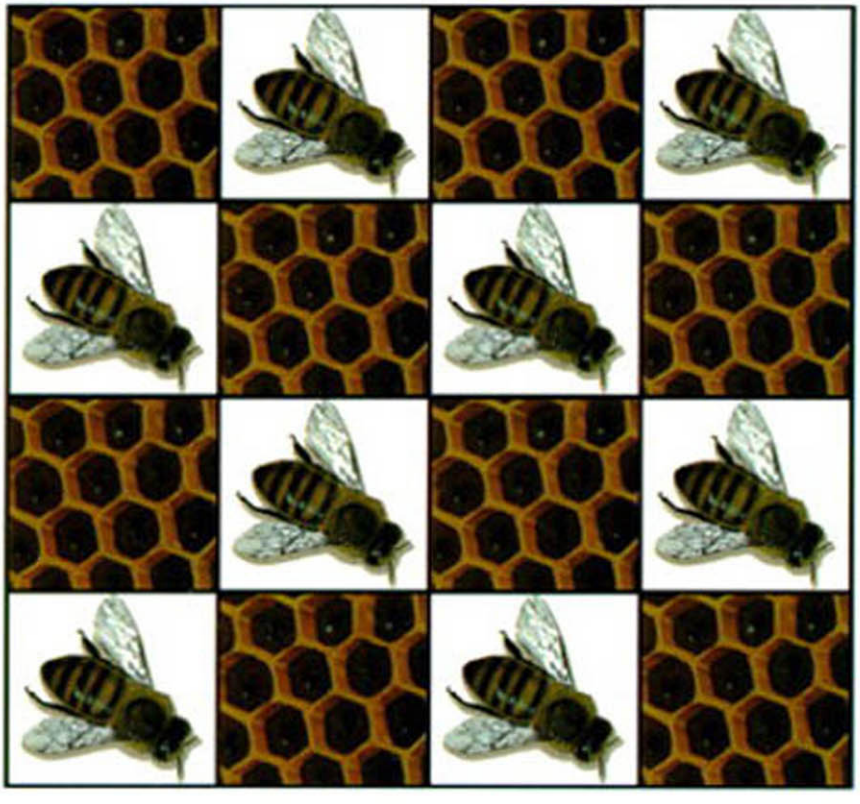
SMALLTALK
BEST PRACTICE PATTERNS



KENT BECK

Smalltalk Best Practice Patterns

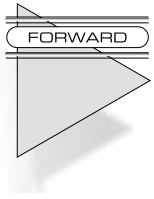
Kent Beck

Pearson

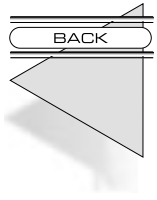
sending method.

One example of where you want to extend superclass behavior is initialization, where not only does the state defined by the superclass need to be initialized, but also the state defined by the subclass.

Always check code using "super" carefully. Change "super" to "self" if doing so does not change how the code executes. One of the most annoying bugs I've ever tried to track down involved a use of super that didn't do anything at the time I wrote it and invoked a different selector than the one for the currently executing method. I later overrode that method in the subclass and spent half a day trying to figure out why it wasn't being invoked. My brain had overlooked the fact that the receiver was "super" instead of "self," and I proceeded on that assumption for several frustrating hours.



Extending Super (p. 60) adds behavior to the superclass. Modifying Super (p. 62) changes the superclass' behavior.



Extending Super

You are using Super (p. 59).

- How do you add to a superclass' implementation of a method?

Any use of super reduces the flexibility of the resulting code. You now have a method that assumes not just that somewhere there is an implementation of a particular method, but that the implementation has to exist in the superclass chain above the class that contains the method. This assumption is seldom a big problem, but you should be aware of the tradeoff you are making.

If you are avoiding duplication of code by using super, the tradeoff is quite reasonable. For instance, if a superclass has a method that initializes some instance variables, and your class wants to initialize the variables it has introduced, super is the right solution. Rather than have code like:

```
Class: Super
        superclass: Object
        instance variables: a

Super class>>new
        ^self basicNew initialize
Super>>initialize
        a := self defaultA
```

and rather than extending initialization in a subclass like this:

```
Class: Sub
        superclass: Super
        instance variables: b

Sub class>>new
        ^self basicNew
                initialize;
                initializeB
Sub>>initializeB
        b := self defaultB
```

using super you can implement both initializations explicitly:

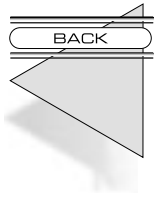
```
Sub>>initialize
        super initialize.
        b := self defaultB
```

and not have Sub override "new" at all. The result is a more direct expression of the intent of the code. Make sure Supers are initialized when they are created and extend the meaning of initialization in Sub.

- *Override the method and send a message to "super" in the overriding method.*

Another example of Extending Super is display. If you have a subclass of a Figure that needs to display just like the superclass, but with a border, you could implement it like this:

```
BorderedFigure>>display
        super display.
        self displayBorder
```



Modifying Super

You are using Super (p. 59).

- How do you change part of the behavior of a superclass' method without modifying it?

This problem introduces a tighter coupling between subclass and superclass than Extending Super. Not only are we assuming that a superclass implements the method we are modifying, we are assuming that the superclass is doing something we need to change.

Often, situations like this can best be addressed by refactoring methods with Composed Method so you can use pure overriding. For example, the following initialization code could be modified by using super.

```
Class: IntegerAdder
        superclass: Object
        instance variables: sum count

IntegerAdder>>initialize
        sum := 0.
        count := 0

Class: FloatAdder
        superclass: IntegerAdder
        instance variables:

FloatAdder>>initialize
        super initialize.
        sum := 0.0
```

A better solution is to recognize that IntegerAdder>>initialize is actually doing four things: representing and assigning the default values for each of two variables. Refactoring with Composed Method yields:

```
IntegerAdder>>initialize
        sum := self defaultSum.
        count := self defaultCount
IntegerAdder>>defaultSum
        ^0
IntegerAdder>>defaultCount
        ^0

FloatAdder>>defaultSum
        ^0.0
```

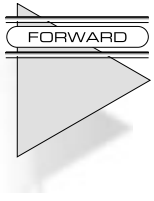
However, sometimes you have to work with superclasses that are not completely factored (i.e. the superclass does not implement #defaultSum). You are faced with the choice of either copying code or using super and accepting the costs of tighter subclass/superclass coupling. Most of the time, the addi-

tional coupling will not prove to be a problem. Communicate your desired changes with the owner of the superclass. In the meantime:

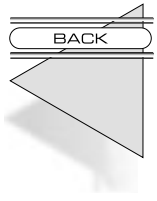
- *Override the method and invoke "super," then execute the code to modify the results.*

Another example from the display realm is if you have a subclass whose color is different from the superclass'.

```
SuperFigure>>initialize
        color := Color white.
        size := 0@0
SubFigure>>initialize
        super initialize.
        color := Color beige
```



Again, the better solution would be to use a Default Value Method (p. 86) to represent the default color, and then override just that method.



Delegation

A Composed Method (p. 21) needs work done by another object. A Message (p. 43) invokes computation in another object.

- How does an object share implementation without inheritance?

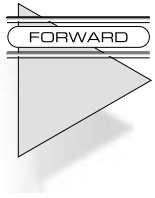
Inheritance is the primary built-in mechanism for sharing implementation in Smalltalk. However, inheritance in Smalltalk is limited to a single superclass. What if you want to implement a new object like A but also like B? Also, inheritance carries with it potentially staggering long-term costs. Code in subclasses isn't just written in Smalltalk. It is written in the context of every variable and method in every superclass. In deep, rich hierarchies, you may have to read and understand many superclasses before you can understand even the simplest method in a subclass.

Factored Superclass explains how to make effective use of inheritance at minimal development cost. You will encounter situations where you will rec-

ognize common implementation, but where Factored Superclass is not appropriate. How can you respond?

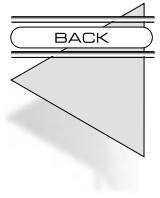
- *Pass part of its work on to another object.*

For example, since many objects need to display, all objects in the system delegate to a brush-like object (Pen in Visual Smalltalk, GraphicsContext in VisualAge and VisualWorks) for display. That way, all the detailed display code can be concentrated in a single class and the rest of the system can have a simplified view of displaying.



Use Simple Delegation (p. 65) when the delegate need know nothing about the original object. Use Self Delegation (p. 67) when the identity of the original object or some of its state is needed by the delegate.

Simple Delegation



You need Delegation (p. 64) to a self-contained object. You may be implementing one of the following methods: Collection Accessor Method (p. 96), Equality Method (p. 124), or Hashing Method (p. 126).

- How do you invoke a disinterested delegate?

When you use delegation, there are two main issues that help clarify what flavor of delegation you need. First, is the identity of the delegating object important? This might be true if a client object passes itself along, expecting to be notified of some part of the work actually done by the delegate. The delegate doesn't want to inform the client of its existence so it needs access to the delegating object. Second, is the state of the delegating object important to the delegate? Delegates are often simple, even state-less objects, in order to be as widely useful as possible. If so, the delegate is likely to require state from the delegating object to accomplish its job.

There are many cases of delegation where the answer to these two questions is "no." The delegate has no reason to need the identity of the delegating object. The delegate is self-contained enough to accomplish its job without additional state.

- *Delegate messages unchanged.*