

Applied Programming and Design Principles



**Object Oriented
Programming using C#**

Lecture 6

The Unified Modelling Language (UML) - Part 2

Lesson's objectives

- UML Class Diagrams
- Denoting Relationships
- Types of Association

An Introduction to UML

- This session will introduce you to the roles of the Unified Modelling Language (UML) and explain the purpose of four of the most common diagrams (**class diagrams, object diagrams, sequence diagrams and package diagrams**)
- These diagrams are used extensively when describing software designed according to the object-oriented programming approach

An Introduction to UML

- In UML 2, there are two basic categories of diagrams: **structure diagrams** and **behavior diagrams**
- Every UML diagram belongs to one these two diagram categories
- The purpose of **structure diagrams** is to show the **static structure** of the system being modeled. They include the class, component, and or object diagrams
- **Behavioral diagrams** show the **dynamic behavior** between the objects in the system, including things like their methods, collaborations, and activities

An Introduction to UML

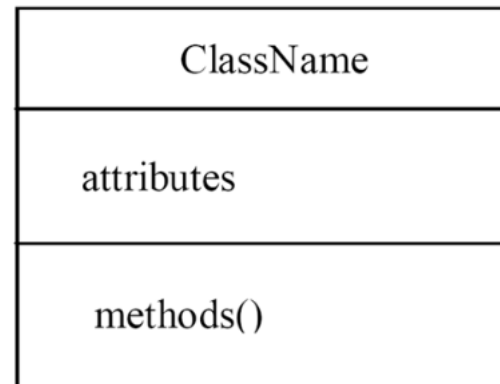
- It is, on the other hand, a **precise diagramming notation** that will allow **program designs to be represented and discussed**
- The diagrams represent technical information they **must be precise and clear** – in order for them to work - therefore there is a **precise notation** that must be followed

UML Class diagrams

- Classes are the basic components of **any object oriented** software system and UML class diagrams provide an easy way to represent these
- Thus a class diagram shows the **architecture of a system**

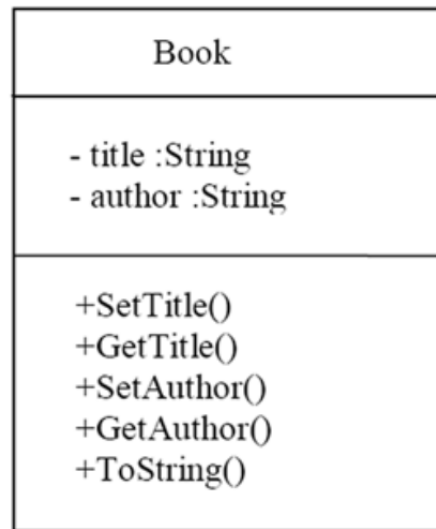
A class consists of

- A unique name (conventionally starting with an uppercase letter)
- A list of attributes (int, double, boolean, String etc)
- A list of methods



Class diagram

- For attributes and methods visibility modifiers are shown (+ for public access, – for private access).
- Attributes are normally kept private and methods are normally made public.



Exercise

- Draw a diagram to represent a class called 'BankAccount' with the attribute balance (of type int) and methods DepositMoney(), WithdrawMoney() and DisplayBalance(). Show appropriate visibility modifiers.

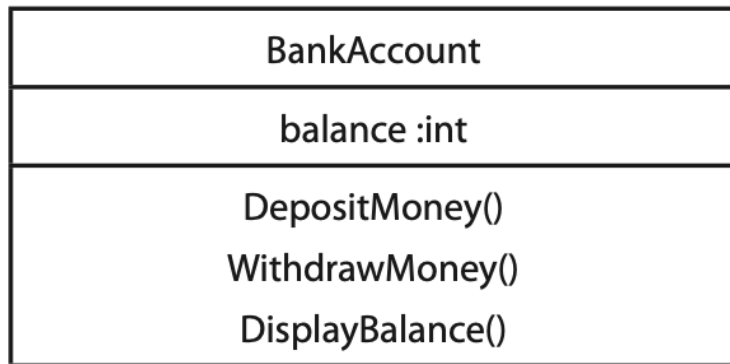
Solution

BankAccount
- balance :int
+DepositMoney() +WithdrawMoney() +DisplayBalance()

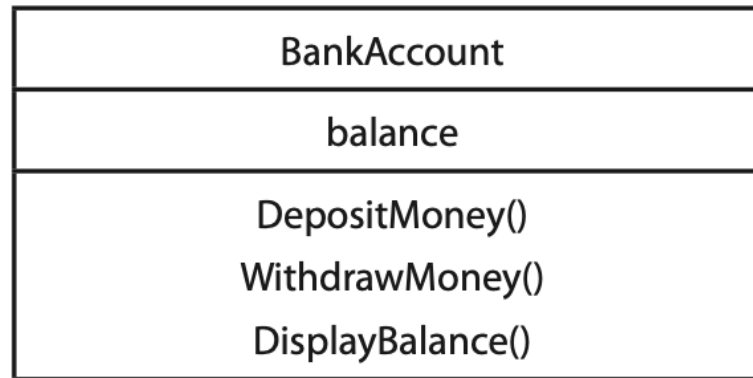
Solution

- UML allows us to suppress any information we do not wish to highlight in our diagrams
- This allows us to suppress irrelevant detail and bring to the readers attention just the information we wish to focus on.

The following are all valid class diagrams



The access modifiers not shown



Access modifiers and the data types not shown



The attributes and methods not shown

Relationships

- Some classes will make use of other classes.
- These relationships are shown by arrows
- Different type of arrow indicate different relationships (including inheritance and aggregation relationships).

Generalization / specialization

- Inheritance
- Interfaces

Relationships

- Navigability
- Multiplicity
- Dependency
- Aggregation
- Composition

Exercise

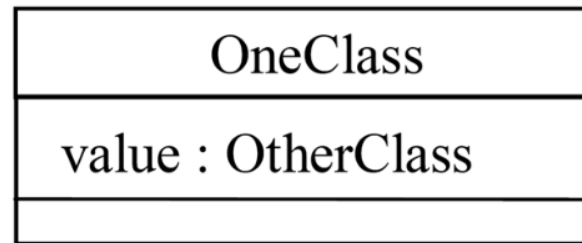
- Draw a diagram to represent a class called 'BankAccount' with a private attribute balance (this being a single integer) and a public method DepositMoney() which takes an integer parameter, 'deposit' and returns a boolean value. Fully specify all of this information on a UML class diagram.

Solution

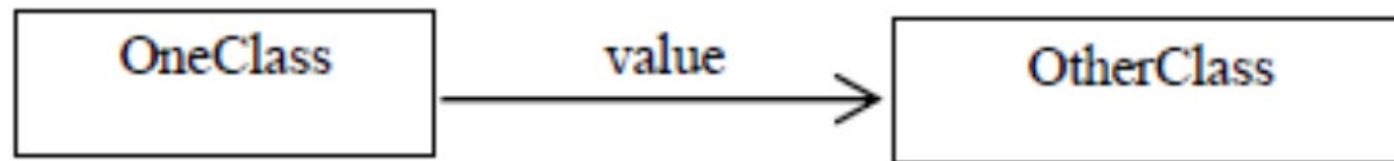
BankAccount
- balance :int[1]
+DepositMoney(deposit :int) : boolean

Denoting Relationships

- The figure above shows a class 'OneClass' that has an attribute 'value'.
- We use **an association** when we want to give two **related classes**, and their relationship, prominence on a class diagram

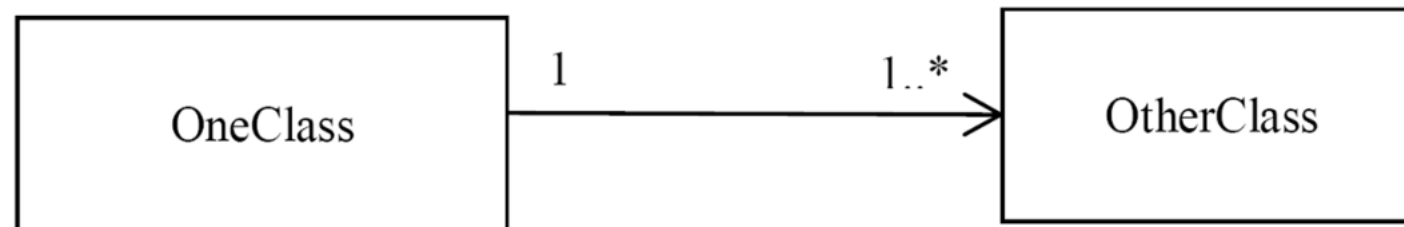


- We could denote exactly the same information by the diagram below.



Multiplicity at both ends of an association

- This implies that 'OneClass' maintains a collection of objects of type 'OtherClass'.



Exercise

Draw a diagram to represent a class called 'Catalogue' and a class called 'ItemForSale' as defined below :-

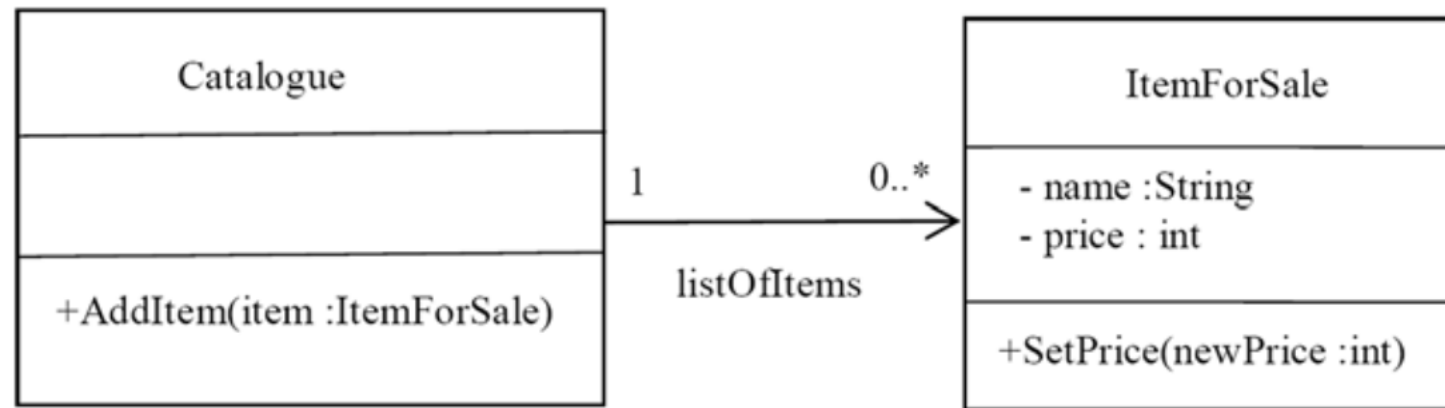
ItemForSale has an attribute 'name' of type String and an attribute 'price' of type int. It also has a method SetPrice() which takes an integer parameter 'newPrice'.

'Catalogue' has an attribute 'listOfItems' i.e. the items currently held in the catalogue. As zero or more items can be stored in the catalogue 'listOfItems' will need to be an array or collection. 'Catalogue' also has one method AddItem() which takes an 'item' as a parameter (of type ItemForSale) and adds this item to the 'listOfItems'.

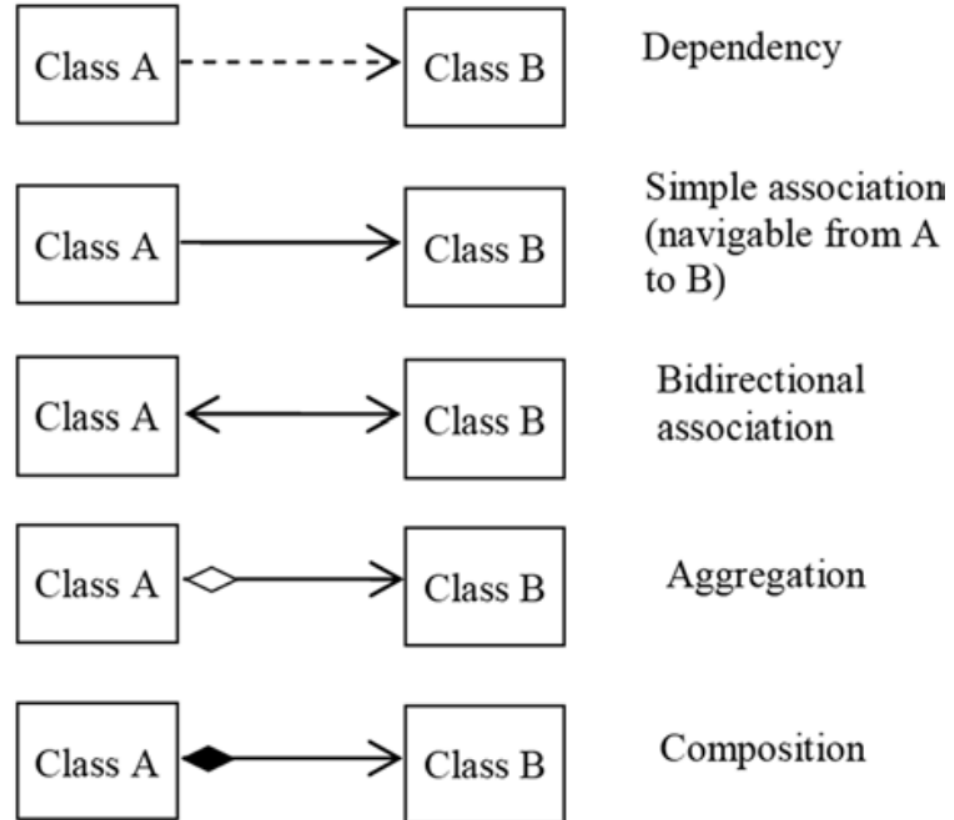
Draw this on a class diagram showing appropriate visibility modifiers for attributes and methods.

Solution

- The class ItemForSale describes a single item (not multiple items).
- 'listOfItems' however maintains a list of zero or more individual objects.

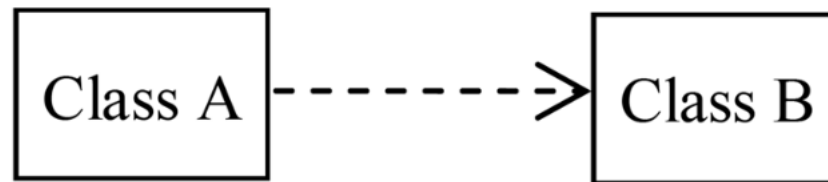


Types of Association



Dependency

- Dependency is the most unspecific relationship between classes (not strictly an „association’)
- Class A in some way uses facilities defined by Class B
- Changes to Class B may affect Class A

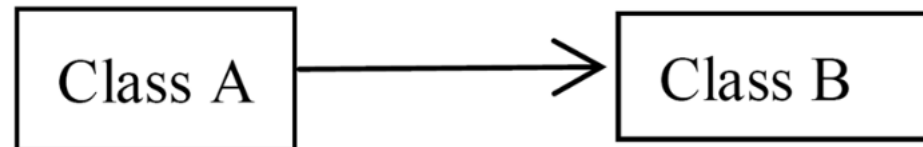


Dependency

- Typical use of dependency lines would be where Class A has a method which is passed a parameter object of Class B
- Or uses a local variable of that class, or calls 'static' methods in Class B.

Simple Association

- Typically Class A has an attribute of Class B
- Navigability is from A to B:
- A simple association typically corresponds to an instance variable in Class A of the target class B type.

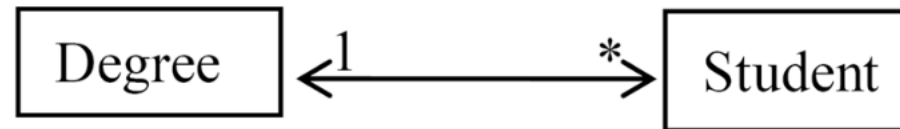


Bidirectional Association

- Bidirectional Association is when Classes A and B have a two-way association
- A bidirectional association is complicated because each object must have a reference to the other object(s).

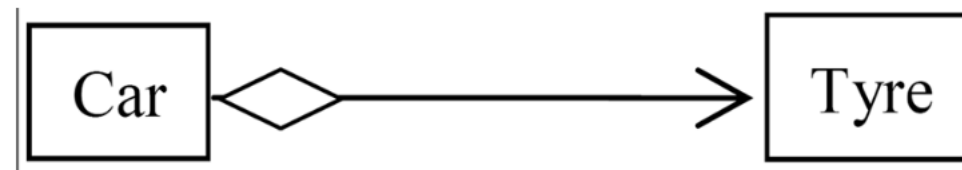
Bidirectional Association

- An example of a bidirectional association may be between a 'Degree' and 'Student'.
- A Degree we may wish to know which Students are studying on that Degree
- Alternatively starting with a student we may wish to know the Degree they are studying.
- As many students study the same Degree at the same time, but students usually only study one Degree



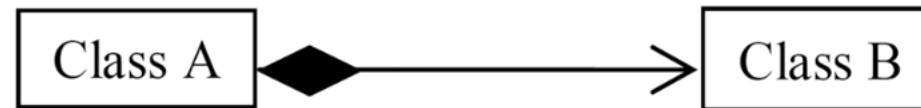
Aggregation

- Aggregation denotes a situation where Object(s) of Class B 'belong to' Class A
- Some designers believe there is **no real distinction between aggregation and simple association**



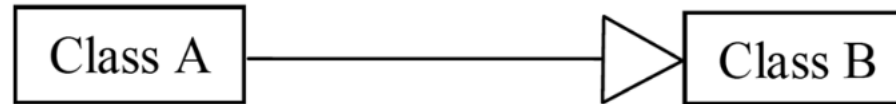
Composition

- Much '**stronger**' than aggregation in this case Class B objects are an integral part of Class A and in general objects of Class B never exist other than as part of Class A, i.e. **they have the same 'lifetime'**



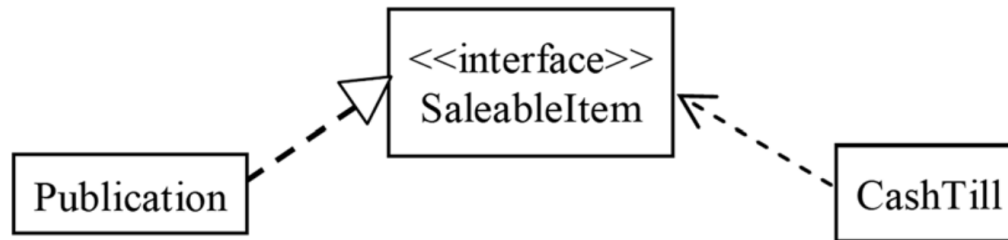
Inheritance

- Class A 'inherits' both the interface and implementation of Class B, though it may override implementation details and supplement both.

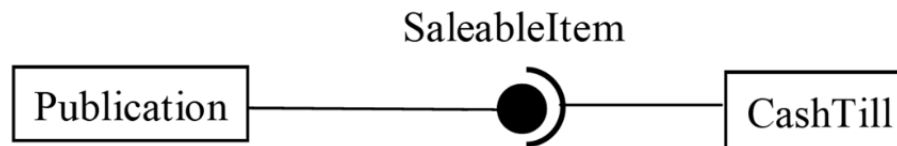


Interfaces

- Interfaces can be represented using the <<interface>> keyword



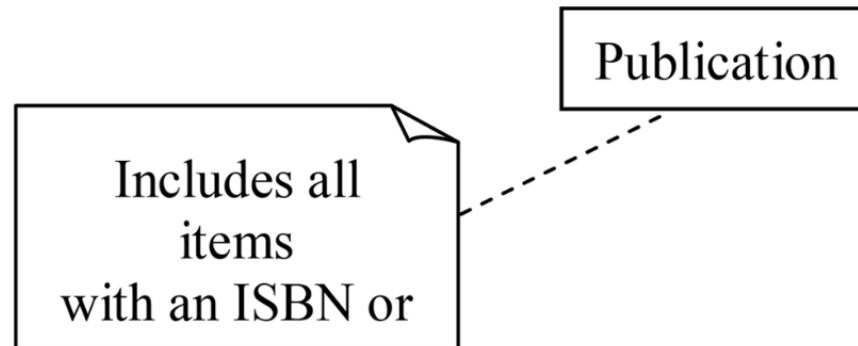
- There is also a shorthand for this



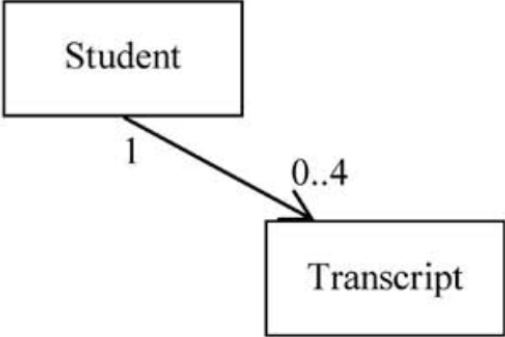
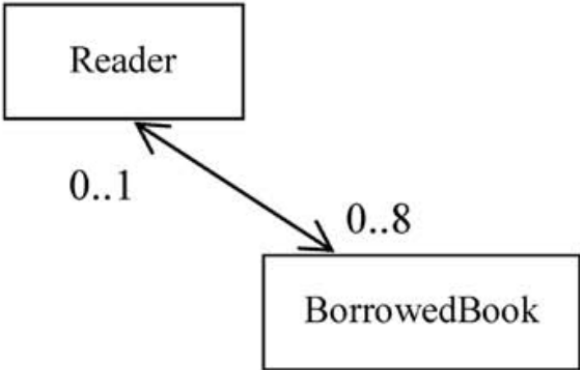
- In both cases these examples denote that the SaleableItem interface is **required by CashTill** and **implemented by Publication**

Notes

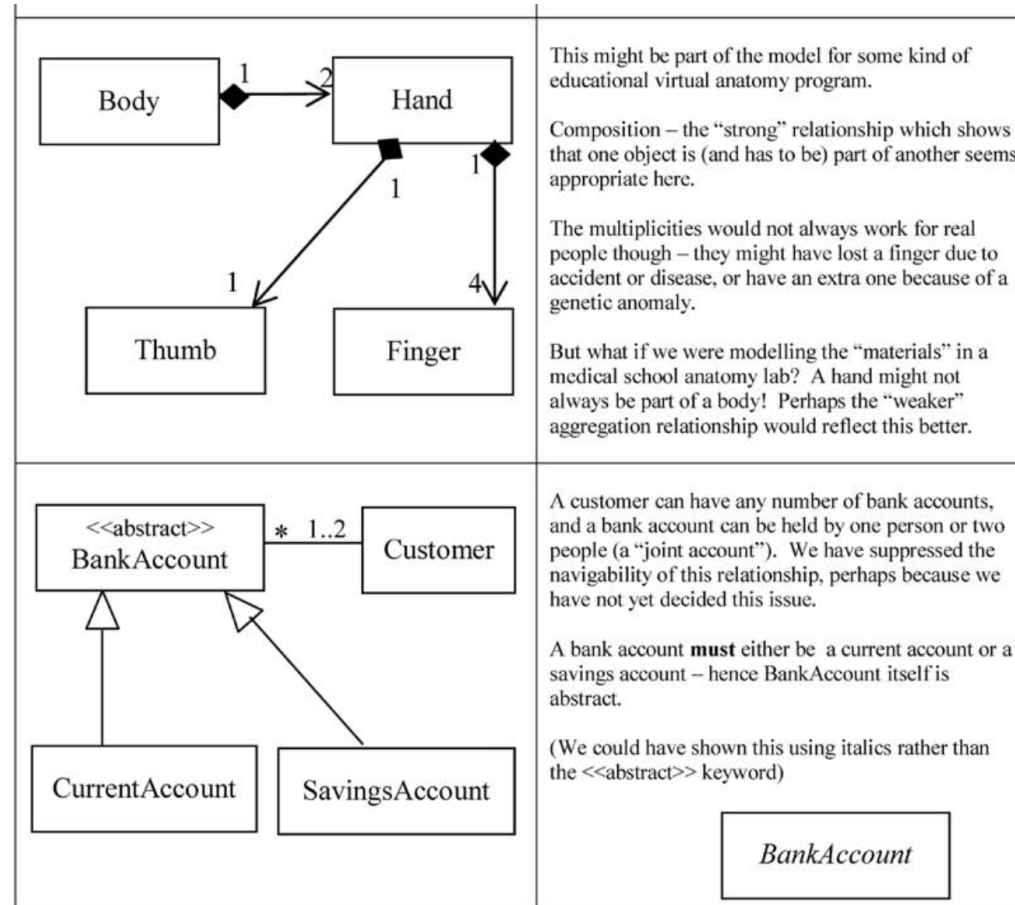
- We can add notes to comment on a diagram element. This gives us a 'catch all' facility for adding information not conveyed by the graphical notation



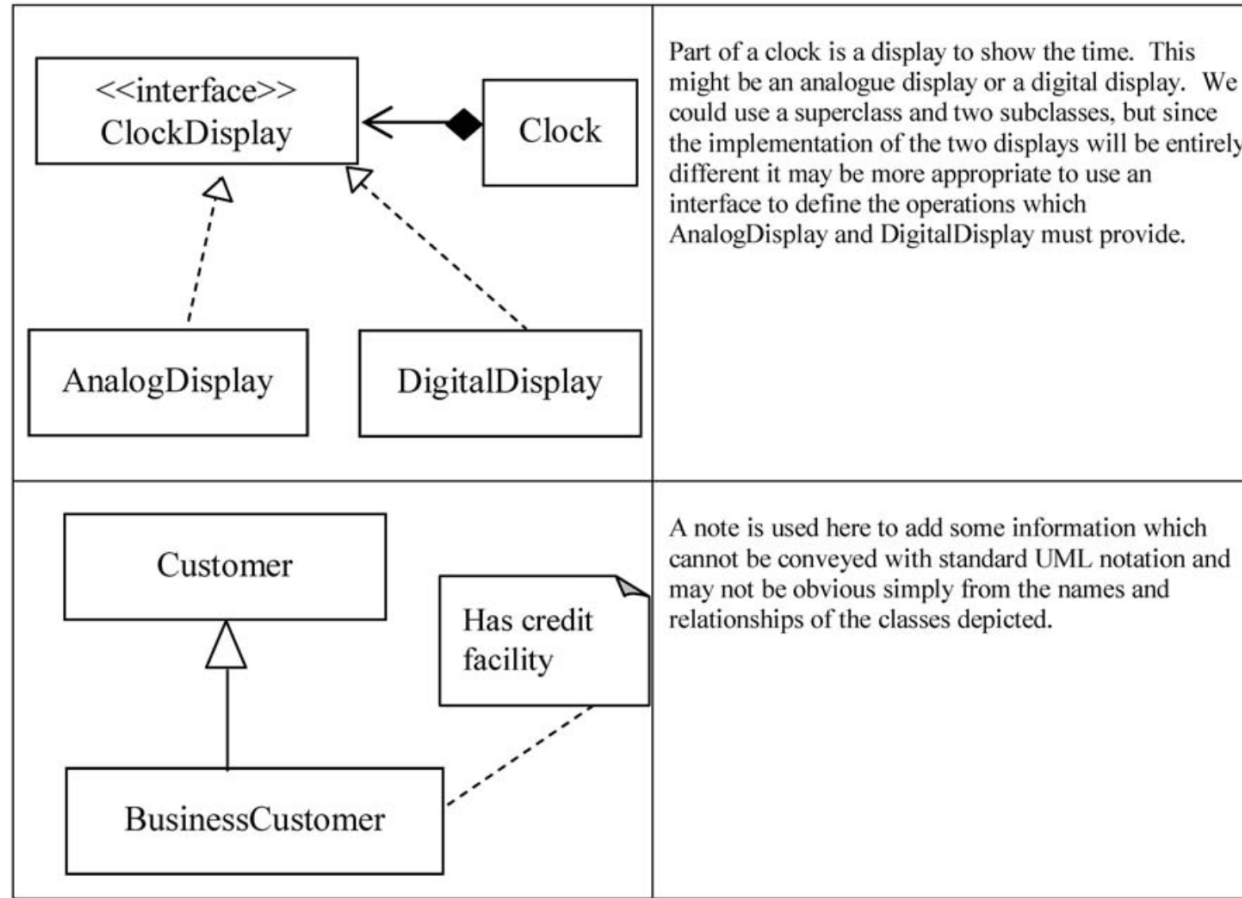
Some examples

 <pre>classDiagram Student "1" -- "0..4" Transcript</pre>	<p>In a University administration system we might produce a transcript of results for each year the student has studied (including a possible placement year).</p> <p>This association relationship is naturally unidirectional – given a student we might want to find their transcript(s), but it seems unlikely that we would have a transcript and need to find the student to whom it belonged.</p>
 <pre>classDiagram Reader "0..1" <--> "0..8" BorrowedBook</pre>	<p>In a library a reader can borrow up to eight books. A particular book can be borrowed by at most one reader.</p> <p>We might want a bidirectional relationship as shown here because, in addition to being able to identify all the books which a particular reader has borrowed, we might want to find the reader who has borrowed a particular book (for example to recall it in the event of a reservation).</p>

Some examples



Some examples



Summary

- This session covered the overview of UML's diagrams
- We discussed about class diagrams
- Then we learned how to use relationships in a class diagram

References

- *Object Oriented Programming using C#*. 1st edn (2011). Ventus Publishing ApS.