Object Models

- Object models describe a system by dividing the objects in the system into *classes* with common *attributes* and *behaviour*, and specifying the *relationship* between the classes.
- Object models are *static* models — they do not show how the objects react over time.
- The object model should be independent of the implementation language.

Reading

- Sommerville 9ed Ch 5.3
- Supplementary material on StudyDesk

Why Use Object Models?

- The objects and the relationships tend to be less likely to change during requirements analysis than the exact functionality. (A win compared with DFD functional models).
- Certain kinds of systems can be naturally modelled as interacting objects: e.g. a mouse sends a double click message to a window. (But many systems are not of this kind.)
- OO approach uses modularity, encapsulation, and abstraction.
  Note: these are *not* unique to OO — the Abstract Data Type approach to program design exhibits these characteristics.

What is an object?

- It is a “thing” that you can interact with.
  - Physical object (book, person) or a conceptual object like a data structure.
- It reacts to *messages* which are sent to it.
  - An object may receive many different kinds of messages, so each kind has a name to distinguish it. E.g. `Start`, `Stop`.
  - A message may have ≥ 0 arguments.
    E.g. `wait 30`, `moveto 10,20`
  - An object understands a fixed set of messages.

What is an object? ....

- Its behaviour (when sent a message) depends on its current *state*.
  - Object state is represented by its *attributes* (also called instance variables).
  - Attributes may be constants (Customer-id) or variables (Account-balance).
- It has *identity*. Each object is unique, and can be identified in some way to distinguish it from other similar objects.
  - Two separate objects may have precisely the same state but are not considered identical.
    E.g. Two identical widgets manufactured by a machine tool.
  - Objects are not defined just by the current attribute values.
  - Objects have a continued existence.

Interfaces

- An object has a *public* interface:
  a list of messages than will be accepted (from any other object).
- Interface will also define message arguments.
- Attributes can be included in a interface.
  - This declares an object attribute as available for inspection and possible modification.
  - This precludes redeclaration of an attribute without rewriting the code which accesses the attribute.
- An object can have a *private* interface:
  defines message that it can send itself, and private attributes.
  (The object can also use the public interface.)
- Sometimes there is an intermediate (between public and private) interface available to selected other objects.

Classes

- Don't want to fully define object every time it is created.
- A *class* is a convenient way of describing objects with similar characteristics:
  - Defines Attributes, Messages etc
  - An object is an *instance* of a class.
  - We create an object by *instantiating* a class.
- A class *method* describes how a message is responded to.
- An object has it's own attributes, but "shares" the class methods with other objects of the same class.
- Class attributes are also possible (C++ `static` attribute).
- The semantics of object creation varies between languages:
  - C++/Java must include constructor method
  - Others: possible to define initial values in class definition.

UML notation for Classes & Objects

- Objects

:Class name	object name : Class name

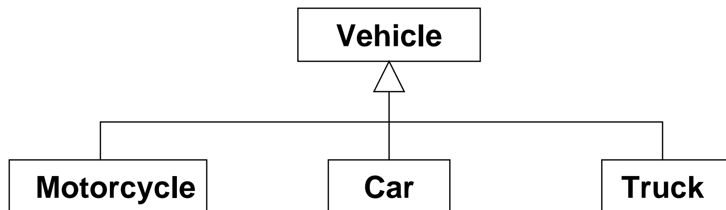
  Object diagrams are not commonly used, but can help during requirements analysis because they provide a physical model, unlike class diagrams.

- Classes. Typically some attributes and methods are omitted in early stages of modelling.

Class name

Class Name
atribute:type=initial value
operation(args):return type

Inheritance/Generalisation



- Inheritance saves effort and possible error by allowing a *subclass* to use methods and attributes defined in the *superclass*.
- The subclass can redefine any superclass properties.
- The subclass can define new properties.
- Car *inherits from* / *is a specialisation of* / *is a derived class (subclass) of* Vehicle.
- Vehicle *is a generalisation of* / *is a base class (superclass) of* Car.

A small example

- From Pooley & Stevens *Using UML*
- Introduction to class (object relationship) diagrams
- How to identify classes
- Developing relationships between classes
- Consider a simple University library

The library contains books and journals. It may have several copies of a given book. Some of the books are for short term loan only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but staff can borrow up to 12 items. Only staff can borrow journals.

The system must keep track of when items are borrowed and returned, enforcing the rules above.

Identifying classes

- This is perhaps the most important single step in OO modelling.
- Key method: *noun identification*
  - Start with a clear description.
  - Highlight nouns and noun phrases: i.e. identify *things*.
    This identifies candidate classes.
  - Only record the singular version of the noun, as we can always create collections of a class.
  - Examine this list and remove the spurious classes.

Noun Identification

The **library** contains **books** and **journals**. It may have several **copies of a given book**. Some of the books are for **short term loan** only. All other books may be borrowed by any **library member** for three **weeks**. **Members of the library** can normally borrow up to six **items** at a **time**, but **staff** can borrow up to 12 items. Only staff can borrow journals.

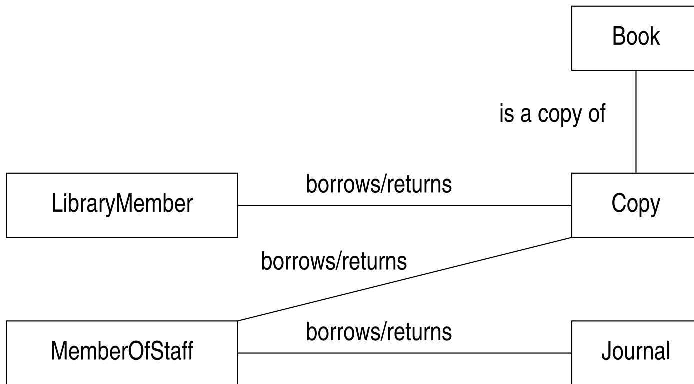
The **system** must keep track of when items are borrowed and returned, enforcing the **rules** above.

Rejecting classes

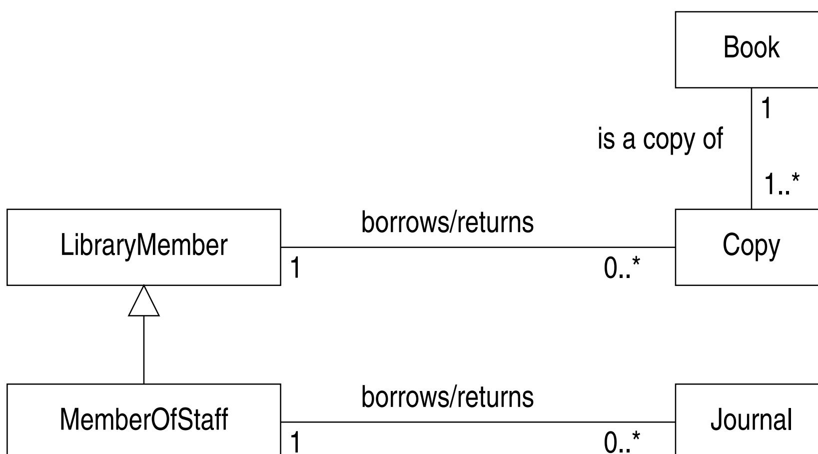
- ✗ library — outside the scope of our system
- ✓ book
- ✓ journal
- ✓ copy of a book
- ✗ short term loan — an event not a thing
- ✓ library member
- ✗ week — time; not a thing
- ✗ member of the library — redundant
- ✗ item — too vague; use book or journal instead
- ✗ time — not a thing
- ✗ system — a general term, not descriptive of our problem
- ✗ rule — a general term, not descriptive of our problem

Look for associations between the selected classes

- "**copies** of a given **book**"
- "**books** may be borrowed by any **library member**"
  – note that library member can include staff
- "**staff** can borrow **journals**"

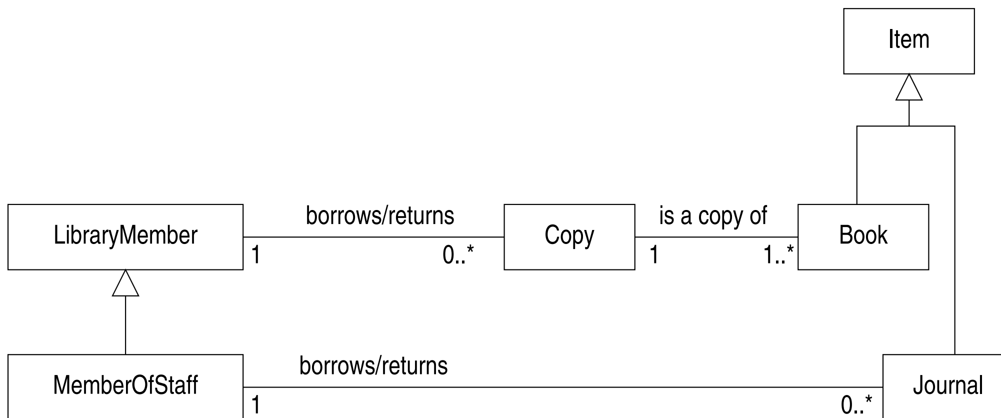


Add multiplicity; look for generalisations



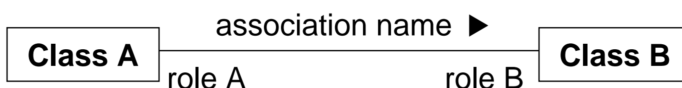
Further possible generalisation?

Item is not associated with another class — it would hold common attributes and methods. It is an implementation convenience not a feature of the library.



Associations

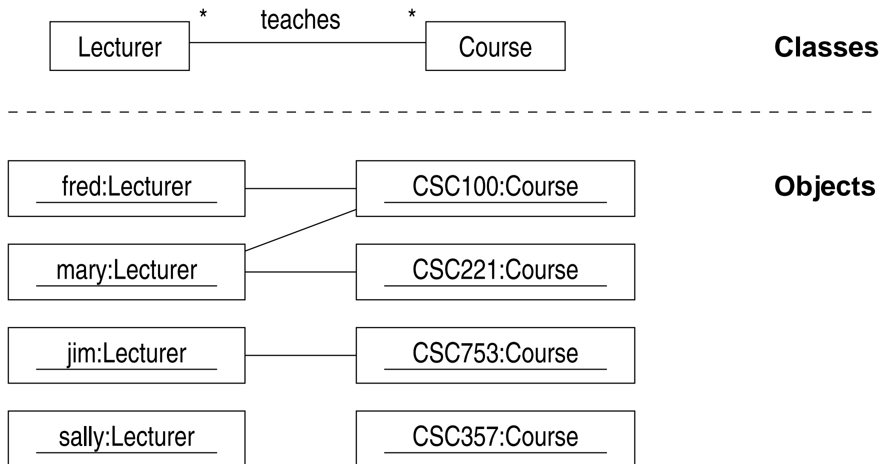
- How are objects of two or more classes related?
- An association connects classes and represents a possible link between a pair of objects at run time.
- Look for verbs and verb phrases: e.g. *contains, works for, is married to*, etc.



- Role names are optional.
- Solid triangle shows direction of association. In example above if the association was *works for*, then an object of Class A would work for an object of Class B.

Links and Associations

- An association connects classes and represents a possible link between a pair of objects at run time.
- a link is an instance of an association



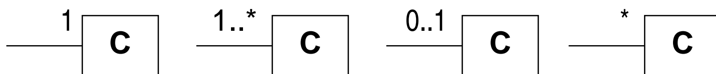
Associations ...

If classes A and B are related, then we expect that

- an object of class A could send a message to an object of class B
- an object of class A could create an object of class B
- an object of class A could receive a message from an object of class B
- an implementation of class A could have an attribute whose value is an object (or objects) of class B)

Multiplicity

- How many objects of Class A are associated with an object of class B? (And vice versa.)
- Very important information:
  – for understanding the system being modelled
  – for implementing a software version of the model
- UML notation



- standard syntax $lower$ .. $upper$, where $upper \geq lower$
  - integer values: e.g. 0..1, 0..10, 1..5, 8..8
  - $upper$ may also be *; e.g. 1..*
  - if $lower = upper$, use just upper bound; e.g. 8..8 $\equiv$ 8
  - can use * as shorthand for "0..*"

Attributes

- Identify attributes of an object: look for possessive phrases (population of) and enumerations (ascending or descending).
- Attributes will not always be obvious — can be added later.
- Don't use derived attributes like *age* if *dateOfBirth* is present.
- Don't use a (redundant) attribute when an association can provide the information:
  - No need for a *numberOfCopies* attribute in *Book*
  - You may well add such an attribute for efficiency reasons at implementation time.

Operations (Methods)

What operations are needed?

- Ask:
  - What "actions" could/does this object perform?
  - What operations/processes does the object participate in?
  - What messages could be sent to the object?
  - If the object could speak, what could it tell us?
    What could we ask it to do?
- Operations usually change the state of the object.
  State = attribute values + links to other objects
- Need operations to set and report the local state (attributes).
  - attributes are usually not (& should not be) directly available

Generalisation

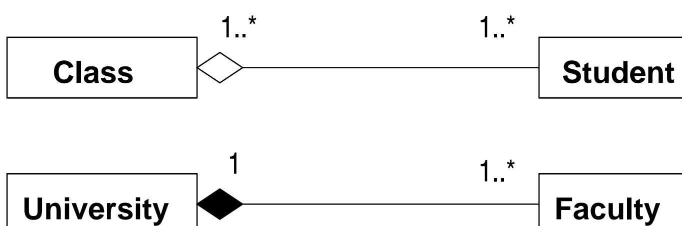
- Generalisation is a relationship between classes.
- An object of a specialised class can be used wherever an object of a more general class is expected; a general object *cannot* be used in place of a specialised one.
- The more specialised object should *perform a similar operation* to the general object when passed the same message.
- To check if a generalisation relationship exists: if B is a specialisation of A then we can say *every B is a kind of A*.
  – or more precisely: *every instance of B is an instance of A*

Generalisation ...

- Inheritance is the mechanism which implements generalisation.
- Problem: changes to superclass may affect subclass behaviour (unless subclass redefines changed method).
- Superclass is (from a programming point of view) independent of subclass so it is hard to predict effects of changing the superclass.
- We may wish to invent base classes to simplify programming: e.g. Book and Journal derived from Item.
- Use generalisation with care — many object models need not contain generalisation.

Aggregation & Composition

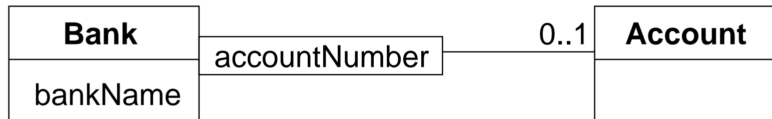
- special case of association, where one object *is comprised of* many members of another
- aggregates have independent lifetimes
  – Student exists after class has been completed
- "part" of the composition cannot outlive the "whole"
  – A faculty cannot survive if a University closes
- association name is optional: assume *is a part/member of*
- The multiplicity of the composition owner must be 1 or 0..1



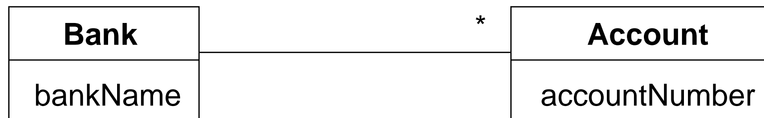
Qualified association

- noting qualified associations is not mandatory
- a qualified association can add precision to a model
- a qualified association can reduce multiplicity to 1
- qualified and non-qualified associations are implemented in the same way

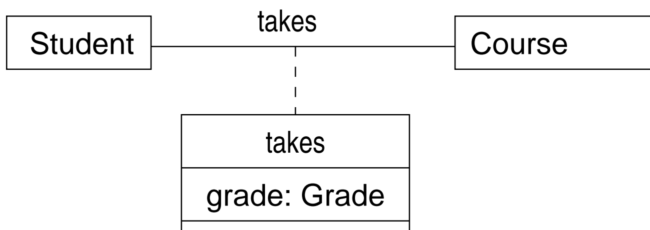
Qualified



Not qualified

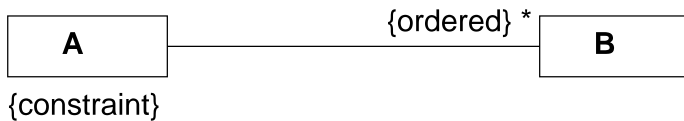


Association class



- We can't store grade in the Student or Course as grade is not a unique property or either.
- The marks are a consequence of the "taking" of the course, which involves both Student and Course.
- Class and association have the same name

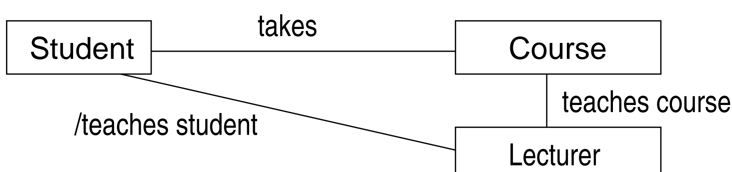
Ordering and Class constraints



- Constraints place restrictions on the way objects can be linked.
- (Multiplicity annotations are a form of constraint.)
- Constraints appear inside {...}.
- {Ordered} says the objects are stored in an ordered collection.
- General constraints can be attached to classes — perhaps to define relationship between attributes.

Derived association

- An association can often be deduced from a pair of associations.
- If shown, should be marked as a *derived association*
- Derived associations are sometimes implemented for efficiency reasons.
- Note: *teaches student* can be derived from *teaches course* and *takes* but we cannot deduce *teaches course* from *teaches student* and *takes*.



Navigability

- This is an *implementation* issue.
- Is is possible for o1:class A to send message to o2:class B?
- Implementation: does class A have an attribute which contains a reference to one or more class B objects?
- The arrow shows that Course contains an attribute which refers to a list of Students.
- Any Course object can find out the names of all the students.
- A Student object cannot (easily) find out the name of all courses taken. (This would require an exhaustive search of Course objects).

Implementation

- Class diagrams are conceptual view.
- Implementation: translate class diagrams to Class definitions
- Translation depends upon language.
- Typically:
  - attributes → class variables
  - methods → member functions
  - associations → pointers or references

Key points

- Object models many be more stable than functional models.
- Not all systems are well suited to object modelling.
- Objects are things with state, behaviour, and identity.
- A class is a convenient way of describing similar objects.
- Inheritance implements code reuse between classes.
- Use noun identification to discover possible classes.
- Classes are related by associations.
- Classes may be related by "is a kind of" generalisation.
- Aggregation and composition are kinds of association.

Key points...

- Qualified association adds precision.
- Association classes add attributes to associations.
- Constraints can be added to classes and associations.
- Derived associations should be annotated with /.
- Navigation shows reachability — used for implementation.