CSE 341: Programming Languages

Autumn 2005

Lecture 27 — Advanced Issues in Object-Oriented Languages

Smalltalk Benefits and Drawbacks

We have a purely-OO dynamically-typed, class-based language:

- We can send any message to any object
- Subclassing and dynamic dispatch allow shared and specialized behavior.

This elegance leads to certain conveniences (good) and awkwardness (bad) . . .

- convenience: classes-are-objects makes “factories” trivial
- awkwardness: class of a class of a class . . .
- awkwardness: “fragile” superclasses
- multiple inheritance (not supported, although implementable in Smalltalk)
- multimethods (not supported, although they can be simulated)

Motivating the “Factory Pattern”

Consider a Java method using a Windows GUI to do some stuff:

```
void doStuff() {
   Frame f = new WindowsFrame(); // a subclass of Frame
   f.addButton(...);
   f.displayMessage(...);
   ...
}
```

And of course we have 100s of methods that build GUI objects in this way.

And now we want to be platform-independent (support Linux and Mac, which use different subclasses for each kind of GUI thing).

What can we do?

Options:

- Duplicate 100s of methods
- Pass a “platform” flag everywhere and use if-statements
- Like previous but put flag in a global scope
- Like previous but abstract if-statements to helper methods

Even with helper methods, the if-statements are very un-OO.

Using the "Factory Pattern"

An OO solution uses "object factories":

```
abstract class FrameFactory { Frame makeFrame(); }
class WindowsFrameFactory extends FrameFactory {
   Frame makeFrame() { return new WindowsFrame(); }
}
class LinuxFrameFactory extends FrameFactory {
   Frame makeFrame() { return new LinuxFrame(); }
}
...
```

Now we can have a global g holding a `FrameFactory` and `doStuff` begins with `Frame f = g.makeFrame();`.

And we've written 3 classes before our first cup of coffee. :)

Convenience of First-Class Classes

Wouldn't it be easier to skip the factory classes and just:

- Store in `g` either `WindowsFrame` or `LinuxFrame`
- Change `doStuff` to begin `Frame f = new g();`

Yes . . . but you can't do that in Java because *classes aren't objects*. It works perfectly in Smalltalk (`f := g new`).

But if classes are objects . . .

“Classes are objects” is great, but Java is avoiding some crazy stuff that:

- Doesn’t affect most day-to-day Smalltalk-80 programming
- Can mostly be brushed under the rug when teaching Smalltalk (up to a point)
- Does affect the Smalltalk-80 definition and implementation
- Does affect instance creation and initialization if you want to know The Whole Truth

Method Lookup

Before we go further, keep firmly in mind how message lookup is handled in Smalltalk. Suppose you are sending a message + to an object p.

1. Find the class of p (say Point).

2. Look in Point's method dictionary. If you find a method with that name (+ in this case), run it.

3. If not, look in Point's superclass, and so on up the superclass chain, until either you find it or you run off the end (and get an error).

The Smalltalk-76 Approach

Here's the catch:

- What is the class of 3? What is the class of 'hi mom'?
- Okay, so what is the class of SmallInteger? Of String?

In Smalltalk-76, 'hi mom' is an instance of the class String.

The class String is an instance of class Class.

Class Class is an instance of . . . (drumroll) . . . itself!

Consequence: every class must understand exactly the same messages.

No class-specific initialization methods (Point x: 10 y: 20), or class constants (Float pi).

What We Should Have Done (Editorial)

To accomodate class-specific initialization methods, class constants, and such, it would have been better to make message lookup special for classes: each class would have its own method dictionary of class methods.

When a class gets a message, first look in that class's individual method dictionary, then in the method dictionary of its class (namely class Class).

Consistency is not always the most important goal.

Metaclasses

What was actually done: Metaclasses.

Each ordinary class is an instance of a unique *metaclass*. This allows complete consistency regarding how method lookup is handled: go to the object's class, look in its method dictionary, and if the method isn't found, continue up the superclass chain.

Cost: beginners can typically not help but worry about where it all ends (is there a meta-meta class? A meta-meta-meta class? Does it go on forever?)

Well, there is eventually a cycle back, so it doesn't go on forever . . . but it is very confusing for beginners.

See: Alan Borning and Tim O'Shea, "An Empirically and Aesthetically Motivated Simplification of Smalltalk-80," *Proceedings of the European Conference on Object-Oriented Programming*, Association Française pour la Cybernétique Économique et Technique, Paris, 1987.

Fragile Superclasses

A common problem in OO languages: What if you want/need to change a class that has been subclassed? "No problem?"

- What if you add a method (new functionality, shared helper, etc.)
- What if you "optimize" a method implementation?
- What if, as a result, you can remove a method?

Bottom line: inheritance reuses implementations; and there is little control over how subclasses reuse public methods and extend objects.

For the latter, distinguishing "add" vs. "override" can improve the situation (see C# "versions" for example)

Multiple Inheritance

If code reuse via inheritance is so useful, why not allow multiple superclasses?

- C++ does, Java and Smalltalk don't
- Because it causes some semantic awkwardness and implementation awkwardness (we'll discuss only the former)
- Because it can interact awkwardly with static typing (not discussed here)

Is it useful? Sure: A simple example is "3DColorPoint" assuming we already have "3DPoint" and "ColorPoint".

Naive view: Subclass has all fields and methods of all superclasses

Multiple Inheritance Semantic Problems

What if multiple superclasses define the same message m or field f?

Options for m:

- Reject subclass—too restrictive (the diamond problem)
- "Left-most superclass wins" (leads to silent weirdness and really want per-method flexiblity)
- Require subclass to override m (can use *directed resends*)

Options for f: one copy or two copies?

C++ provides two forms of inheritance:

- One always makes two copies
- One makes one copy if fields were declared by same class (diamonds)

Beyond this course: Other ways to compose behavior (e.g., mixins)

Multimethods

Remember our semantics for message send (with late-binding):

1. We use the receiver's class to determine what method to call.
2. We evaluate the method body in an environment with `self` bound to the receiver and the arguments bound to the parameters.

The second step *does not* really make `self` so special; we could require methods to give an explicit name for this "$\mathbf{0}^{th}$" argument.

The first step *does* make `self` special; the classes of the other arguments does not affect what method we call.

Multimethods let us do just that!

Why multimethods

Consider these reasonable methods:

```
"in Point"
distTo: p2
   ^ (((self getX - p2 getX) raisedTo: 2))
      + ((self getY - p2 getY) raisedTo: 2)) sqrt
"in 3DPoint"
distTo: p2
   ^ (((self getX - p2 getX) raisedTo: 2))
     + ((self getY - p2 getY) raisedTo: 2)
     + ((self getZ - p2 getZ) raisedTo: 2) sqrt
```

What might happen when we do `p distTo: q`?

Multimethods Example

Neither Smalltalk nor Java has multimethods, so we have to make up syntax.

```
multimeth  p1@Point distTo: p2@Point
  ^ (((p1 getX - p2 getX) raisedTo: 2))
     + ((p1 getY - p2 getY) raisedTo: 2)) sqrt
multimeth p1@3DPoint distTo: p2@3DPoint
  ^ (((p1 getX - p2 getX) raisedTo: 2))
    + ((p1 getY - p2 getY) raisedTo: 2)
    + ((p1 getZ - p2 getZ) raisedTo: 2) sqrt
```

Now we're commutative and we can change the behavior for "one Point and one 3DPoint" by writing two more methods (and one can call the other)

Thoughts on multimethods

On the one hand, ordinary methods introduce an undesirable asymmetry for binary operations (e.g. distTo: or $+$). Multimethods fix this.

On the other hand, they are “less OO” because if the “$\mathbf{0}^{th}$” argument isn't special, then the semantics is less “receiver-oriented” so it's less tied to the “interacting objects” analogy.

And there are pragmatic questions like:

- where do programmers define multimethods
- how does the implementation build the necessary tables for resolving message-sends
- what if there's no best match