Chapter 4

Object oriented Programming

Chapter 4
Object oriented Programming

System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.

Different models present the system from different perspectives

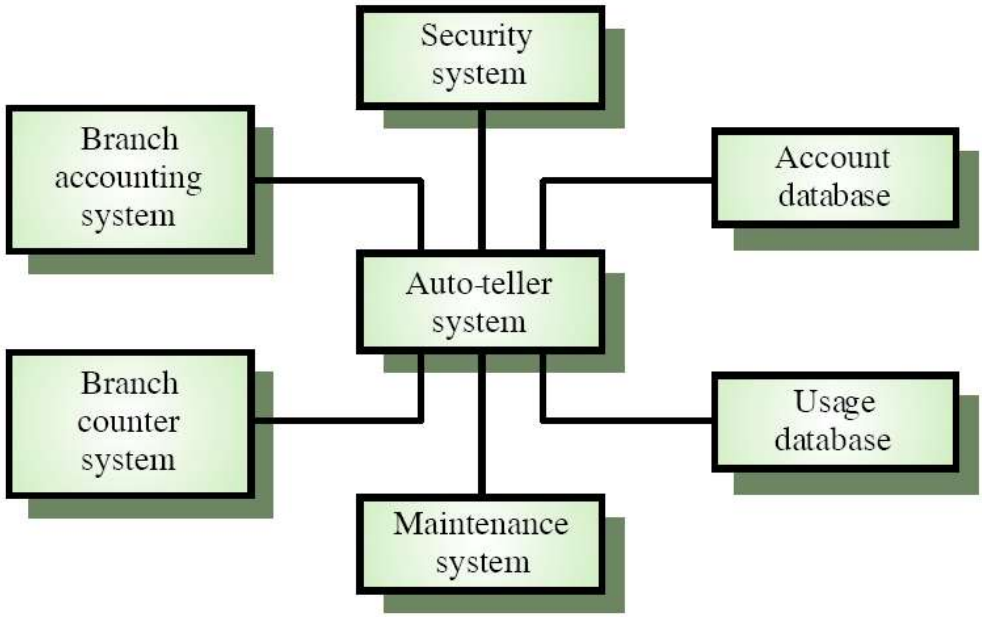
- **Context models** are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- **Behavioural models** are used to describe the overall behaviour of a system.
- **Data models** Used to describe the logical structure of data processed by the system.
- **Object models** describe the system in terms of object classes and their associations.

[1] Context models

Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.

Social & organisational concerns may affect decision on where to position system boundaries. Architectural models show the system and its relationship with other systems.

Example: The context of an ATM system



[2] Behavioural models

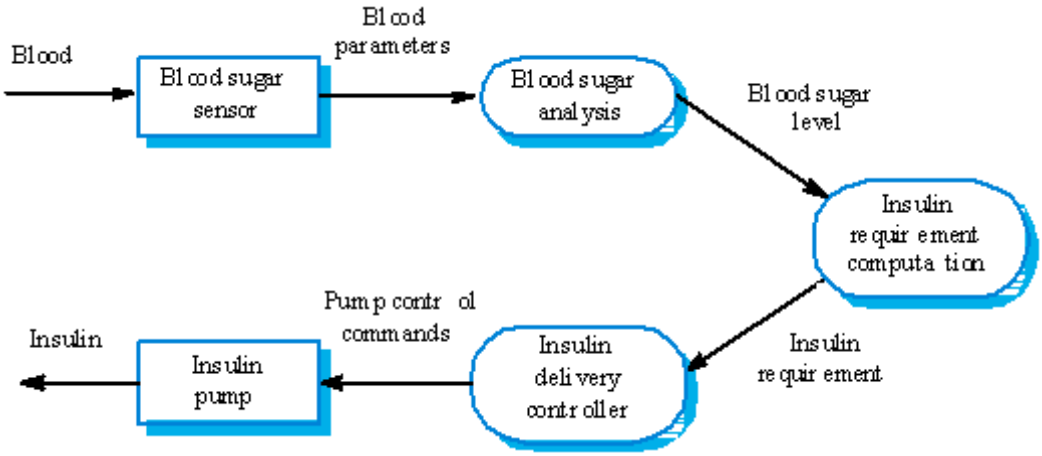
Behavioural models are used to describe the overall behaviour of a system. Two types of behavioural model are:

- **Data processing models** that show how data is processed as it moves through the system;
- **State machine models** that show the systems response to events.

[2.1] Data-processing models

Data flow diagrams (DFDs) may be used to model the system's data processing. These show the processing steps as data flows through a system. DFDs are an intrinsic part of many analysis methods. Simple and intuitive notation that customers can understand. Show end-to-end processing of data.

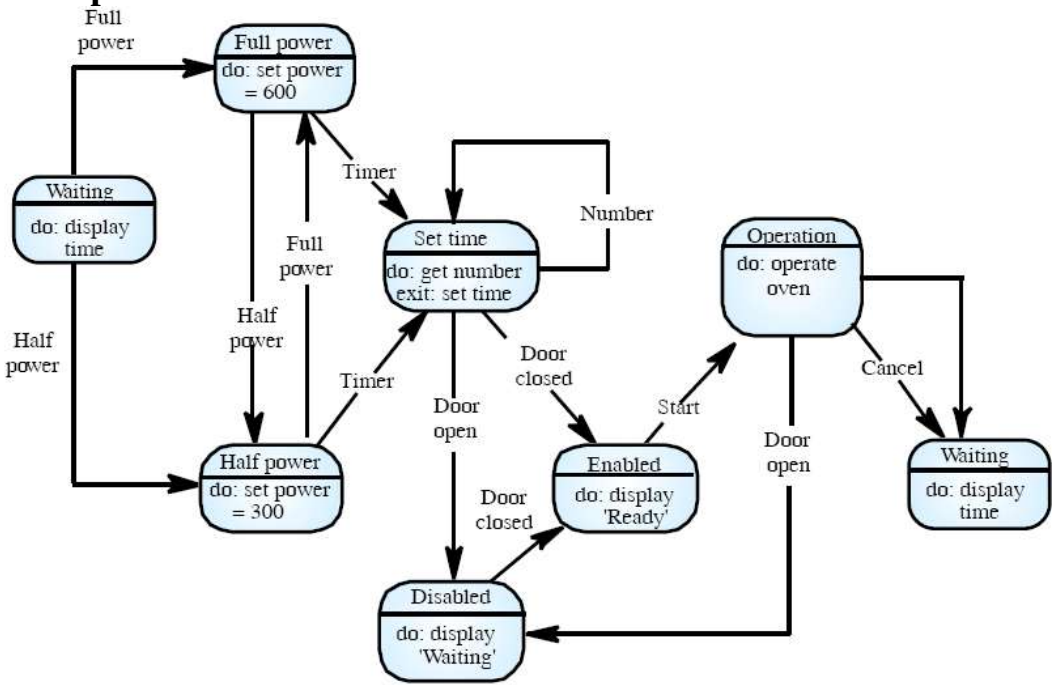
example: Insulin pump DFD



[2.2] State machine models

These model the behaviour of the system in response to external and internal events. They show system's responses to stimuli so are often used for modelling real-time systems. State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.

Example : Microwave oven model



Microwave oven state description

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	Cooking time is set to user's input. The display shows the cooking time selected
Disabled	Oven operation is disabled for safety. Interior light is on. Display 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.

Microwave oven stimuli

Stimulus	Description
Half power	The user has pressed the half power button
Full power	The user has pressed the full power button
Timer	The user has pressed one of the timer buttons
Number	The user has pressed a numeric key
Door open	The oven door switch is not closed
Door closed	The oven door switch is closed
Start	The user has pressed the start button
Cancel	The user has pressed the cancel button

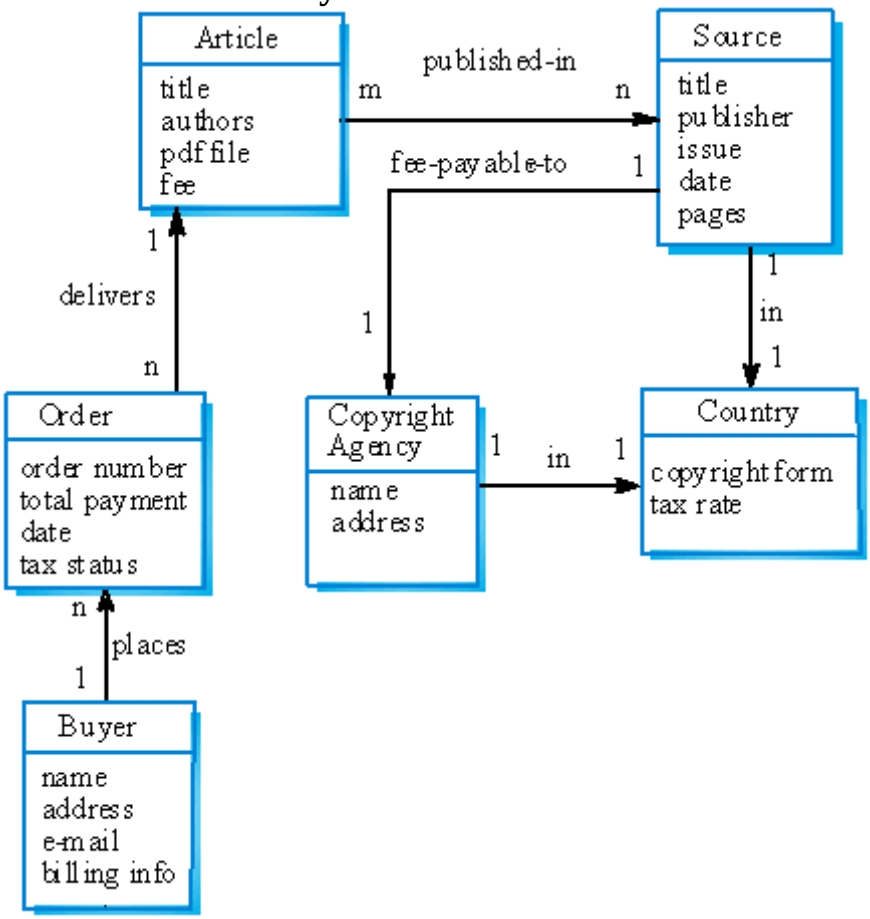
[3] Data models

Used to describe the logical structure of data processed by the system. An **entity-relation-attribute (ERA)** model sets out the entities in the system, the relationships between these entities and the entity attributes Widely used in database design. Can readily be implemented using relational databases.

Advantages

- Support name management and avoid duplication;
- Store of organisational knowledge linking analysis, design and implementation;

EXAMPLE: Library semantic model



Data dictionary entries

Name	Description	Type
Article	Details of the published article that may be ordered by people using LIBSYS.	Entity
authors	The names of the authors of the article who may be due a share of the fee.	Attribute
Buyer	The person or organisation that orders a copy of the article.	Entity
fee_payable_to	A 1:1 relationship between Article and the Copyright Agency who should be paid the copyright fee.	Relation
Address (Buyer)	The address of the buyer. This is used to any paper billing information that is required.	Attribute

[4] Object models

Object models describe the system in terms of object classes and their associations. An object **class** is an abstraction over a set of objects with common attributes and the services (operations) provided by each object. Various object models may be produced

- Inheritance models;
- Aggregation models;
- Interaction models.

ADVANTAGES

- Natural ways of reflecting the real-world entities manipulated by the system
- object classes reflecting domain entities are reusable across systems

DISADVANTAGES

- object class identification is difficult process requiring understanding of application domain

Object models and the UML

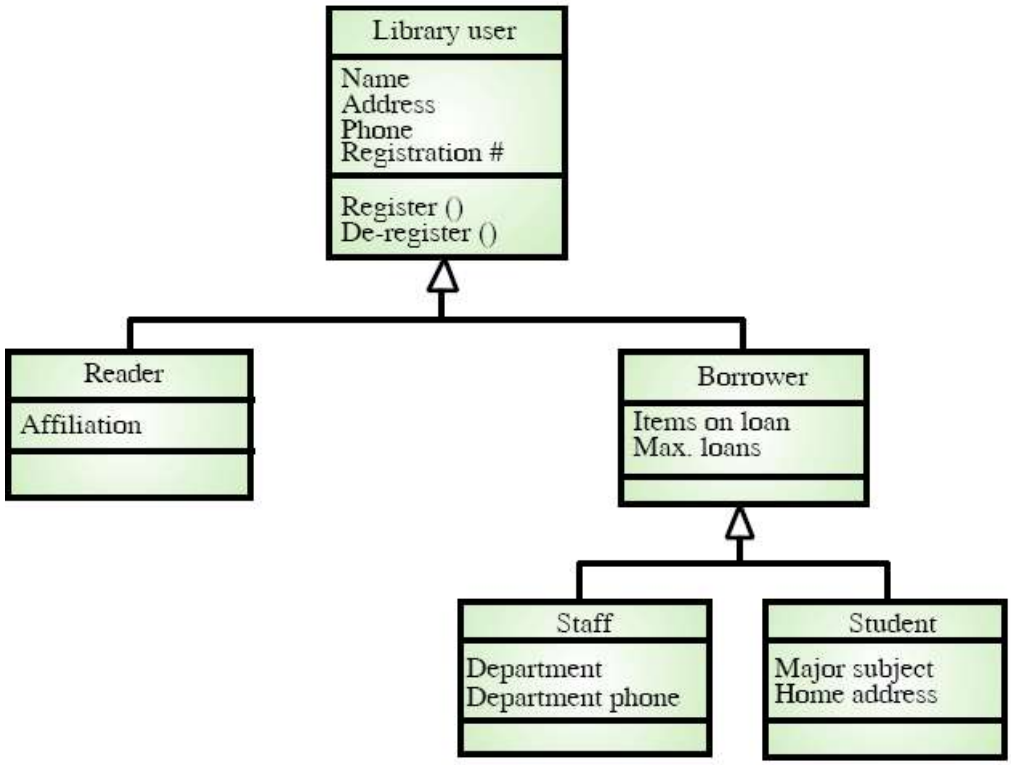
The UML (unified modelling language) is a standard representation devised by the developers of widely used object-oriented analysis and design methods.

It has become an effective standard for object-oriented modelling. Object **classes** are rectangles with the name at the top, attributes in the middle section and operations in the bottom section;

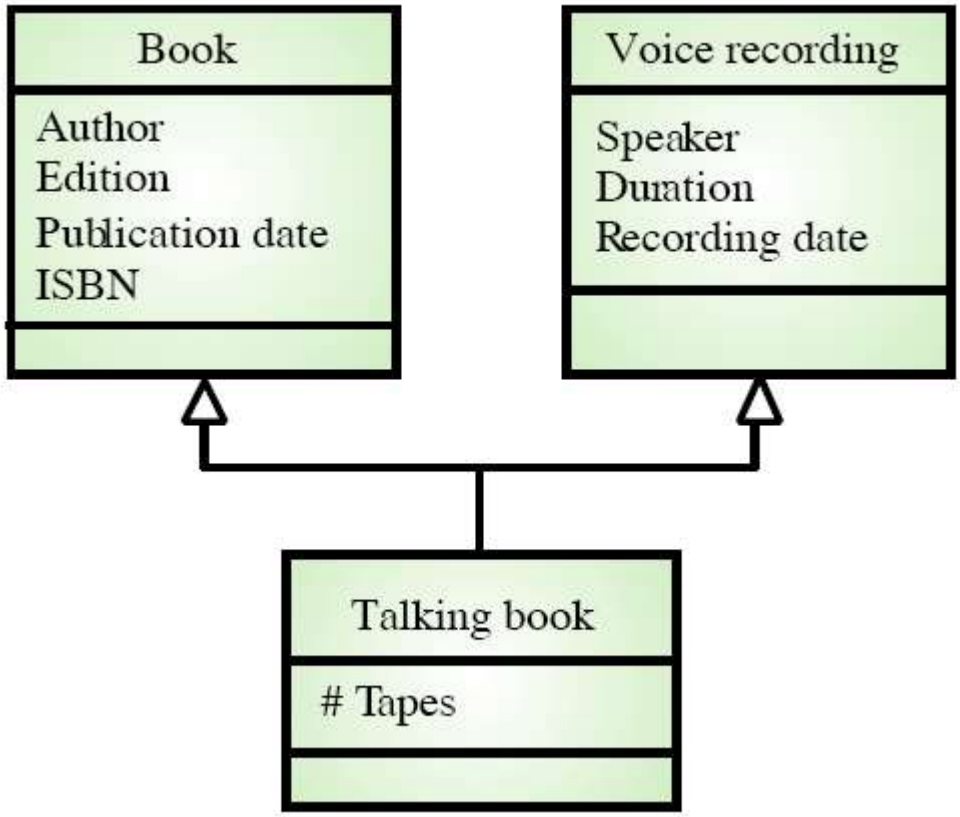
[4.1] Inheritance models =generalisation = is-a relationship

Organise the domain object classes into a hierarchy. Classes at the top of the hierarchy reflect the common features of all classes. Object classes inherit their attributes and services from one or more super-classes. these may then be specialised as necessary.

example: User class hierarchy

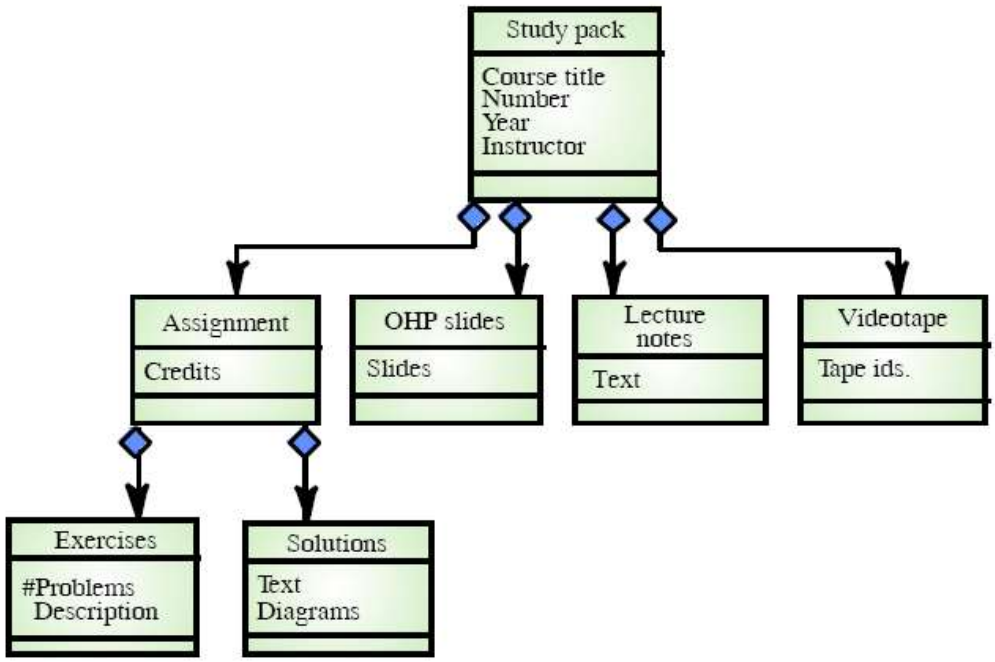


Multiple inheritance : Rather than inheriting the attributes and services from a single parent class, a system which supports multiple inheritance allows object classes to inherit from several super-classes. This can lead to semantic conflicts where attributes/services with the same name in different super-classes have different semantics. Multiple inheritance makes class hierarchy reorganisation more complex.



[4.2] Object aggregation = composition = part-of relationship

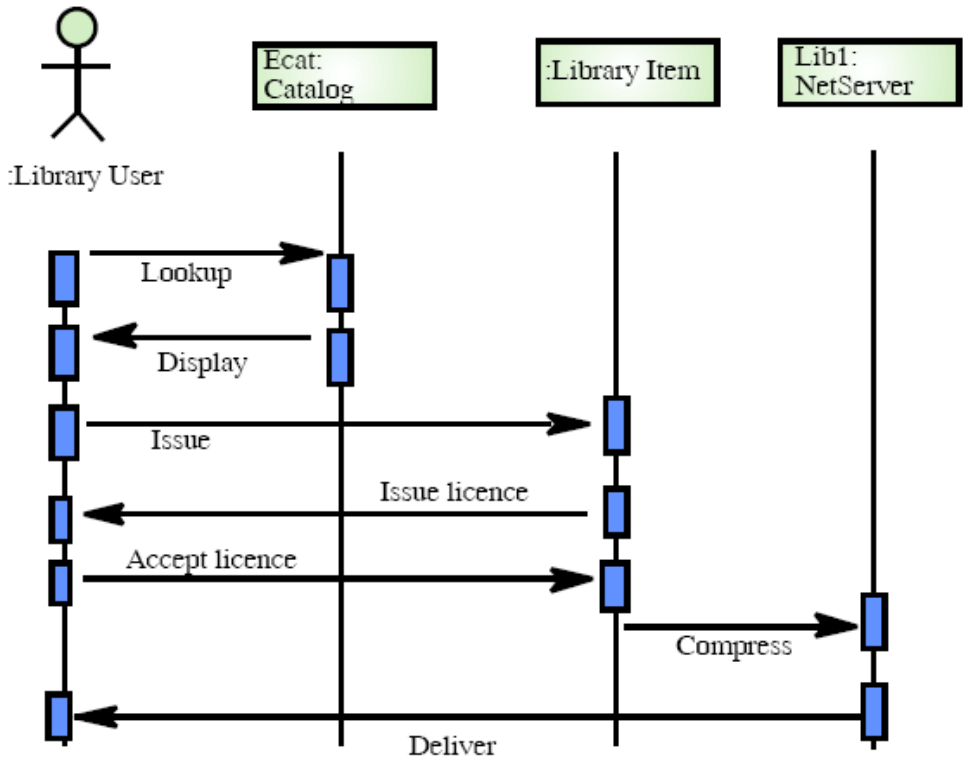
An aggregation model shows how classes that are collections are composed of other classes.



[4.3] Object behaviour modelling

A behavioural model shows the interactions between objects **Sequence diagrams (or collaboration diagrams)** used to model interaction between objects.

example: Issue of electronic items


Ecat:
Catalog
:Library Item
Lib1:
NetServer
:Library User
Lookup
Display
Issue
Issue licence
Accept licence
Compress
Deliver

Object oriented Programming in C#

Object-Oriented Programming (OOP) is a software development paradigm that splits a program in building blocks known as **objects**.It uses "objects" to design applications and computer programs. The OOP paradigm allows developers to define the object's data, functions, and its relationship with other objects.

Fundamental Concepts Of OOPS :

Class: A class defines the abstract characteristics of a thing (object), including the thing's characteristics (its **attributes** or **properties**) and the things it can do (its **behaviors** or **methods** or **features**)

Objects can be grouped into classes.A class is a detailed description of an object. It is the blueprint for an object. For example, the class Dog would consist of traits shared by all dogs for example fur color, and the ability to bark.

Object:A particular **instance** of a class is called **Object.** The class of Dog defines all possible dogs by listing the characteristics that they can have.

Object oriented Programming Features :

i)**Encapsulation**

ii)**Inheritance**

iii)**Polymorphism**

iv)**Data Abstraction**

i) Inheritance

Getting the properties from one class to other class is called inheritance.A parent class can inherit its behavior and state to children classes. This concept was developed to manage generalization and specialization in OOP and is represented by a is-a relationship. Inheritance offers Re-usability ,Consistency ,Less redundancy .

The idea of inheritance is at the heart of Object Oriented programming. In essence, an object can be created which inherits some of its functionality from another object or objects.

In order to implement this correctly, attention must be paid during the class design phase. This process is basically a process of identifying the common features of many specific objects and abstracting them to a higher level.

The classic modeling example is modeling motorized transportation types. If we look at a car, a bus and a train we can see certain common aspects that they all share. All have wheels, all have a maximum speed, all have an engine, all need a way to stop. Rather than creating member elements within each class we can abstract the common elements to another more generic class called Vehicle which contains the common elements.

```
class Vehicle{

int numberOfWheels;
string enginetype;
int maxSpeed;
public bool brake();

}
```

We can now spend our time modeling the specifics of the car Class:

```
class Car{

int numberOfDoors;
float horsePower;
string interior;
string bodyStyle;
string make;
string model;
}
```

The Car class as it stands right now cannot tell us the engineType, maximum speed, or number of wheels. In order to do this it must inherit from the Vehicle class. This is the way of saying that a Car **IS A** Vehicle. The **IS A** relationship is what allows objects to be used as different types under different circumstances.

Inheriting a class is implemented by adding a colon and the class name that you want to inherit from.

```
class Car:Vehicle{
…
…
}
```

The Car is now defined as inheriting from Vehicle. A Car IS A vehicle. Vehicle is considered the **base class** for Car.

The Car now has 2 types defined for it: 1)Car 2) Vehicle.

The beauty of OOP becomes apparent when we begin to use the Car object in different situations.

This can be seen if we look at three methods, all requiring a parameter:

```
public void getMaxSpeed(Vehicle v)

public void getModel(Car c)

public printString(Object 0)
```

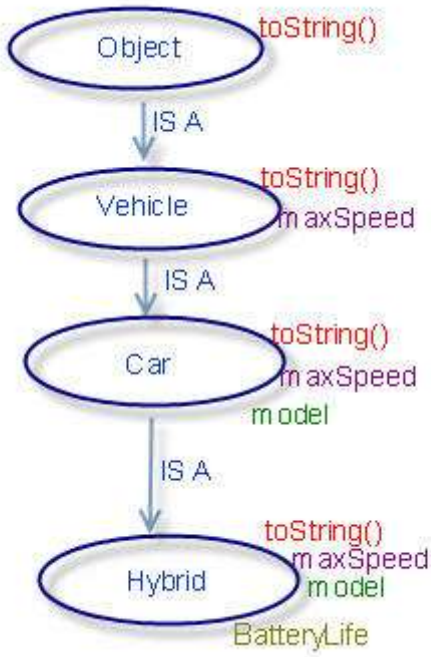
If we create the object theCar as type Car, we can use theCar for each one of these method calls.

```
Car theCar  = new Car() ;

public void getMaxSpeed(theCar) //theCar IS A Vehicle

public void getModel(theCar)  // theCar IS A Car …duh

public printString(theCar)  // theCar IS A Object
```

This use of the Car object as three different types is termed **polymorphism** (many shapes…or in our case, many types). **Polymorphism** allows objects to be used as any type in their inheritance chain. All objects inherit from Object when they are created, so any created object can be used as an Object.

In Windows programming, many of the UI elements such as Listbox, TextBox etc all derive from the Window class, and therefore can be used in any situation that is expecting a Window object.

When a class is defined as inheriting from another object, it **derives** some of its functionality from that object, and then can add additional functionality to the object. A class will inherit any of the public and protected members of its base class.



C# does not support multiple inheritance of classes within a single class definition. A single class can only inherit from ONE class. For example, you could not have the Hybrid class inheriting from Vehicle and Car in its definition.

```
class Hybrid:Vehicle:Car   // incorrect
```

The Hybrid class can only inherit from one class, in this case Car. Because Car was derived from Vehicle, Hybrid also inherits the members of Vehicle, but it inherits them through Car, not through Vehicle directly.

```
class Hybrid:Car  //correct
```

One problem with this is that Object hierarchies can become nested to very deep levels if all general functionality specifications needed to be inherited. This can degrade performance if the hierarchy is nested to deeply. We will see how this can be resolved when we look at Interfaces.

.Net Inheritance Capabilities :

- Ability to use the functionality contained in an existing class in a newly created class.
- All objects in .NET are derived from, and inherit methods from, a base class called System.Object.
- In .net we can utilize cross language inheritance (classes developed in VB.Net can inherit from classes written in other .Net languages as long as the base class is CLS-compliant).
- Multiple implementation of inheritance not supported. Can have multiple levels of inheritance, however each derived class can have one and only one base class.

Base Class : Provides default implementation of properties and methods It is also known as **parent class or superclass**

Derived class : Inherits members from the base class and all ancestors of the base class

Overrides or extends base class to become specialized. Each level of specialization typically adds new functionality,it is called as **sub class or child class**

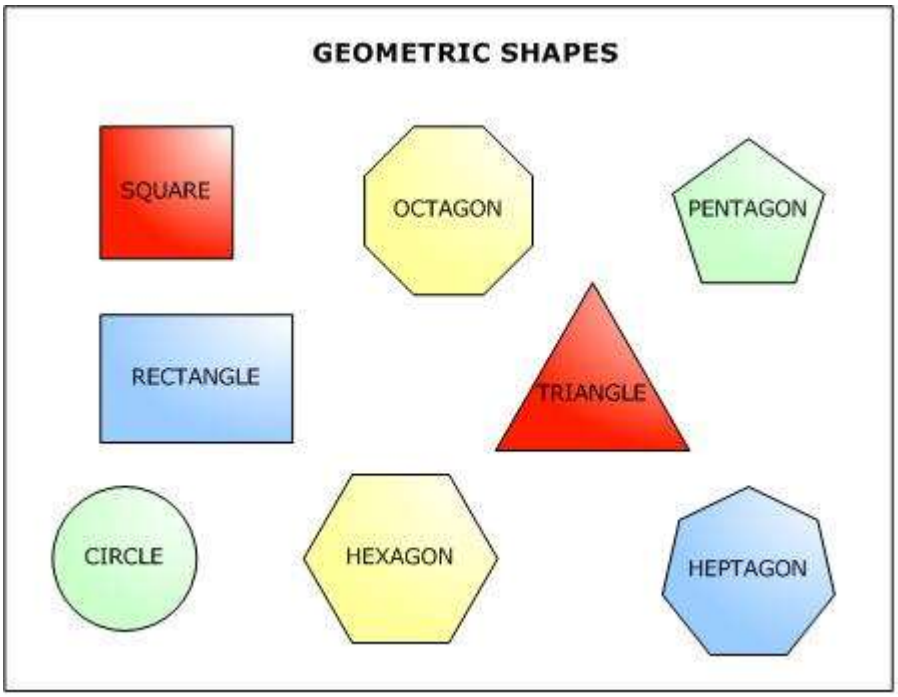
Example:

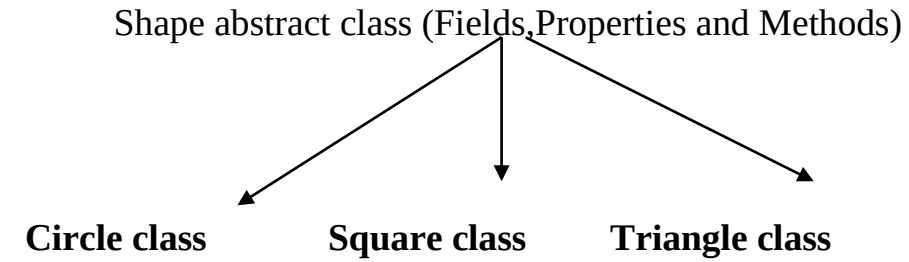
I) Geometric Shape→General Object→Area,Perimeter,color

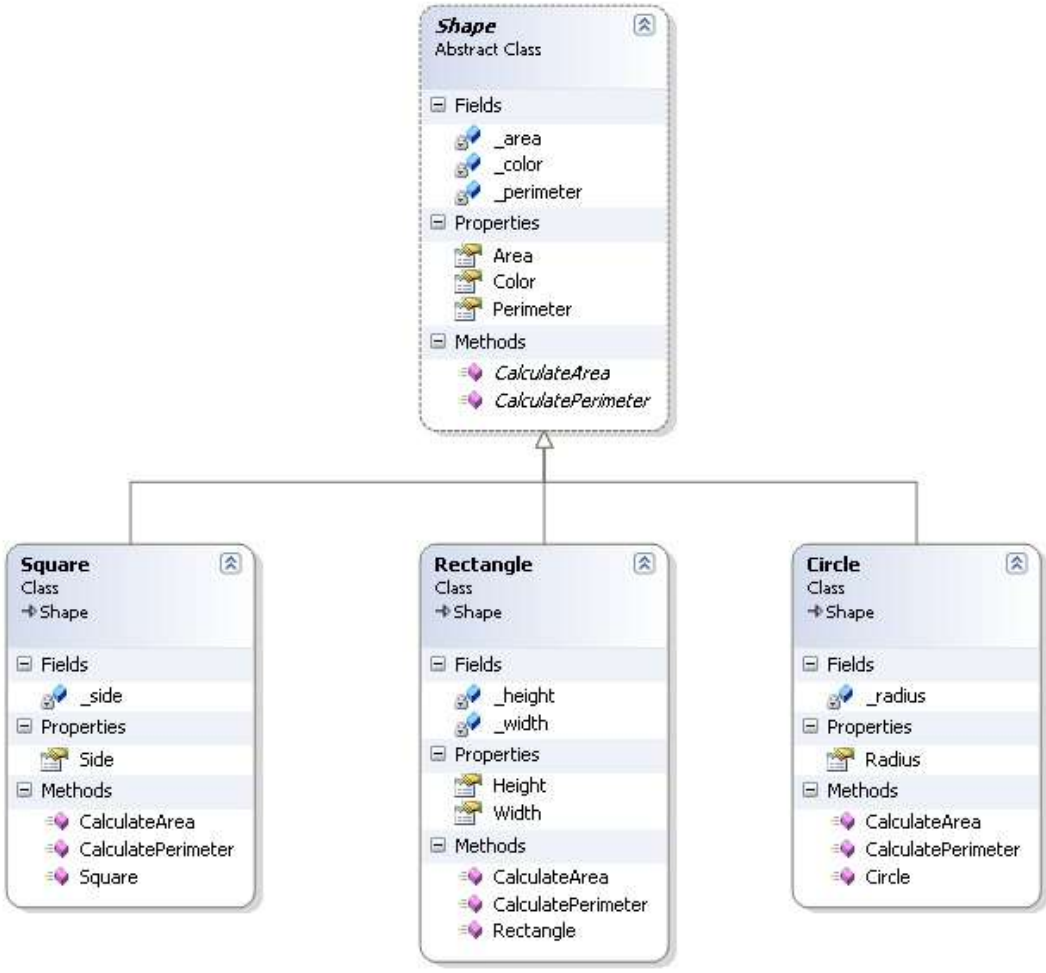
Generalization → Object encapsulates common state & behaviour for a category of Objects

Specialization →Object can inherit the **common state and behavior of a generic object**

Each object needs to define its own special and particular state an behavior. Each shape has its own color. Each shape has also particular formulas to calculate its area and perimeter.

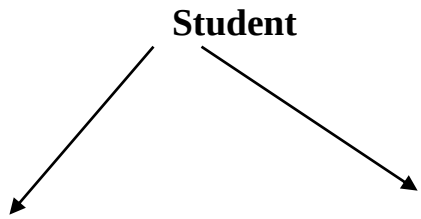






In the above example generalization and specialization are explined

II)
Classes that inherit from a class are called ***subclasses*.** The class a subclass inherits from are called ***superclass***. In the example, **Student** is a superclass for **Graduate** and **Undergraduate**. **Graduate** and **Undergraduate** are subclasses of **Student**.



Graduate **Under Graduate**

Types of inheritance supported by .net :
Is a Relationship:
Implementation Inheritance: It is a form of inheritance in object-oriented programming languages. The derived classes of the same parent (the base class) to share code implemented in the parent. Methods, variables, properties, events and other actual code elements are implemented in the parent and used by the children. Code from base class can be extended or overridden in derived class. Examples: single implememtation inheritence-one super class and one sub class. Multi level inheritance -the sub class of one level forms the super class of another level.

Visual Inheritance :
Ability to reuse and extend code and visual objects from an existing form or control .

C# doesn't support multiple implementation inheritance. C# & VB.NET supports only multipletype/**interface inheritance**, i.e. you can derive an class/interface from multiple interfaces. There is no support for multiple implementation inheritance in .NET.

That means a class can only derived from one class. since a lot of ambiguity comes in disigning object model.

.Net do not support Multiple implementation inheritance. But you can very well **have multiple interface inheritance**.

Uses of Inheritance:

- **Code Reuse:** Code that utilizes existing components usually takes less time to write, Reduced ongoing bug fix.

- **Facilitates Polymorphism :** Allows you to use an object created from a derived class in any situation where an object created from the base class is expected. Code written in terms of a generic base class can be leveraged even when working with a object created from a derived class.

ii) Polymorphism :

Polymorphism is the one of the feature of OOP'S,it means one name existing in multiple Forms , it has the ability to redefine methods for derived classes. polymorphism is concerned with how a class presents itself to the outside world. Polymorphism roughly means "**many forms**," and specific named behavior can be implemented in different ways by different classes.
To really understand polymorphism we need **method signature**, also sometimes called a **prototype**. All methods have a signature, which is defined by the method's name and the data types of its parameters. In Polymorphism **different objects have different implementations of the same**

Encapsulation has to do with hiding the internal implementation of an object,where as Polymorphism has to do with multiple classes having the same interface.

Polymorphism in Methods

As we have seen, the ability to use an object as a different type is known as polymorphism. This provides a powerful tool for extending and reusing code but there are some situations that must be dealt with when dealing with inheritance. One of these is the use of method names within the object hierarchy.

For example, our Vehicle class provides a brake() method. All objects that inherit from vehicle will also inherit this method. But suppose we want to have our car perform a different action when we call the brake() method? If we think about this, we may want to have any other object derived from Vehicle to be able to provide its own implementation of the brake() method.

In short, we want the brake() method to be treated polymorphically, we want the appropriate method to be called depending on the object type. To establish this in our base class we add the virtual keyword to the method.

```
Vehicle
public virtual bool brake(){}
```

The virtual keyword tells any class that inherits from this that it is free to override this method with its own method of the same name. Any class that wants to include their own version of brake() can do so by adding the override keyword to that method. Without this virtual keyword any derived class cannot override the method. All methods are non-virtual by default which means that classes which inherit the method cannot override the method. By adding the virtual keyword to the method in the base class we allow the derived classes to override it if they so choose.

```
Car
public override int brake(){}
```

Any car object will now use the brake() method in the Car class, and not rely on the method inherited from Vehicle. However, the Car class can still call the brake method in the Vehicle class by using the reserved keywrod name base.

```
public override int brake(){
        base.brake();     // call the vehicle brake method.
        // now do some special car braking things here
}
```

Versioning in Class Libraries

The C# language is designed so that versioning between base and derived classes in different libraries can evolve and maintain backwards compatibility. This means, for example, that the introduction of a new member in a base class with the same name as a member in a derived class is completely supported by C# and does not lead to unexpected behavior. It also means that a class must explicitly state whether a method is intended to override an inherited method, or whether a method is a new method that simply hides a similarly named inherited method.

C# allows derived classes to contain methods with the same name as base class methods.

- The base class method must be defined **virtual**.
- If the method in the derived class is not preceded by **new** or **override** keywords, the compiler will issue a warning and the method will behave as if the **new** keyword were present.
- If the method in the derived class is preceded with the **new** keyword, the method is defined as being independent of the method in the base class.
- If the method in the derived class is preceded with the **override** keyword, objects of the derived class will call that method rather than the base class method.
- The base class method can be called from within the derived class using the **base** keyword.
- The **override**, **virtual**, and **new** keywords can also be applied to properties, indexers, and events.

By default, C# methods are not virtual — if a method is declared as virtual, any class inheriting the method can implement its own version. To make a method virtual, the **virtual** modifier is used in the method declaration of the base class. The derived class can then override the base virtual method by using the **override** keyword or hide the virtual method in the base class by using the **new** keyword. If neither the **override** keyword nor the **new** keyword is specified, the compiler will issue a warning and the method in the derived class will hide the method in the base class.
Using the new keyword will hide the base class method name and tells the compiler to use only the derived class method within an instance of that class. This has the effect, however, of hiding the derived class method if the object is downcast to the base object.

Abstract and Sealed Classes

Abstract classes are classes defined so that they can only be inherited from, an object instance cannot be created from them. These are useful for specifying top level functionality in an inheritance heirarchy.

You cannot create an instance of an abstract class, you can only inherit from the class.

Classes are defined as abstract by using the abstract keyword. If any method is defined as abstract within a class, the class is also considered abstract and cannot be instantiated.

```
public abstract Vehicle {

 abstract public brake();

}
```

Adding the abstract keyword to our Vehicle basically defines this class as a "template" class which other classes can inherit from, but from which no object can be created directly as an instance of this class. In other words, you can never have an instance object of the Vehicle abstract class. Although an abstract class can contain public methods with implementation logic, any abstract methods of the class will contain no implementation logic, as the implementation is required in the derived class.

Sealed classes

The opposite of an abstract class, which can never create an instance object, is the sealed class, which can never be inherited. A sealed class is a class which can only be used as an instance object of that class, it can never be inherited from to create sub classes. An example of a sealed class in the framework is the FileInfo class which provides methods for manipulating the file system.

```
public sealed class FileInfo : FileSystemInfo
```

A sealed class cannot be used as a base class. For this reason, it cannot also be an abstract class. Sealed classes are primarily used to prevent derivation. Because they can never be used as a base class, some run-time optimizations can make calling sealed class members slightly faster.

A class member, method, field, property, or event, on a derived class that is overriding a virtual member of the base class can declare that member as sealed. This negates the virtual aspect of the member for any further derived class. This is accomplished by putting the **sealed** keyword before the override keyword in the class member declaration.

We can achieve polymorphic behavior by using several techniques :

- Late binding
- Multiple interfaces(Early binding →at compile time)
- .NET Reflection
- Inheritance

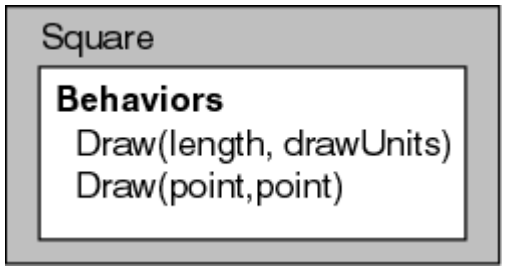
Using Polymorphism we can implement Overloading and Overriding .

Overloading :

A class may override its own methods based on parameter lists. Overloading which means the use of same thing for different purposes.Using Polymorphism we can create as many functions we want with one function name but with different argument list. The function performs different operations based on the argument list in the function call. The exact function to be invoked will be determined by checking the type and number of arguments in the functions.

Example :

It represents a class, Square, and its draw methods. The implementation of the draw behavior differs based on the information that is passed into the Draw method. This is a special case of overriding called *overloading*.



Overriding :

Overriding is to override the base class implementation or provide a new implementation in the derived class. We can override members in the base class in VB.Net using the keyword Overrides and in C# using the key word override. To override a base class member in VB.Net it should be marked as Overridable and in C# they should be marked with virtual.

Method signature of the overiddden methods should be same as the base class methods.

Shadowing :

This is a VB.Net Concept by which you can provide a new implementation for the base class member without overriding the member. You can shadow a base class member in the derived class by using the keyword "Shadows". The method signature,access level and return type of the shadowed member can be completely different than the base class member.

Hiding :

This is a C# Concept by which you can provide a new implementation for the base class member without overriding the member. You can hide a base class member in the derived class by using the keyword "new". The method signature,access level and return type of the hidden member has to be same as the base class member.

Differences Between Shadowing,Overriding & Hiding :

1) The access level , signature and the return type can only be changed when you are shadowing with VB.NET. Hiding and overriding demands the these parameters as same.

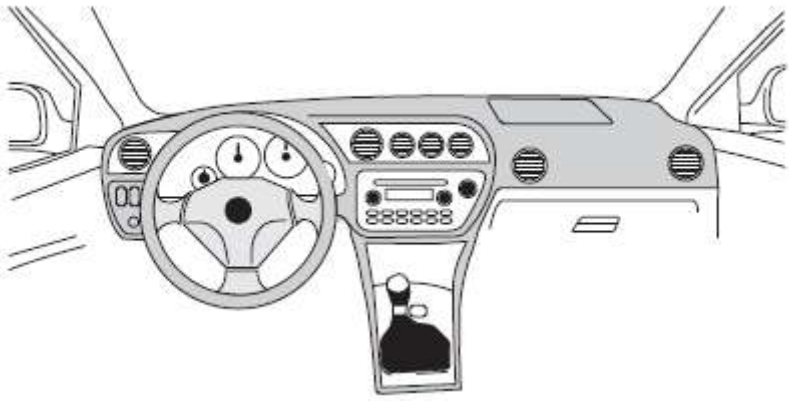
2) The difference lies when you call the derived class object with a base class variable.In case of overriding although you assign a derived class object to base class variable it will call the derived class function. In case of shadowing or hiding the base class function only will be called.

iii) Encapsulation

Encapsulation is the ability to **hide the internal workings of an object's behavior and its data.** object should totally separate its interface from its implementation. All the data and implementation code for an object should be entirely hidden behind its interface. encapsulation is the exposure of properties and methods of an object while hiding the actual implementation from the outside world. In other words, the object is treated as a black box—developers who use the object should have no need to understand how it actually works.

Example :

For instance, let's say you have a object named Car and this object has a method (behavior) named start(). When you create an instance of a car object and call its start() method you are not worried about what happens to accomplish this, you just want to make sure the state of the car is changed to 'running' afterwards. This kind of behavior hiding is encapsulation and it makes programming much easier. When you want your car object to be in a 'running' state, instead of calling: fuel.on(), starter.on(), etc., you just call start(). This not only makes it easier to work with, but if the internal workings of this start() method have to change, the results will be the same.



iv) Data Abstraction :

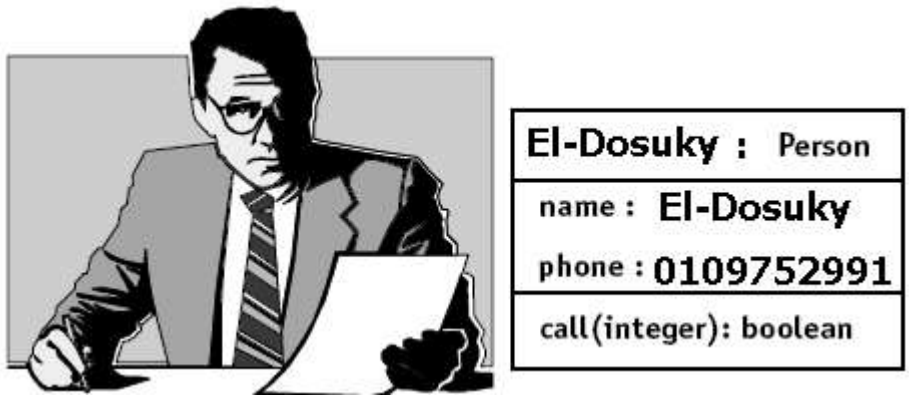
Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes.

Abstraction is the ability to generalize an object as a data type that has a specific Set of characteristics and is able to perform a set of actions.

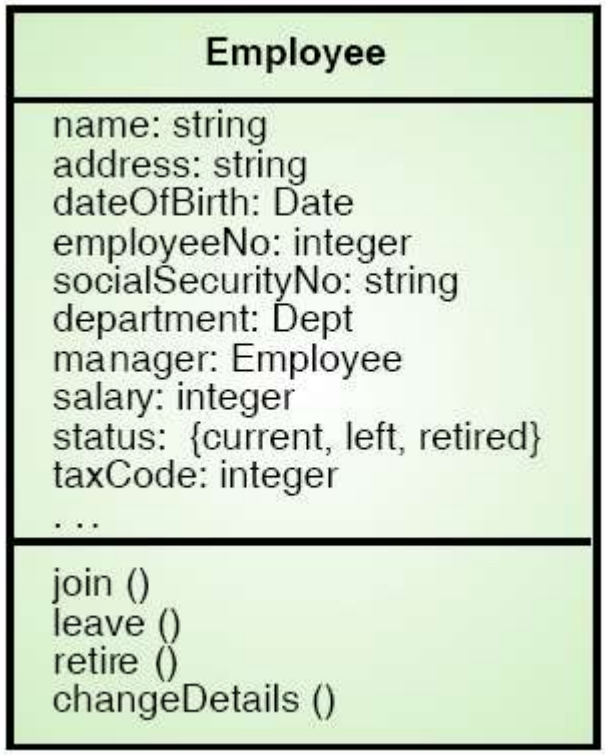
Object-oriented languages provide abstraction via **classes**. Classes define the **properties and methods of an object type**, but we cannot use a class directly, instead, an object must be created from a class—it **must be instantiated**.

For example, you can create an abstraction of a dog with characteristics, such as color, height, and weight, and actions such as run and bite. The characteristics are called **properties**, and the actions are called **methods**.

El-dosuky on the left (hard at work); and abstraction of El-dosuky on the right (an object icon)



Here is a detailed description of an employee .

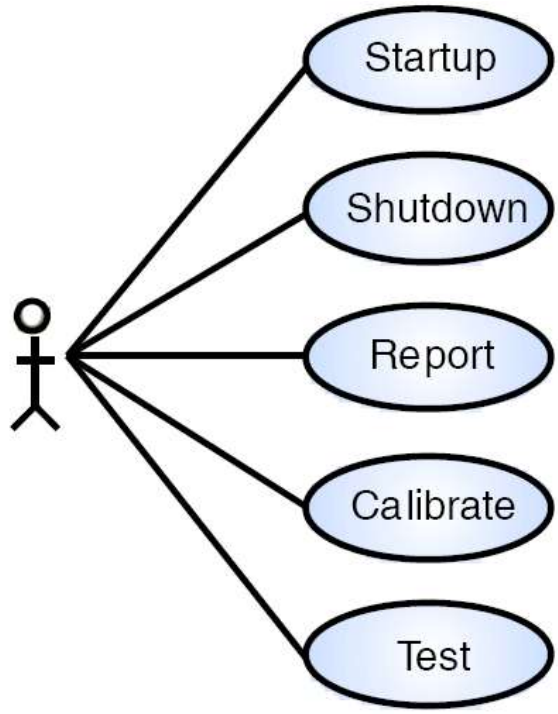


Case study: Weather system description

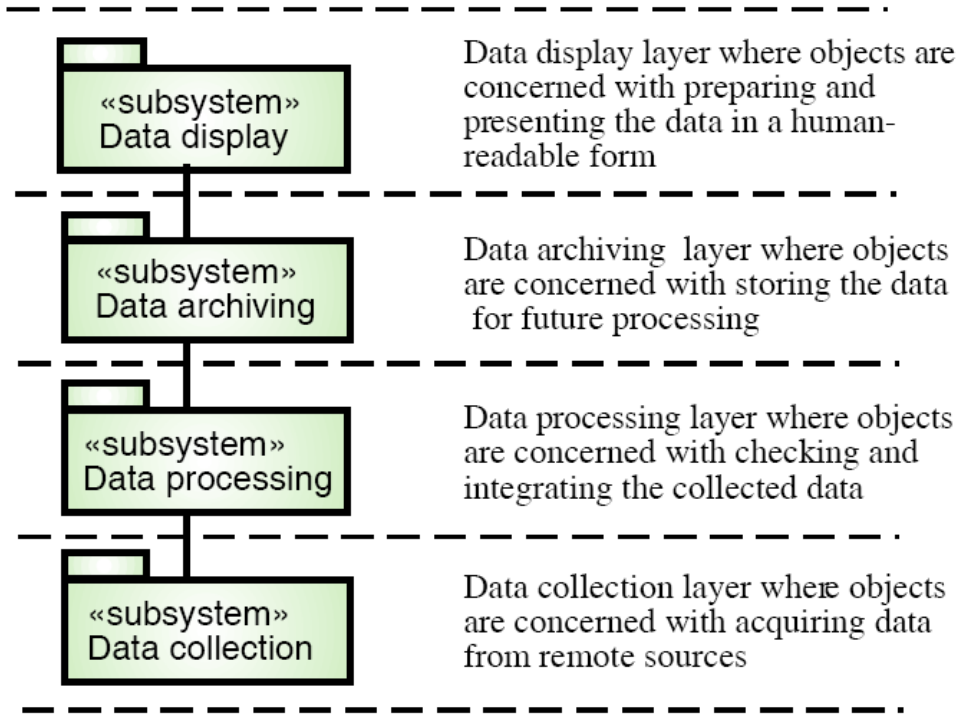
A weather station is a package of software controlled instruments which collects data, performs some data processing and transmits this data for further processing. The instruments include air and ground thermometers, an anemometer, a wind vane, a barometer and a rain gauge. Data is collected every five minutes.

When a command is issued to transmit the weather data, the weather station processes and summarises the collected data. The summarised data is transmitted to the mapping computer when a request is received.

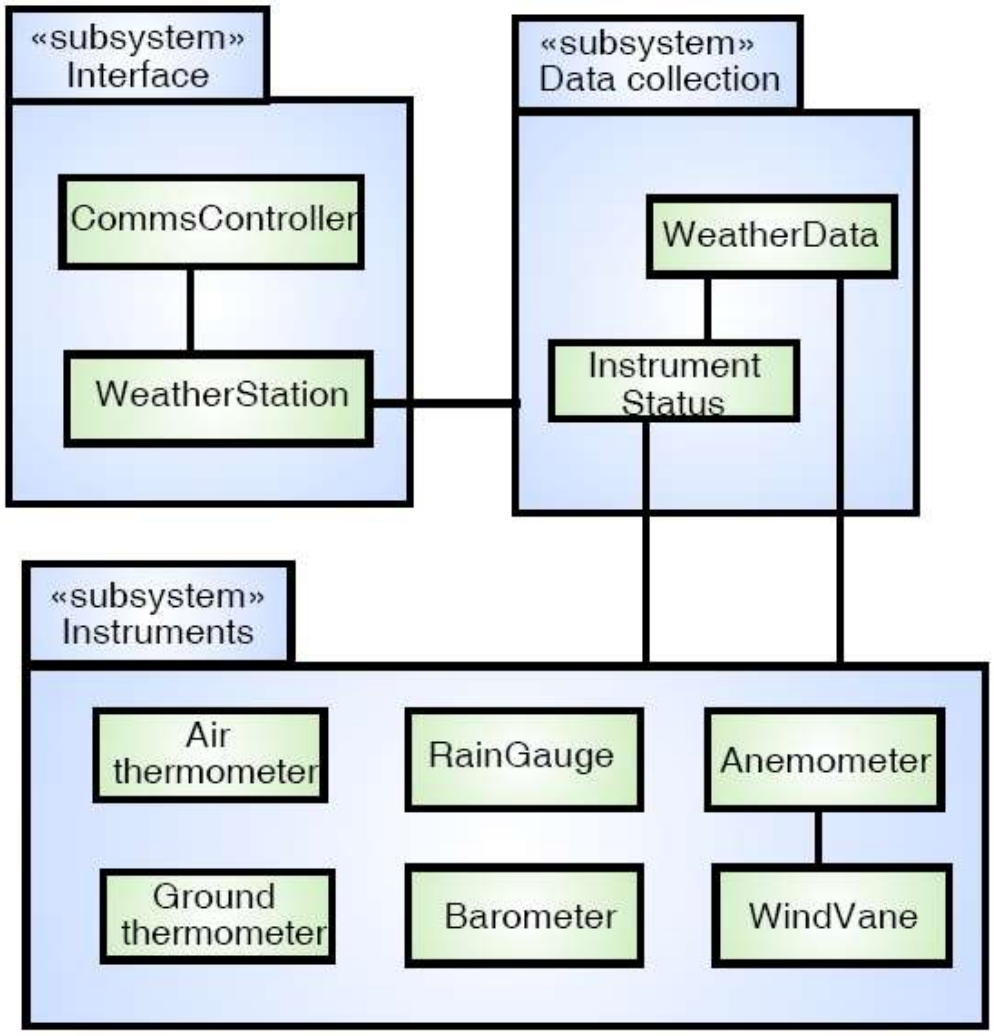
1. Use cases



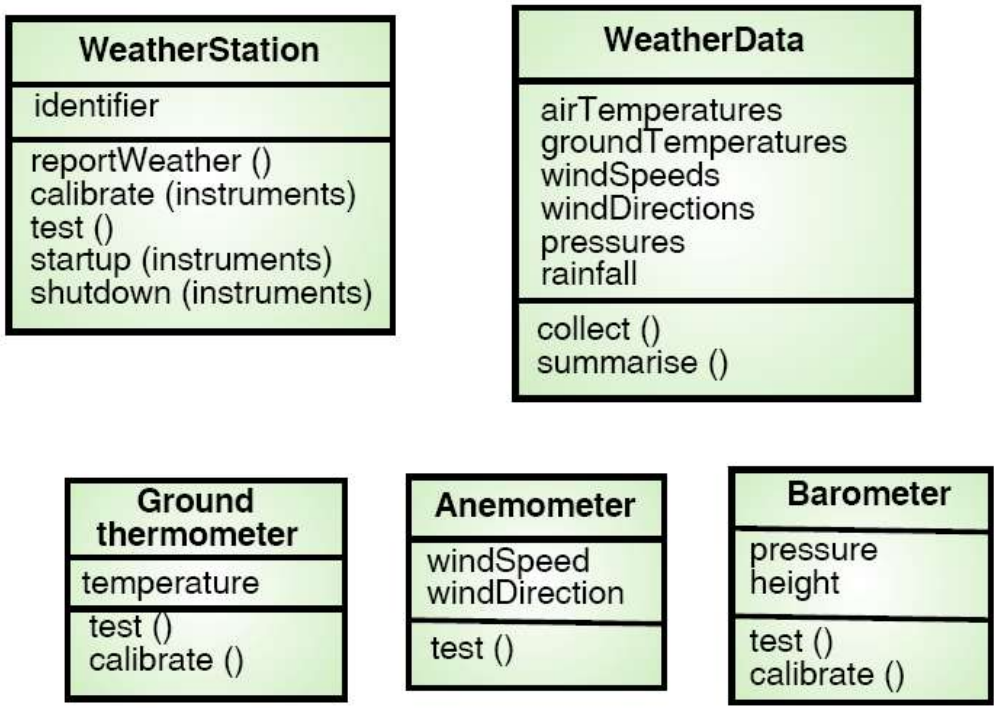
2.Weather station layers



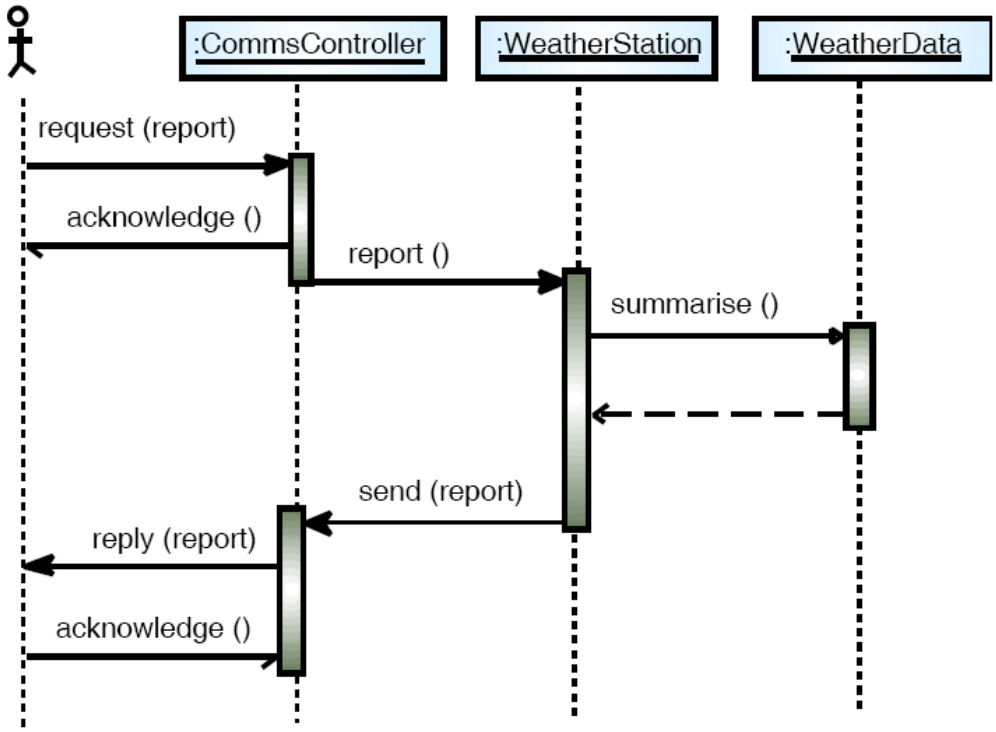
3. Weather station subsystems



4. Classes



5. Data collection sequence diagram



6. Weather station interface

```
interface WeatherStation {
        public void WeatherStation () ;
        public void startup () ;
        public void startup (Instrument i) ;
        public void shutdown () ;
        public void shutdown (Instrument i) ;
        public void reportWeather ( ) ;
        public void test () ;
        public void test ( Instrument i ) ;
        public void calibrate ( Instrument i) ;
        public int getID () ;
} //WeatherStation
```