UML & Object Modelling

- UML & Object Modelling - Interaction Diagrams
  - Sequence diagrams
  - Examples
  - GUIs
  - Communication diagrams

Interaction Diagrams

- The class diagram shows the **static structure of the system**, but does not show its **dynamic behaviour**, i.e. how objects interact to carry out some task.
- UML has **interaction diagrams** to show how objects can communicate with one another.
- We consider here two kinds of interaction diagram: **sequence diagrams** and **communication diagrams** (previously known as *collaboration diagrams)*.
- Here we show how sequence diagrams can represent **scenarios**, i.e. sequences of events which show a particular system behaviour.

Interaction diagrams

- We can represent an object of class `ATM` as `theATM:ATM`.
- We always underline objects in a diagram to emphasise their difference from classes.
- Typically we want to indicate a *typical object of a class*. This can be represented as `:ATM`.
- We will show the sequence diagram for successful withdrawal of money at an ATM. This use case has already been given.

Interaction Diagrams

- **Sequence diagrams** can be useful to:
  - better understand and specify what may happen in a use case.
  - help us identify the operations offered by each object.
  - help us identify new classes that are required to support the behaviour.
- As interaction diagrams are dealing with **behaviour**, they are presented in terms of **objects** rather than **classes** (interaction happens through method calls between objects).

Sequence Diagrams

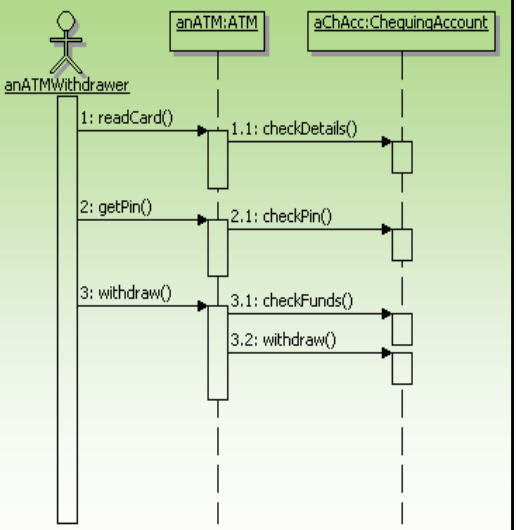
A diagram shows the **objects** that take part in a scenario and the **messages** that are sent between them.

The vertical axis represents **time**, and so the order in which the messages are sent is specified.

Each object has a vertical column; its **lifeline**. While the **object exists**, it is shown by a dashed line.

An arrow **to** an object shows a call of an operation offered by the object. An arrow **from** the object shows a call made by the object.

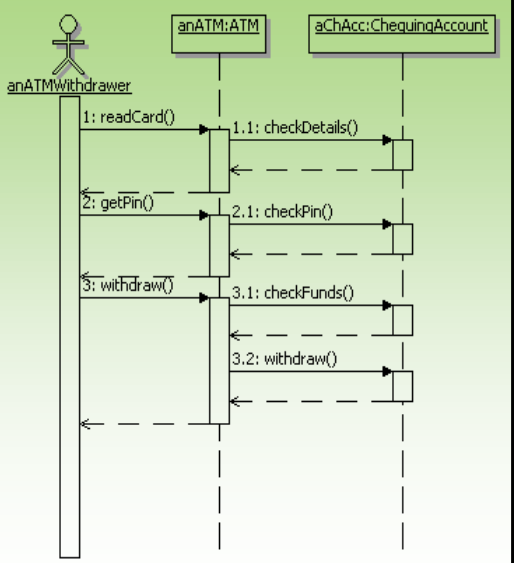
A vertical bar in the lifeline show the **activation of a method** in the object, ie. when the method is active.



Sequence diagrams

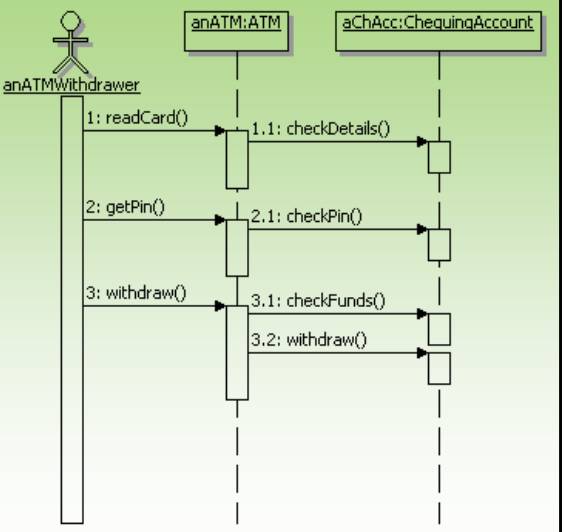
Every method call has a **return** shown as a dashed line. We do not usually show them in a sequence diagram to avoid clutter.

Here we see the previous sequence diagram with the returns added in explicitly.



Sequence diagrams

- **Actors** are represented in a sequence diagram analogously to objects.
- Here, `anATMWithdrawer` is an actor and so does not appear in our class diagram.
- Note that this sequence diagram concentrates on the system as it appears to the actor.
- In the design phase, we are likely to identify more objects within the system and interactions between these objects.

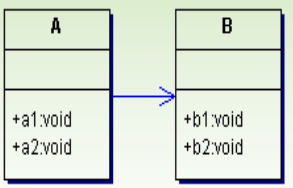


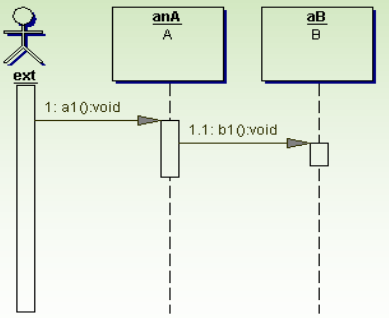
Class versus Sequence Diagrams

- Remember that a class node in a **class diagram** shows the operations **offered** by that class: a **static view** of the system.
- It does **not** show the operations **called** by objects of the class.
- We show object **calls** in a **sequence diagram**. It exposes the **dynamic behaviour** of the system.

Class versus Sequence Diagrams

- The class diagram shows that an `A` object offers operations `a1` and `a2` while `B` offers `b1` and `b2` (and `A` "knows about" `B`).
- The sequence diagram shows that a result of calling `a1` (by the external actor) is a call of the operation `b1` offered by a `B` object.





Corresponding outline Java code

The corresponding outline Java classes are:

```
class A
{
    ...
    private B aB = new B();
    ...
    public void a1(...) {
       ... aB.b1(...) ...
    }
    ...
}

class B
{
    ...
    public void b1(...) { ... }
    ...
}
```

Class & Sequence Diagrams

- A potential problem with sequence diagrams is that we need to create a large number to show the dynamic behaviour of a system.
  - Typically one sequence diagram per use case.
  - Hard to avoid!
- In contrast to the class diagram where one diagram can show the entire static structure (but it doesn't really tell us how the system works, just what is possible).

The Development Process

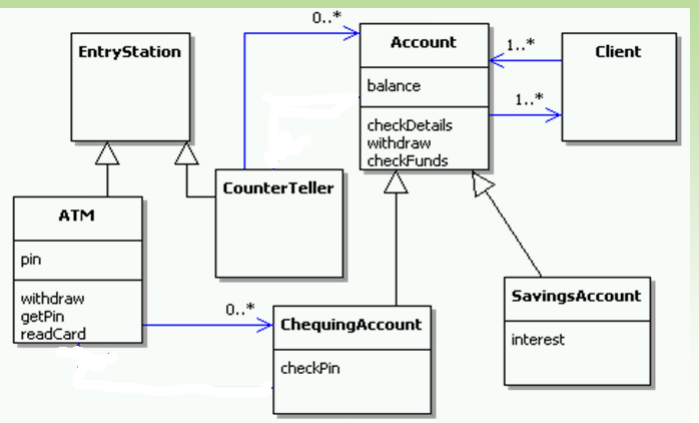
- In the development process, we create our **use case** models and **initial class diagram** in parallel.
- Use cases are initially described in **natural language**.
- The next step is to create **sequence diagrams** to show the scenarios that can result from the use cases.
  - Sequence diagrams are initially developed from use cases.
  - Initially, we create sequence diagrams for standard correct behaviour.
  - Later, we can deal with **error situations**.

Sequence Diagrams

- The advantage of sequence diagrams is that their creation enables us to:
  - determine **new classes** that are required to support a scenario
  - identify the **operations** that each class must offer
  - identify the **other objects** that need to carry out these operations
  - determine **navigability**, e.g. the direction in which messages are sent from one object to another.

Class diagram

- Using the information from the sequence diagram, we can expand the class diagram as follows:



Determining Properties of a Class

- When determining the attributes and operations of a class, it can be useful to take an **anthropomorphic** view:
  - Pretend **you** are the class and determine your attributes, the operations you can offer and the other objects you need to collaborate with in order to carry out your operations.
- To put it another way:
  - **You identify the responsibilities of each class and the operations required to carry out each responsibility.**

Core OO Design Principles

- **Abstraction**
  - only provide relevant information
- **Modularity/decomposition**
  - clearly identifiable sub-problems "connected" through clear "interfaces"
- **Encapsulation/information hiding**
  - information encapsulated in objects
  - accessible through clearly specified protocols
  - hide implementation details

Core OO Design Principles

- How **many classes, how big should a class be?**
  - **Monolithic system**, only one class:
    - too many responsibilities
    - no good decomposition/modularity
      - split into smaller classes.
- **Fragmented system**, too many classes?
  - trivial classes
  - no good decomposition/abstraction/efficiency?
- Generally a class should have one clearly defined job to do and it should do it well.

CRC Cards

- As a side note: There are other "formalisms" that may help determine the properties of classes.
  - See, for instance, **CRC** cards [**Class, Responsibilities** and **Collaborations**].
- On each card, you write:
  - the name of the class,
  - the responsibilities of the class
    - a brief description of the purpose of the class,
  - the collaborators of the class - the other classes that are needed for this class to carry out its responsibilities.
- A class will have an association with each of its collaborators. CRC cards can be used to **collect** together **information** from the different sequence diagrams.

GUIs

- A major question is to decide what system we are actually designing. Does it include the GUI or is it the underlying software system?
  - Object-oriented modelling is primarily concerned with modelling the underlying system.
  - We will often define one or more **boundary classes** to handle information that has to be passed between the environment and the system.
  - However, we will be concerned with the **information** to be passed, **not** with **graphical components** such as text boxes or buttons.

GUIs

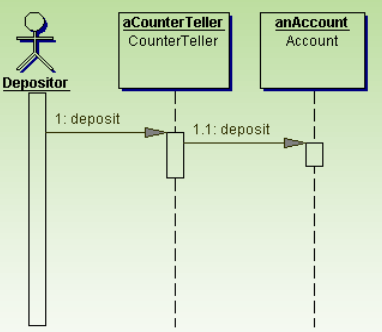
- The GUI will be built as part of the **implementation**.
  - By separating the GUI from the underlying system, we make the final product more resilient to change.
  - It can be given different (graphical or other) interfaces which will work with different hardware.
  - As hardware changes, we need only change the GUI, not the underlying system design.
- Some modellers do not explicitly include boundary classes in their models, but they can generally help us understand how information is passed between the environment and the system.

Example: Depositing Cash

- Let us now consider another use case: **deposit cash with counter teller**
  - *A customer gives cash to a counter teller.*
  - *The counter teller enters the amount and the account number.*
  - *The system updates the account and issues a receipt.*

Depositing Cash

- If a use case is very simple, we do not have to create a sequence diagram.
- It could be argued that is the case here, it depends on your level of **experience**.
- Note the use of `anAccount` where `Account` is an abstract class.
- What we are modelling is that `anAccount` represents an instance of any of the subclasses of `Account`.



Example: Depositing a Cheque

To show how sequence diagrams can help us identify classes and operations, consider a more complex use case:

deposit cheque with counter teller

- A customer deposits a cheque with a counter teller.
- The systems reads the information on a cheque and updates the appropriate chequing account. However, the system does not confirm the updating of the account until the cheque has been cleared, i.e. until the system has checked that there are sufficient funds in the account on which the cheque is drawn.
- This may involve the system sending a message to another bank.

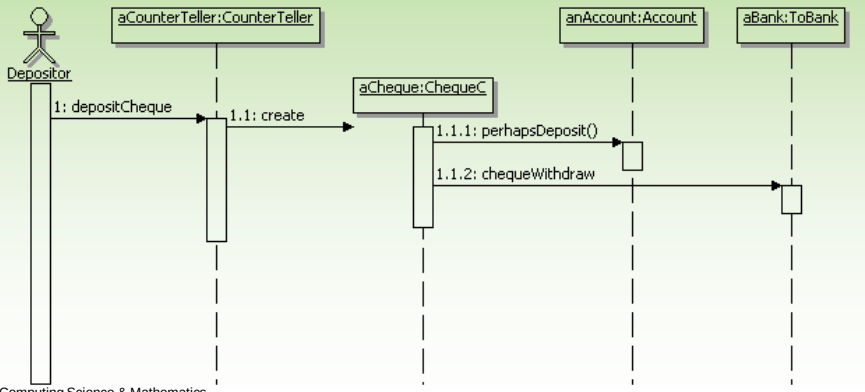
Example: Depositing a Cheque

- This use case is therefore quite complex. To help us understand what is required, we need to create a sequence diagram.
  - Initially we will omit parameters from the operations; our aim is to get the **structure** right and we do not want to get bogged down in detail.
  - It seems likely that we will need a `cheque object`, but what exactly will it be like?
  - Will it be an entity object that merely contains information about the cheque?

Deposit Cheque : Sequence Diagram

An alternative view is that it will be a **control object** whose purpose is to co-ordinate the actions needed to deposit a cheque.

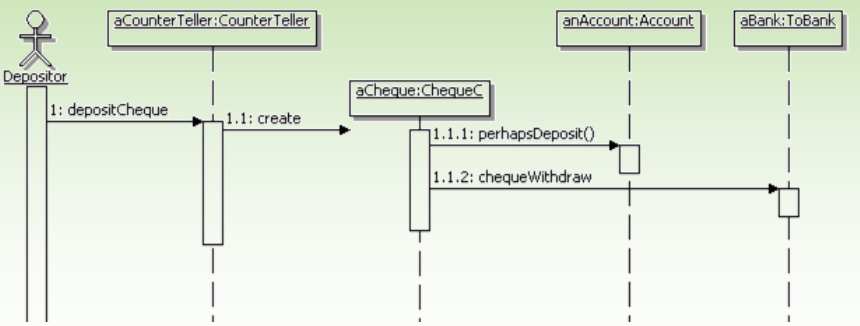
There are a couple of new features introduced in this diagram.



Deposit Cheque : Sequence Diagram

A new `Cheque` object is being created and so we put its object box at the point where it is created. The purpose of the `ChequeC` object is **to coordinate** activity:

- to communicate with the `Account` object and
- to send a message to another bank via the `ToBank` boundary class
- and finally to confirm the deposit.



Deposit Cheque : Sequence Diagram

Note that in creating this use case and associated sequence diagram, we have identified **new classes**:

- the **control class** `Cheque`
- the **boundary class** `ToBank` through which we can check that the other account has sufficient funds.

We also identify **operations** such as

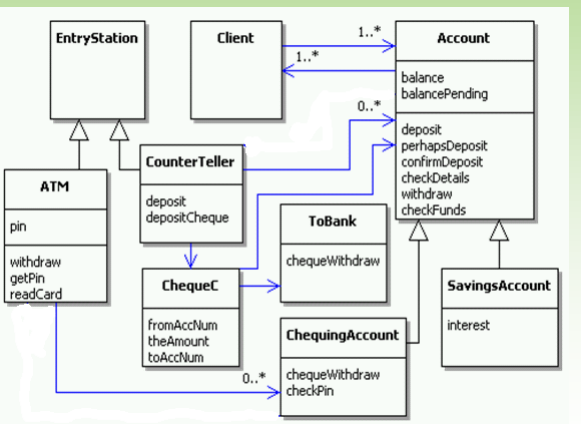
- `perhapsDeposit` and `confirmDeposit`

Note that `Account` will need to have **an extra attribute** to hold the amount deposited by cheques, but which have yet to be confirmed.

Class Diagram

We can now add the information from these sequence diagrams to expand the class diagram. Note how we are **incrementally** adding to our understanding of the problem and in parallel, dealing with:

- use cases
- sequence diagrams
- the class diagram



Operations

- Many authors do not show attributes and operations at an early stage in the development of the class diagram.
- In disagreement with that approach, other authors feel that one only really understands what a class does, when its main attributes and operations have been identified .
- The list does not have to be complete; we can always add more attributes and operation as our knowledge increases.

Communication Diagrams

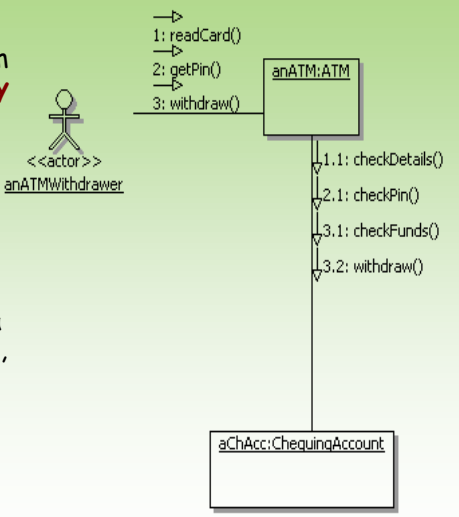
As an alternative to sequence diagrams, we can use **communication diagrams** (previously known as collaboration diagrams).

- The objects that interact to perform some task, together with the links between them, are known as a **collaboration**.
  - Remember that a Class diagram contains classes. A communication diagram is like **part** of a class diagram in which we represent **objects** rather than classes.
  - Communication diagrams show how **instances** of some of the classes in a class diagram **collaborate** with each other to carry out some task.
  - Tools enable sequence and communication diagrams to be converted from one to the other.

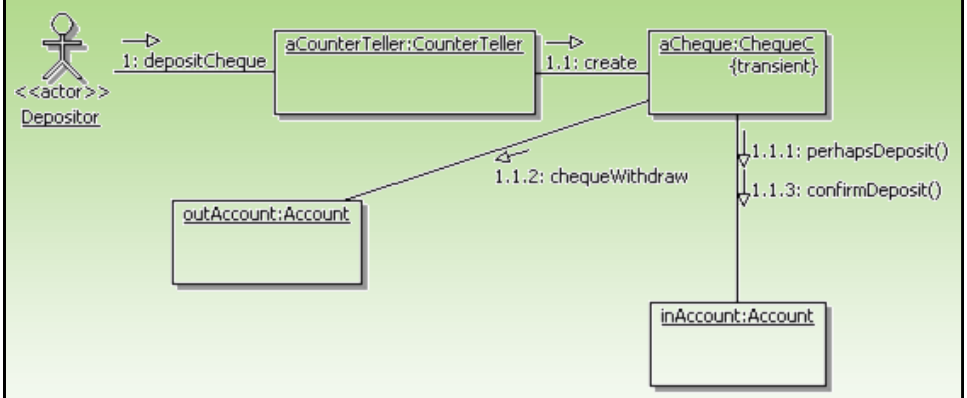
Communication diagrams

We show the communication diagram for **successful withdrawal of money from an ATM**.

- The ordering of the operation calls is shown by giving them **sequence numbers**.
- When an object receives a message with number `x`, the messages the object sends as a consequence are numbered `x.1`, `x.2` etc.



Communication Diagrams



This is a communication diagram for **depositing a cheque**.

Analysis and Design Models

- UML models are used in different phases of software development.
- Remember that in the requirements analysis phase our aim is to understand the problem and we do that by determining the behaviour that the actors want the system to provide.
- As we move to more detailed analysis, and then design, our view of actors and use cases can change.

Analysis and Design Models

Let us look at this through another example.

We identified a `Depositor` as an actor. The `Depositor` interacted with a `Counter Teller`.

- In our initial analysis model, we are not concerned with computers and so we can regard a human `Counter Teller` as part of the `Bank` system with which the `Depositor` interacts.
- As we go into more detail, we need to determine the **boundary** between the eventual system that we are to create and its environment.

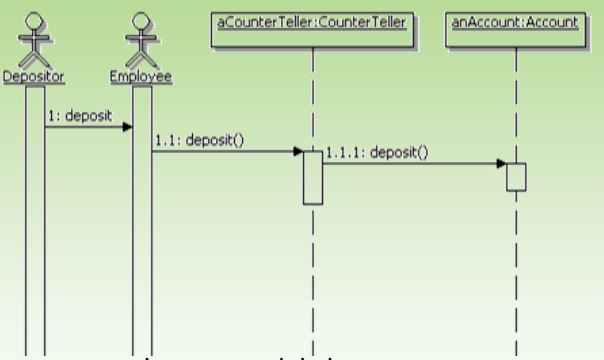
Analysis and Design Models

- More detailed analysis shows that `Depositor` interacts with a bank employee (a human `Counter Teller`) who interacts with a `Counter Teller` machine.
- To help understand the problem, we can model the interaction between `Depositor` and a human `Counter Teller`, but we must realise that this interaction is between entities in the environment and is not part of the system that we are to design and implement.
- This also means that some of our actors do not directly interact with the computer system, but interact with other actors.

Analysis and Design Models

Our sequence diagram for **`deposit cash with counter teller`** might now become:

We are now explicitly regarding `Counter Teller` as a machine through which the system interacts with the environment.



However, it is important to note that we model the `CounterTeller` machine to be the same as the human `CounterTeller` that we had earlier.

Design Models

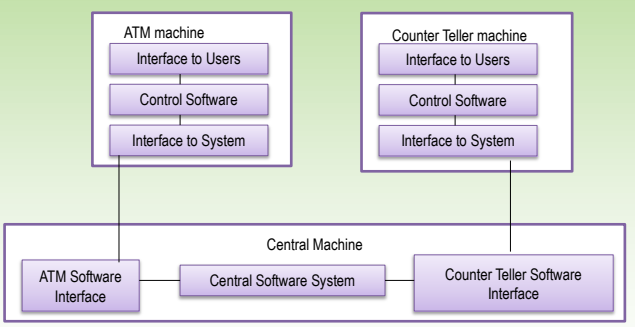
In the design phase, we move from **understanding the problem** to **modelling a solution**. Hence, deciding on the division between the system and the environment is now very important. We must also decide on the precise system that is to be created.

Consider our `Bank` system:

- The `ATM` and the `CounterTeller` machine are physical devices separate from our central computer.
- If our task is to design a software system that will run on the central computer then the `ATM` and the `CounterTeller` machine should be regarded as actors that interact with our system.

Design Models

- Our overall software structure is now



Design models

Our Class diagram is unchanged.

- The classes that we previously defined for `ATM` and `CounterTeller` now represent boundary classes on the central machine.
- **Boundary objects** therefore give us great flexibility.
  - As they only pass information and do no processing, they can occur virtually unchanged in models at different levels of abstraction.