UML: Use cases and Case study 1: identifying classes

CSCU9P5 - Software Engineering I

UML & Object Modelling

- UML & Object Modelling - Use Cases & Case Study I
  - Requirements engineering
  - Use cases
  - Bank account: use cases
  - Identifying classes
  - Bank account: classes and class diagram

Requirements Engineering

- Purpose of Requirements Engineering/Analysis is to identify and clarify requirements.
- Issues
  - Initial informal requirements are usually vague, inconsistent and incomplete, eg:
    - Badly defined requirements: customers do not always clearly state what they need
    - Frequently changing requirements
    - Different users/clients with conflicting needs.

Requirements Engineering

- Issues
  - Communication barriers:
    - Managers may have no direct experience of actually using the system or a clear understanding of technical issues/possibilities.
    - The Analyst and Designer may have communication issues, too.
  - It can be difficult to visualise how a system works in practice until you have actually used it.

Requirements Engineering

- Risk management (more later…)
  - When you make a decision, you run the risk that it is wrong.
- Discover errors as soon as possible
  - the later you find out the error, the more expensive it is to put it right.
- A major risk is that we misunderstand the requirements
  - Frequent evaluation steps (e.g. iterative methods)
- Prototyping the solution or part of the solution and showing the results to customers/users helps cut down this risk
- Use case are a useful first step in prototyping

Use Case Models

- Object-oriented development methods usually take a user-oriented approach to system development.
  - We identify the users of the system and the tasks that they require the system to perform.
- UML uses the term actor.
  - An actor is the role played by a user.
  - Users can be humans, hardware devices or other software systems, i.e. external entities that interact with the system.
  - A user may interact with the system in more than one role and hence be represented by more than one actor.
    - For example, if a bank clerk withdraws money from an ATM, they are playing the role of an ordinary customer and are treated in our model as an ordinary customer.

Use Case Models

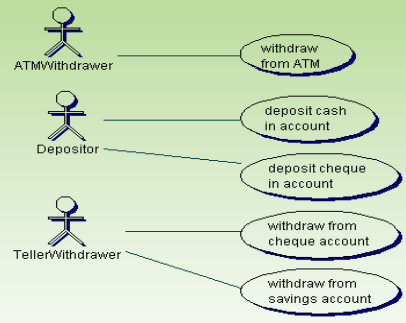
- A **use case** is a task that an actor needs to perform with the help of the system.
- A **use case** documents some behaviour that the actor needs from the system **from the actor's point of view**.
- It **does not** specify **how** the behaviour will be carried out.

Use Case Model

- Use cases enable the informal requirements to be reformulated in a more structured manner.
- The set of use cases should cover the set of **requirements** as seen by the users of the system.
- Concentrating on use cases means that **users are involved** in the early-phase development of the new system. This cuts down resistance to change.
- Remember that a use case **is not** involved with design details. It is concerned **with high-level behaviour** that is to be provided, **not with how** it is to be carried out.

Use Case Diagram

- We represent the use cases graphically in a use case diagram.
- It consists of actors and the individual use cases with which they are involved.
- Each use case is given a natural language description.
- Several other use cases can be added to the diagram.



Case Study: Automated Banking System

- UML (Unified Modelling Language) provides a notation for use cases.
- Let us consider a case study of an Automated Banking System:
  - Clients may take money from their accounts, deposit money or ask for their current balance.
  - All these operations are accomplished using either automatic teller machines (ATM) or counter tellers.
  - Transactions on an account may be done by cheque, standing order or using the teller machine and card.
  - There are two kinds of account: savings accounts and chequing accounts.
  - Savings accounts give interest and cannot be accessed by the automatic tellers.
  - When a cheque is deposited it must be cleared before the funds can be used by the depositor.

Example Use Case

- An example of a use case is **withdraw money from ATM**.
  - The textual description of the use case is written in terms of what the user expects the system to do rather than as a passive description of what has to be done.
  - *An* `ATMwithdrawer` *inserts a card in an ATM and types in a PIN number.*
  - *The* ***system*** *checks that the PIN matches the card.*
    - *If it does not, the system rejects the card.*
    - *If valid, the* `ATMwithdrawer` *types in the amount to be withdrawn.*
      - *The system checks that the account has sufficient funds.*
        - » *If it does, the system debits the account and gives the money.*
        - » *Otherwise, the system refuses the transaction.*

Use Cases

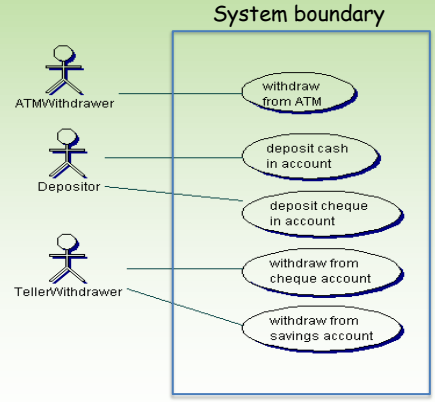
- In constructing use cases, we should only put in what the users have told us.
- However, the process of creating the use cases is likely to give us ideas on how the system can be improved / extended.
- These should act as the basis for questions to the users, rather than be added directly to the use cases.
- In developing our model, it is best to concentrate on only a subset of the use cases. Others can be added later.
- It is important to note that there is not a single correct model
  - Different groups may arrive at different solutions, each of which is acceptable.
- Creating good models is **not easy**
  - We seldom (never?) get it right first time.

Why Use Cases are Important

- Each use case will be realised by **one or more UML diagrams** of various kinds.
- The use cases help us construct these diagrams.
- Once we have a model, it is important to ensure that it satisfies the requirements -- called **validation**.
- Use cases are important in determining the set of **test cases** to be used in validation.

The "System Boundary"

- In a use case diagram it is helpful to draw the "system boundary"
  - Helps emphasise what is *outside* and what is *inside* the system



Identifying Classes

- The next step is identifying the classes (or the objects, at this level of abstraction).
- As a first step, we can take our informal requirements and identify the nouns or noun phrases, i.e. identify the words that potentially represent things in our system.
- In the description on the next slide, the nouns and noun phrases are in bold.

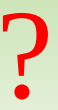
Identifying Classes : Example

- ***Clients*** *may take* ***money*** *from their* ***accounts****, deposit money or ask for their* ***current balance****.*
- *All these operations are accomplished using either* ***automatic teller machines*** *(ATM) or* ***counter tellers****.*
- *Transactions on an account may be done by* ***cheque****,* ***standing order*** *or using the* ***teller machine*** *and* ***card****.*
- *There are two kinds of account:* ***savings accounts*** *and* ***cheque accounts****.*
- *Savings accounts give* ***interest*** *and cannot be accessed by the automatic tellers.*
- *When a cheque is deposited it must be cleared before the* ***funds*** *can be used by the* ***depositor****.*

Identifying classes

- This gives us an initial set of potential classes. Let us consider each in turn: (first group, more later…)

- ***Client***
- ***money***
- ***account***
- ***current balance***
- ***automatic teller machine***
- ***counter teller***



Identifying classes

- This gives us an initial set of potential classes. Let us consider each in turn:
  - ***Client***: We could discard this because it is an actor. However, we may need to ***hold information about*** clients.
  - ***money***: We have to think about what exactly we mean by this.
  - ***account***: An important class.
  - ***current balance***: This is more like the attribute of an account.
  - ***automatic teller machine***: An important class.
  - ***counter teller***: An important class.

Identifying Classes

- ***cheque***
- ***standing order***
- ***teller machine***
- ***card***
- ***savings account***
- ***cheque account***
- ***interest***
- ***depositor***

?

Identifying Classes

- ***cheque*** We have to think about what exactly we mean by this.
- ***standing order*** An important class.
- ***teller machine*** Same as automatic teller machine.
- ***card*** We have to think about what exactly we mean by this.
- ***savings account*** Subclass of account?
- ***cheque account*** Subclass of account?
- ***interest*** This is more like an **attribute** of a savings account.
- ***depositor*** Same as client.

Identifying Classes

- We have identified that `savings account` and `cheque account` are subclasses of `account`, i.e. they are **specialisations** of account.
- Another way of identifying an inheritance hierarchy is to identify several classes which can be **generalised** as they have many properties in common.
- Take, for example, `automatic teller machine` and `counter teller`.
- We could identify a superclass `entry station` which defines the properties common to `automatic teller machine` and `counter teller`.

Identifying Objects & Classes

- When considering a problem, it is often easier to think initially in terms of objects rather than classes.
  - Humans tend to think in terms of a Vauxhall car, i.e. an object rather than in terms of the class Vauxhall car.
  - This is not a problem; it is easy to translate **typical objects** into **classes**.
- During the identification phase, we may often use the term **object** and **class** interchangeably.
  - Assuming that the difference between the two concepts is crystal clear to you by now???
- Note: UML also has *object diagrams*

Identifying Objects & Classes

- Some objects are fairly easy to identify, while others are more difficult.
- Identifying **nouns** in the informal requirements is a good way of identifying **domain objects**, i.e. objects that belong to the problem.
- Two **kinds** of domain object are:
  - **entity objects** that hold information about entities in the problem domain, and
  - **boundary objects** through which we interact with the outside world.
- Hence, `Account` is an entity class while `ATM` is a boundary class.
  - They are the easiest kind of object to identify.

Identifying Objects & Classes

- Some "potential classes" are **both *actors* and *entity classes***
- For example:
  - Client must remain as an actor
  - But we also need an **entity class** *within* the system to hold *information about the client*
- Some "potential classes" may appear to be **actors** but
  - It is not clear whether they are "inside" or "outside" the "system"
  - And they may become "boundary classes"
- For example: Counter teller
- These can be a confusing issues!

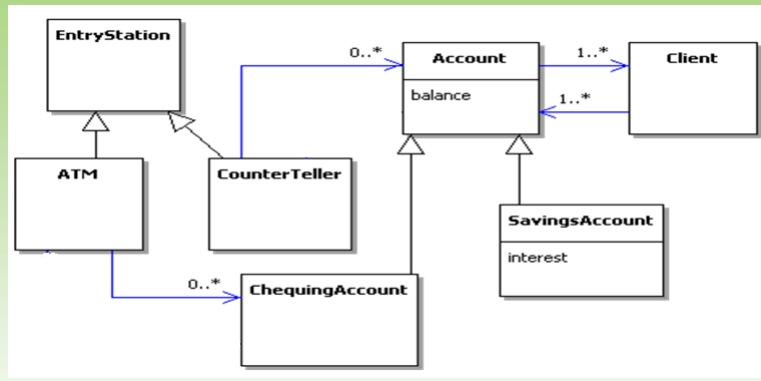
Identifying Objects & Classes

- Other objects are created to help with the solution.
- They can be much harder to identify.
- An example is **control objects** that are created to co-ordinate some process after which they terminate themselves.
- We will see later how **sequence diagrams** can help in the identification of control objects.

Class Diagram

- We can now use these classes to build an initial **class diagram**.
- We have not yet identified many attributes or operations; that can come later.
- An important part of the class diagram is the **associations** that exist between objects of a class.
- We can give the **multiplicity** of these associations and also attach a **label** to them to indicate how they relate to each other.

Class Diagram



- Note that as `CounterTeller` can update any kind of account, the association is between `CounterTeller` and the `Account` superclass.
  - As `ATM` can only update `ChequeAccount`, the association is between `ATM` and the `ChequeAccount` subclass.

The Early Stage Modelling Process

- In the early stages of requirement analysis it is best to create a **class diagram** in **parallel** with the **use cases**.
  - The initial class diagram will consist of classes defining objects from the problem domain.
  - The description of the use cases will often refer to such entities; it is therefore important to always use the same name for the same thing.
  - The initial class diagram can be where such names are defined.
- Also, use cases are not themselves object-oriented; relating them to the initial class diagram helps keep our focus on objects.
- Important: **keep all diagrams coherent!**

Requirement Analysis Models

- The classes that we have initially represented in the class diagram all belong to the **real world**.
- In fact object models can be used to model real world situations where there is no intention of creating software.
- However, in this course, we must always bear in mind that the eventual aim is to create a software system (or a mixture of hardware and software).
- In the **requirements analysis phase**, our **main aim is to understand the problem.**

Requirement Analysis Models

- As modelling is **difficult**, we do not want to get bogged down in detail.
- We therefore do not worry initially about whether an object is hardware, software or really part of the environment.
- Once we have created our initial model, we can explicitly attempt the difficult task of determining the boundary between the system and the environment.