Introduction to object orientation

Perdita Stevens, University of Edinburgh

March 2010

What is an object?

Something you can do things to.

An object has state, behaviour and identity.

State can affect behaviour.

Behaviour can affect state.

Objects communicate by sending messages: the behaviour of an object on receipt of a message is "up to the object".

From objects to classes

Typical systems involve many very similar objects.

A class defines the structure and behaviour of similar objects.

In the languages we'll consider a class C defines:

- C's interface, i.e. what messages any object of class C understands, and some specification of the object's behaviour on receipt of a message
- C's implementation, i.e. some code that implements that behaviour, some state (data) that is used.

Just as for individual objects:

Interface: public

Implementation: secret

Why use objects?

Original hope/hype was to

- reduce cost (time, in development and maintenance)
- improve quality

by

- making the design "naturally" match the domain
- achieving high levels of reuse

The secret of OO

When OO achieves these aims, it is usually *not* because of any of the things that are specific to OO. Especially, reuse is a *much* less important benefit than expected.

Instead it's because

- the object oriented approach takes modularity, encapsulation and abstraction as fundamental
- and OOPLs make them (comparatively!) easy, the obvious way.

OO is a religion: it's the *people* who become oriented towards objects.

"Thinking objects" can be done in any language.

Modularity: why?

"The task of the software development team is to engineer the illusion of simplicity." (Booch)

And it needs engineering:

"The complexity of software is an essential property, not an accidental one" (Brooks)

Already in 1965, Dijkstra was pointing out that real systems are too complex for one person to understand as a whole. They have not got simpler.

Separation of concerns

To accommodate human cognitive limitations and yet make progress, we need some way to limit what needs to be understood.

Dijkstra actually had in mind "correctness", "efficiency", "scalability" etc. as the concerns that needed to be separated.

Focus of modularity in software design is a more structural notion of concern:

allow a developer to understand how one part of a system works well enough to write/improve it, without needing to understand the whole system that well.

What is a module?

A piece of code with a well-defined *interface* which *encapsulates* some information about the code. E.g.

- subroutine
- library function
- class
- component
- subsystem

(Later we'll consider getting the *right* modules with the *right* interfaces: but observe that almost any modularity is better than none.)

Defns from Booch (OOD with Apps)

encapsulation The process of hiding all the details of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the implementation of its methods. The terms *encapsulation* and *information hiding* are usually interchangeable.

abstraction The essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply-defined conceptual boundaries relative to the perspective of the viewer; the process of focussing upon the essential characteristics of an object.

Briefer characterisation

Abstraction limits how much you *need to* know about a module

Encapsulation limits how much you *can* know about a module (...in some sense...)

Q: how exactly can not being able to know something be a benefit?

A: if it changes, you can be sure you weren't relying on it

What's a good module?

The problem was: Real systems are too complex for one person to understand.

So aim of modules is to increase

- the understandability of the code, and hence
- the probability of correctness, and
- the ease of maintenance.

Ultimately understandability is determined in the human's head.

Modules should have: meaning; high *cohesion*; low *coupling*.

Modules in OO

In OO systems the modules we think most about are the classes.

Classes are unusual modules though: they are instantiable!

The data of our program appears inside the instances of its modules.

Classes are *abstractions* that should capture the key features of the domain – they *encapsulate* state and the behaviour that alters that state together.

Key domain abstractions

Basic idea: a class for each important *thing* in the problem domain.

1. This is more stable than the requirements for a particular system
2. and the resulting structure is easier for people to understand.

Starting point: take some text about the system, look at the noun phrases.

Only later think about the behaviour required for this particular application.

Encapsulation example

Format of data is a problem! In OO it's the classic example of something that should normally be hidden.

Even hide knowledge about whether or not it exists! *information hiding*

E.g. suppose an application refers to a Point in 2D space, and can get both its Cartesian co-ordinates (x, y) and its polar co-ordinates (r, θ).

Do we store both sets, or do we store one and use that one to calculate the other on demand?

The outside world shouldn't know or care.

Relationships between classes

An object can't do much on its own – it needs to cooperate with other objects.

Since the behaviour of an object is defined in its class, this gives us relationships between classes.

These are common and relate individual objects (and hence, their classes):

- "sends a message to" (or specifically, "sends message XYZ to")
- "contains" (special case of the above!)

"is a kind of" is rarer and relates only classes.