Composition

Based on Metz Chapter 8:
Combining Objects with Composition

The Gang of Four say:

- The second principle of object-oriented design:
    - *Favor object composition over inheritance*

The Gang of Four say:

- The first principle of object-oriented design:
  - *Program to an interface, not to an implementation*
- The second principle of object-oriented design:
  - *Favor object composition over inheritance*

Inheritance *vs.* Composition

- Inheritance lets us *quickly* create a specialization of an existing object
  - all we need do is program the differences
- But inheritance is not a panacea:
  - the extension must be prepared in advance, as a new class or factory
  - the kind of extension can't be changed at runtime
  - with single inheritance, you have just one shot

Costs of Inheritance

- What happens when you get it wrong?
- Reasonable, usable and Exemplary are coins with two sides!
  - ¬ reasonable: making changes near the top of an incorrectly-modeled hierarchy
  - ¬ usable: recumbentMountainBike (or immutableSet) can't be built
  - ¬ exemplary: can't extend an incorrectly-modeled hierarchy

Composition

- Pros
  - component can be changed at runtime
    - e.g., state pattern
  - clear separation of responsibilities
    - need know only the interface of the component
- Cons
  - more work
    - define separate classes for part, parts ...
  - delegation not supported by most languages
    - must use self delegation pattern (Beck, p.67)

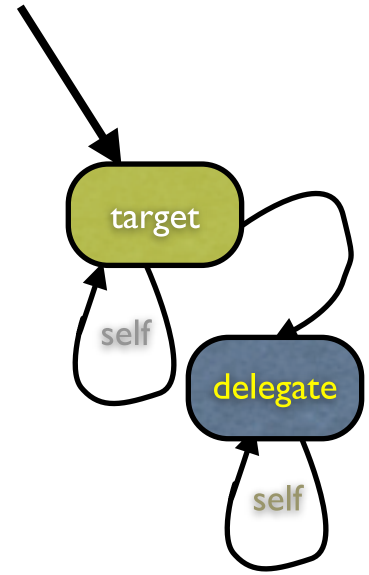
Metz:

- Inheritance:
  - for the cost of arranging objects in a hierarchy, you get message delegation for free
- Composition:
  - reverses these costs & benefits:
    - not restricted to a hierarchy; objects relationships are explicit
    - delegation of messages must *also* be explicit
- when faced with a problem that composition can solve, you should be biased towards using composition

Delegation

- Delegation allows you to share implementation without inheritance
- Pass part of your work on to another object. Put that object in one of your instance variables
    - e.g., a *Path* contains a field *form*, the bit mask responsible for actually drawing on the display.
    - e.g., a *Text* contains a *String*

What about **self**?



- When you delegate, the receiver of the delegating message (the *delegate*) is no longer the target
  - Does it matter? Does the delegate need access to the target? Does the delegate send a message back to the client?
- If it doesn't matter, *forward* messages unchanged — Beck calls this *Simple Delegation*

Simple Delegation Example

```
method do(aBlock) {
   collectionOfPoints.do(aBlock) }

method map(aBlock) {
      def newPath = path.withForm(self.form)
      newPath.points :=
               (collectionOfPoints.map(aBlock)
      newPath }
```

Portland State UNIVERSITY

Simple Delegation works when:

- you don’t need the state of the original target object
- you don’t need the behaviour of the original target object
- you don’t need the identity of the original target object

If you need these things, use *self delegation*

Self Delegation

- When the delegate *needs* a reference to the delegating object…
- Pass along the delegating object as an additional parameter.

Self Delegation Example

```
Dictionary: method at(key) put(value) {
    self.hashTable.at(key) put(value) for(self)
}
HashTable: method at(key) put(value) for(aCollection) {
    def hash = aCollection.hashOf(key)
}
Dictionary: method hashOf(anObject) {
    anObject.hash
}
PlugableDictionary: method hashOf(anObject) {
    injectedHash(anObject)
}
```