# *Object-Orientation (Cont'd)*

## *Wolfgang Emmerich*

© Wolfgang Emmerich, 1998/99

# Lecture Overview

- **Basic principles of object-orientation**
  - *objects*
  - *classes of object*
  - *encapsulation*
  - *inheritance*
  - *polymorphism*
- **Object-orientation vs. structured methods**

© Wolfgang Emmerich, 1998/99



- This slide presents an overview of the concepts that are fundamental to object-orientation and that we are, therefore, going to discuss this week.
- We are going to define what objects are in terms of the concepts that define the object.
- We are then going to review classes as a mechanism to define the properties that similar objects have in common. Classes will also provide a mechanism to create objects in the first place.
- We are then going to look at encapsulation, i.e. how information hiding is supported and how properties that are local to an object can be hidden from the outside in order to improve the changeability of the analysis, design and implementation decisions made for that object.
- The first three characteristics relate to the object itself. It is somehow single, a recognisable entity; it is self contained and it is like other objects. The fourth and fifth characteristics concern its associations with other objects, the ability to communicate with different varieties of object and inherit qualities which make it the same as other objects.
- Inheritance can be considered as a notation for defining the similarity of several classes of objects. It provides the basic mechanism for abstraction, which allows properties common to multiple classes of objects to be defined only once in an abstract class. Then more specific classes inherit all properties from the abstract class; the do not need to be redundantly defined.
- Finally polymorphism refers to the fact that object of different classes can be treated by operations and be stored to variables.
- To start looking at these concepts in detail, we examine what objects exactly are?

## What’s object-orientation all about ?

- **_Principles and techniques for system modelling which:_**
  - ***aim to produce a model of a system***
  - ***provide notations and methods***
- **_Advantages for software development:_**
  - ***reduces the ‘semantic gap’ between reality and models***
  - ***makes system understanding easier***
  - ***allows local modification to models***

© Wolfgang Emmerich, 1998/99

*3*

- Object-orientation provides concepts, notations and methods for producing a model of a system.
- The Oxford English Dictionary defines the term model as: *“A simplified representation or description of a system or complex entity, esp. one designed to facilitate calculations and prediction”.*
- As we saw last week there are at least two main types of model suggested by Booch, ‘logical’ and ‘physical’, each themselves comprising ‘static’ and ‘dynamic’ versions. Jacobson requires five (domain object, analysis, design, implementation, testing). We are able to use object-orientated principles and notations in any of these models, which simplifies the incremental development of those models.
- Notations and methods are at the core of all software engineering practice. It is important that object-orientation can provide a sound semantic basis for both (and for models).
- Object-orientation simplifies the development of the models as object-oriented notations are fairly expressive and therefore can express models that are 'closer' to what has to be modelled from the real world.
- The expressiveness of object-orientation also facilitates a dense notation of models, which again makes it easier for consumers of models to understand and digest them.
- Probably the most important advantage of object-orientation is that it supports the important principle of information hiding. Therefore, design decisions athat are local to a certain part of the model can be kept local and be hidden from the outside. As they are hidden, they can be freely changed without interfering with other parts of the model.

# An Object

- ***An object is: "An abstraction of something in a problem domain, reflecting the capabilities of a system to keep information about it, interact with it, or both" (Coad & Yourdon 91)***
- ***"An entity able to save a state (information) and which offers a number of operations (behaviour) to either examine or affect this state" (Jacobson 92)***

© Wolfgang Emmerich, 1998/99

4

- This slide displays two definitions of what an object is. The definitions were given by Coad & Yourdon and Jacobson.
- Depending on which phase of the system life cycle a method is used in, the definitions given for objects refer to something more or less abstract or practical., In other words, they may be more or less close to something in the real world (the problem domain) or to the way in which that something might be represented in a software system.
- For the course of this lecture we would consider the following characteristics of an object to be essential:
  - Objects have an internal state that is recorded in a set of *attributes*.
  - Objects have a behaviour that is expressed in terms of *operations*. The execution of an operations changes the state of the object and/or stimulates the execution of operations in other objects.
  - Objects (at least in the analysis phase) have an *origin* in a real world entity.
- This relationship to real world entities is an issue of abstraction as we shall see on the next slide..

# *Objects & Abstraction*

- ***"An abstraction denotes the essential characteristics of***
- ***an object that distinguish it from all other kinds of objects***
- ***and thus provide crisply defied conceptual boundaries,***
- ***relative to perspective of viewer " (Booch 94)***

© Wolfgang Emmerich, 1998/99

5

- It is obviously essential to be able to differentiate one object from another; there must be a clear boundary. However different people with different perspectives or roles in a problem domain may view and define the characteristics of a single object quite differently.
- The (sometimes very difficult) task of a system analyst is then to capture these different views and model them in a coherent way in appropriate objects.
- In software engineering and object database research there have been several approaches proposed as to how the life of a system analyst can be made easier through the explicit capturing of views that coherently define a single perspective on a set of (related objects). We are, however, not detailing these approaches during the course of this lecture as they have not (yet) been introduced into the mainstream object-oriented approaches.

# Sample Objects

- **From a "Food Manufacturing Company"**
- **Passive objects :**
  - one individual sack of lentils
  - invoice 63501 sent to A Farm, Lincolnshire
- **Active objects :**
  - lorry "M235 BCM"
  - van "N683 CNM"
- **Human agents :**
  - Richard Green
  - David Brown (Executive)
  - Hill, D (Truck driver)
- **Structure objects :**
  - Marketing Department

© Wolfgang Emmerich, 1998/99

6

- This slide displays a number of examples for objects. Objects can be very different, they can be passive or active things, they can represent humans, organisational structures or even processes such as a holiday.
- Objects in isolation are of limited value; they need to be considered within a certain context and in order to express dynamics and to show a certain behaviour, they need to be stimulated through other objects and stimulate other objects.
- In this example, Richard Green is using a fax machine and has David Brown as his manager. Hence, it is necessary to set the object representing Richard Green in proper context and associate it to the object representing David Brown and the object representing the Fax machine.
- On the next slide we are going to see the structural characteristics of an object...

# Attributes and Associations

- *Any object has both attributes and associations*
- *Attributes characteristic features or properties name / value pairs*
- *Association any kind of link or connection between one object and one or a set of other objects*
- *Characteristics private to an object best represented as attributes*

© Wolfgang Emmerich, 1998/99



- An object has both attributes and associations.
- *Attributes* determine the characteristic features or properties that belong to the object itself. An attribute is a name value pair. The name of the attribute should be carefully chosen so that it suggests the proper semantics of the attribute. The name is then used to access the value of that object. The value of an object's attribute frequently change over time, though the name of the object's attribute never change. Names may be changed if the object model evolves but then the object becomes a different object.
- *Associations* are used to identify certain kinds of relationships that an object has with one or more other objects. Hence associations are used to set an object into context with other objects. Associations also have names and they sometimes also have attributes of their own.
- Modelling of objects often allows a choice between using an attribute or an association to represent a particular feature. Choice will depend on particular purpose of model; as a rule characteristics that are internal only should be attributes, even though the value of the attribute may be another object. If that other object, however, should be aware that it is an 'attribute' of some object it should rather be modelled as an association.
- We will now see that there is a great variety of associations...

## Associations are Relationships

- **Two essential oppositions :**
- **aggregation vs reference relationships**
  - **aggregation relationships where connection creates composite objects from simple objects**
  - **reference relationships where connection only refers to another object**
- **static vs dynamic relationships**
  - **static relationships, where coupling of objects is stored over a long period of time,**
  - **dynamic relationships, which are established by operations**

© Wolfgang Emmerich, 1998/99



- *Association* will be used as the most general term, covering all types of relationships. It is now also incorporated into a 'catch-all' term in the UML.
- Different kinds of relationships can be distinguished. Aggregation relationships are used to form *composite objects*. A composite object is an object that has component objects. These component objects are related to the composite object through an aggregation relationship. The composite object has a certain behaviour that applies to all components, e.g. deletion. If the composite object is deleted then all component objects are deleted as well.
- Reference relationships, however, only related two objects. Then one of these object can easily identify the other one in order to, for instance, stimulate a the execution of a certain behaviour.
- Static relationships are relationships that are stored in order to keep them over a longer period of time. This is appropriate if the relationship exists for long, i.e. the objects participating in the relationship do not change frequently.
- Dynamic relationships are computed by operations and are not stored. In that way updates are not required when the relationship has to be changed, This is used if membership in relationships frequently change.
- Aggregation relationships tend to be static, whereas reference relationships can be both static and dynamic.

# Viewpoint of Association

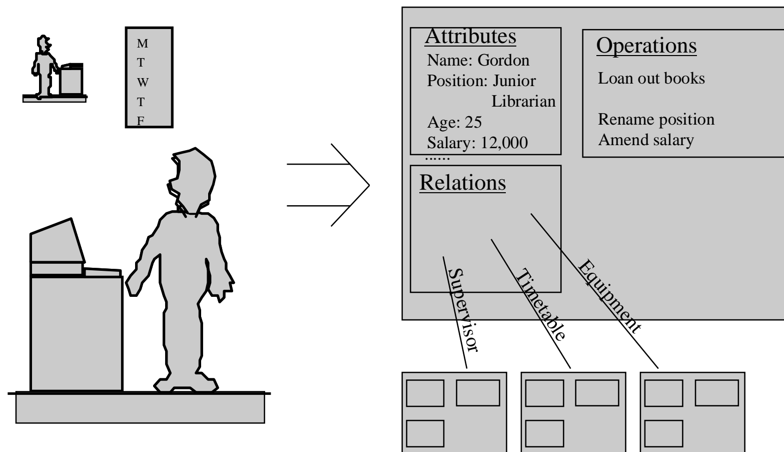
- View and purpose affect definition of assoc.
- Partition or division: Association indicating new objects created by splitting other objects apart , e.g. A book 'consists of' or 'can be divided into': title page, introduction, chapters, conclusion, index
- Aggregation or amalgamation: Association indicating new objects created by adding other objects together e.g Covers, binding, and end papers are

© Wolfgang Emmerich, 1998/99


- The distinction between types of association is dependent in part on the view of the object and the purpose of the observation.
- If considered from the component objects the term aggregation is correct but if we consider the composite object's perspective then the object is partititioned into component objects.
- A potential author may see his or her work as partitioned, but the publisher's printer will see it as an aggregation. In an hierarchical representation arrows indicating association will be in different directions.
- This is why sometimes relationships are given two names for the links they have in different dirctions.
- Other forms of associations depend on the meanings attached to objects and their purposes e.g. all associations defining social or organisational structures.
- Please note, that objects may have more than one association e.g. 'sibling' between members of a 'family' 'consisting of' 'adults' and 'children'.
- Having described generally an exterior view of an object, we are going to look now what can be said about its specific characteristics, that might be defined within it?

# Structure and Behaviour of an Object

## ■ Gordon, Junior Librarian, as object




© Wolfgang Emmerich, 1998/99



- An important aspect of o-o analysis is the transformation of the type of view on the left to that on the right, by the abstraction of characteristics, for which there is no standard algorithm. This process can uncover a multiplicity of different associations of different kinds and between same objects, e.g. Gordon has both static and dynamic relations with his terminal.
- An object has :
  - fixed, private attributes,
  - relationships with other objects that are static,
- In addition an object behaves in particular ways via :
  - operations computing dynamic relationships with other objects, and,
  - operations relevant to itself.
- Hence the Junior librarian gordon is seen as an object that has attributes that identify his name, position, age and salary. These are modelled as attributes because they inherently belong to the object that represents Gordon. Likewise Gordon has static relationships to other objects modelling entities, such as his supervisor, a timetable and certain equipment that he can use. These are modelled as relationships because these other objects also have to know about the object modelling Gordon. Gordon has a certain behaviour.
- Please note that it is not unlikely that associations that are modelled as relationships at some stage will become attributes (when it is recognised that they are private to the object) or become operations (when it is recognised that they should be dynamically computed).

# Behaviour and Structure

## ■ *Example in a Food Company*

One packet of herbal tea

Attributes
Price: 2.95
Weight: 1kg

Operations
Is Stored
Is Sold
Is Delivered

Relations
Order: 57891
Delivery: 57891A
Supplier: A Farm

Richard Green

Attributes
Age: 45
Sex: Male

Operations
Decide policies
Attend meetings
Manage Board meetings

Relations
Managing Director

Lorry M235 BCN

Attributes
Reg: M235 BCN
Weight: 7.5 tons
Fuel level: 40 litres

Operations
Deliveries Goods
Repeat Until End
Go To Destination
Unload Goods
EndRepeat
Refuel....

Relations
Driver Hill, D
Trailor SSFG12

Marketing Department

Attributes
Staff: 50 people
Location: High St

Operations
Market All Products
Choose New Products
Determine Market Niches

Relations
Marketing Director
Marketing Team

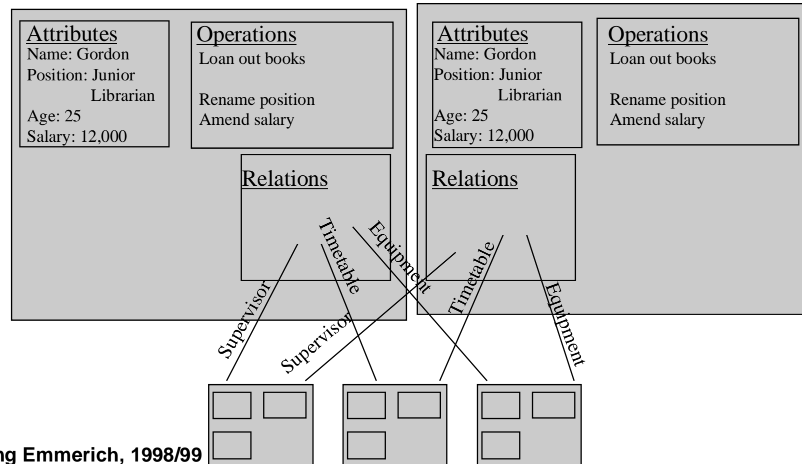
© Wolfgang Emmerich, 1998/99

11

- This slides includes examples of passive, active, human and structure objects and details the examples of objects that we had on an earlier slide.
- The slide shows relationships by indicating the identity of objects as names which is a rough approximation. In reality relationships will be established between two or more objects that are properly identified.
- We are, therefore, going to see now how objects are identified (e.g. for the participation in a relationship).

# Object Identity

■ ***Separate objects each have a unique object identity***

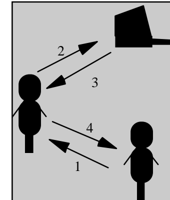



© Wolfgang Emmerich, 1998/99 12

- In the relational data model, different tuples cannot be separated if all their attribute values are the same. Hence it is important in relational data modelling to identify primary key attributes. These are then used to identify tuples.
- The object-oriented approach to identification is different. Any object has its own identity. Object identity is implemented by the run-time environment of object-oriented programming languages. Hence it does not need to be modelled. The object-oriented approach also separates *equality* from *identity*. Two objects are usually considered equal if their attribute values and the relationships they participate in are the same, they may still be not identical though. Two objects are identical if they have the same identity.
- The example given on this slide displays two objects that are equal, but not identical. They represent two different persons (e.g twins) that happen to have the same names, age, salary and are working in the same position. Still in the object-oriented approach we can tell that they are different people without having to introduce artificial primary key attributes.
- In object-oriented programming languages object identity is made available to the programmer. Object identity has different names, such as 'object references', 'unique object identifiers' or 'object pointers'. In essence these are all the same in that they uniquely identify objects.
- As we will see on the next slide object identity is also needed for stimulating the dynamics of objects...

# DYNAMICS OF OBJECTS

- **_Dynamics are generated through stimuli or messages passing between objects_**
  - **_Receipt of a stimulus cause operation by (or in) the receiving object_**
  - **_Receipt of a stimulus can trigger sending of another stimulus to other objects_**




© Wolfgang Emmerich, 1998/99



In order to be come active objects have to be activated or stimulated. This is usually achieved by passing messages between objects.

Note that the object identity is used to identify the target object to which a message is to be sent.

The receipt of a messages activates the object and the object will execute an operation that is identified by the message. Operations are, therefore named and this name is usually used in the message. The message may also include parameters that the operation will need for execution. Once the operation is finished the result is sent back to the sender of the first message.

The example in the lower right corner displays a set of messages that would be sent between an object representing librarian Gordon, an object for student Natalie and a library management system.

1 Message for "Loan-out books" operation by Gordon is request to loan books from Nathalie, a student

2 Gordon registers loan

3 System flags excess borrowing message

4 Gordon tells Natalie she’s over limit

In this exchange the internal operations of each ‘object’ are hidden unless, for example, Gordon describes outloud every step he takes (which he might do in a requirements gathering exercise).

In object-orientation, the principle of encapsulation embodies this idea of self-contained objects which only expose their exterior, their ‘interfaces’ to other objects.

# Encapsulation

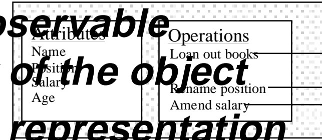
- ***"Behaviour and information are encapsulated in objects" (Jacobson 1992)***
- ***"Encapsulation is the process of compartmentalising the elements of an abstraction that constitute its structure and behaviour" (Booch 1995)***
  - ***Only the 'interface' of an object is 'visible' to other objects***
  - ***Need, and can, only know operations on an object, not how they work nor about other characteristics***
  - ***Necessary prerequisite for information hiding***

© Wolfgang Emmerich, 1998/99


- The two quotes shown on the top of this slide indicate that encapsulation can be seen as both :
- a) an essential quality of object-orientation and
- b) one of the activities involved in analysis (the emphasis of Booch)
- The idea behind encapsulation is information hiding. Encapsulation means, literally, turning something into a capsule, i.e. into a small, sealed container whose contents are normally invisible from the outside, with consequences for 'visibility' of both operations and data.
- The motivation for both information hiding and encapsulation is that changes of only those concepts can affect other concepts if the other concepts use them. If a concept is hidden other concepts cannot rely; hence they are not affected if the hidden concept is changed. Hence encapsulation is a way to improve the changeability and maintainability of a system composed of objects.
- This is rather abstract in more practical terms the application of this principle in an object-oriented setting means that attributes are hidden because the name and types of attributes frequently change. Also the implementation of operations are hidden as other objects should not need to know anything how the operation is achieved.
- To achieve encapsulation in analysis and design phases we are usually only interested in the externally visible part of objects that is sometimes referred to as the object's interface.
- Object-oriented programming languages that are used in coding phases usually have concepts to separate the externally visible interface of an object from the code that implements the object.

# Encapsulation (cont'd)

- ***Encapsulation and abstraction are complementary concepts***
  - ***Abstraction focuses on the observable characteristics and behaviour of the object***
  - ***Encapsulation focuses on the representation derived from these characteristics***




- ***Encapsulation requires :***
  - ***explicit division between abstractions***
  - ***clear separation of their concerns***

© Wolfgang Emmerich, 1998/99

15

- Let us get back to Gordon and see what encapsulation means in his case. Gordon will have particular personal characteristics and responsibilities in relation to the users of the library. These factors are being encapsulated in attributes and relations, e.g. salary, age, loan out books. The only way to access them is to use the operations provided as the operation are exported as it is suggested by the little circles that break through the capsule.
- Note that encapsulation and abstraction are complementary concepts. The encapsulation has to be defined for classes at every level of abstraction.
- The universe has, so far, become populated with a potentially vaste number of objects. The concept of class that we are going to introduce on the next slide begins to simplify this profusion.

# Classes of Objects

- ***Classes represent groups of objects which have the same behaviour and information structures.***
- ***Class is a kind of type, an ADT (but with data), or an 'entity' (but with methods)***
- ***Classes are the same in analysis and design***
- ***"A class represents a template for several objects ...Objects of the same class have the same definition both for their operations and for their information***

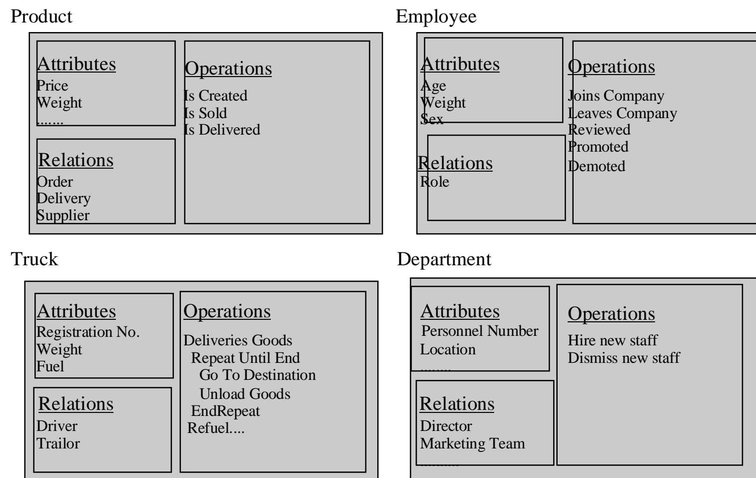
© Wolfgang Emmerich, 1998/99

*16*

- In any information system, there are many objects that share the same behaviour and information structure. In a library, for instance, there are thousands of book copies and a library information system will have as many objects. Each of them has the same set of attributes (names, not values) and relationships modelling the relevant concepts of a book copy and the same behaviour. As it is unfeasible to define these concepts for thousand of book copy objects individually we have to find a mechanism to define the concept once for all objects that have the same information structure and behaviour. The concept that we use for that purpose are *classes*.
- A class is a type that defines the common properties of a number of objects that are similar in structural and behavioural properties. For a class we define the attributes it has, the relationship instances of the class can participate in and the operations that instances of the class can execute.
- We also define for a class which of these properties are externally visible, i.e. that are exported, and which are private and hidden from the outside.
- Classes are the essential common feature of both analysis and subsequent design.
- Let us know look at examples of classes...

# Sample Classes

## ■ *From a Food Company*




© Wolfgang Emmerich, 1998/99 

- Principle concern of object-oriented analysis is the class, not the instance.
- Examples here show classes at the earliest analysis stage, identifying all common characteristics of each class, including the (static) relations with other classes and the behaviours that will come to be defined as specific internal and external operations.
- Please note that the term 'object' is often used very loosely, eg a class of objects can itself be refered to as an object.
- Having defined classes, each object becomes a particular instance of the class to which it belongs and we look at the class/instance relationship now.

# Objects are Instances of Classes

- Every object is an instance of a single class
- A class defines the possible behaviours and the information structure of all its object instances.
- Different instances may have their operations activated in different ways and in different sequences; hence they may be in different states.



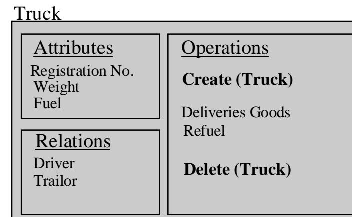
© Wolfgang Emmerich, 1998/99

18

- An object is an instance of a class. That class defines the operations that the object supports and the data structures that are used for storing information in that object.
- A class therefore defines structure and operations for all of its instances. Hence every object that is an instance of that class supports the same operations and has the same structure as every other instance of that class.
- Different objects of the same class can only differ in two respects. They have a different identity and they may be in different states if their attribute values are different.
- Note that classes can be considered as objects themselves on a meta level. Operations that would be provided on that meta level would be creating a new class, compiling it, adding a new attibute and so on. Attributes of these objects would be the attributes an object has and the operations that the object provides. This approach is supported by some object-oriented programming languages, most notably SmallTalk, in object database management system for schema management purposes and in the CORBA framework as an interface repository.
- Note also, that attributes of objects may be instance variables or class variables in some languages. An instance variable implements an attribute of an object while a class variable implements an attribute of a class, that is shared by all instances. A typical application of the latter would be counting the extent of a class, i.e. the number of objects that exist.
- The class of which an object is an instance is determined at object creation time and we now look at how objects are actually created...

# Instantiation of objects

- ***Instantiation of a class generates an object, an instance of its class***
- ***Instantiation demands a specific create operation in every class***



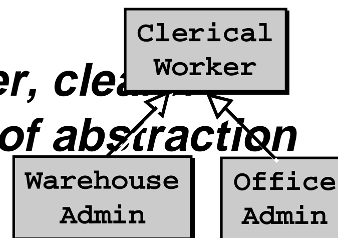

© Wolfgang Emmerich, 1998/99

19

- As we mentioned earlier classes are templates for objects. Objects come into existance by instantiating a class. Hence classes not only define properties of objects, but also how new objects are to be created.
- During the instantiation process a unique identity for an object is determined and typically all attribute values are initialised. As a run-time environment can only support default initialisations, application specific initialisations have to be determined as special operations of the class the object is instantiated from (rather than an object). These operations are sometimes also referred to as *constructors*.
- Instantiation has particular importance in object orientation because it is the essential activity linking class and object. It is crucial in implementation and it has no complement in any vernacular definition of a class.
- On the next slide, we consider an essential characteristic of the relationships between classes as opposed to objects...
-

## Inheritance

- ***Inheritence: a relationship between different classes with common characteristics***
- ***"If class B inherits class A, then both the operations and information structure in class A will become part of class B" (Jacobsen 92)***
- ***Major benefits are simpler, cleaner classes, at higher levels of abstraction***



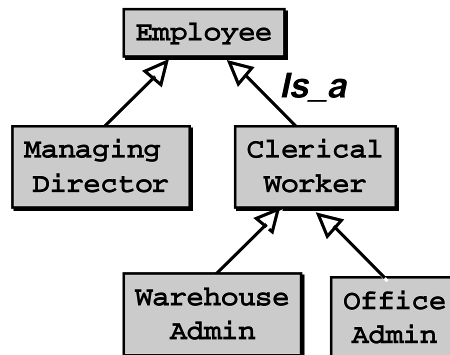

© Wolfgang Emmerich, 1998/99



- The common properties between different objects that have the same data structures and operations are specified in a class. Hence classes specify attributes, relationships and operations. At another level of abstraction, different classes might again share attributes, relationships and operations. Again it is highly undesirable to specify these properties more than once. In fact the important principle of abstraction that we had introduced last week suggested that these should be factored out and only specified once.
- Inheritance is the vehicle to do so. We can identify more abstract classes that specify a number of attributes, relationships and operations. Then we can specify other classes to inherit these more abstract classes. The more abstract class is called super-class and the more concrete class is called sub-class. A sub-class inherits all properties from its super-class.
- The inheritance relationship is transitive. A class C inheriting from another class B inherits also properties from A if B is a subclass of A.
- In the example given above, ManagingDirectory is a subclass of Employee. Hence class ManagingDirector inherits all properties of class Employee. Likewise TruckDriver inherits all properties of Employee.
- Subclasses may define attributes, relationships and operations themselves that are then specific to the subclass. Hence in the above example ManagingDirectory may have certain properties that a TruckDriver has not.
- Two important analytic activities, 'generalisation' and 'specialisation', are associated with the principle of inheritance and creation of new classes and these will be discussed on the next slide.

# Generalisation & Specialisation

- ***Generalisation - Creation of an 'ancestor'***
- ***Specialisation - Creation of a 'descendant'***



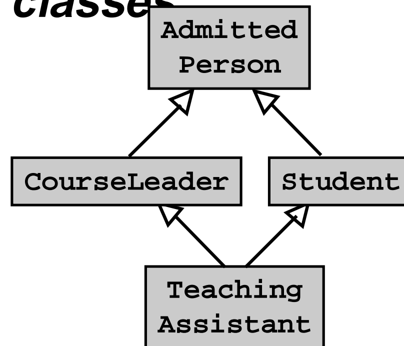

© Wolfgang Emmerich, 1998/99

21

- Generalisation means extracting common properties from a collection of classes and placing them higher in the inheritance hierarchy, in a 'superclass'.
- Much care has to be taken when generalising (as in the real world) that the property makes sense for every single subclass of the superclass. If this is not the case the property must not be generalised.
- Specialisation involves the definition of a new class which inherits all the characteristics of a higher class and adds some new ones, in a 'subclass'.
- Whether the creation of a particular class involves first, or second activity depends on stage and state of analysis, whether initial classes suggested are very general, or very particular.
- To put it in other words, specialisation is a top-down activity that refines the abstract class into more concrete classes and generalisation is a bottom-up activity that abstracts certain principles from existing classes in order to find more abstract classes.
- Both generalisation and specialisation can lead to complex inheritance patterns, particularly via 'multiple inheritance' that we are going to investigate on the next slide...

# Multiple Inheritance

- One class inherits from two or more existing classes




- Allows more complex class structures, but
- less easily understood

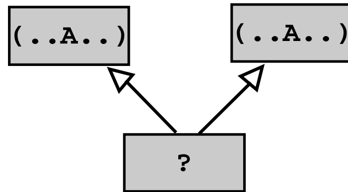
© Wolfgang Emmerich, 1998/99



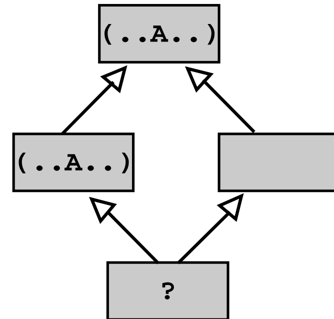
- Multiple inheritance refers to the fact that a class may inherit more than one existing classes. In the example above other staff inherits from both counter staff and repair staff.
- The advantage of multiple inheritance is that it facilitates re-use existing classes much better than with single inheritance. If properties of two existing classes had to be re-used and only single inheritance were available then one of the two classes would have to be changed to become a subclass of the other class. This may not always be sensible if the artificial subclass inherits properties that it ought not have just for the sake of enabling them to be reused in the third class.
- However, multiple inheritance should be used rather carefully as the inheritance relationships that will be created through it can become rather complex and are fairly difficult to understand themselves.
- It is also a controversial aspect of object-orientation and therefore not implemented in some object-oriented languages, eg Smalltalk, because multiple inheritance can lead to ambigous situations itself. We investigate an examples of these situations on the next slide...

# Multiple Inheritance & Ambiguities

- **Name clashes :**




- **Incorrect repeated inheritance**




© Wolfgang Emmerich, 1998/99

23

- Ambigouities can arise through multiple inheritance as in the examples above.
- The simplest form of these ambiguities are *name clashes* as displayed on the left hand side of the slide. A name clash arises when the subclass inherits two properties (irregardless whether attribute, relationship or operation) with the same name from two independent branches. If instances of that class receive a message identifying that property it is then ambigous whether they should react according to the definition in the left or the right superclass.
- A more subtle form of ambiguity that may arise through multiple inheritance is an *incorrect repeated inheritance* as displayed on the right hand side. Here an ambiguity arises because a property that has been repeatedly inherited via two branches from a common superclass is redefined, i.e. given a new semantics, in one branch. Then it becomes ambigous whether the original declaration of the property is valid in the subclass or the property that has been redefined in one branch.
- The problem with these name clashes is that they very often cannot be rectified at their origin as the classes to be re-used often cannot be changed. Then renaming concepts are used to rectify them.
- If these ancestors themselves have no common ancestor, names have to be changed or one definition chosen. If these ancestors themselves have a common ancestor, then it is also necessary to ensure than no redefintions have taken place higher in the hierarchy, thus effectively demanding a choice of definition. Detection of such circumstances is a major issue.
- The final concept to be explained is polymorphism...

# Polymorphism

- ***System behaviour is defined by the dynamic behaviour of instances of classes***
- ***"Polymorphism means that the sender of a stimulus does not need to know the receiving instance's class. The receiving instance can belong to an arbitary class" (Jacobsen 1992)***
- ***Polymorphism enables different instances of different classes to be associated***
- ***A receiving instance interprets stimuli according to its own class***

© Wolfgang Emmerich, 1998/99



- *Polymorphism* is a concept in type theory in which a single name (such as a variable type) may denote objects of many different classes that are related by some common superclass; any object denoted by this name is therefore able to respond to some common set of operations.
- (Also the term 'limited polymorphism', meaning a restriction of the classes receiving messages)
- Polymorphism reduces the complexity of implementing such dynamic behaviour, by allowing the association of different objects and interpretation of the same message in different ways based on the class they are an instance of.
- Example: 'Draw' message received by an instance of class 'line' will cause a different behaviour from that of an instance of 'filled irregular polygon' receiving same message.

## Structured Methods

- ***SSADM (Cutts 1987), SA (de Marco 1978), SADT (Ross 1977).***
- ***Existing structured methods treat separately***
  - ***functions (behaviour) and***
  - ***data (information held)***
- ***Problems:***
  - ***Difficulties with maintenance (because need knowledge of data storage)***
  - ***Division of knowledge (whereby "what" is transformed into "how")***
- ***Instability of functions***

© Wolfgang Emmerich, 1998/99



- This and the next two slides compare object-oriented and structured methods. This comparison has already been touched on from a more general perspective when we compared functional and object-oriented decompositon but now we shall try to get it down to the point.
- The problem with structure-oriented methods, such as SSADM, Structured Analysis or SADT, is that they treat functions (i.e. the behaviour) of the system differently from the data (i.e. the information held somewhere within the system).
- This complicates maintenance and the evolution of a system as both data and functions need to be changed. Moreover it is more difficult to isolate changes. If a certain aspect has to be changed, this almost certainly involves both the change of data structures and of algorithms. Finally the change of algorithms and data structures in structural methods often involves a number of subsequent changes to places where these data structures are used as well.
- Object-oriented decomposition, on the other hand has evolved from the idea of information hiding which significantly contributes to the changeability of the system as motivated on the next slide...

## *Object-Oriented Methods*

- ***Better able to cope with change***

| *Item* | *Freq. Of Changes* |
|---|---|
| *Object from application* | *Low* |
| *Long-lived information structures* | *Low* |
| *Passive object's attribute* | *Medium* |
| *Sequence of behavior* | *Medium* |
| *Interface with outside world* | *High* |
| *Functionality* | *High* |

- ***OO focuses analysis on problem domain***
- ***Promotes reuse***
- ***Continuity of representation***

© Wolfgang Emmerich, 1998/99

*26*

- Object-oriented decompositions of systems tend to be better able to cope with change. This is because they manage to encapsulate those items that are likely to change (such as functionality, sequence of behaviour and attributes) within an object and hide them from the outside world. This provides the advantage that the outside cannot see them and therefore cannot be dependent on them and does not need to be changed if these items change.
- Also object-oriented decompositons are closer to the problem domain, as they directly represent the real-world entities in their structure and behaviour.
- The abstraction primitives built into reuse have a huge potential of reuse as commonalities between similar objects can be factored out and then the solutions can be reused.
- Finally, object-orientation has the advantage of continuity througout analysis, design implementation and persistent representation.

## Suggested object-oriented Methods

- Coad & Yourdon (91) for OOA
- Booch (94) also for OOA
- Jackson (83) for system design
- Jacobson (92) OOSE
- Approach of this course based on Jacobson because employs ‘use cases’ throughout
  - essential user role
  - focus on domain
  - integration in process
- All likely to be superseeded by ‘retreads’ employing the UML

© Wolfgang Emmerich, 1998/99



- Unfortunately, there is no single object-oriented analysis and design method that we could readily teach you and this slide lists some of the proposed methods.
- For this course we have selected Ivar Jacobson´s object-oriented software engineering approach because it supports use case scenarios.
- We largely agree with industry that these scenarios are particularly useful during the elicitation of complete user reqirements. OOSE has domain focus built into it and is integrated in the process.
- We should, however, note that we are going to use the unified modelling language notation. UML is currently being developed at Rational, a major consulting company in the US and also the vendor of the market leading OOAD environment. Grady Booch, Ivar Jacobson and James Rumbaugh are jointly working on the modelling language. While we are giving this lecture, UML is being evaluated by the Object Management Group to become the de-facto industry wide notation for object-oriented modelling.
- It is very likely that the methods presented so far in the various books identified at the beginning will be revisited by their authors and be expressed in terms of the Unified Modelling Language.
- Although UML is an important consolidation step forward in the maturation process of object-orientation, it will not be a silver bullet. As the next slide suggests, there are also problems inherent to UML.

# Drawbacks of OO

- ***Large scale reuse not yet achieved***
- ***Few available reusable libraries***
- ***Managing reusable libraries is a problem***
- ***Extensive retraining before pervasive***

© Wolfgang Emmerich, 1998/99

28

- Although object-orientation is very favourable to reusing requirements, parts of the design and implementation, large-scale reuse has not yet been achieved. We believe that this is not necessarily only a problem of object-orientation, but also of the mindset of many software professionals that do not believe in anything they have not developed themselves.
- As there is little demand for reuseable components a market for components has not yet been established. Vendors are scattered and their products are rarely standardised and therefore are not exchangeable. A notable exception is the standard template library that has been standardised last year by the ANSI.
- With the possibility of deploying previously developed components from reuse library, whether bought off the shelve or built inhouse, the problem of configuration management arises which has not yet been fully understood.
- Also a considerable amount of retraining of staff is required before object-oriented projects start to fly and industry is still in the process of building up an experience base.
- As a member of a development team (whether student, academic or commercial) you may have your mind made up for you, but essential to recognise true status of ideas and techniques.

# Summary

- **Basic concepts:**
  - ***object: entity combining essential characteristics abstracted from a domain***
  - ***class: expression of objects' common characteristics***
  - ***encapsulation: combination of attributes & operations in a single self-contained object***
  - ***inheritance: relationship between super-class and sub-class defining levels of commonality***
  - ***polymorphism: facility allowing stimuli to ignore class of receiving object***

© Wolfgang Emmerich, 1998/99

29

- These concepts provide the foundations for all object-oriented methods, each with a different flavour
- It is difficult, in practice, to separate out the application of these principles within the process of analysis and thus give useful guidance. In general the ideas of an object as an abstraction and as an instance of a class help object identification; the ideas of inheritance and encapsulation aid object modelling and abstraction; behaviour modelling requires all these ideas plus polymorphism.
- For your background reading, we would suggest the following references. Bertrand Meyer gives a very comprehensive introduction and motivation for the concepts that we have discussed here. Luca Cardelli's a seminal paper provides a formal semantics for many of the concepts we have presented:

[Mey88] B. Meyer: Object-Oriented Software Construction. Prentice- Hall. 1988.

[Car85] L. Cardelli: The Semantics of Multiple Inheritance. Information and Computation 76:138-164. Academic Press.

- The next lecture begins the detailed description of Jacobson's OOSE method. For the notations of objects and classes we are going to use the Unified Modelling Language that is being officially revealed as these notes are printed.

# Summary (Cont'd)

- ***Application of concepts enables:***
  - ***object identification***
  - ***object modelling***
  - ***behaviour modelling***

© Wolfgang Emmerich, 1998/99