7SENG003W Advanced Software Design

Use Case Modelling

Recap

- So far, we've looked at:
  - Classes and objects in UML
    - Icons and diagrams
  - How objects communicate
    - Through message-passing
  - Their relationship to Java classes and objects
    - How to code UML classes and objects
    - Vice-versa
- Now, we will start looking at how you start the development process
  - How, from a statement of requirements, you can produce a design that can be turned into code

Getting started with design

- Previous lectures have shown some of the basic features of UML, and how they can be used to document code structure
- But how is this structure – the design - arrived at in the first place?
  - 2 important questions:
  1. How does a software designer arrive at the set of classes that make up the fundamental architecture of a system?
  2. How do you know that a given set of classes will support the functionality required?

  - These kinds of questions are answered by the “process” element of a methodology?

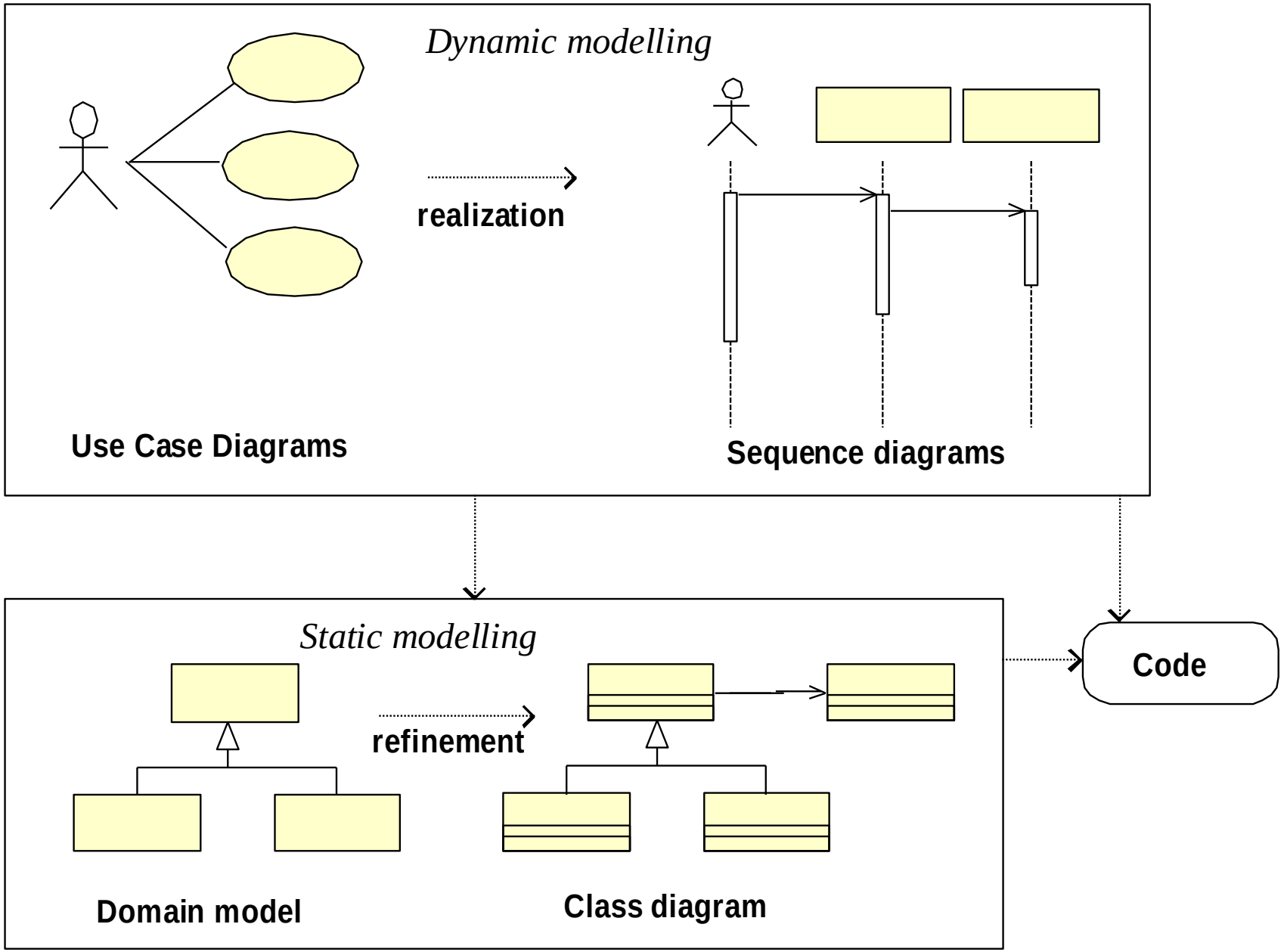
Two approaches

- There are two main approaches to the problem of getting started with the design of a new system

1. Data-driven approaches
   - Look first at the data that is to be handled by the system
   - Try to model the data as a set of classes
   - Implement the required functionality in these classes
   - A static approach to system modelling

2. Behaviour-oriented approaches
   - Look first at the behaviour or functionality required
   - Try to deduce what classes are required to support this functionality
   - What data does each class need
   - A dynamic approach to system modelling

General Process



Static Models

- Static models describe the structure of the data in the system
  - Domain models describe the problem space
    - The real-world entities and data that the system will deal with in terms of the object model
    - The environment within which the system will be used
  - The structure of the solution space (the program) will be different and usually more complex, but the basic structure of the domain model should still be present in the final code

- In this way, because the object model is used throughout the life-cycle, the structure of an object-oriented program often reflects aspects of the real-world structure
- Producing a domain model, therefore, can often be a useful starting point for development
  - Helps us identify classes we can use in the system design

Dynamic Models

- Dynamic models describe the system's behaviour:
  - Use Case models classify the users of the system and the various tasks that they can perform with it
  - Sequence diagrams break these high-level tasks down into detailed interactions between objects
  - Statecharts (not shown on the process diagram) summarize dynamic properties of objects of a particular class

- Consistency:
  - The structure described in the static models must support the interactions described in the dynamic models
  - For example, objects shown on sequence diagrams must be instances of classes in the class model

Use-Case Driven

- UML is often described as supporting use-case driven design methodologies
  - Development starts with a use case model which aims to model in some abstract way the user's **functional requirements**
  - All subsequent work depends heavily on the information contained in this model
  - Aim is to ensure that the final developed system fully meets the users requirements
- *Domain modelling* is used as part of the development and helped inform use-case driven design
  - Develops understanding of the problem domain
  - Develops vocabulary
- Knowledge of the domain can ease the description of use cases enormously

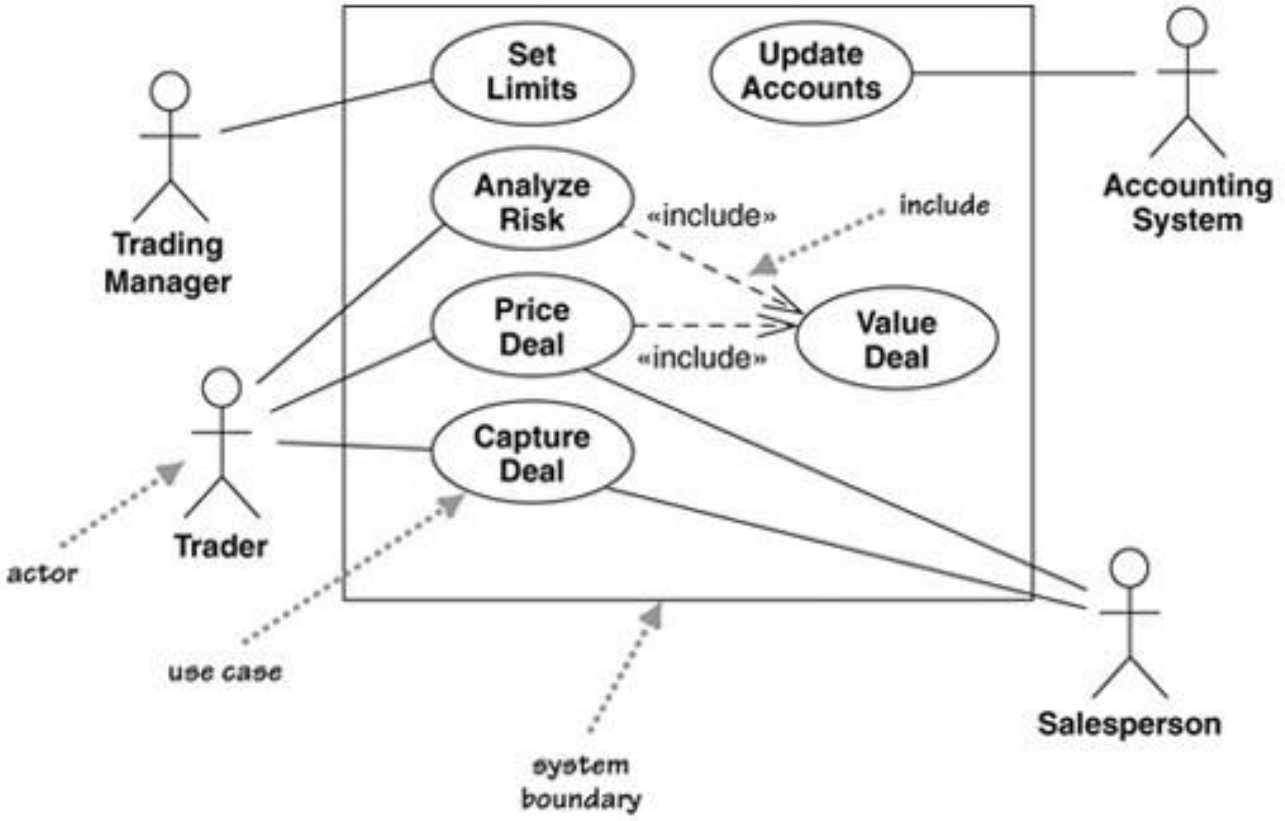
Use Case Modelling

- One problem with starting design straight from requirements is that they are a static description of what the system should do. But in object-oriented development (OOD), we're concerned with objects and their interactions, so we need to understand dynamic behaviour.
- One technique for capturing dynamic behaviour, which has caught on in OOD circles, is use case modeling.
- A use case is a statement about how a user will interact with the system to achieve some goal.
  - the goal is something that the user wants to do (and which has value to the user) and which the system can help the user to achieve
  - the user interacts with the system, with both the user and the system taking a sequence of actions, to achieve the stated goal
  - "Use cases document the behavior of the system from *the user's point of view*" (Stevens and Pooley, *Using UML*)

Functional requirements

- “Use cases are a technique for capturing the **functional requirements** of a system. Use cases work by describing the typical interactions between the users of a system and the system itself, providing a narrative of how a system is used.” (UML Distilled, Chap. 9)
- Functional requirements
  - Expected behaviour or capabilities of the system as demanded by the user, e.g.,
    - Order products from a company’s catalogue
    - Checkout shopping basket
    - Book a dental appointment
    - Find out train times for Thursday afternoon from London Euston to Coventry
- Non-functional requirements
  - Things about a system, e.g,
    - Web-based
    - Encrypts user data
- https://www.geeksforgeeks.org/functional-vs-non-functional-requirements/

UML Distilled example use case diagram



Use Case Modelling

- Use case diagrams model a system in terms of:
  - A set of *actors*: each actor describes a *role* that users can play when interacting with the system. Notice that actors are not the same as individual users.
  - A set of *use cases*: each describes a possible kind of interaction between an actor and the system.
  - A number of relationships between these entities
- Use cases provide a structured view of the system's functionality, and make a good starting point for system development.

Example application

- In this lecture, we're going to use a car dealership as an example application area
- The kind of people who work in a car dealership are:
  - Receptionists, who answer the phone and book appointments for cars to go into the garage, sales visits etc., and who process payments
  - Car salesmen/women, who sell cars
  - Engineers
  - Chief engineers
  - Drivers, etc.

- The example application will be for the garage side of the car dealership, which fixes, services, and MOTs cars

Use Cases

- A use case expresses **a goal**. For example:

- The use case additionally tells the story (or set of stories) about
  - how this goal is achieved
  - Alternative paths that might be taken but which still achieve goal success
  - what happens if the goal cannot be achieved (failures)
- The story or **scenario** is
  - expressed as an informal, semi-structured narrative in native language
  - details the steps taken by the user and by the system towards fulfilling the goal, or what happens if the goal cannot be met
  - Helps clarify the use case
- A use case may be composed of a number of scenarios
  - Typically there will be one main scenario, which describes what happens when everything goes according to plan… the **happy days** scenario

Example happy days scenario

Goal: Book MOT

1. Receptionist enters date
2. System displays available timeslots for MOTs
3. Receptionist selects timeslot
4. System prompts for car registration number
5. Receptionist enters registration
6. System displays customer details
7. Receptionist confirms customer details
8. System confirms MOT booking

There are many ways to write scenarios – the numbered list format is one popular format.

Actors

- A use case model describes users as actors, denoted in UML use case diagrams as:

- These may be users or other systems (since even systems have goals!). An actor, however, is not a representation of a real user or system, but of **the role** that a person or system plays when they interact with the system.
- "Actors are idealized users"
- Typically, each use case has one principal/primary actor, the one with the goal!
- However, each use case may have secondary actors, external entities which help the system achieve sub-goals as part of meeting the principal actor's goal.

actors

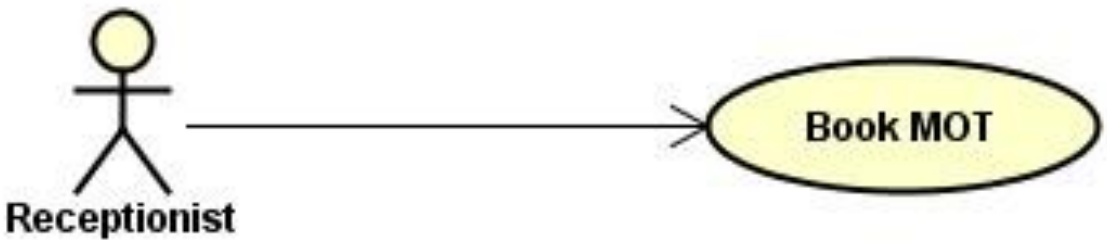
We really mean “roles” rather than “actors” in the traditional sense…

Him

But not him…

Actors and Use Cases

- The primary actor is the one who seeks to fulfill the use case goal by interacting with the system.
- They supply the trigger that starts off the sequence of steps detailed in the scenario. This is shown as:

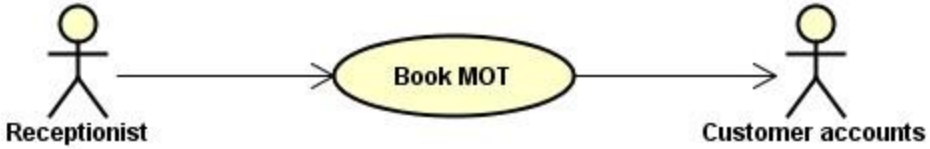


- The primary actor is simply an idealized form of someone/something that needs the system and derives value from the interaction.

Primary & Secondary actors

- The “primary” actor is the actor that has the goal

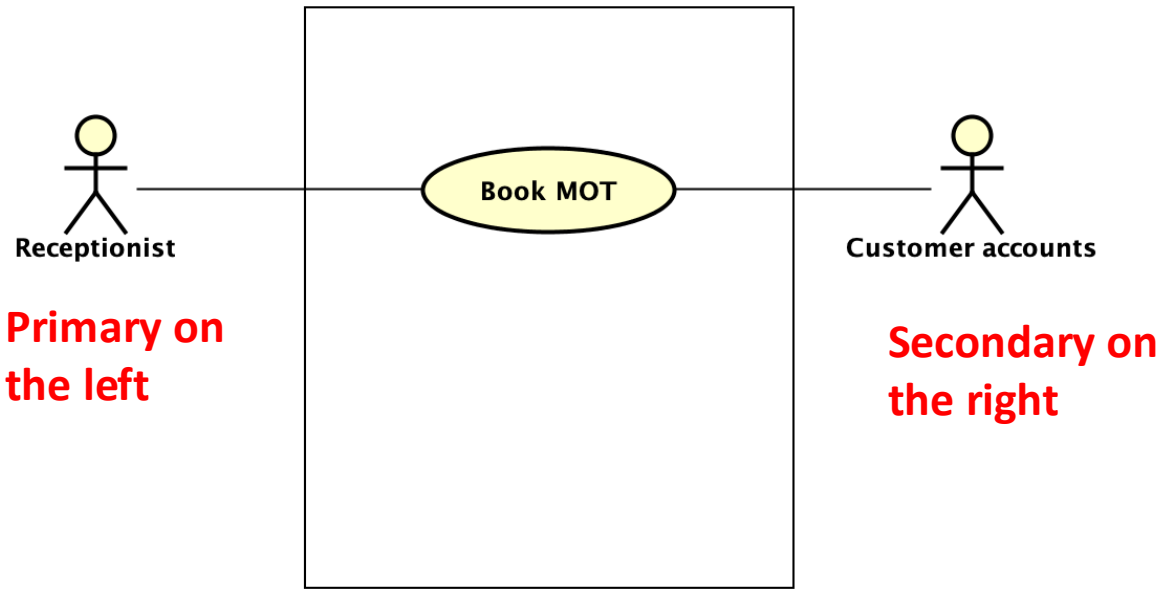
- Some use cases require assistance of a “secondary” actor



- Direction of arrow on association between actor and use case indicates who **initiates** dialogue – if actor initiates, then they are primary; if system initiates, then actor is secondary

Primary and secondary actors – alt format

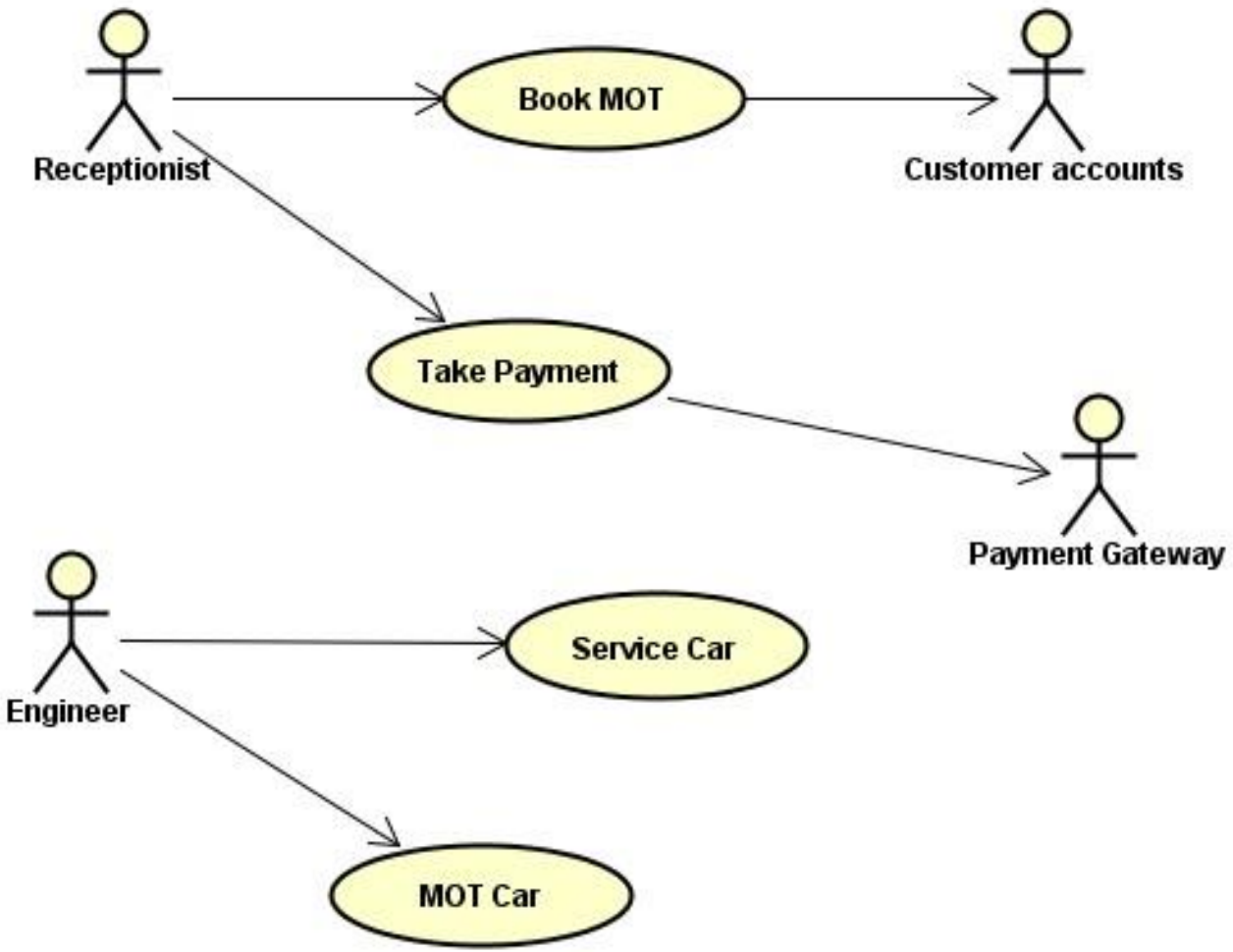
- An alternative format for showing primary and secondary actors
- Also show box around use cases that represent goals that the system helps the user achieve



Creating use cases

- The first step is to identify who will be the actors
  - an actor really identifies a type or category of behaviour that external entities exhibit when using the system for a particular purpose
- What are the goals of these actors? What are they trying to achieve, for which they need to interact with the proposed system?
- Task:
  - what are the actors in a car dealership garage?
  - What are their goals?

Example use case diagram



Describing use cases

- Use cases are described informally in UML. Each use case corresponds to a class of possible interactions that a user can have with a system.
- It is normal to distinguish the *basic course of events* in a use case from the various errors or exceptions that can occur.
- Often, a use case description will consist of a description of the normal case together with brief descriptions of exceptional cases at the points where they can occur.
- *Scenarios* are often used to help clarify use cases. A scenario is a detailed description of a single interaction that a user has with a system. Whereas a use case may be described in quite general terms, a scenario will spell out every detail of a particular interaction.

Example scenario

Use Case: Book MOT

Principal Actor: Receptionist

Trigger: Customer telephones to book MOT appointment

Main scenario

1. Receptionist enters date
2. System displays available timeslots for MOTs
3. Receptionist selects timeslot
4. System prompts for car registration number
5. Receptionist enters registration
6. System displays customer details
7. Receptionist confirms customer details
8. System confirms MOT booking

Exceptions and Variations

- Every other scenario for a use case should be a variation or exception of this main scenario.
  - *variations* branches or variations in the happy days scenario - these can rejoin the main scenario at a later step
  - *exceptions* branches in the happy days scenario to deal with things that go wrong - i.e., the exceptions - usually terminate the use case
- Task: what could go wrong in the main scenario?
- Variations and exceptions are useful since they highlight unusual conditions/situations.
  - often not envisaged in the statement of requirements, and only come to light as a result of asking "what-if" type questions.
  - help identify test cases for the implementation
- Task: any alternatives in the main scenario?

Variations

- Use variations to extend main scenario in some way, e.g,

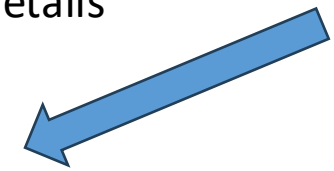
Main scenario

1. Receptionist enters date
2. System displays available timeslots for MOTs
3. Receptionist selects timeslot
4. System prompts for car registration number
5. Receptionist enters registration
6. System displays customer details
7. Receptionist confirms customer details
8. System confirms MOT booking

Variations

6.1 Unknown customer

1. Prompt customer for their details
2. Customer provides details
3. Customer added to Account Manager's list of customers



Extends main scenario with registration

Exceptions

- Use exception scenarios to deal with things that go wrong – either to fail gracefully or deal with the exception and recover. E.g.,

Main scenario

1. Receptionist enters date
2. System displays available timeslots for MOTs
3. Receptionist selects timeslot
4. System prompts for car registration number
5. Receptionist enters registration
6. System displays customer details
7. Receptionist confirms customer details
8. System confirms MOT booking

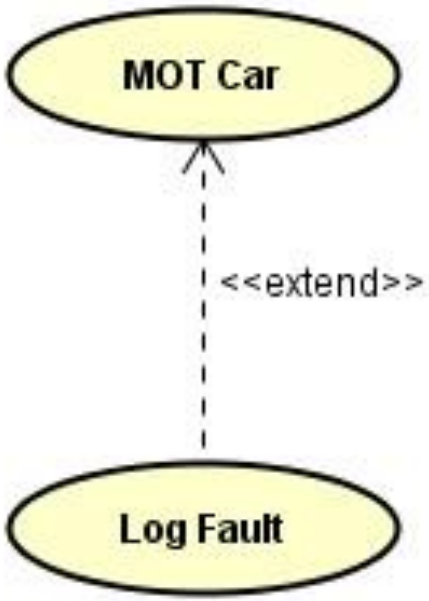
Exceptions

5.1 Correct invalid registration

1. Ask customer to provide registration number again
2. Receptionist enters registration
3. System confirms registration is correct
4. Rejoin (6)

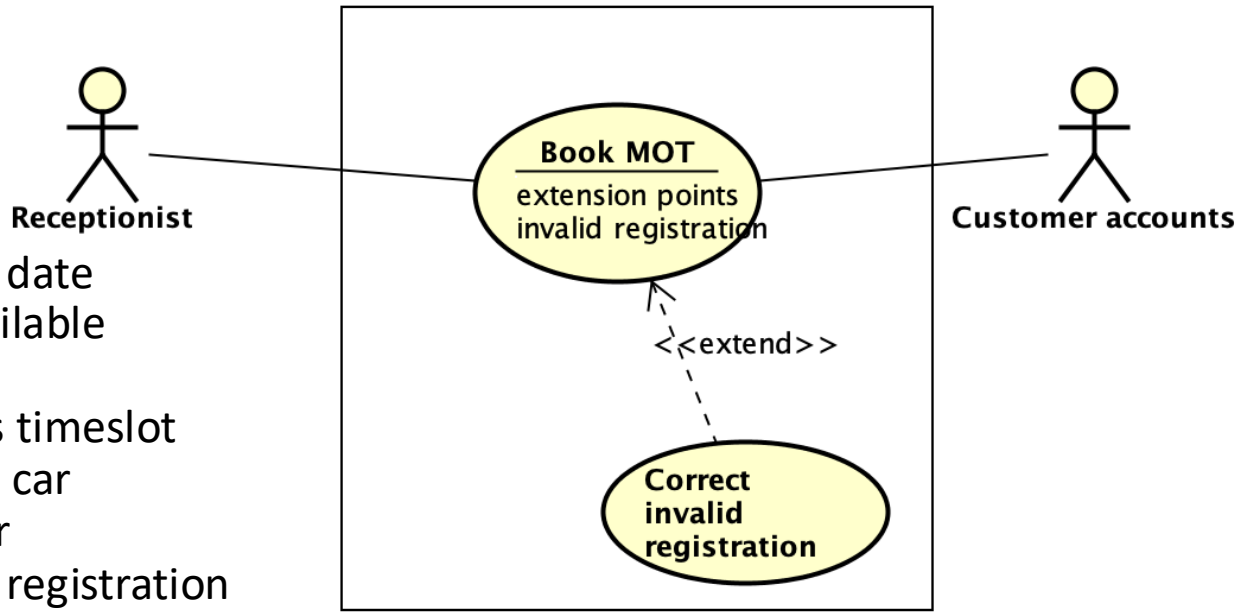
Use case extension

- The extends relationship between use cases allows you to represent optional behaviour or behaviour that is executed only under certain conditions
- This example use case diagram shows that we follow the “MOT Car” use case up to a certain point (indicated in the use case scenario) and, if certain conditions are met, then follow the “Log Fault” use case
  - Hence the “Log Fault” use case extends the “MOT Car” use case, by providing additional behaviour to be performed by under certain conditions



- Why do this instead of exception scenario in “MOT Car” use case
  - If “Log Fault” includes a significant amount of work. If not, then write as exception scenario

Exception points



Main scenario

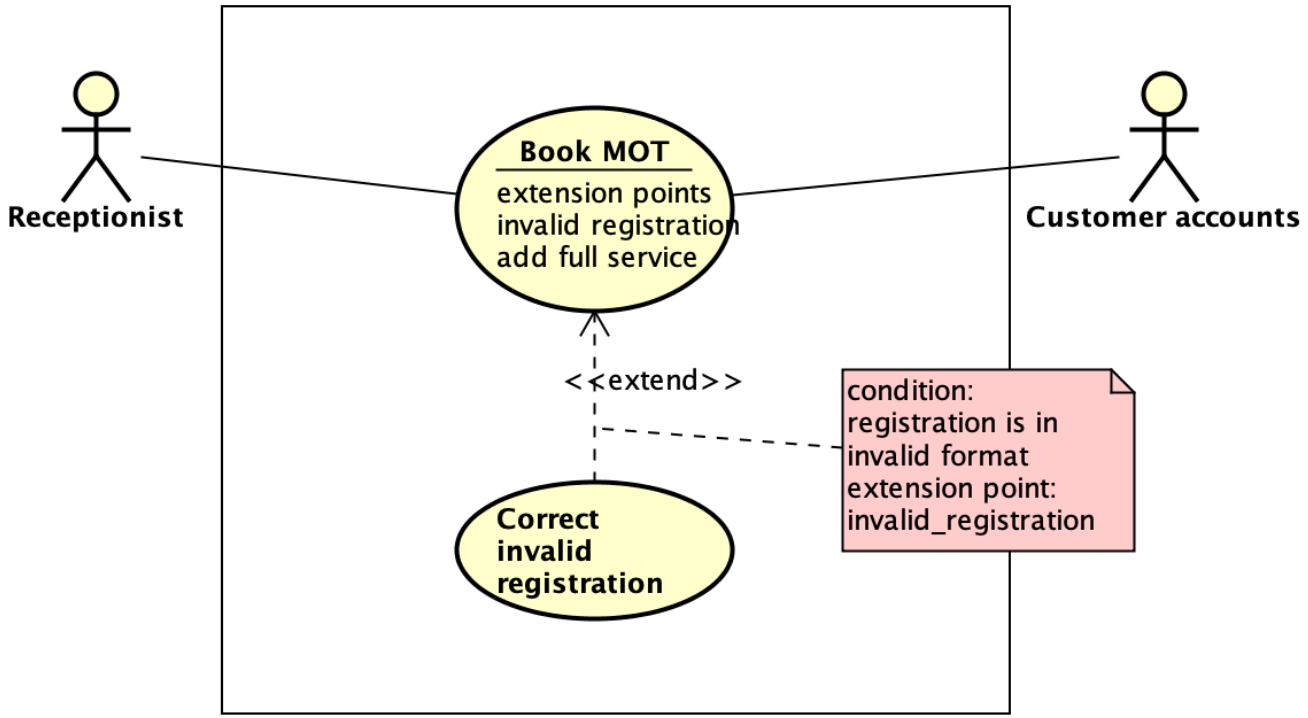
1. Receptionist enters date
2. System displays available timeslots for MOTs
3. Receptionist selects timeslot
4. System prompts for car registration number
5. Receptionist enters registration

Extension point: invalid_registration

1. System displays customer details
2. Receptionist confirms customer details
3. System confirms MOT booking

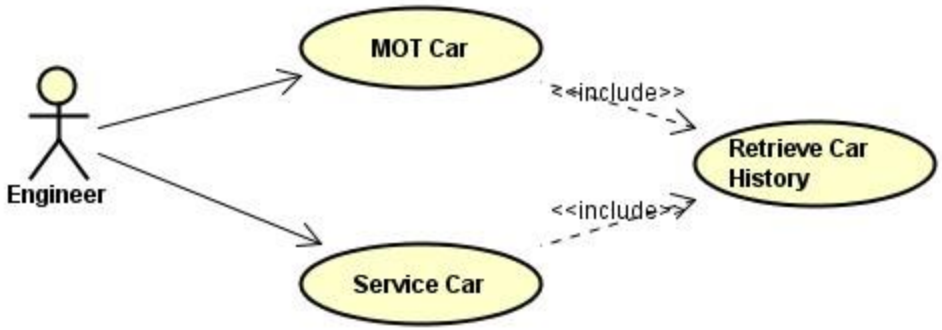
Extension points allow us to indicate where in the extended use case scenario the extension occurs

- If there are multiple extension points, we can add notes to extends relationships to indicate which extension point they use



Including common functionality

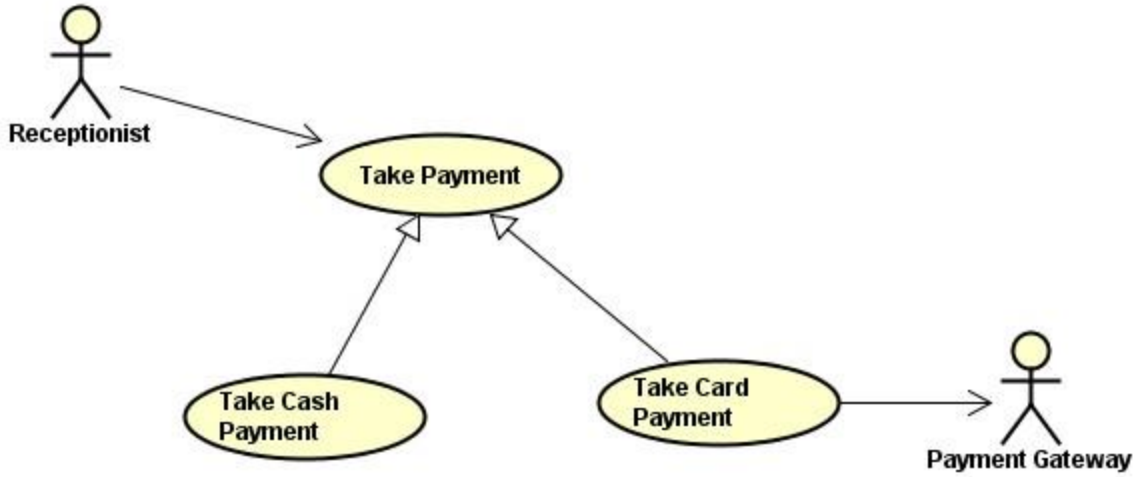
- Some use cases may represent functionality that is common to other use cases



- The includes relationship between “MOT car” and “Retrieve car details”, for example, means, in principle, that the steps taken to retrieve the car details are ALWAYS done when we execute the main scenario for “MOT car”
- Note the relationship between the use cases is a DEPENDENCY

Use case generalization

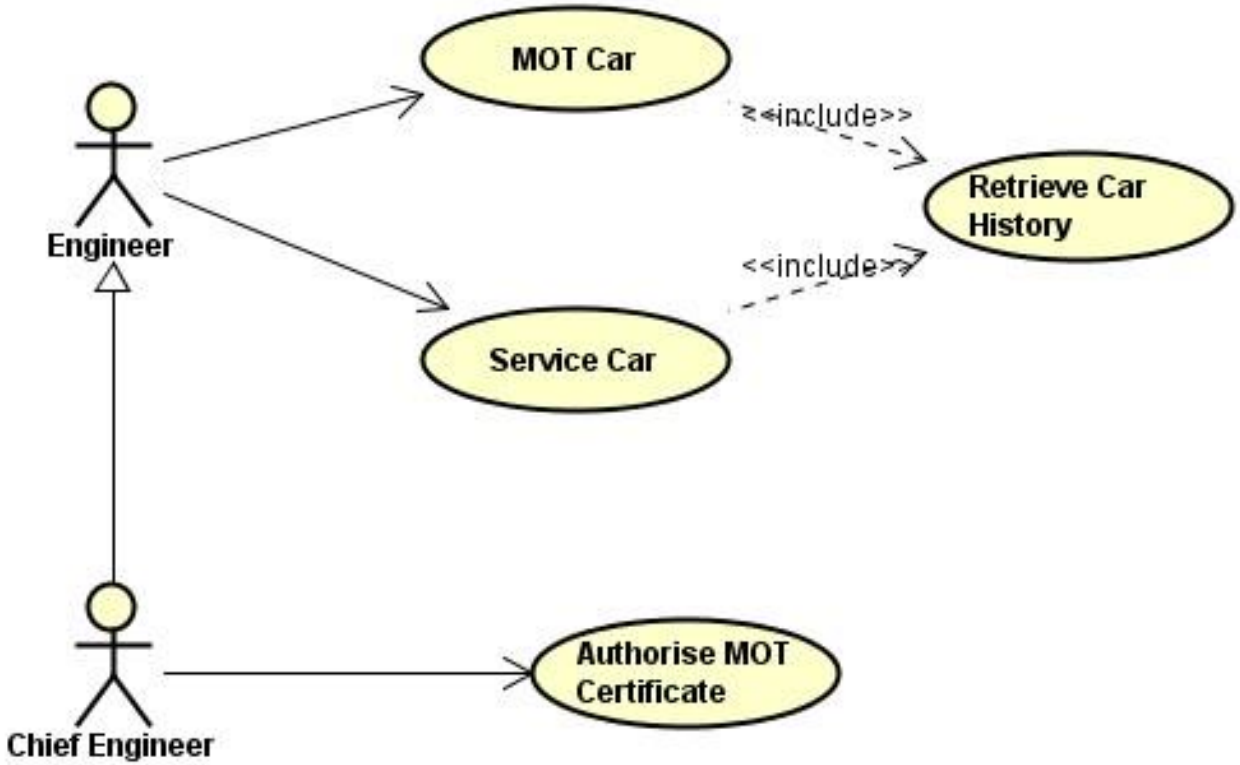
- There may be cases where two use cases represent the same basic goal and flow of events, but each is a special case
- We can use generalization to represent this relationship



- Note that the base use case is abstract
- The “child” use cases specify what happens when either a cash payment or a card payment is processed

Actor generalization

- As well as use cases, actors can be related by generalization. This provides a convenient way for different levels of access to a system to be provided



The final use case diagram