Chapter 9: Data Abstraction and Object Orientation

July 2, 2015

Three fundamental concepts to object-oriented programming

- Encapsulation
- Inheritance
- Dynamic method binding

Object-oriented programming

- What we would like from any module-based approach: ** Reduce conceptual load by minimizing the level of detail needed at any one point ** Fault containment, so that programmers don't misuse a component, and limiting where a component might be used ** Independence: it would be nice to be able to be agnostic with respect to the actual implementation; if we later change out one implementation for another, then it should not have any evident impact on code using the module

Object-orientation

- However, just using modules alone doesn't seem to be adequate; when you want to extend functionality or replace some method, module syntax alone doesn't seem to have any convenient way of expressing these minor modifications.

Refinement

- "Object-orientation can be seen as an attempt to enhance opportunites for code reuse by making it easy to define new abstractions as *extensions* or *refinements* of existing abstractions." [page 451]

Derivation

- In an object-oriented language, one of the more powerful ideas is that the idea of a *derived* class, which *inherits* the fields and methods of its parent class, and which can be augmented, hidden, or supplanted by the programmer with other functionality.

Encapsulation and inheritance

- Modules: some languages allow a module to be split into the declaration and definitions needed for outside consumers (often called a "header"), and the internal bits needed for the implementation (generally called the "body").
- As the book points out, it is common for a method to utilize a "self" (or "this" or "current") that allows the module to refer to the calling instance variable; this generally can be regarded as turning a call of the form var->method(x) to method(var,x).

Modules and types

- It has been common for languages to conflate modules and types.
- Here's an introduction to Haskell's rules for modules, for instance.

Initialization and finalization

- Generally, initialization in an object-oriented paradigm has been called a "constructor"; some languages have also allowed for "destructors", though this is comparatively rare.
- Lots of issues with constructors can arise: conventions on passing arguments and their meaning; execution order in deeply structured (or even multiply inherited!) objects that have many levels of constructors; garbage collection for languages that have no explicit destructors...

Dynamic method binding and virtual methods

- Consider the situation where each of the following derived classes have redefined a method called print_classes():

```
class person { ...
class student : public person { ...
class professor : public person { ...

student s;
professor p;

person *x = &s;
person *y = &p;

x->print_classes();
y->print_classes();
```

Smalltalk

- Smalltalk is where the ideas for object orientation were first fleshed out, and in many ways is the canonical exemplar of object orientation, using only dynamic type-checking and dynamic method lookup. This imposes speed penalties that are not present in languages that allow the compiler to do more of the work.