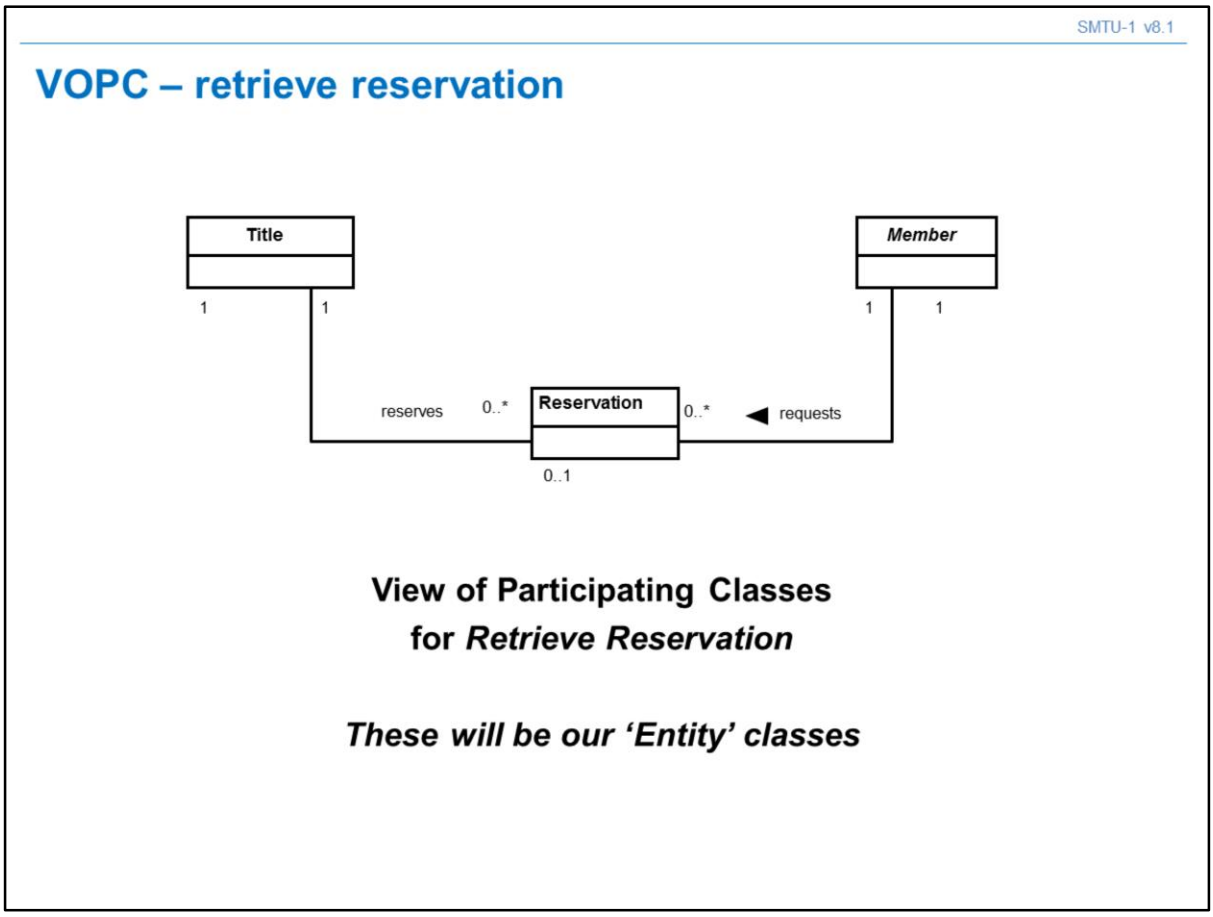
- **Main Flow**

  1. *Librarian* enters reservation number into system [A1]
  2. *System* retrieves reservation details

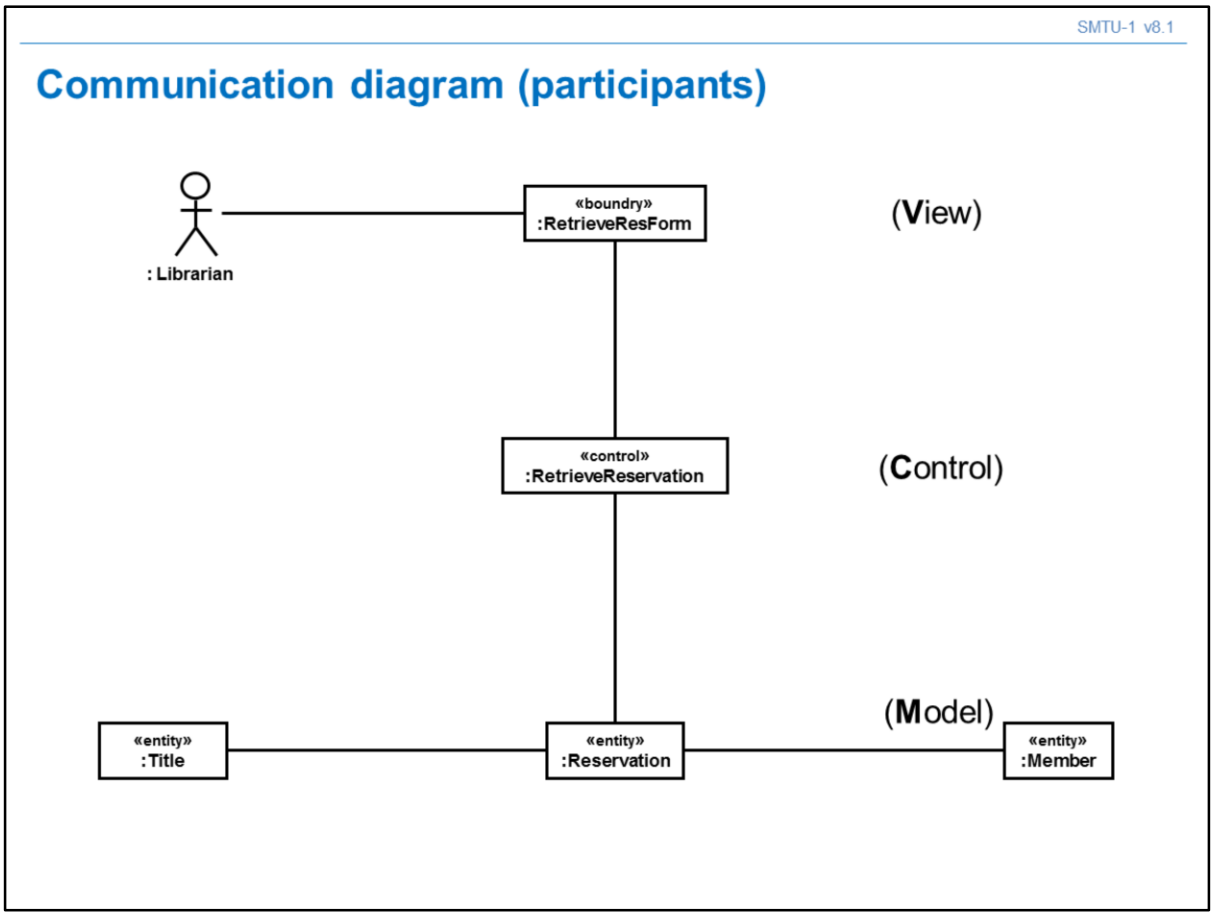
- **Alternates**

  A1. Reservation number not known

  1. Librarian enters membership number into system
  2. System retrieves all “Unfulfilled” reservations for member



Sub-set of the library class diagram that will support an enquiry on a reservation.

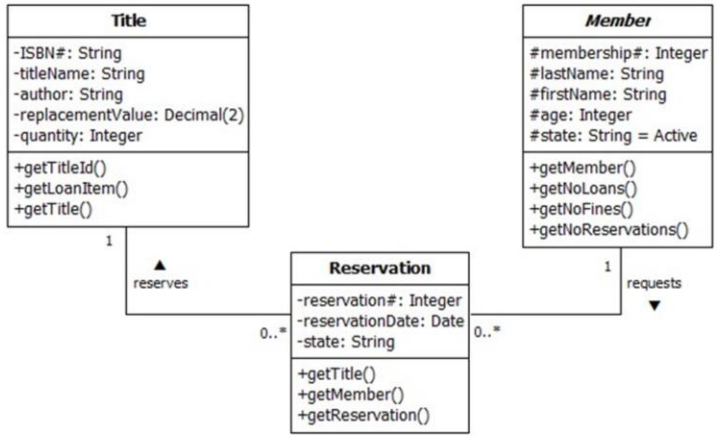


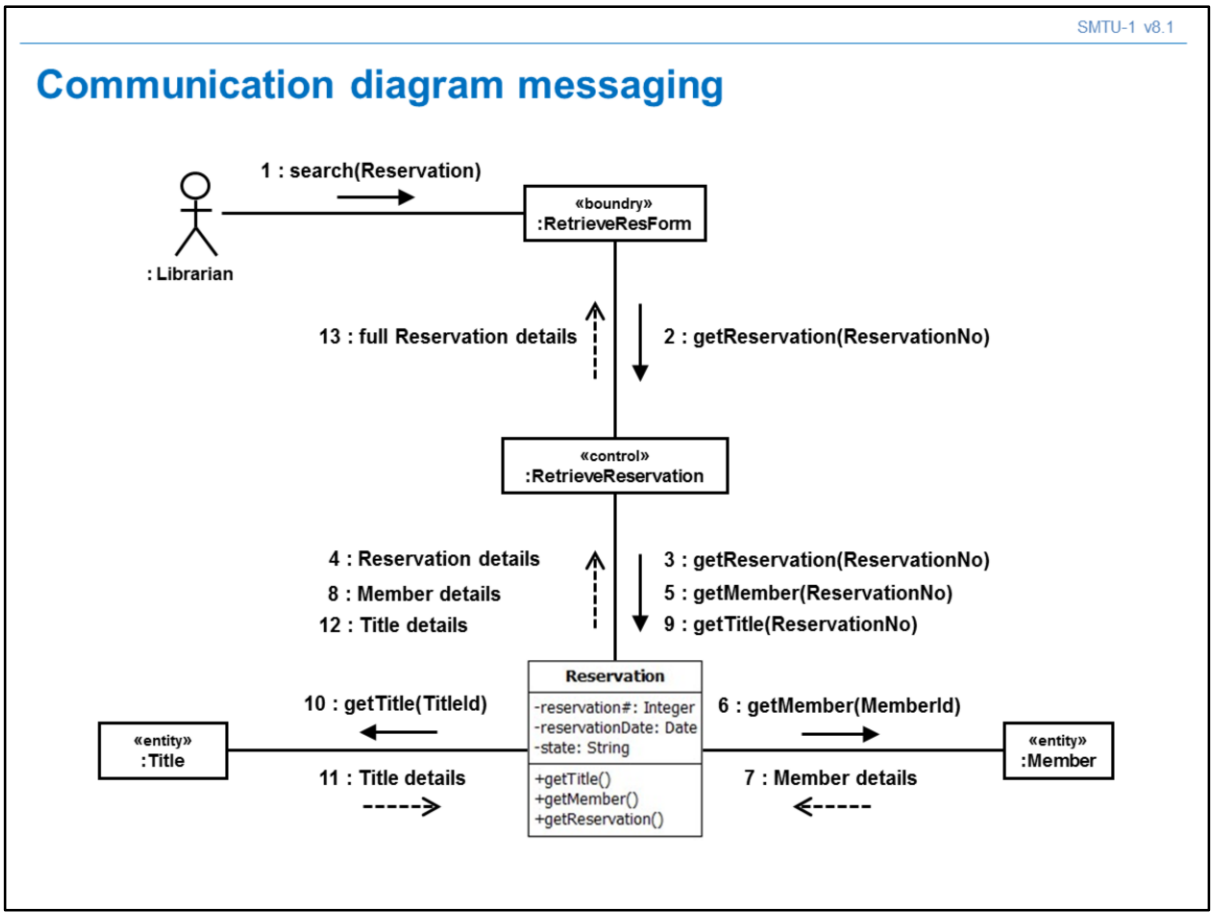
Entity objects, now enhanced with control and boundary participants, together with the actor.

SMTU-1 v8.1

Messages in Interaction Diagrams

- **For 1 object to invoke the behaviour of another it is said that it sends the target object a *Message***
  - This is done by invoking the relevant *operation* on the target object
- **The message must have the same *signature* as the operation**
  - Name, parameters and return type

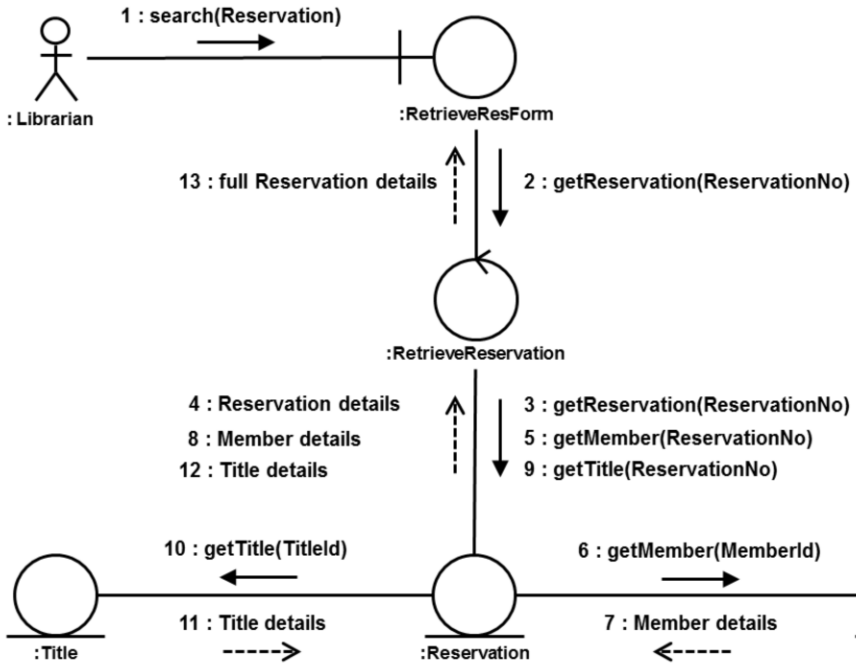




Remember that the operations called by the messages relate to the operations defined in the class.

By mapping out the expected flow of messages, we are confirming the operations required on each participating class, and we may get insight into the attributes of these classes too. The operations subsequently have to be designed and coded, as we move towards producing the solution.

Communication diagram using the standard icons

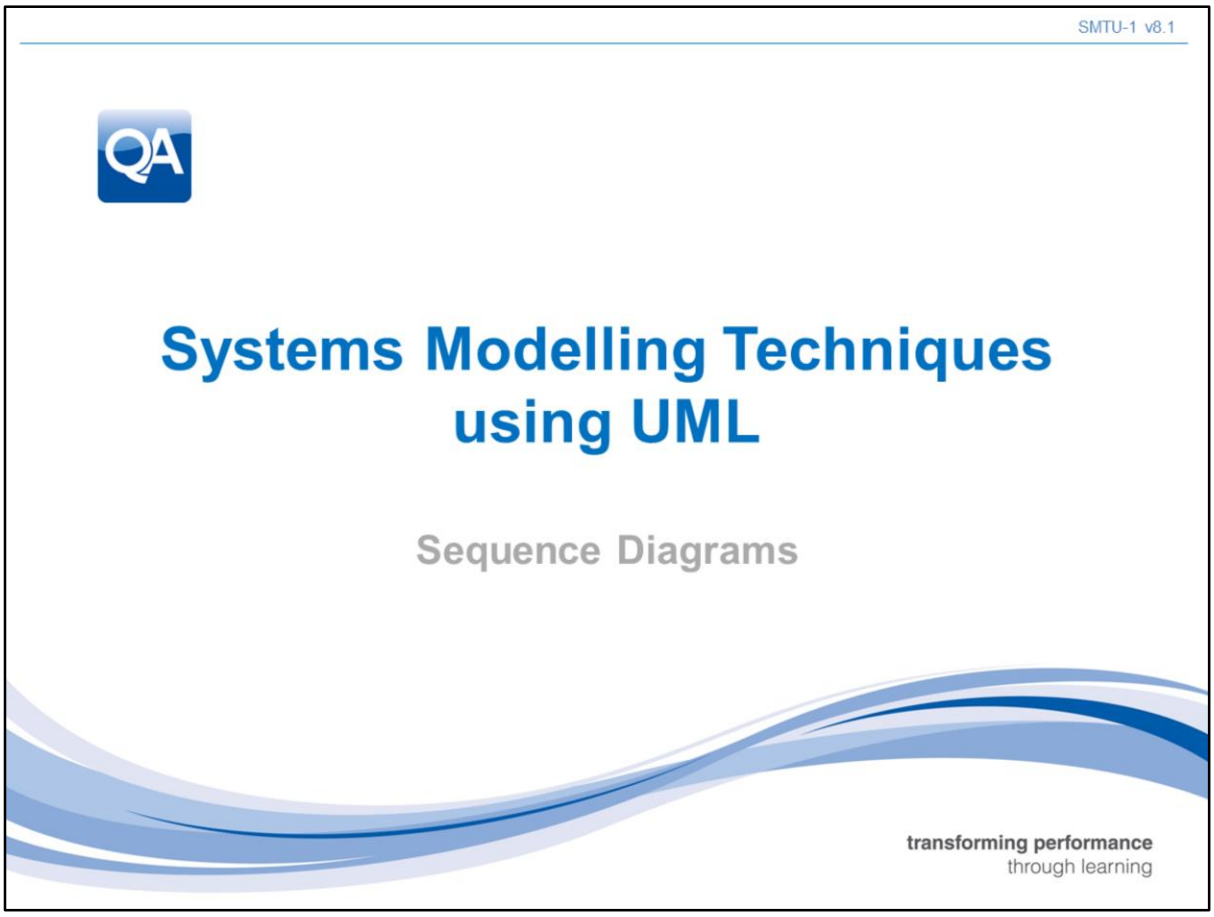


This is a ‘use case realisation’ diagram. Often they are produced quite informally, just checking that everything seems to ‘hang together’ (robustness check).

SMTU-1 v8.1

Summary

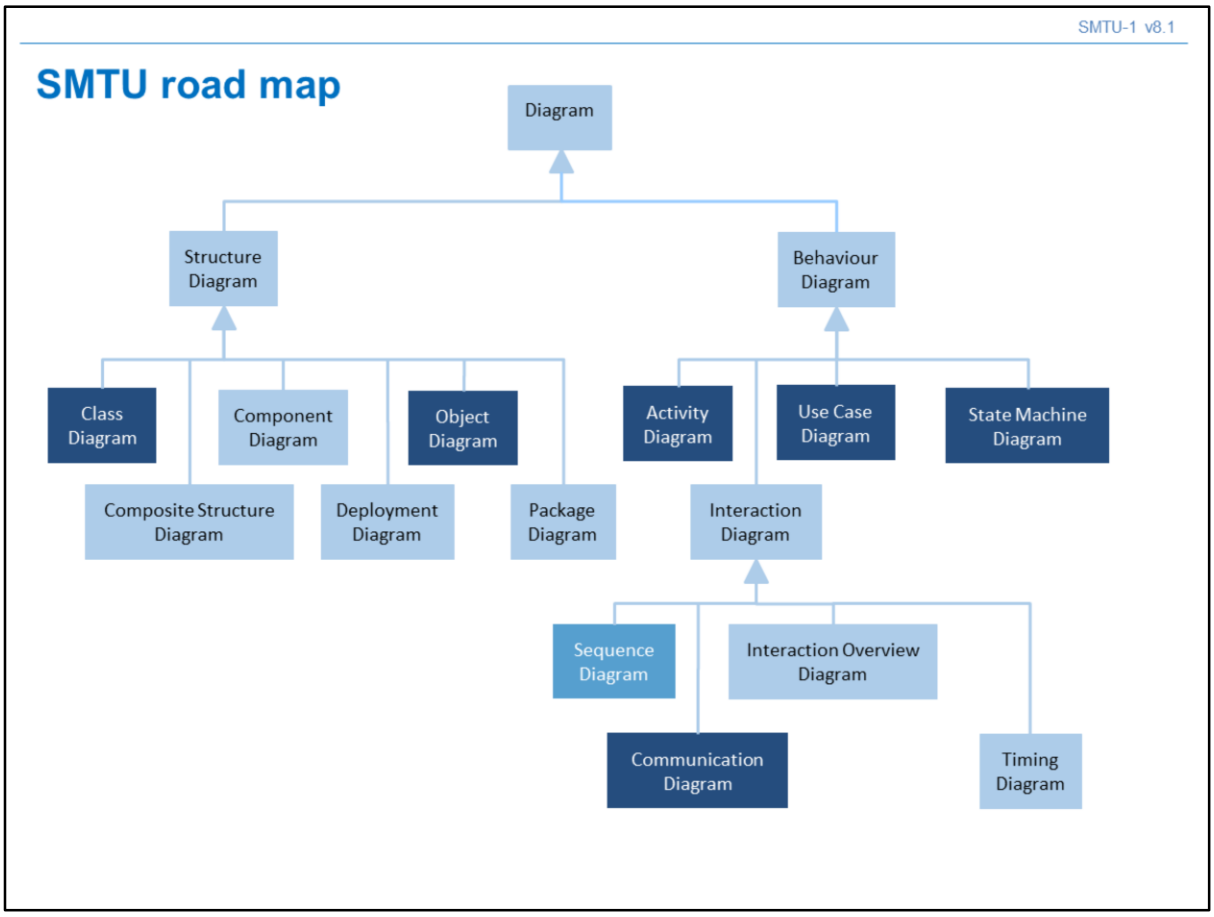
- Use case realisations
- MVC pattern
- UML messages
- Communication diagrams


SMTU-1 v8.1
Systems Modelling Techniques using UML
Sequence Diagrams
transforming performance
through learning

SMTU-1 v8.1

Topics

- **Sequence diagrams**
- **Message types**
- **Combined Fragments**
  - ALT
  - LOOP
  - OPT
  - PAR
- **Organising the UC Realisation**

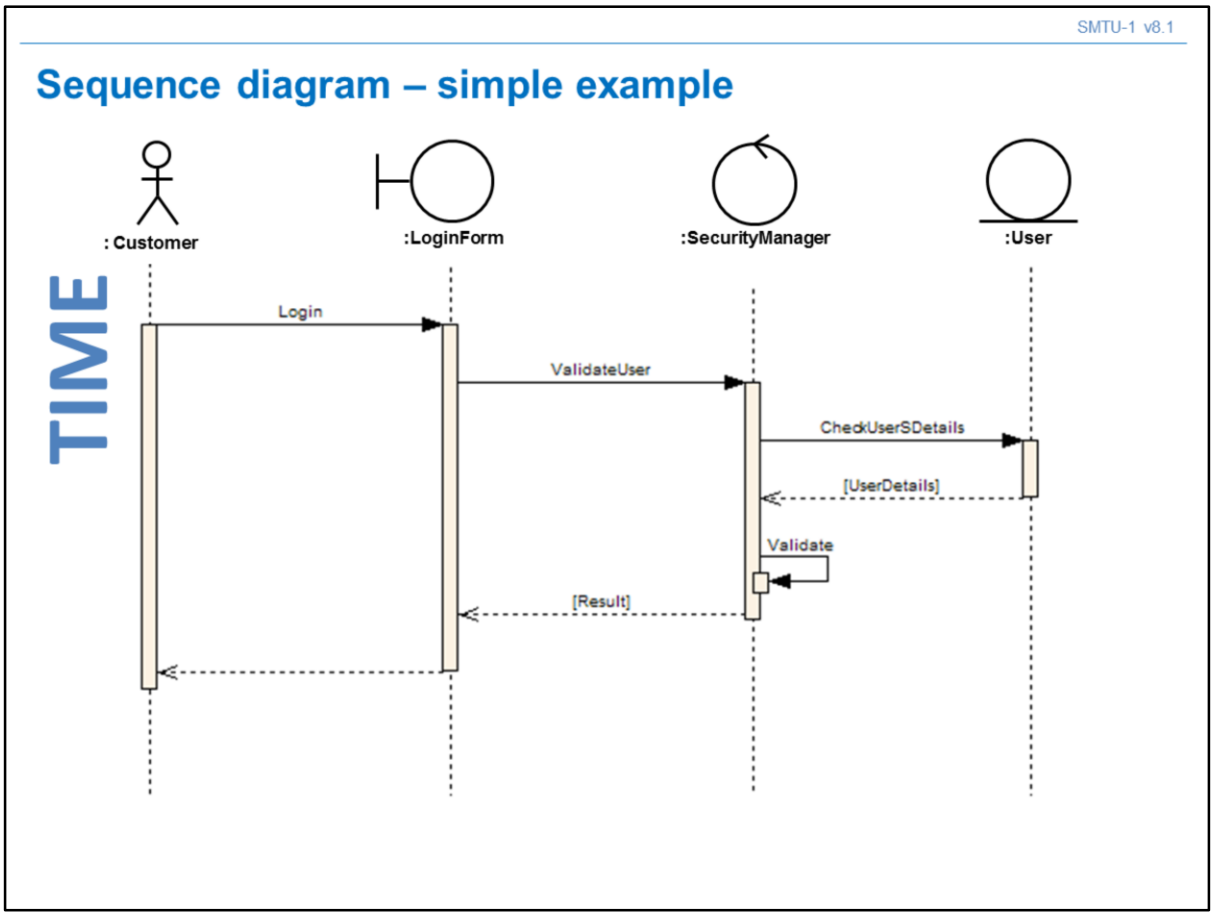


Sequence diagrams show participant objects interactions arranged in time sequence. They depict the sequence of messages exchanged to carry out the functionality of the scenario. Sequence diagrams typically are associated with use case realisations in the Logical View of the system under development. As such they are significant tool for designing the application.

SMTU-1 v8.1

Sequence diagrams

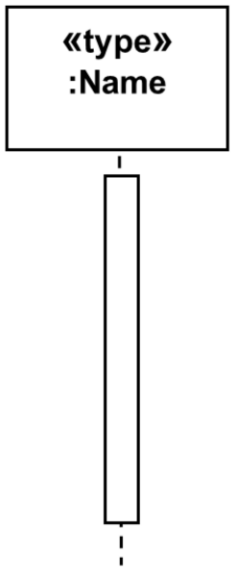
- **Sequence and communication diagrams do exactly the same job – most tools can switch between them automatically**
- **Hence, a sequence diagram may be used as a graphical representation of a use case realisation, instead of a communication diagram**
  - Analysts tend to favour one or the other
- **An advantage of a sequence diagram over a communication diagram is that it's vertical axis shows time and so the numbering required for communication diagrams can be removed**
- **Communication diagrams are perhaps more widely used in Analysis, while Sequence diagrams are more useful in Design**
  - However as models they cover the same view

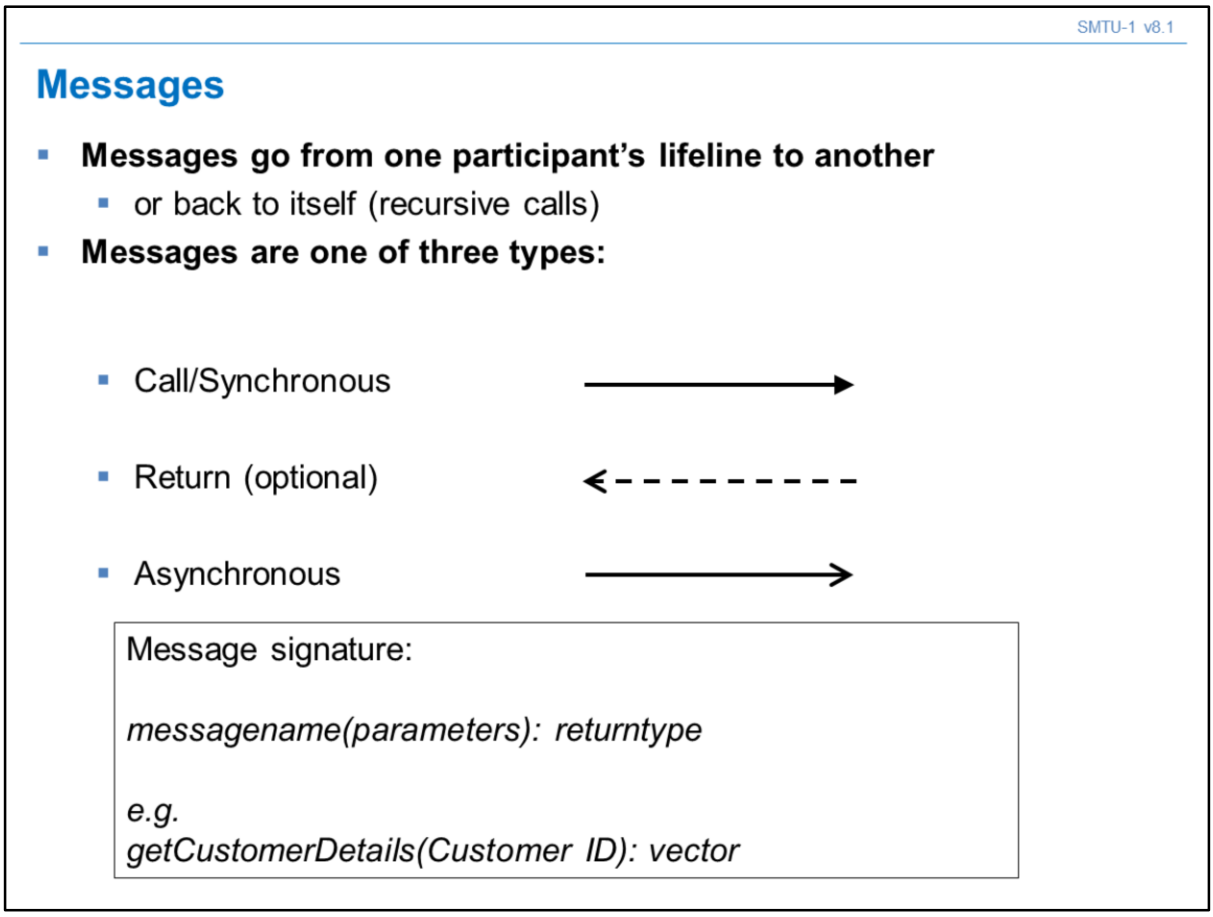

SMTU-1 v8.1
Sequence diagram – simple example
:Customer
:LoginForm
:SecurityManager
:User
TIME
Login
ValidateUser
CheckUserSDetails
[UserDetails]
Validate
[Result]

SMTU-1 v8.1

Participants and lifelines

- Participants are shown across the top as objects
- Each participant has a corresponding lifeline running down
- An activation bar shows that the participant is busy (active)
- Activation bars are optional, but useful for visual effect





A message that goes from one object to another goes from that object's lifeline to the other's. An object can also send messages to itself, so the message will loop back to its own lifeline.

The *signature* of a UML message is shown above. This must correspond to the signature of the operation being invoked. However in UC realisation the format may be used very informally – the objective is to capture the general *flow* of messages, not the details of each message.

UML uses arrows to indicate message passing, and the type of arrow shows what type of message it is. UML 2 uses the three shown above:

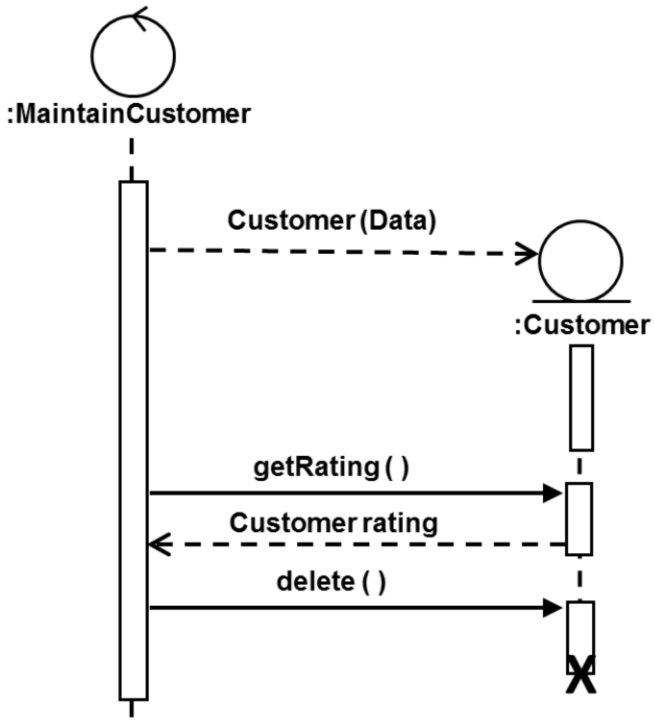
Call is a request from the object sending the message for the receiving object to carry out one of its operations. This is a synchronous call, i.e. the sender waits for the receiver to reply before continuing.

An asynchronous message indicates that the sending object invokes required behaviour on the receiving object, but it doesn't wait for the operation to complete before continuing. A simple example of this is using email or sending something for printing.

SMTU-1 v8.1

Creation and destruction of participants

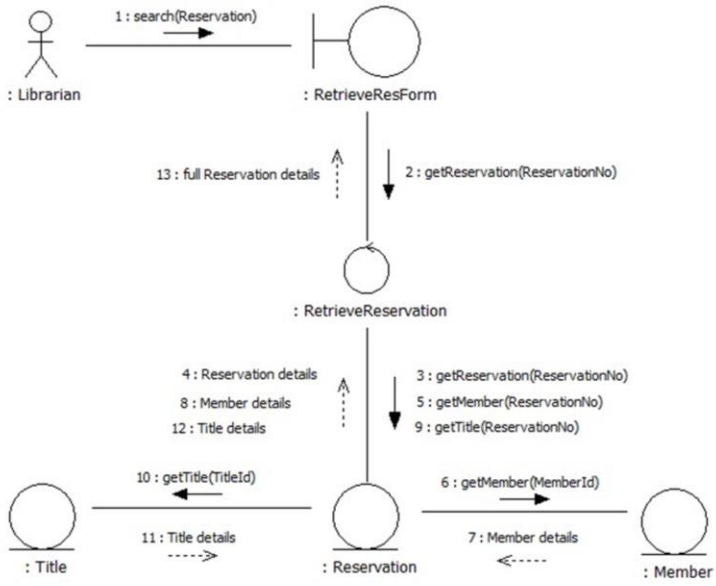
- *Creation* is shown as a dashed line from the sending participant to the receiving participant
- *Deletion* is shown as a message from the sending participant to the receiving participant



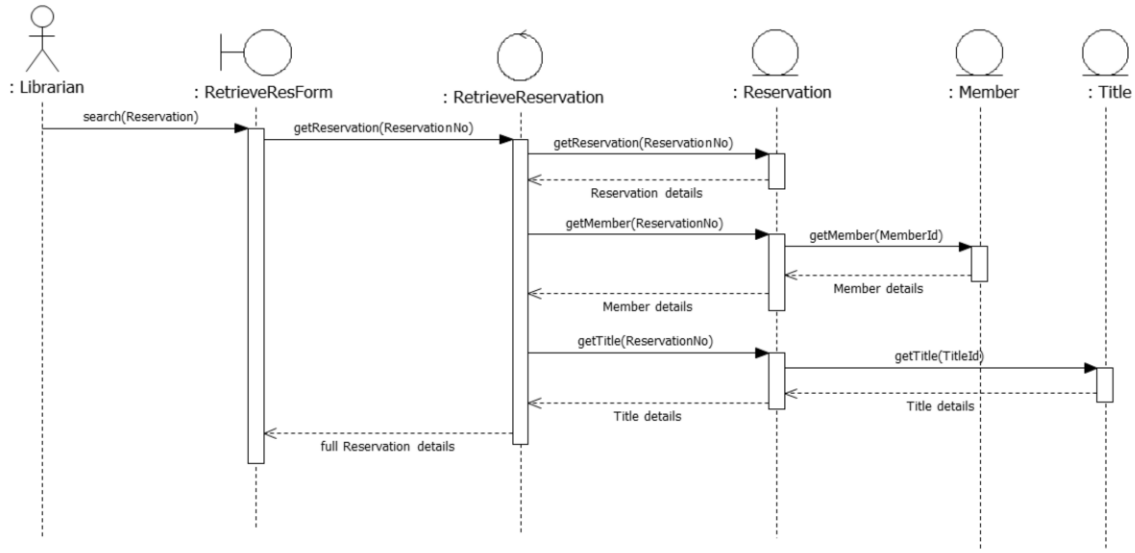
The customer message creates a new object of the Customer class (this is typically handled by the constructor of the class, which has the same name as the class). The delete operation destroys an object.

These messages are often omitted in UC realisation and left for the design model versions.

Communication Diagram ...



… equivalent Sequence Diagram

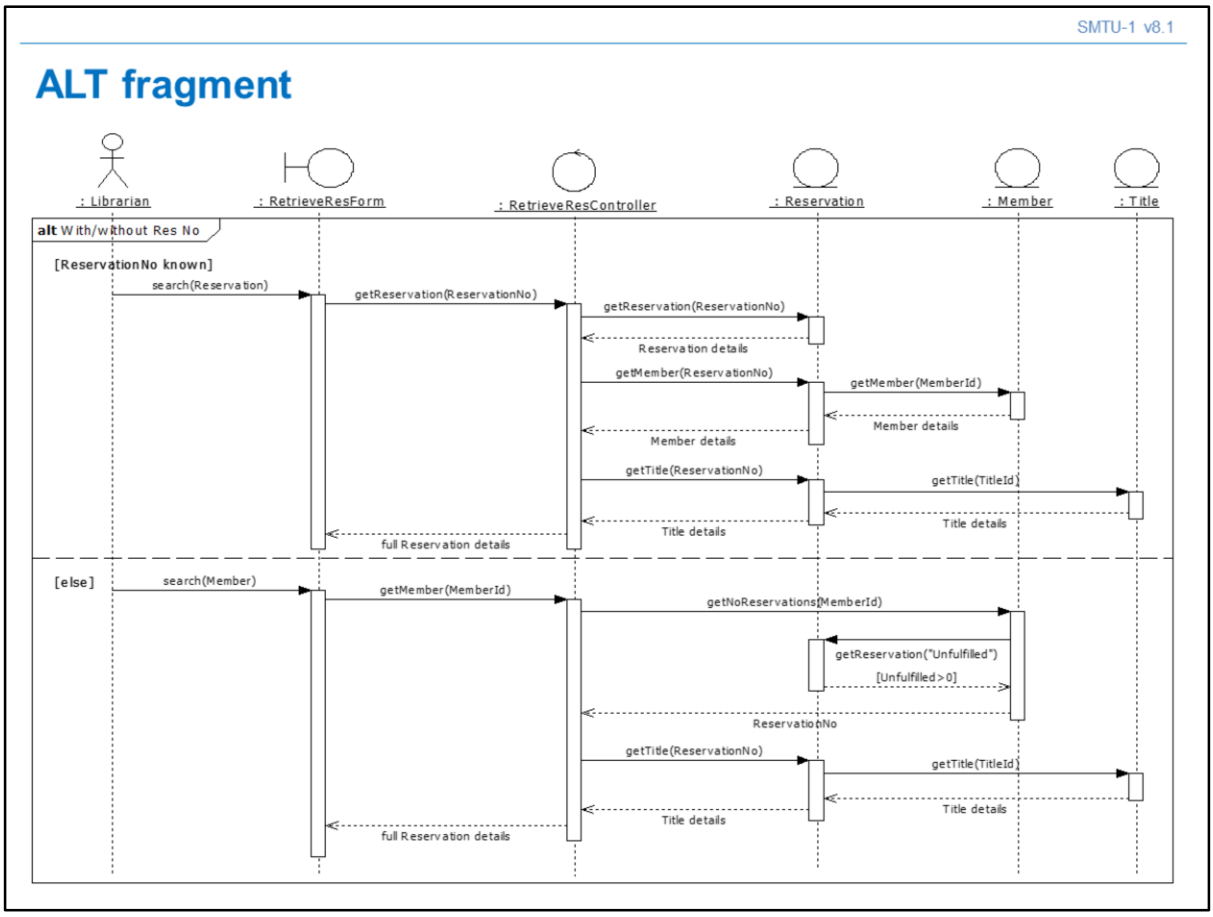


SMTU-1 v8.1

Combined Fragments

- **Sequence Diagrams are designed to show … sequences!**
- **Logic however is not always so straightforward**
  - Need for loops
  - Need for conditions
  - …
- **Answer is to use *combined fragments* overlaid on the diagram**
  - LOOP
  - ALT
  - OPT
  - PAR
  - There are about 15 types altogether
- **Use with care – can make the diagram very difficult to read**

Activity Diagrams are more suited to showing complex logic. It is not advisable for example to use SD to represent all the UC scenarios in 1 diagram.

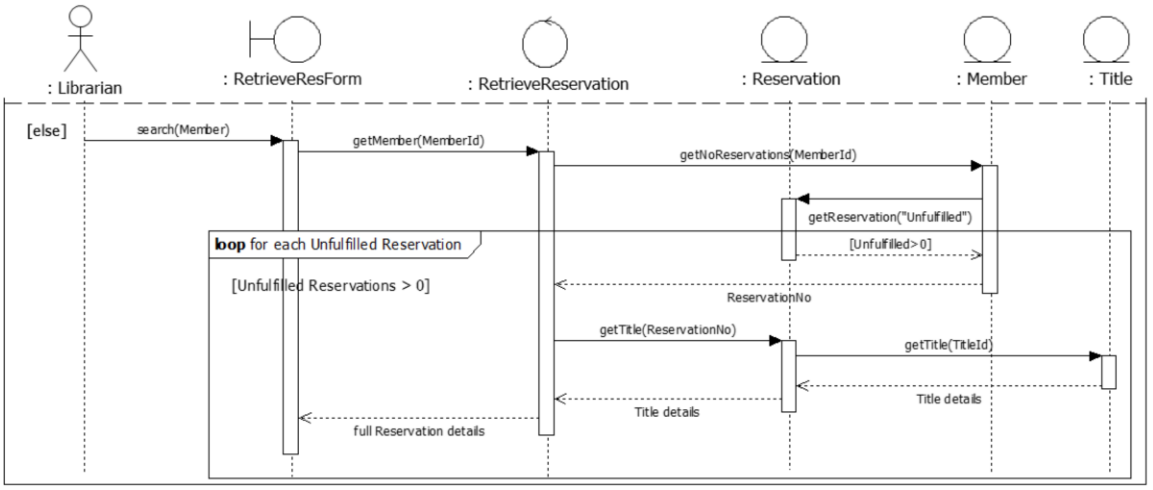


ALT is equivalent to the programming construct: If … elseif … else … endif

SMTU-1 v8.1

LOOP fragment

- **When member ID is used to search for reservations more than one unfulfilled reservation may be returned**
- **A loop fragment may be used to show this**



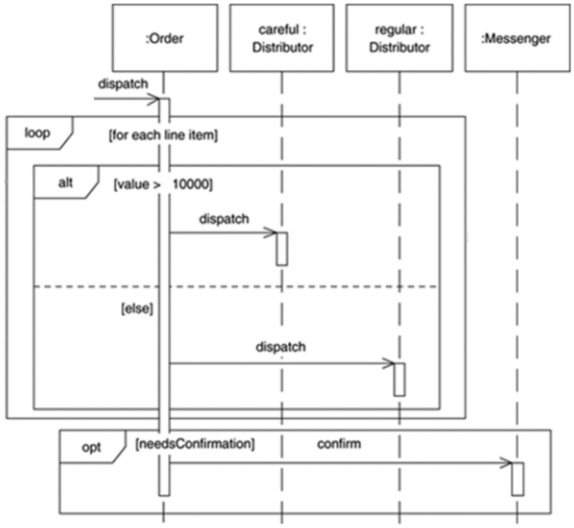
Occasionally there will be a need to model a repetitive sequence. In UML 2, this is achieved using the **Loop** combined fragment. A frame is placed on to the sequence diagram and the text "loop" is placed in the frame's name box. Inside the frame's content area the loop's guard is placed towards the top left corner. Then the loop's sequence of messages is placed in the remainder of the frame's content area.

In a loop, a guard can have two special conditions tested against in addition to the standard Boolean test. The special guard conditions are minimum iterations written as "minint = *n*" and maximum iterations written as "maxint = *n*".

SMTU-1 v8.1

LOOP, ALT and OPT combined

- **Fragments are often combined**



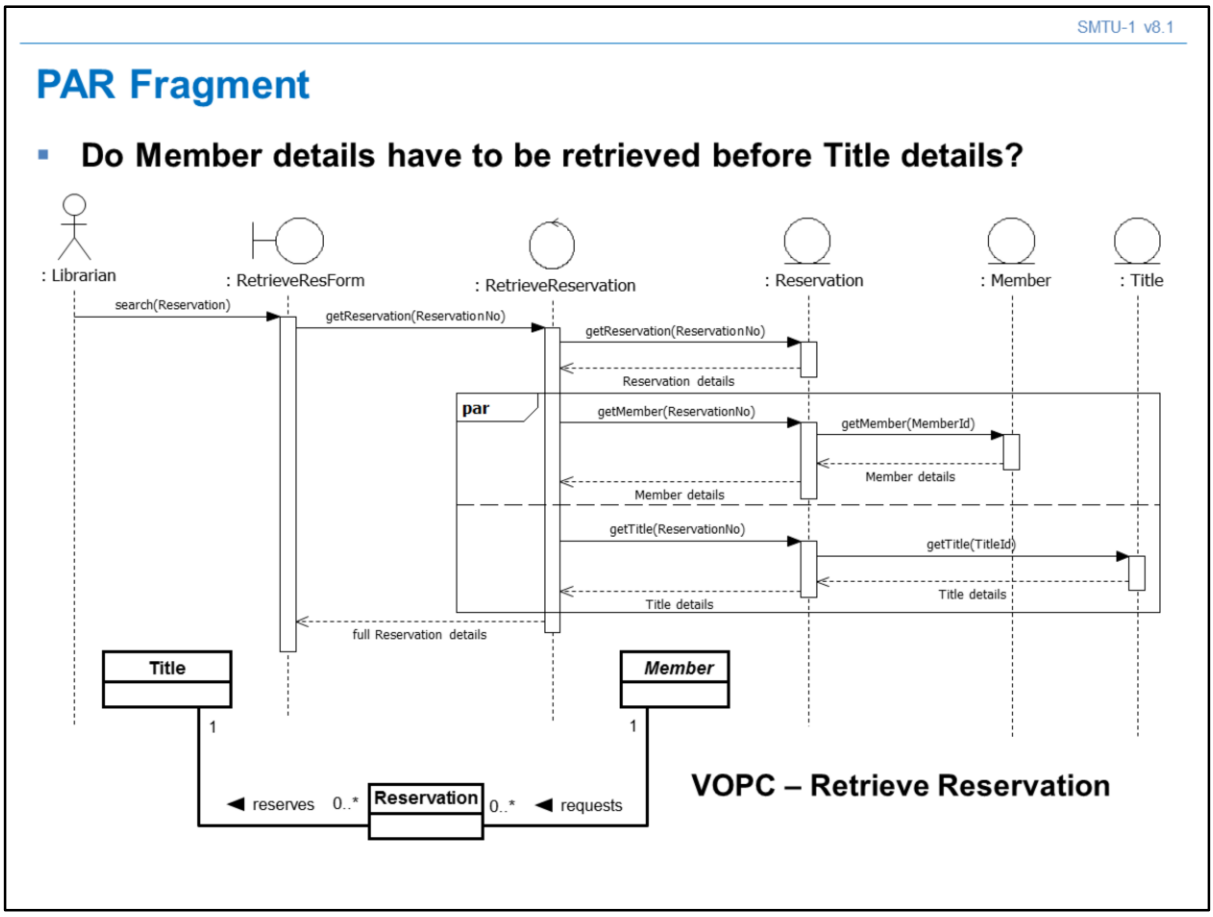
What does these fragments state?

Answer:

An order consists of at least one line item.

If the line item has a value of more than 10000 it will be dispatched by a careful distributor otherwise a regular distributor will be used.

If the Order requires confirming then a confirmation message will be sent.



The sequence of messages from Reservation to Member and Title can happen in either order, or indeed at the same time if the system is capable of multi-threading. Therefore we should not impose sequence if it is not required. The use of a parallel frame clearly shows that there in no dependency between the two messages.

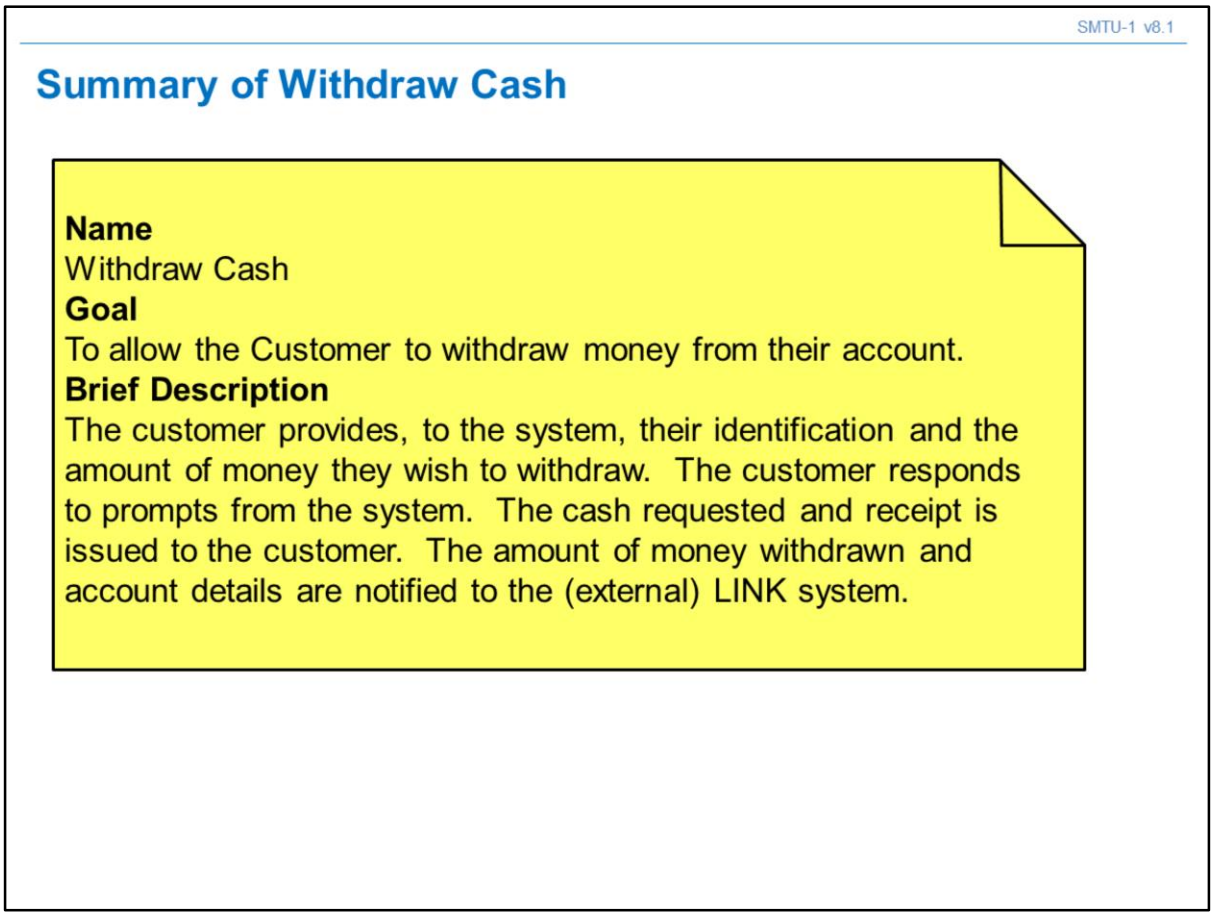
The **par** combination fragment is drawn using a frame, and the text "par" is placed in the frame's name box. The frame's content section is the broken into horizontal operands separated by a dashed line. Each operand in the frame represents a thread of execution done in parallel.

SMTU-1 v8.1

Use Case Realisation – Robustness Analysis

- **In Analysis, the *core* UCs will be analysed for 'robustness'**
- **Given the short UC narrative, a sequence or communication diagram is drawn to check that the flow makes sense, using the Domain Objects, across the MVC architecture**
- **It is essential to derive a standard layout for this, so all UC are analysed in the same way; lifelines are typically:**
  - *Actor*: the primary actor
  - *UI*: represents the User Interface components
  - *Control*: this is the UC control object, dealing with the flow of events from and to the UI, and accessing the data. The logic here could be based on the AD logic (if any).
  - *Entity*: 1 or more objects that represent the domain data – each of these objects should correspond to a Domain Class on our Class Model.
- **This analysis work could lead to changes in the UC diagram, the UC narrative and/or the Domain Model**

Note: the messages are often expressed very informally at this stage. Later in design more attention will be paid to the exact message signatures required.



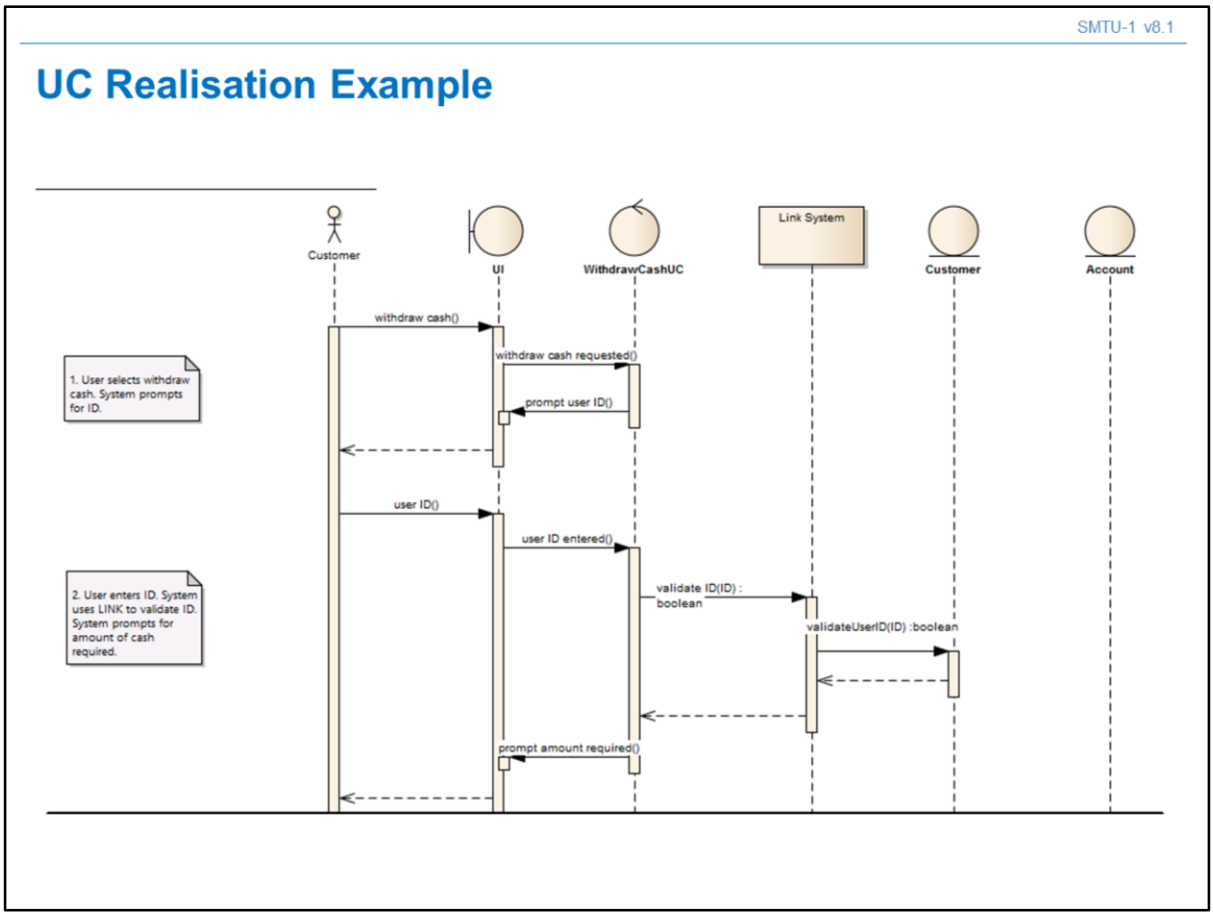
Summary UC description, as noted earlier

SMTU-1 v8.1

Organising a UC realisation

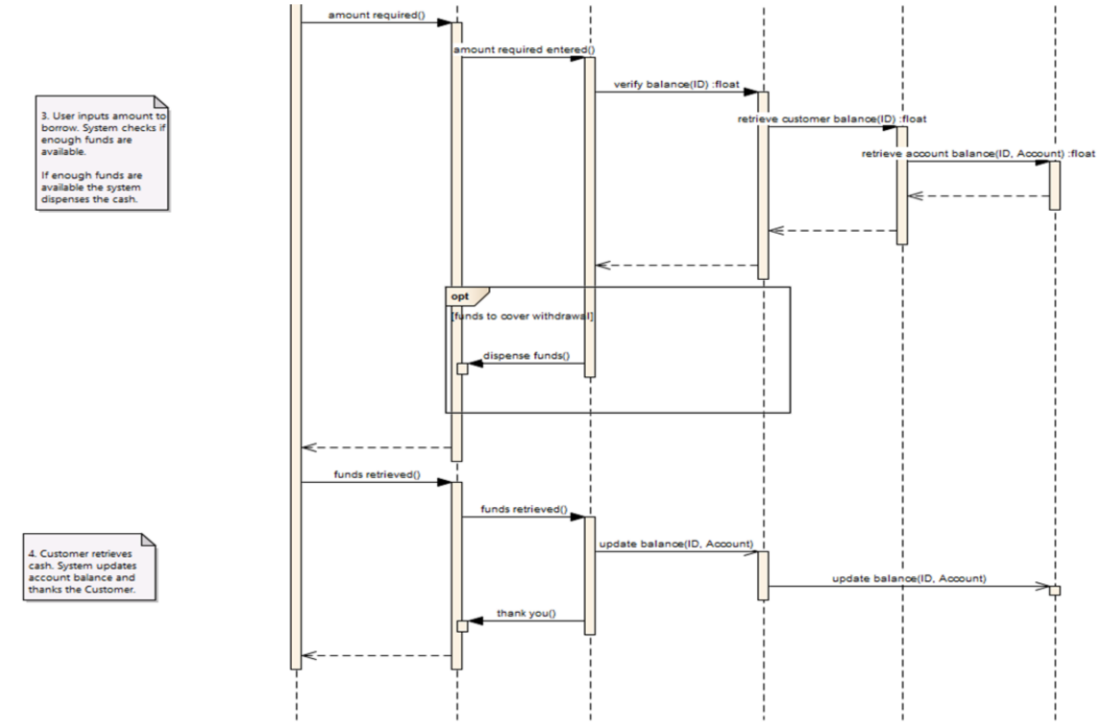
- The Summary or Detailed UC description is the starting point
- Organise the dialogue into Actor/System *round trips*
- Each round trip can then be mapped onto the interaction diagram

Trip	Customer	System	LINK System
1.	Selects *Withdraw Cash*.	Prompts for ID.	
2.	Supplies ID.	Sends message to validate ID. Prompts for amount.	Validate ID.
3.	Selects Amount.	Verifies funds available. If funds available, supplies Cash.	Retrieve Balance.
4.	Retrieves Cash.	Sends message to update Balance. Thanks Customer.	Updates Balance.
	Use case ends.		



Although a tool has been used here, it is quite common to hand sketch these robustness checks, the idea being to *analyse* not *design*

UC Realisation Example (II)



SMTU-1 v8.1

Summary

- **Sequence diagrams**
- **Message types**
- **Fragments**
  - ALT
  - LOOP
  - OPT
  - PAR
- **Organising a UC realisation**