Class diagrams and architectural design

Perdita Stevens

School of Informatics
University of Edinburgh

More unified modelling language

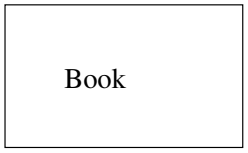
We saw use case diagrams, which are part of UML, the Unified Modelling Language.

(What's that, and why?)

A class diagram crept into last lecture...

Now we look properly at the basic features of class diagrams.

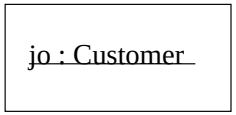
A class



A class as design entity is an example of a **model element**: the rectangle and text form an example of a corresponding **presentation element**.

UML explicitly separates concerns of actual symbols used vs meaning.

An object



This pattern generalises: always show an instance of a classifier using the same symbol as for the classifier, labelled instanceName : classifierName.

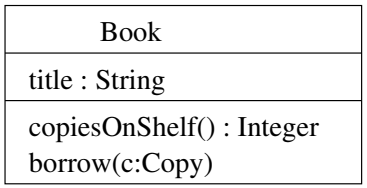
Classifiers and instances

An aspect of the UML metamodel that it's helpful to understand up front.

An **instance** is to a **classifier** as an object is to a class: instance and classifier are more general terms.

UML defines many different classifiers. In fact, UseCase and Actor are classifiers.

Showing attributes and operations



Syntax for signature of operations (argument and return types) adaptable for different PLs. May be omitted

Compartments

We saw the standard:

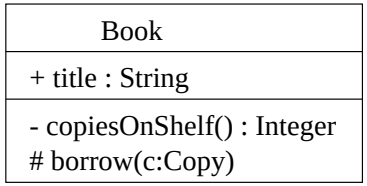
- a compartment for attributes
- a compartment for operations, below it

They can be suppressed in diagrams.

They are omitted if empty.

You can have extra compartments labelled for other purposes, e.g., responsibilities of the class...

Visibility

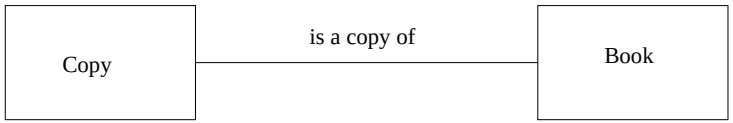


Can show whether an attribute or operation is

- public (visible from everywhere) with $+$
- private (visible only from inside objects of this class) with $-$

(Or protected ($\#$), package ($\sim$) or other language dependent visibility.)

Association between classes

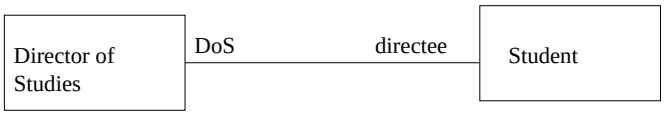


This generalises: association between classifiers is always shown using a plain line. (Recall the associations between actors and use cases!)

An instance of an association connects objects (e.g. Copy 3 of War and Peace with War and Peace).

An **object diagram** contains objects and links: occasionally useful.

Rolenames on associations



Can show the role that one object plays to the other.

Useful when documenting the class: e.g. a *class invariant* for DirectorOfStudies could refer to the associated Student objects as self.directee (a set, if there can be more than one).

Can use visibility notation + − etc on role names too.

Class invariants

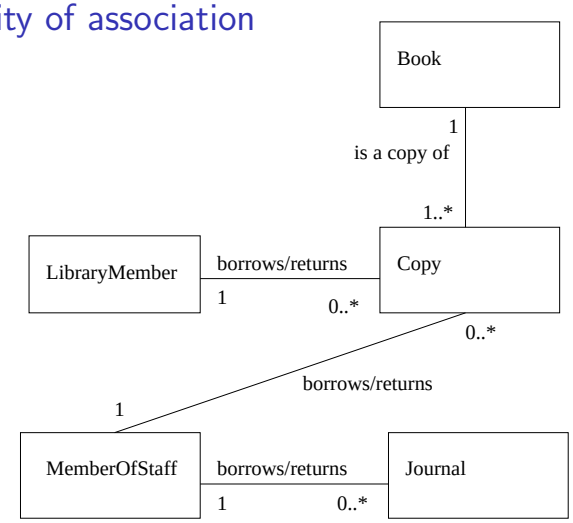
A class invariant is a statement which is supposed to be true of every object of the class, all the time - a "sanity check".

Very useful to make these explicit. Can be included as comments on class diagrams, and in code.

May be formal, e.g. `x + y = z`, or informal, e.g. "the attribute `docstring` describes the action of the button in concise English".

If formal, it can be useful to have class invariants automatically checked.

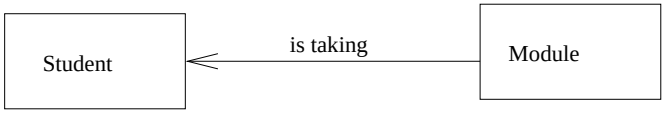
Multiplicity of association



Commas for alternatives, *two* dots for ranges, * for unknown number. E.g. each Copy is a copy of exactly one Book; there must be at least one Copy of every Book.

Navigability

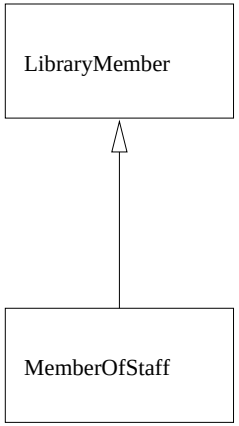
Adding an arrow at the end of an association shows that some object of the class at one end can access some object of the class at the other end, e.g. to send a message.



Crucial to understanding the coupling of the system. NB direction of navigability has nothing to do with direction in which you read the association name.

We'll return to this in the context of interactions and interaction diagrams.

Generalisation



This generalises: generalisation between classifiers is always shown using this arrow.

Usually, but not necessarily, corresponds to implementation with inheritance.

Abstract operations and classes

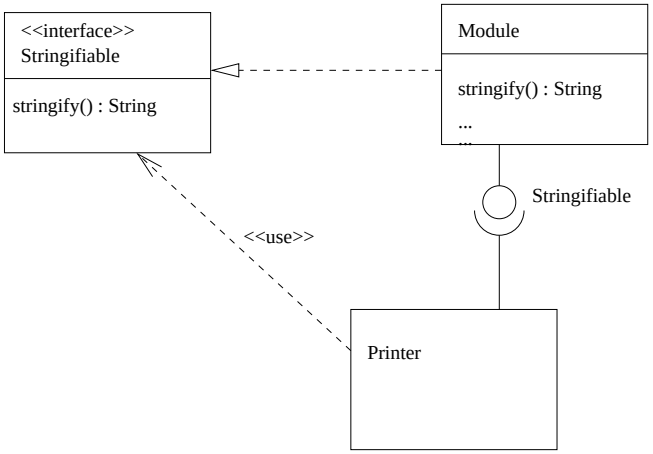
An operation of a class is abstract if the class provides no implementation for it: thus, it is only useful if a subclass provides the implementation.

A class which cannot be instantiated directly – for example, because it has at least one abstract operation – is also called abstract.

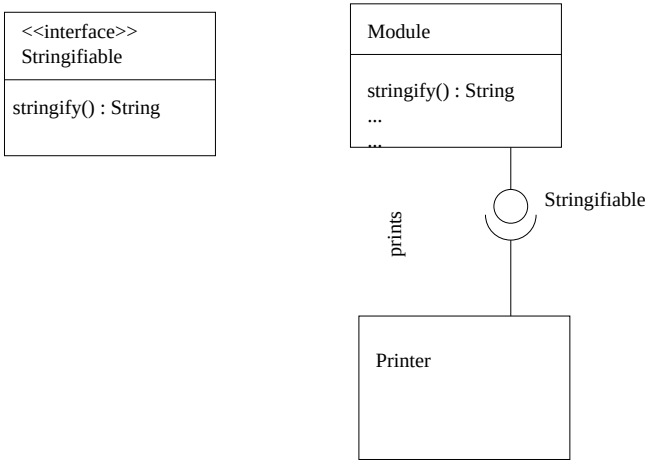
Can show *abstract* operation or class using italics for the name, and/or using the *property* {*abstract*}.

Interfaces

In UML an interface is just a collection of operations.



Simpler diagram: WRITE ONCE



Many things other than classes can realise interfaces: can use the lollipop symbol on e.g. components, actors.

Architecture

Identifying objects and classes

Simplest and best: look for noun phrases in the system description!

Then abandon things which are:

- redundant
- outside scope
- vague
- attributes
- operations and events
- implementation classes.

(May need to add some back later, especially implementation classes: point is to avoid incorporating premature design decisions into your conceptual level model.)

Similarly, can use verb phrases to identify operations.

So what is architecture?

Many things to many people.

> The way that components work together

More precisely, an architectural decision is a decision which affects how components work together.

Includes decisions about the high level structure of the system – what you probably first think of as "architecture".

Pervasive, hence hard to change. Indeed an alternative definition is "what stays the same" as the system develops and between related systems (Stuart Anderson).

Classic structural view

Architecture specifies:

- what are the components?
  Looked at another way, where shall we put the encapsulation barriers? Which decisions do we want to hide inside components, so that we can change them without affecting the rest of the system?
- what are the connectors?
  Looked at another way, how and what do the components really need to communicate? E.g., what should be in the interfaces, or what protocol should be used?

The component and connector view of architecture is due to Mary Shaw and David Garlan – spawned specialist architectural description languages, and influenced UML2.0, but beyond scope of this course.

Reading

Suggested: Browse SEI's collection of architecture definitions at `http://www.sei.cmu.edu/architecture/definitions.html`

Suggested: (architecture) Somerville ch 11-13

Suggested: (class diagrams) Stevens ch 5.

More examples of architectural decisions

- what language and/or component standard are we using? (C++, Java, CORBA, DCOM, JavaBeans...)
- what conventions do components have about error handling?
- what framework does the product use?

Clean architecture helps get reuse of components.

Indeed by some definitions parts of the architecture can *be* components – frameworks, product-line architectures etc.