More on object orientation

Perdita Stevens, University of Edinburgh

April 2010

Agenda

- Discussion of last time's exercises
- More on abstraction and encapsulation: common pitfalls and idioms to avoid them
- Overview of current OO languages, and languages with OO features
- Conventions for designing using objects in C

Exercises

Please tick to show how you got on...

Questions, comments, problems?

Meta-comments: too easy/hard/short/long?

Aims for design

1. Flexibility (as opposed to rigidity)
2. Robustness (as opposed to fragility)
3. Reusability (as opposed to immobility)

Robert C. Martin wrote a great sequence of articles for the C++ report, on which I draw heavily in the next section.

See `http://www.objectmentor.com`.

The Open-Closed principle

"Software entities like classes, modules and functions should be open for extension but closed for modifications."

That is, you should be able to add new functionality for clients that need it without needing to modify old stuff, and thus interfere with clients that were already happy with the old version.

OCP classic example

You have 3 kinds of some entity already, and there may be more in future. You could:

1. Write one class in which each method contains a case statement with a case for each kind of entity.
2. Define a class for the entity, with a subclass for each kind of entity.

2. satisfies the OCP better, because adding a new kind of entity just means writing a new subclass, without modifying any existing code.

Liskov substitution principle

“What is wanted here is something like the following substitution property:

If

for each object o1 of type S

there is an object o2 of type T

such that

for all programs P defined in terms of T,

the behavior of P is unchanged when o1 is substituted for o2,

then S is a subtype of T. ”

Other ways to express LSP

Informally: if no client, that's never heard of subclass S, can tell anything's changed when you give it an object of subclass S, then creating S won't have broken any client, so all is well.

"What you don't know won't hurt you."

Alternatively:

Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be true for objects y of type S where S is a subtype of T.

Is LSP too strong?

All except the last version, yes. They make LSP independent of any contract for T's behaviour, so they force S to preserve *all* of T's behaviour – even things nobody ought to care about, e.g. the order in which elements come out of an unordered set.

(Note the subtlety of the last formulation: provable vs true.)

Design by contract

says: there should be explicit contracts attached to the responsibilities a class exists to fulfill.

A method has:

- a precondition – this must be true when the method is invoked, or all bets are off
- a postcondition – the method promises to ensure this, provided its precondition was met

A class has:

an invariant, which it must maintain.

Subclassing in DbyC

In Design by Contract terms, a subtype is a subcontractor to its superclass.

It must

Demand no more; promise no less.

OK to strengthen postconditions;

OK to weaken preconditions;

but never the other way round.

Contracts vs no contracts

Explicit, formal contracts are great, and in principle let you check that subclassing is safe.

But they are a pain to write and maintain.

Worth thinking about and documenting at least informally, at least for crucial classes.

(Java Modeling Language...)

Common pitfall: high to low dependencies

You have your beautiful class C that offers high level services to its clients. They can't tell what low-level gubbins it depends on: that's all encapsulated.

So far so good.... but

1. when the low-level gubbins changes, clients are affected nonetheless because your class has to be rewritten;
2. you can't reuse C in a new application unless exactly the same low-level gubbins is available.

Dependency inversion (Robert C. Martin)

(Slightly confusing name for a very basic principle)

“A. High-level modules should not depend on low-level modules. Both should depend on abstractions. B. Abstractions should not depend upon details. Details should depend upon abstractions.”

High-level module depends upon an abstract interface capturing what it needs from a low-level module.

Low-level module depends on that interace too, as it implements it.

Law of Demeter

in response to a message m, an object O should send messages *only* to the following objects:

1. O itself
2. objects which are sent as arguments to the message m
3. objects which O creates as part of its reaction to m
4. objects which are *directly* accessible from O, that is, using values of attributes of O.

OOPLs

A biased selection:

Pure OO: Simula, Smalltalk

Less pure OO: Java, Python, $C\sharp$,...

Extensions of non-OOPLs: Objective C, C++, Perl5+,...

Others: CLOS, Oberon, Self, Scala...

Most relevant to you

- Java
- C++
- C♯

Bonus: techniques for object orientation in C!

Common features

Java, C++ and C♯ are very similar:

- imperative (“do this; then do that” as opposed to e.g. “here’s a function definition; here’s another”)
- class-based (as opposed to, say, prototype-based)
- statically typed (mostly)[1]

[1] Do not confuse static typing with static binding! A language is statically typed in as far as the compiler will find the type errors.

In the beginning there was C

and then Bjarne Stroustrup came along.

C++ permits OO programming without sacrificing efficiency.

But Bjarne himself said:

C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off

Genesis of C++

1980 - 1983 “C with classes” (Bjarne Stroustrup)

1983/4 “C with classes” redesigned and renamed C++ (virtual fns and operator overloading)

1985 C++ generally available

1989 C++ standardisation starts: X3J16 committee formed

1998 ISO/IEC 14882, Standard for the C++ Programming Language

...

Guiding principles 1

1. Retain C as a (near) subset.
2. *"Simplicity was an important design criterion: where there was a choice between simplifying the language definition and simplifying the compiler, the former was chosen"*
3. *"Features that would incur run-time or memory overhead even when not used were avoided"*

Quotes from Stroustrup, *The C++ Programming Language*.

Guiding principles 2

"C++ type-checking and data-hiding features rely on compile-time analysis of programs to prevent accidental corruption of data. They do not provide secrecy or protection against someone who is deliberately breaking the rules."

Stroustrup ibid, my emphasis.

How C++ adds OO features to C

New construct "class". Classes are essentially structs (but with default access control private).

Multiple inheritance.

Dynamic binding ("proper inheritance") possible, but so is static binding.

Lots of tightening up (e.g. enums more like real types: can't do arithmetic on them).

Memory management still explicit (new and delete).

Resources for learning more about C++: books

Bibles:

Bjarne Stroustrup, *The C++ Programming Language*
Margaret Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*

Bjarne Stroustrup, *The Design and Evolution of C++*

My favourite:

James O. Coplien, *Advanced C++ Programming Styles and Idioms*

Other people's favourites:

Scott Meyers, *Effective C++ : 50 Specific Ways to Improve Your Programs and Designs*

Scott Meyers, *More effective C++*

Java

Sun: James Gosling. Originated in 1991 as Oak targeted at programming set-top boxes.

Java 1.0 released 1995, i.e. beginning of the web era.

C++ without the C-compatability-led compromises. Other aims: robustness, security, portability, support for threads...

C$\sharp$

Microsoft's answer to Java, version 1.0 late 2001/early 2002.

Opinions differ about how novel it is... lots of jokes about how close it is to Java, but Anders Hejlsberg, the main language designer, says it is closer to C++.

In recent years influences have flowed both ways.

Shooting yourself in the foot 1

C You shoot yourself in the foot.

C++

You accidently create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical assistance is impossible since you can't tell which are bitwise copies and which are just pointing at others and saying "That's me, over there."

Shooting yourself in the foot in Java

You locate the Gun class, but discover that the Bullet class is abstract, so you extend it and write the missing part of the implementation. Then you implement the ShootAble interface for your foot, and recompile the Foot class. The interface lets the bullet call the doDamage method on the Foot, so the Foot can damage itself in the most effective way. Now you run the program, and call the doShoot method on the instance of the Gun class. First the Gun creates an instance of Bullet, which calls the doFire method on the Gun. The Gun calls the hit(Bullet) method on the Foot, and the instance of Bullet is passed to the Foot. But this causes an IllegalHitByBullet exception to be thrown, and you die.

or

Anybody can shoot themselves in any kind of foot with any kind of gun that's loaded with any kind of bullet. However, they can only do it if their system is running a compatible Java Runtime Environment.

Shooting yourself in the foot in C♯

Copy how Java shot itself in the foot. Explain how you did it better.

or

After searching for five hours, you find three seperate Foot and Gun implementations on the MSDN. None of them works as described.

or

Of course you can shoot yourself in the foot as long as you declare the code unsafe

or

Microsoft shoots you in the foot, then declares that this is the standard way for people to shoot themselves in their feet.

Shooting yourself in the foot in Smalltalk

You spend so much time playing with the graphics and windowing system that your boss shoots you in the foot, takes away your workstation and makes you develop in COBOL on a character terminal.

or

You send the message shoot to gun, with selectors bullet and myFoot. A window pops up saying Gunpowder doesNotUnderstand: spark. After several fruitless hours spent browsing the methods for Trigger, FiringPin and IdealGas, you take the easy way out and create ShotFoot, a subclass of Foot with an additional instance variable bulletHole.

Sources for that digression

There are many, of course. I liked:

`http://www-users.cs.york.ac.uk/susan/joke/foot.htm`

`http://digg.com/programming/How_to_Shoot_Yourself_in_the_Foot_in_Any_Programming_Language`

`http://www.fullduplex.org/humor/2006/10/how-to-shoot-yourself-in-the-foot-in-any-programming-language/`

Doing OO in C

... sort of a bit...

Recall: the main benefit of an OO approach is that it makes good design – low coupling, high cohesion – easy. E.g. by

packaging data together that corresponds to the same domain concept

encapsulating it

with a well-defined interface

so that clients depend on the interface, not the implementation

How to do it

Defining a set of "objects":

```
typedef struct {
  uint8 *data_storage;
  ...
} Data_buffer;
```

Then "methods" are functions that take a pointer to a `Data_buffer` as their first argument.

Don't access data directly – use accessor functions.

Use .h files as interfaces that can have multiple implementations.