Languages as a Tool for Software Engineering
Piercing the Mysteries of Object-Oriented Programming

Lecture 25

Lecture Outline

- What is the relationship of PL and productivity?
  - Search for the silver bullet
  - Fads: Systematic, nth Generation, Object Oriented, …
  - Side issues: batch vs. time-sharing vs. graphic interaction
- OO
  - Message passing vs Generic functions
  - The CLOS model
  - Implementation

Programmer Productivity is Low

If a programmer produces on average only 5-10 lines of correct code per day, and it is irrelevant whether the code is in assembler or C or …. Then the programmer using assembler is less productive.

What can we do? What is the silver bullet?

Various proposals for higher productivity

Structured Requirements etc
Formal methods
Team dynamics

Hardware and software support (above the language level)

Testing

Rapid turnaround

Higher Level Languages and Productivity

Use more concise programming languages?
Verify correctness
Automatic test generation

Fire the below-average programmers?
Hire more programmers?
(Re-)educate the programmers in mathematics and logic?
Genetic engineering: clone the best programmers?
Smart pills? ☺
BETTER SUPPORT FOR THE SOFTWARE "LIFE CYCLE"
    maintenance
    portability

Change the PL "paradigm" to model the application so that the translation from real-world to virtual program world is easier.

Object Oriented Programming tries to model the world

Is this really a language issue? Yes if you are using a language that interferes; No if you can just add these concepts to the language.

Lisp supports many forms of language, and at least 4 OO additions have been proposed and implemented (Flavors, Loops, NewFlavors, CLOS).

Lisp allows one to use the "meta object protocol" to define variations on its object system.

Review of Object Oriented Programming Ideas

Base some or all of your language about data objects: running this down to the "ground"….
An instance or object is a block of memory that can hold some data in slots, and is associated with some class of similar objects.

There is a way of finding out the class (es) of any object.

Associated with the class is a collection of methods for manipulating objects, referring to their "slots" and other methods. Sometimes classes have data too.

Typically there is a class hierarchy, so that an object is a member of a class, but also a member of its "super" class. Methods can also be attached to the super-classes.

Objects – Classes – Hierarchy-Inheritance-Methods

Two common models for thinking about it

The recent first wave was to think of "message passing" … for example, to find the area of an object

tell obj area

If obj has a method for area, or a superclass has a method, then we "set self:= obj" and then run the method.

Extra arguments are possible like

tell obj move 10

X := 4 + 5 ;; 4 has a + method, aux argument 5. X has a := method.. Etc [this is like Smalltalk)

Limits of the message model

This gets pretty boring, since EVERYTHING is

tell RECEIVER MESSAGE EXTRA_ARGS

So why not leave out the "tell" and put the "active part" first; all we need is () to put it in lisp syntax:

(MESSAGE RECEIVER EXTRA_ARGS)

And now you see that the OO message-passing idea says in effect that we are calling a function … which is the meaning of MESSAGE, but that its meaning is modified by its first argument (only). (+ 4 5) is 'find the + method of 4, apply to 5.'

An obvious generalization: we could allow all the arguments to participate in modifying the meaning of MESSAGE, which brings us to the next model: generic functions.

Generic functions are another cut through the same programming text

GROUPED BY METHOD

```
(defmethod area((z rectangle))  ;; length X width..
(defmethod area ((z circle))    ;; pi X r^2
(defmethod area  ((z triangle)) .....
```

GROUPED BY CLASS

```
 class rectangle
  method area ....
class circle
  method area ...
```

Generic functions consist of "all the methods with the same name"

discriminated by argument types

```
(defmethod area  ((z rectangle))  ;; length X width..
(defmethod area  ((z circle))     ;; pi X r^2
(defmethod area  ((z triangle)) .....
```

Textually, you could write these defmethods far apart in the program, if you chose to do so.

CL Generic functions can be generic with respect to each argument's type*

(defmethod add((z rational)(w rational)) …

(defmethod add((z doublefloat)(w rational)) …

(defmethod add((z rational)(w doublefloat)) …

(defmethod add((z interval)(w rational)) …

Etc.
We could program combinations of rational + interval + complex + float etc.

In general we still need to define the types, their slots, their inheritance

*kinds of objects. The Object system mirrors the built-in types in Lisp.

More general types can also be used for defmethod…

…This is how inheritance can be used effectively

(defmethod add((z number)(w number)) …

CLOS will use the MOST SPECIFIC method. Thus if number is a superclass of rational, doublefloat, complex, interval etc. this last method will be used as a backup (only) when a more specific pair of arguments cannot be found.

A clumsy example for CLOS

```
(defclass plane-figure() ()) ;no superclass, no slots

(defclass rectangle(plane-figure)(height width))

(setf r1 (make-instance 'rectangle))
(setf (slot-value r1 'height) 10)
(setf (slot-value r1 'width) 15)
(defmethod area ((r rectangle))(* (slot-value r 'width) (slot-value
r 'height)))

(area r1)
150
(area 'foo)
  error: no methods applicable
```

A neater way, declaring slots

```
(defclass plane-figure() ())

(defclass rect (plane-figure)
 ((height :accessor hi)
   (width  :accessor wi)))

(setf r1 (make-instance 'rect))
(setf (hi r1) 10)
(setf (wi r1) 15)
(defmethod area ((r rectangle))(* (wi r) (hi r)))

(area r1)
150
```

Even neater way, declaring initargs

```
(defclass plane-figure() ())

(defclass rect (plane-figure)
 ((height :accessor hi :initarg :hi)
   (width  :accessor wi :initarg :wi)))

(setf r1 (make-instance 'rect :hi 10 :wi 15))

(defmethod area ((r rect))(* (wi r) (hi r)))

(area r1)
150
```

Superclasses

Complex precedence rules for inheritance from multiple superclasses (single inheritance is often dictated by some language optimized for "efficiency")

CLOS allows multiple inheritance, e.g. from geometric shape as well as graphical representation

```
(defclass screen-circle (circle graphic) ….;
(setf sc (make-instance 'screen-circle …))

(radius sc)
(color sc)
```

Method combination

Complex instructions for use of methods / before, after, in-between. These are all plausible for various applications.

There may be zero or more applicable methods: all the arguments in the call come within the specializations of all of its parameters.

Zero= error; otherwise the most specific gets used.

But auxiliary methods can be invoked e.g. "do this first" or "do this afterward" or even "around" ..which can call the primary method via "call-next-method".

(Polite speaker example…)

Implementation Criteria

- Flexibility/ Generality is part of the CL design
  - Dynamic class definition: you can add methods or even slots to an object AFTER instantiation
- Efficiency should come out of the implementation: If you never need this dynamic facility, set fields in concrete as early as possible
- MOP "meta object protocol" provides tools for rolling your own version of object system. (Used by matlisp

General Techniques

- One way: Method/Function call is 2 steps: given the method, use a "case" on the argument type(s) to find the right version.
- Another way (not for CLOS), given the type of the receiver of the message, use a "case" on the method name to find the right method.
- Either way you have to deal with inheritance, when there is no immediate success in the search.
- It is possible to compile out flexibility at compile time or repeatedly "just-in-time" (CLOS recompiles classes on redefinition; other systems require recompiling everything.)

A simplified "How to do OO"

(ref. ANSI Common Lisp chapter 17)
YOU ARE ASSUMED TO HAVE READ THIS CHAPTER!!

It uses a simple "message passing" model. Starting here. Classes are almost the same as objs.

Each object obj is a hash table. If you want to find a method meth associated with that object, you compute (gethash meth obj). Thus tell obj meth becomes (funcall (gethash meth obj) obj)

We can define tell

```
(defun tell(ob m &rest args)
      (apply (gethash m ob) ob args)
```

That's it

A more elaborate "How to do OO"

Repeat from previous version:

Each object obj is a hash table. If you want to find a property prop associated with that object, you compute (gethash prop obj)

New part: inheritance

if there is no such property, then get obj's parent, under its :parent property, and look there, recursively if necessary. That is, you look at (gethash prop (gethash :parent obj)) call this recursive gethash rget

Tell is almost the same. Use rget

Tell obj message extra-args

is implemented by

```
(defun tell (obj message &rest args)
  (apply (rget message obj)  obj args))
```

It is assumed here that (rget message obj) will return a function

We could stop here but..

Let us try for multiple inheritance.

An object then does not have a single parent, but a list of parents. We must compute an order in which to visit the parents, and have rget use that.

When do you compute the list of parents?

Parent precedence list for multiple inheritance

Object A inherits from B and C, B inherits from D, C inherits from D. When do you look at D? Compute a precedence list.

When do you compute its parents?

(a) At compile time (efficient but inflexible).
(b) Every time you use rget and need to know where to look? (flexible but very inefficient)
(c) Every time the class hierarchy changes you recompute the :parents property for every object (inefficient if there are many objects)
(d) When you use rget, you ask, has the class hierarchy changed so as to affect this object? If so, redo :parents (almost as efficient as (a)).

Make this efficient

It pays to separate out classes and instances.

Associate methods with classes, so instances can be smaller.

Classes become more efficient

Instead of hashtables, use arrays. Compile away the "names" of methods the same way we compiled away the symbol table. For example, Method PRINT might be location 13 in the method array associated with the class of an object. Method X inherited from parent A is also given a slot in the array.

(a) Do as much as possible at compile time (efficient but inflexible).

(b) Instead of rget you are just doing an array ref.

(c) Make it impossible to change the class hierarchy unless you recompile.

Instances become more efficient

Instead of hashtables, use arrays. Compile away the "names" of instance variables the same way we compiled away the symbol table. For example, slot "radius" might be location 2 in the instance array associated with an object. Slot 0 might be a pointer to the class of this object. Look there for methods.

Object orientation has many variants

If this one is too simple, make it fancier. If this is too fancy or slow, make it simpler.

In some sense object orientation and first-class functions are different ways of viewing the same style of computation; languages with a flexible functional approach can implement objects easily. (CS61a OO was written in Scheme..)