CSE 341: Programming Languages

Dan Grossman

Spring 2004

Lecture 22— Advanced Issues in Smalltalk and Other Dynamically Typed OO Languages

Where We Are

We have the basics:

- Everything is an object.
- Every object has a class.
- An object's behavior is determined by its class.
- Subclassing inherits, extends, and overrides behavior.
- `self` is resolved with *late-binding* (dynamic dispatch)

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

Today's "Advanced Issues"

- convenience: classes-are-objects makes "factories" trivial
- awkwardness: class of a class of a class...
- awkwardness: "fragile" superclasses

And there extensions with their own conveniences and awkwardenesses:

- multiple inheritance
- multimethods (next class)

Next time we will start considering type-system issues for OO; today we can still send any message to any object.

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 Apple, 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();`

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

An interesting connection to our equivalence lecture:

- If `new` is just a message and `WindowsFrame` is just an object, then sending `makeFrame` to `WindowsFrameFactory` is equivalent to sending `new` to `WindowsFrame`
- Just like `(fn x => C x)` is equivalent to `C` where `C` is a datatype constructor (if constructors-are-functions).

But if classes are objects...

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

- Doesn't affect day-to-day Smalltalk-80 programming
- Does affect the Smalltalk-80 definition and implementation

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?
  - If the same class (Smalltalk-76, Java), then they share methods because class methods are class-class instance methods.
  - Class (static) methods in Java are special

But we're not done...

Metaclasses

- Okay, what is the class of SmallInteger class? Of StringInteger class?
  - If we keep stuff separate forever, we'll have an infinite number of classes!
- Okay, so what is the class of `Metaclass`?
- Okay, so what is the class of `Metaclass class`?

Clever, huh? But the "instance-of relation" ends in a cycle!

Moral: Even elegant systems often have their "dark corners"

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 today)

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 "$\mathbf{0}^{\boldsymbol{th}}$" 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 happend 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, multimethods are "more OO" because they do more late-binding which is the essence of OO.

On the other hand, they are "less OO" because if the "$\mathbf{0}^{\boldsymbol{th}}$" 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