SMALLTALK BEST PRACTICE PATTERNS

KENT BECK

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Smalltalk Best Practice Patterns

Kent Beck

Library of Congress Cataloging-in-Publication Data
Beck, Kent.
Smalltalk best practice patterns / Kent Beck.
p. cm.
Includes index.
ISBN 0-13-476904-X (pbk.)
1. Smalltalk (Computer program language) I. Title.
QA76.73.S59B43 1997
005.13'3--dc20 96-29411
CIP

Editorial/Production Supervision: *Joe Czerwinski*
Acquisitions Editor: *Paul Becker*
Manufacturing Manager: *Alexis R. Heydt*
Cover Design Director: *Jerry Votta*
Cover Design: *Design Source*

©1997 by Prentice Hall PTR
Prentice-Hall, Inc.
A Division of Simon and Schuster
Upper Saddle River, NJ 07458

The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact:

Corporate Sales Department
Prentice Hall PTR
One Lake Street
Upper Saddle River, NJ 07458

Phone: 800-382-3419
Fax: 201-236-7141
E-mail: corpsales@prenhall.com

All rights reserved. No part of this book may be reproduced in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN: 0-13-476904-X

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall of Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Pte. Ltd., *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

PREFACE vii

1. INTRODUCTION 1

CODING 1
Talking Programs *3*
GOOD SOFTWARE. 4
STYLE 6
WHAT'S MISSING? 7
BOOK ORGANIZATION 9
ADOPTION 9
LEARNING A PATTERN 10

2. PATTERNS 13

WHY PATTERNS WORK 14
ROLE OF PATTERNS 15
Reading *15*
Development *15*
Review *16*
Documentation. *16*
Clean Up *16*
FORMAT 16

3. BEHAVIOR 19

METHODS 20
Composed Method 21
Constructor Method 23
Constructor Parameter Method. 25
Shortcut Constructor Method 26
Conversion 28
Converter Method 28
Converter Constructor Method 29
Query Method 30
Comparing Method 32
Reversing Method 33
Method Object. 34
Execute Around Method 37
Debug Printing Method 39
Method Comment 40
MESSAGES 43
Message 43
Choosing Message 45
Decomposing Message 47
Intention Revealing Message 48
Intention Revealing Selector 49
Dispatched Interpretation 51
Double Dispatch 55
Mediating Protocol 57
Super 59
Extending Super 60
Modifying Super 62
Delegation. 64
Simple Delegation. 65
Self Delegation 67
Pluggable Behavior. 69
Pluggable Selector 70
Pluggable Block 73
Collecting Parameter. 75

4. STATE. 79

INSTANCE VARIABLES 80
Common State 80
Variable State 82
Explicit Initialization. 83
Lazy Initialization. 85
Default Value Method 86
Constant Method. 87
Direct Variable Access 89
Indirect Variable Access 91

Getting Method 93
Setting Method 95
Collection Accessor Method 96
Enumeration Method 99
Boolean Property Setting Method 100
Role Suggesting Instance Variable Name 102
TEMPORARY VARIABLES 103
Temporary Variable 103
Collecting Temporary Variable 105
Caching Temporary Variable 106
Explaining Temporary Variable 108
Reusing Temporary Variable 109
Role Suggesting Temporary Variable Name 110

5. COLLECTIONS 113

CLASSES 114
Collection 115
OrderedCollection 116
RunArray 118
Set 119
Equality Method 124
Hashing Method 126
Dictionary 128
SortedCollection 131
Array 133
ByteArray 135
Interval 137
COLLECTION PROTOCOL 139
IsEmpty 139
Includes: 141
Concatentation 143
Enumeration 144
Do 146
Collect 147
Select/Reject 149
Detect 150
Inject:into: 152
COLLECTION IDIOMS 153
Duplicate Removing Set 154
Temporarily Sorted Collection 155
Stack 156
Queue 157
Searching Literal 159
Lookup Cache 161
Parsing Stream 164
Concatenating Stream 165

6. CLASSES 167

Simple Superclass Name 168
Qualified Subclass Name 169

7. FORMATTING 171

Inline Message Pattern 172
Type Suggesting Parameter Name 174
Indented Control Flow 175
Rectangular Block 177
Guard Clause 178
Conditional Expression 180
Simple Enumeration Parameter 182
Cascade 183
Yourself 186
Interesting Return Value 188

8. DEVELOPMENT EXAMPLE 191

PROBLEM 191
START 192
ARITHMETIC 194
INTEGRATION 198
SUMMARY 201

APPENDIX A: QUICK REFERENCE 203

INDEX 217

Preface

This preface will explain what this book is about. It will convince you to buy this book, or you will know why you shouldn't (more of the former than the latter, I hope).

What's it all about?

This book is about the simple things experienced, successful Smalltalkers do that beginners don't. In a sense, it is a style guide. I have tried to penetrate beneath the surface, though, to get at the human realities that make the rules work instead of focusing solely on the rules themselves.

The topics covered are the daily tactics of programming:

- How do you choose names for objects, variables, and methods?
- How do you break logic into methods?
- How do you communicate most clearly through your code?

These are small scale issues. There are also many bigger technical reasons why projects fail (and many more nontechnical reasons).

The attraction of this set of issues is that they are so tractable. You don't have to be a programming wizard to pick good names, you just have to have good advice.

The advice is broken into 92 patterns. Each pattern presents:

- a recurring daily programming problem;
- the tradeoffs that affect solutions to the problem; and
- a concrete recipe to create a solution for the problem.

For example, here is a summary of a pattern called "Role Suggesting Temporary Variable Name":

Problem: What do you name a temporary variable?

Tradeoffs:

- You want to include lots of information in the name.
- You want the name to be short so it is easy to type and doesn't make formatting difficult.
- You don't want redundant information in the name.
- You want to communicate why the variable exists.
- You want to communicate the type of the variable (i.e. what messages it is sent).

Solution: Name the variable after the role it plays. The type can be inferred from context, and so doesn't need to be part of the name.

You will see in the body of the book that each pattern occupies a page or two. Each pattern includes examples (and counter-examples) from the standard Smalltalk images. Each pattern also talks about related patterns.

The patterns don't stand in isolation, 92 independent bits of advice. Patterns work together, leading you from larger problems to smaller. Together they form a system or language. The system, as a whole, allows you to focus on the problem at hand, confident that tomorrow you can deal with tomorrow's problems.

Why should you read it?

Learning—If you are just learning Smalltalk, these patterns will give you a big jump start on making effective use of the system. Because the patterns aren't just rules, you can smoothly go from merely following the patterns, to understanding why they are the way they are, to formulating your own patterns. You will need a good basic introduction to Smalltalk in addition to this book, but reading them together will greatly accelerate your learning.

Programming—If you program in Smalltalk, these patterns will give you a catalog of techniques that work well. You will have discovered or invented many of them yourself, but the patterns may give you a fresh perspective on why they work or present nuances you hadn't considered.

Teaching—If you teach Smalltalkers, either as a mentor or in classroom training, these patterns will give you large bag of instructional material. If you are trying to explain why code should be different, it is much more satisfying for you and the learner to be able to discuss the pattern and how it applies to the particular situation.

Managing—If you manage Smalltalk projects, you may be struggling with how to apply good software engineering principles to Smalltalk. These patterns don't address that topic directly, but they can become the basis of a common vocabulary for your developers.

What isn't it about?

This is not a book of methodology. It will not guide your entire development process. You can use it with your existing process, whether you invented it or it came out of a book. This book is about making code that works for you.

This is not a book of philosophy. If you want to understand what makes programs good in the abstract, if you want to learn to write patterns yourself, or understand their philosophical or psychological basis, you won't find any help here. This book is for people who have programs to write and want to do so as quickly, safely, and effectively as possible.

This is not a book of design. If design is the process of defining the relationships among small families of objects, the resulting problems repeat just as surely as do implementation problems. Design patterns are very effective at capturing that commonality. They just aren't the topic of this book. This book is about making Smalltalk work for you. Making objects work for you is an entirely different topic.

Acknowledgments

I would like to thank the many people who contributed to this volume. First I would like to thank the Xerox PARC Learning Research Group (Alan Kay, Adele Goldberg, Dan Ingalls, Diana Merry-Shapiro, Ted Kaehler, Larry Tesler, and Bob Flegel) for having the insights in the first place, so I had something to write down. I would like to thank my mentor and intellectual partner, Ward Cunningham, for showing me the way and sharing his insights. Many of the patterns here he identified and/or named. Thanks to my reviewers (Dirk Riehle, David N. Smith, Mitchell Model, Bill Reynolds, Dave Smith, Trygve

Reenskaug, Ralph Johnson, John Brant, Don Roberts, Brian Foote, Brian Marick, Joe Yoder, Ian Chai, Mark Kendrat, Eric Scouten, Charles Herring, Haidong Ye, Kevin Powell, Rob Brown, Kyle Brown, Bobby Woolf, Harald Mueller, Steve Hayes, Bob Biros, David Warren, Gert Florijn, Mark L. Fussell, Martin Fowler, Chuck Siska, Chris Bird, Ron Jefferies, Volker Wurst, Peter Epstein, Thomas Murphy, Michel Brassard, Ron Jefferies, John Sellers, Steve Messick, Darrow Kirkpatrick, Phoenix Tong, Doug Lea, Randy Stafford, Sharry Fealk and all the reviewers who didn't put their names on their comments) for reading early rough drafts carefully. Finally, this book would never have been finished without my ever patient but gently prodding editor, Paul Becker.

Chapter 3

Behavior

Objects model the world through behavior and state. Behavior is the dynamic, active, computational part of the model. State is what is left after behavior is done, how the model is represented before, after, and during a computation.

Of the two, behavior is the more important to get right. The primacy of behavior is one of the odd truths of objects; odd because it flies in the face of so much accumulated experience. Back in the bad old days, you wanted to get the representation right as quickly as possible because every change to the representation bred changes in many different computations.

Objects (done right) change all that. No longer is your system a slave of its representation. Because objects can hide their representation behind a wall of messages, you are free to change representation and only affect one object.

Behavior in systems of objects is specified in two ways; with messages and methods. I saw a great comment at OOPSLA (the Object Oriented Programming Languages, Systems and Applications conference). It said, “This seems an awful fuss for a fancy procedure call.” Well, separating computation into messages and methods and binding the message to the method at run time,

based on the class of the receiver, may seem like a small change from an ordinary procedure call, but it is a small change that makes a big difference.

This section tells you how to specify behavior so that your intent is clearly communicated to your reader. Many constraints affect your choices when specifying behavior. The more centralized the flow of control, the easier it is to follow in the sense that you don't have to go bouncing around all over the place to understand how work is accomplished. However, centralizing control kills flexibility. You want to have lots of objects involved so you have many opportunities to replace objects to change the system, and so you can completely factor code.

Methods

Methods are important to the system because they are how work gets done in Smalltalk. Just as important, methods are the way you communicate to readers how you intended for work to get done. You must write your methods with both of these audiences in mind. Methods must do the work they are supposed to do but they must also communicate the intent of the work to be done.

Methods decompose the function of your program into easily digestible chunks. Carefully breaking a computation into methods and carefully choosing their names communicates more about your intentions to a reader than any other programming decision, besides class naming.

Methods are the granularity of overriding. A well factored superclass can always be specialized by overriding a single method, without having to copy part of the superclass code into the subclass.

Methods don't come for free. Managing all those bits and pieces of code—writing them in the first place, naming them, remembering, rediscovering, and communicating how they all fit together—all take time. If there is no benefit to be gained, bigger methods would be better than small because of the reduced management overhead.

Methods cost in performance as well. Each method invocation takes precious computer cycles. The trick to getting good performance is using methods as a lever to make your performance measurement and tuning more effective. In my experience, better factored code, with lots of small methods, both allows more accurate and concise performance measurement (because there aren't little snippets of code duplicated all over) and provides leverage for tuning (through techniques like Caching Instance Variable).

Overall, the goal of breaking your program into methods is to communicate your intent clearly with your reader, provide for future flexibility, and set yourself up for effective performance tuning where necessary.

Composed Method

You are implementing a method named with an Intention Revealing Selector (p. 49).

- How do you divide a program into methods?

Programs need to do more than just instruct a computer, they need to communicate to people as well. How your program is broken into methods (as well as how big those methods are) is one of the most important decisions you will make as you refine your code so that it communicates as clearly as possible. The decision is complicated by the many factors affecting it and the history of programming practice that has traditionally optimized machine resources at the cost of people's time.

Messages take time. The more small methods you create, the more messages you will execute. If all you were worried about was how fast your program would run, you would arrange all of your code in a single method. This radical approach to performance tuning invokes enormous human costs and ignores the realities of performance tuning well-structured code, which often results in several order-of-magnitude improvements.

Simple minded performance tuning is not the only factor suggesting that large methods are best. Following the flow of control in programs with many small methods can be difficult. Novice Smalltalk programmers often complain that they can't figure out where any "real" work is getting done. As you gain experience, you will need to understand the flow of control through several objects less often. Well chosen message names let you correctly assume the meaning of invoked code.

The opportunity to communicate through intention revealing message names is the most compelling reason to keep methods small. People can read your programs much more quickly and accurately if they can understand them in detail, then chunk those details into higher level structures. Dividing a program into methods gives you an opportunity to guide that chunking. It is a way for you to subtly communicate the structure of your system.

Small methods ease maintenance. They let you isolate assumptions. Code that has been written with the right small methods requires the change of only a few methods to correct or enhance its operation. This is true whether you are fixing bugs, adding features, or tuning performance.

Small methods also make inheritance work smoothly. If you decide to specialize the behavior of a class written with large methods, you will often find yourself copying the code from the superclass into the subclass and changing a few lines. You have introduced a multiple update problem between the

superclass method and the subclass method. With small methods, overriding behavior is always a case of overriding a single method.

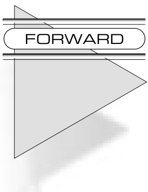
- *Divide your program into methods that perform one identifiable task. Keep all of the operations in a method at the same level of abstraction. This will naturally result in programs with many small methods, each a few lines long.*

You can use Composed Method top-down. While you are writing a method, you can (without having an implementation yet) invoke several smaller methods. Composed Method becomes a thought tool for breaking your development into pieces. Here is an example of a top-down Composed Method:

```
Controller>>controlActivity
	self controlInitialize.
	self controlLoop.
	self controlTerminate
```

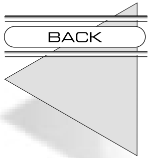
You can also use Composed Method bottom-up, to factor common code in a single place. If you find yourself using the same expression (which might be only 3 or 2 or even 1 line of code), you can improve your code by putting the expression in its own method and invoking it as needed.

Perhaps most importantly, you can use Composed Method to discover new responsibilities while you are implementing. Any time you are sending two or more messages from one object to another in a single method, you may be able to create a Composed Method in the receiver that combines those messages. Such methods are invariably useful from other parts of your system.



Create objects with a Constructor Method (p. 23). Put boolean expressions into a Query Method (p. 30). Invoke Messages to get work done elsewhere, sometimes by Delegation (p. 64). Use a Temporary Variable (p. 103) for temporary storage. Represent constants with a Constant Method (p. 87).

Constructor Method



A Composed Method (p. 21) has had to create an object.

- How do you represent instance creation?

The most flexible way to represent instance creation is by a simple "new" method, followed by a series of messages from the client to the new instance. That way, if there are different combinations of parameters that make sense, the client can take advantage of just those parameters it needs.

Creating a Point in this style looks like this:

```
Point new x: 0; y: 0
```

Further flexibility is provided in this approach to half-way construct an object in one place, and then pass it off to another to finish construction. This can simplify communications if you don't have to modify the design to put all the creation parameters in one place.

On the other hand, what is the first thing you want to know about a class, once you've decided it may do what you want it to do? The first question is "What does it take to create an instance?" As a class provider, you'd like the answer to this question to be as simple as possible. With the style described above, you have to track down references to the class and read the code before you get an inkling of how to create a useable instance. If the code is complex, it may take a while before you figure out what is required and what is optional in creating an instance.

The alternative is to make sure that there is a method to represent each valid way to create an instance. Does this result in a proliferation of instance creation methods? Almost never. Most classes only have a single way to create an instance. Almost all of the exceptions only have a handful of variations. For the rare case where there really are hundreds or thousands of possible correct combinations of parameters, use Constructor Methods for the common cases and provide Accessor Methods for the remainder.

With this style of instance creation, the question "How can I create a valid instance?" can be simply answered by looking at the "instance creation" protocol of the class methods. The Intention Revealing Selectors communicate

what the instance will do for you, while the Type Suggesting Parameter Names communicate the parameters required.

- *Provide methods that create well-formed instances. Pass all required parameters to them.*

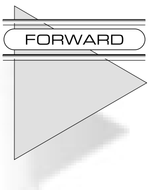
Point class>>x:y: is a Constructor Method because it takes both of the required numbers as parameters.

Some people think that the keywords in the Constructor Method have to be named the same as the instance variables that will eventually be initialized while constructing an instance. You should always look for a way of expressing more intention with a selector (Intention Revealing Selector). For example, Point class>>r:theta: is a Constructor Method I add when I am working in polar coordinates:

```
Point class>>r: radiusNumber theta: thetaNumber
	^self
		x: radiusNumber * thetaNumber cos
		y: radiusNumber * thetaNumber sin
```

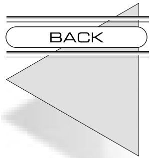
SortedCollection class>>sortBlock: aBlock is a Constructor Method because it returns a SortedCollection that is ready to use. SortedCollection class>>new is also a Constructor Method because it returns a SortedCollection that is ready to use, too. It just has a default sort block.

Put Constructor Methods into a method protocol called "instance creation."



If the method takes parameters, you will need a Constructor Parameter Method (p. 25). Give your method an Intention Revealing Selector (p. 49) that describes the roles of the parameters, not their type. A Constructor Method that is used extensively may deserve a Shortcut Constructor Method (p. 26).

Constructor Parameter Method



A Constructor Method (p. 23) needs to pass parameters on to the new instance. You need to initialize Common State (p. 80).

- How do you set instance variables from the parameters to a Constructor Method?

Once you have the parameters of a Constructor Method to the class, how do you get them to the newly created instance?

The most flexible and consistent method is to use Setting Methods to set all the variables. Thus, a Point would be initialized with two messages:

```
Point class>>x: xNumber y: yNumber
	^self new
		x: xNumber;
		y: yNumber;
		yourself
```

The problem I have run into with this approach is that Setting Methods can become complex. I have had to add special logic to the Setting Methods to check whether they are being sent during initialization; if so I just set the variable.

Remember the rule that says "Say things once and only once?" Special casing a Setting Method for use during initialization is a violation of the first part of that rule. You have two circumstances—state initialization during instance creation and state change during computation—but only one method. You have two things to say and you've only said one thing.

The Setting Method solution also has the drawback that if you want to see the types of all the variables, you have to look at the Type Suggesting Parameter Names in several methods. You'd like the reader to be able to look at your code and quickly understand the types of the Instance Variables.

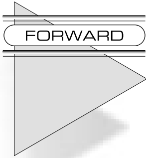
- *Code a single method that sets all the variables. Preface its name with "set," then the names of the variables.*

Using this pattern, the code above becomes:

```
Point class>>x: xNumber y: yNumber
	^self new
		setX: xNumber
		y: yNumber
Point>>setX: xNumber y: yNumber
	x := xNumber.
	y := yNumber.
	^self
```

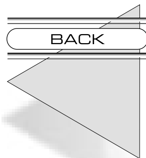
Note the Interesting Return Value in setX:y:. It is there because the return value of the method will be used as the return value of the caller.

Put Constructor Parameters Methods in a method protocol called "private."



If you are using Explicit Initialization (p. 83), now is a good time to invoke it, to communicate that initialization is part of instance creation.

Shortcut Constructor Method



You have identified a pervasive Constructor Method (p. 23).

- What is the external interface for creating a new object when a Constructor Method is too wordy?

The typical way you create a new object is to send a message to the class that creates a new instance for you; "Point x: width y: height". This is good because it is very explicit about what object is being created. If you want to find out what happens as a result of this expression, you know just where to look.

There are two problems with this style of interface for object creation. The most important is that it is wordy. If Point class>>x:y: were the only interface for creating points, I dare say the Smalltalk source file would grow by a few percent. For very commonly used objects, you can create a more concise interface by sending a message to one of the arguments that then turns around and sends the longer form.

The second problem with an explicit class-based interface for object creation is that it can be misleading. There are times when differences in the classes of the arguments change the concrete class returned by the message. For example, different kinds of Collections might need different kinds of Streams.

The very conciseness of representing object creation as a message to one of the arguments is also its weakness. Such a message can easily be mistaken for built in language syntax ("@" is the classic example). It puts a burden on the programmer to remember the message. It cannot be easily looked up by looking at the instance creation methods of the class. However, the clarity or concision gains for a constructor method can be substantial.

- *Represent object creation as a message to one of the arguments to the Constructor Method. Add no more than three of these Shortcut Constructor Methods per system you develop.*

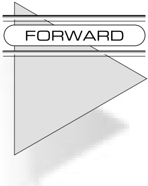
The classic example in Smalltalk is Point creation. The Constructor Method is:

```
Point class>>x: xNumber y: yNumber
	^self new
		setX: xNumber
		y: yNumber
```

The Shortcut Constructor Method is:

```
Number>>@ aNumber
	^Point
		x: self
		y: aNumber
```

Interestingly, the ParcPlace image has been moving away from using Point>>extent: and Point>>corner: as Shortcut Constructor Methods for Rectangles.



Put Shortcut Constructor Methods in a method protocol called "converting."

Conversion

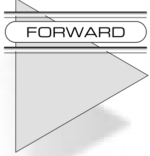
- How do you convert information from one object's format to another's?

Different clients may need the same information presented with different protocol. For example, one object may need to look at a Collection sorted, another with duplicates removed.

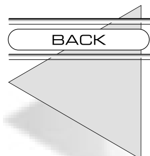
The simplest solution is to add all of the possible protocol needed to every object that may be asked of it. This might result in unnecessarily large public protocols with the resulting difficulty in publication and understanding. The same selector might need to mean different things to different clients, making this approach simply unworkable.

- *Convert from one object to another rather than overwhelm any one object's protocol.*

Some conversions are between similar objects, like changing a String of 8-bit ASCII characters to a String of 16-bit ISO characters. Some conversions are between different objects, like changing a String to a Date or a Number to a Pointer.



Conversions that return objects with similar responsibilities should use a Converter Method (p. 28). To convert to an object with different protocol use a Converter Constructor Method (p. 29).



Converter Method

You are implementing a Conversion.

- How do you represent simple conversion of an object to another object with the same protocol but different format?

For a long time, it bothered me that there was a String>>asDate method. I couldn't quite put my finger on what it was that bothered me about it, though. Then, I walked into a project where they had taken the idea of conversion to extremes. Every domain object had twenty or thirty different conversion methods. Every time a new object was added, it had to have all twenty or thirty methods before it would start working with the rest of the system.

One problem with representing conversion as methods in the object to be converted is that there is no limit to the number of methods that can be added. The protocol grows and grows without limit. Another is that it ties the receiver, however tenuously, with a class of which it would otherwise be oblivious.

I avoid the protocol explosion problem by only representing conversions with a message to the object to be converted when:

- The source and destination of conversion share the same protocol.
- There is only one reasonable way to implement the conversion.

- *Provide a method in the object to be converted that converts to the new object. Name the method by prepending "as" to the class of the object returned.*

Here are some examples. Notice that the object returned has the same protocol as the receiver (Sets act like Collections, Floats act like Numbers).

```
Collection>>asSet
Number>>asFloat
```

Put Converter Methods in a method protocol called "private."

Choose an Intention Revealing Selector (p. 49) for your conversion.

Converter Constructor Method

You need to implement Conversion (p. 28) to a new kind of object.

- How do you represent the conversion of an object to another with different protocol?

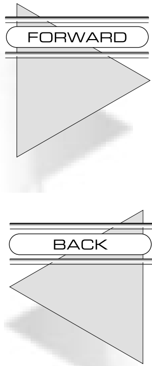
In many ways, the simplest way to communicate the presence of a conversion is a Converter Method. If I am explaining Date to you and you already know about Strings, it is tempting to say, "You can just convert a String to a Date by sending asDate to the String."

This solution risks cluttering common sources of conversion like Strings and Numbers with protocol that is irrelevant to their primary mission. The Visual Smalltalk implementation of String has 36 as... methods, half of which return objects with completely different protocols. I have seen applications where String has been "enhanced" with more than 100 Conversion Methods.

- *Make a Constructor Method that takes the object to be converted as an argument.*

For example, Date class>>fromString: is a Converter Constructor Method. It takes the String to be converted as an argument and returns a Date.

Put Converter Constructor Methods in a protocol called "instance creation."



You need to choose an Intention Revealing Selector (p. 49) for the method.

Query Method

A Composed Method (p. 21) has had to execute a boolean expression.

- How do you represent testing a property of an object?

There are actually two decisions here. The first is deciding what to return from a method that tests a property. The second is what you should name the method.

Designing the protocol for a Query Method provides you with two alternatives. The first is to return one of two objects. For example, if you have a switch that can be either on or off, you could return either #on or #off.

```
Switch>>makeOn
    status := #on
Switch>>makeOff
    status := #off
Switch>>status
    ^status
```

That leaves clients needing to know how Switch stores its status:

```
WallPlate>>update
    self switch status = #on ifTrue: [self light makeOn].
    self switch status = #off ifTrue: [self light makeOff]
```

A maintenance programmer who innocently decides to change the Symbols to #On and #Off will break the client.

It is far easier to maintain a relationship based solely on messages. Rather than status returning a Symbol, it is better for Switch to provide a single method that returns a Boolean; true if the Switch is on and false if the Switch is off.

Whether this is represented in the Switch as a variable holding a Boolean or a variable holding one of two Symbols is irrelevant to designing the protocol.

The naming question is a bit more sticky. The simplest name for a method that tests a property and returns a Boolean is just a simple name. In the example above, I'd be tempted to call the method "on":

```
Switch>>on
    "Return true if the receiver is on, otherwise return false."
```

However, this leads to confusion. Does "on" mean "is it on?" or "make it on?"

- *Provide a method that returns a Boolean. Name it by prefacing the property name with a form of "be"—is, was, will, etc.*

Here are some examples from Smalltalk:

```
isNil
isControlWanted
isEmpty
```

If you use the logical inverse of a Query Method a lot, also provide an inverse method, like notNil or notEmpty. Actually, if you can find a positive way of saying the inverse, that's even better. On the other hand, isUseful and isFull don't make much sense.

Put Query Methods in a protocol called "testing."

Comparing Method

- How do you order objects with respect to each other?

The comparison messages <, <=, >, >= are implemented mostly in Magnitude and its subclasses. They are used for all sorts of purposes—sorting, filtering, and checking for thresholds.

When you create new objects, you have the option of implementing comparison methods yourself. When I was a year or two into Smalltalk, I seem to remember implementing comparison methods any time I put a kind of object into a SortedCollection. As time went on, I used the sort block (see SortedCollection) more and more and implemented "<=" less and less.

I still implement "<=" when there is one overwhelming way to order a new object. That way, those using it can take a collection containing those objects and sort them just by saying "asSortedCollection."

Most uses of sorting in the user interface require more flexibility than can be provided by a single comparison order. Expect to use sort blocks with Temporarily Sorted Collection.

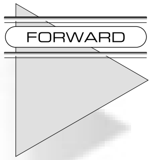
- *Implement "<=" to return true if the receiver should be ordered before the argument.*

Numbers are the obvious example of Comparing Methods. Characters and Strings also implement Comparing Methods.

If you had a Collection of timed Events, the Comparing Method could order them by time:

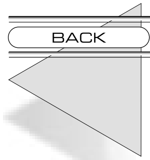
```
Event>><= anEvent
	^self timestamp <= anEvent timestamp
```

Put Comparing Methods in a protocol called "comparing."



Ordering is often done in terms of Simple Delegation (p. 65) to the ordering of other objects. For multiple orderings, use a Temporarily Sorted Collection (p. 155).

Reversing Method



A Composed Method (p. 21) may not read right because messages are going to too many receivers. You may have a Cascade (p. 183) that doesn't look quite right because several different objects need to receive messages.

- How do you code a smooth flow of messages?

Good code has a rhythm that makes it easy to understand. Code that breaks the rhythm is harder to read and understand.

```
Point>>printOn: aStream
    x printOn: aStream.
    aStream nextPutAll: ' @ '.
    y printOn: aStream
```

Here we have messages going to three different objects. We want to read this as a three part operation, but because the operations are on three different objects it is hard to put the pieces together.

We can solve the problem by making sure that all messages go through a single object. However, creating new selectors just for the fun of it is a bad idea. Each selector in the system must justify its existence by solving a real problem; encoding an important decision.

Adding a new method with a new selector to make code read more smoothly is a good use of the selector namespace.

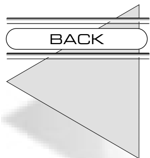
- *Code a method on the parameter. Derive its name from the original message. Take the original receiver as a parameter to the new method. Implement the method by sending the original message to the original receiver.*

By defining Stream>>print:, we can smooth out the above method:

```
Stream>>print: anObject
	anObject printOn: self
Point>>printOn: aStream
	aStream
		print: x;
		nextPutAll: ' @ ';
		print: y
```

This pattern seems to veer perilously close to the realm of pure aesthetics. However, I often find that the desire to use it is followed closely by the absolute need to use it. As soon as you have all the messages going to a single object, that object can easily vary without affecting any of the parameters.

Put Reversing Methods in a method protocol named after the message being reversed. For example, Stream>>print: is in the method protocol "printing."



Method Object

You have a method that does not simplify well with Composed Method (p. 21).

- How do you code a method where many lines of code share many arguments and temporary variables?

The behavior at the center of a complex system is often complicated. That complexity is generally not recognized at first, so the behavior is represented as a single method. Gradually that method grows and grows, gaining more lines, more parameters, and more temporary variables, until it is a monstrous mess.

Far from improving communications, applying Composed Method to such a method only obscures the situation. Since all the parts of such a method generally need all the temporary variables and parameters, any piece of the method you break off requires six or eight parameters.

The solution is to create an object to represent an invocation of the method and use the shared namespace of instance variables in the object to

enable further simplification using Composed Method. However, these objects have a very different flavor than most objects. Most objects are nouns, these are verbs. Most objects are easily explainable to clients, these are not because they have no analog in the real world. However, Method Objects are worth their strange nature. Because they represent such an important part of the behavior of the system, they often end up at the center of the architecture.

- *Create a class named after the method. Give it an instance variable for the receiver of the original method, each argument, and each temporary variable. Give it a Constructor Method that takes the original receiver and the method arguments. Give it one instance method, #compute, implemented by copying the body of the original method. Replace the method with one that creates an instance of the new class and sends it #compute.*

This is the last pattern I added to this book. I wasn't going to include it because I use it so seldom. Then it convinced an important client to give me a big contract. I realized that when you need it, you REALLY need it.

The code looked like this:

```
Obligation>>sendTask: aTask job: aJob
	| notProcessed processed copied executed |
	...150 lines of heavily commented code...
```

First, I tried Composed Method. Every time I tried to break off a piece of the method, I realized I would have to send it both parameters and all four temps:

```
Obligation>>prepareTask: aTask job: aJob notProcessed:
notProcessedCollection processed: processedCollection
copied: copiedCollection executed: executedCollection
```

Not only was this ugly, but the resulting invocation didn't save any lines of code (see Indented Control Flow, below). After fifteen minutes or so of struggle, I went back to the original method and used Method Object. First I created the class:

```
Class: TaskSender
	superclass: Object
	instance variables: obligation task job notProcessed
processed copied executed
```

Notice that the name of the class is taken directly from the selector of the original method. Notice also that the original receiver, both arguments, and all four temps became instance variables.

The Constructor Method took the original receiver and both arguments as parameters:

```
TaskSender class>>obligation: anObligation task: aTask
job: aJob
	^self new
		setObligation: anObligation
		task: aTask
		job: aJob
```

Next I copied the code from the original method. The only change I made was textually replacing "aTask" with "task" and "aJob" with "job," since parameters are named differently than instance variables. Oh, I also deleted the declaration of the temps, since they were now instance variables.

```
TaskSender>>compute
	...150 lines of heavily commented code...
```

Then I changed the original method to create and invoke a TaskSender:

```
Obligation>>sendTask: aTask job: aJob
    (TaskSender
        obligation: self
        task: aTask
        job: aJob) compute
```

I tried out the method to make sure I hadn't broken anything. Since all I had been doing was moving text around, and I did it carefully, the revised method and its associated object worked the first time.

Now came the fun part. Since all the pieces of the method now shared the same instance variables, I could use Composed Method without having to pass any parameters. For example, the piece of code that prepared a Task became a method called #prepareTask.

The whole job took about two hours, but by the time I was done the #compute method read like documentation; I had eliminated three of the instance variables, the code as a whole was half of its original length, and I'd found and fixed a bug in the original code.

Execute Around Method

- How do you represent pairs of actions that have to be taken together?

It is common for two messages to an object to have to be invoked in tandem. When a file is opened, it has to be closed. When a context is pushed, it has to be popped.

The obvious way to represent this is by publishing both methods as part of the external protocol of the object. Clients need to explicitly invoke both, in the right order, and make sure that if the first is called, the second is called as well. This makes learning and using the object more difficult and leads to many defects, such as file descriptor leaks.

- *Code a method that takes a Block as an argument. Name the method by appending "During: aBlock" to the name of the first*

method that needs to be invoked. In the body of the Execute Around Method, invoke the first method, evaluate the block, then invoke the second method.

I learned this pattern from Cursor>>showWhile:

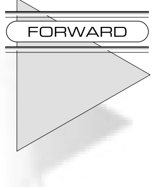
```
Cursor>>showWhile: aBlock
    | old |
    old := Cursor currentCursor.
    self show.
    aBlock value.
    old show
```

I use it lots of places. For example, I use it for making sure files get closed.

```
File>>openDuring: aBlock
    self open.
    aBlock value.
    self close
```

You will often want to wrap the Block evaluation in an exception handler so you are assured the second message gets sent.

```
File>>openDuring: aBlock
    self open.
    [aBlock value] ensure: [self close]
```



Put Execute Around Methods in a method protocol named after the operations they encapsulate. For example, File>>openDuring: goes in the method protocol "opening."

You need to give your method an Intention Revealing Selector (p. 49).

Debug Printing Method

- How do you code the default printing method?

Smalltalk provides a single mechanism for turning objects into printable strings; printOn:. Strings are great because they fit nicely into generic interface components; lists display strings; tables display strings; text editors and input fields display strings.

Strings are also useful in generic programming tools, like the Inspector. As a programmer, you can often look at the string generated by an object and instantly diagnose a problem.

The two audiences for strings generated by objects, you and your client, are often in conflict. You want all the internal, structural details of your object laid out in one place so you don't have to go searching layers and layers of objects to find what you want. Your client assumes the object is working correctly and just wants to see externally relevant aspects of the object in the string.

VisualWorks has taken the valuable step of separating these two uses of object-to-string conversion. If you want a client-consumable string, you send "displayString." If you want a programmer-consumable string, you send "printString." For Smalltalks with a single message for printing, you need to choose which audience you will address.

- *Override printOn: to provide information about an object's structure to the programmer.*

Associations print so that programmers can read them:

```
Association>> printOn: aStream
	aStream
		print: self key;
		nextPutAll: '->';
		print: self value
```

The saving grace of this pattern is that all the user interface builders have ways of parameterizing which message they will send to objects to get strings. Thus, when you create a list and send it some objects, you can also say "...and send the message `userString' to the objects to get strings."

Put Printing Methods in the method protocol "printing."

Method Comment

You have written a Composed Method (p. 21).

- How do you comment methods?

Back in the days of assembly language programming, the distance between what you intended as a programmer and how the computer forced you to express that intention was enormous. Every few lines (sometimes on every line), you needed a little story to help you understand what the next few instructions really meant.

As programming languages progressed, moving the expression closer to what it really meant, the habit of commenting every few lines relaxed somewhat. Many commenting standards settled on a comment at the beginning of a procedure, explaining the purpose of the procedure and describing the arguments and return value.

I find no value in this kind of "template" comment. Someone recently asked me point blank, "What percentage of your methods have comments?" I answered, "Between 0 and 1 percent." Oh the uproar! As a sanity check, I asked a developer at one of my clients (where I had taught Smalltalk based on an earlier version of these patterns) what percentage of the methods of their 200 class system had comments. His answer, "between 0 and 1 percent." "Has that ever been a problem?" "No, never."

I have certainly heard extravagant claims of "self documenting" code over the years. Shoot, Forth was supposed to be self documenting. What is it about Smalltalk code written with these patterns that lets it communicate tactical information without any supporting prose?

The information in the "template" comment is captured in the code with various patterns; Intention Revealing Selector communicates what the method does; Type Suggesting Parameter Name says what the arguments are expected to be; and various types of method patterns suggest return types, like Query Method for methods returning Booleans.

There is another important topic to communicate about a procedure—how it handles the various cases it is coded for. In Smalltalk, important cases become objects in their own right (see Choosing Message below), so each method only computes a single case. The result is code that communicates all the necessary tactical information to the reader.

Regardless of how well the system as a whole is put together, the big picture cannot easily be read method by method. There has to be another way of teaching the reader about the system as a whole. I use literate programs, although class and package comments will do in a pinch. However, trying to shoehorn a description of the architecture into a method comment is unlikely to work well, if only because the reader most likely won't stumble across it.

- *Communicate important information that is not obvious from the code in a comment at the beginning of the method.*

Here are examples of information that can be difficult to communicate solely through the code:

- Method dependencies—Sometimes one method must be invoked before another can execute correctly. A comment can warn the reader not to invoke one without the other. Sometimes you can use Composed Method or Execute Around Method to communicate the same information.
- To-do—I often write comments while I am prototyping to remind myself of some thought I don't want to lose. "Look at using a Dictionary later for efficiency," for example. When I reconsider the thought later, I delete the comment after choosing whether to follow it.
- Reasons for change, particularly base class—If you need to change something, the reason for the change is often not immediately apparent in the code. This often occurs when

changing a method supplied by a Smalltalk vendor. A comment helps a reader understand why you did what you did if you can't make the code say it.

Here is my favorite example of a useless comment:

```
(self flags bitAnd: 2r1000) = 1 “Am I visible?”
    ifTrue: [...]
```

A quick look at Composed Method yields:

```
isVisible
    ^(self flags bitAnd: 2r1000) = 1
```

And the original code turns into:

```
self isVisible
    ifTrue: [...]
```

I expect you to be skeptical of this pattern. Here's an experiment you can perform in the privacy of your own workstation. Write code with comments for every method. Go through your methods one by one and delete only those comments that duplicate exactly what the code says. If you can't delete a comment, see if you can refactor the code using these patterns (Composed Method and Intention Revealing Selector are especially useful) to communicate the same thing. I will be willing to bet that when you are done you will have almost no comments left.

One last example from client code:

```
Bin>>run
    "Tell my station to process me."
    self station process: self
```

You can translate the code directly into the comment:

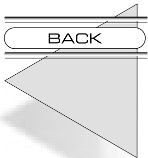
English	Code
Tell my station	self station
to process	process:
me	self

Messages

Messages are the heartbeat of a Smalltalk program. Without messages, there would be no program. Deftly managing this heartbeat is the first skill of the expert Smalltalk programmer. When you learn to see your program in terms of patterns of messages and you learn what can be done to that stream of messages to solve problems, then you will be able to solve any problem you can imagine in Smalltalk.

Procedural languages explicitly make choices. When you code up a case statement, you say once and for all what all the possibilities are. In Smalltalk, you use messages to make choices for you. The extra added bonus is that the set of choices is not set in concrete. You can come along later and add new choices without affecting the existing choices just by defining a new class.

This section talks about the tactical ways you can use the message stream. It gives you a toolbox of techniques for solving problems by manipulating the communication between objects.



Message

A Composed Method (p. 21) needs work done.

- How do you invoke computation?

In the earliest days of computing, this wasn't even a question. A program was one big routine that executed from start to finish.

As soon as programs got at all complicated, "program-as-a-routine" broke down. Conceptually, it was just too hard to manipulate the whole program at once. The limited resources of the era also came into play. When you had the same code duplicated in many places, you could save space by using a single copy of the code and invoking it everywhere you needed. The two factors, mental overload and memory overload, worked with each other. By giving the broken-out parts of the routine names, you saved space and you got a convenient tool for understanding the program a piece at a time.

Here things stood for a number of years. The client would invoke a subroutine. The subroutine would run. The client would regain control.

At the same time, there was a growing realization that a disciplined use of control structures was critical to the quality and cost of a program. If-then-else and case statements were invented to capture common ways to vary the execution of a program.

Simula brilliantly combined these two ideas. Conditional code says "execute this part of the routine or that part." A subroutine call says "execute that code over there." A message says "execute this routine over here or that routine over there, I don't really care."

Smalltalk went a step further by making messages the sole control structure in the system. All procedural control structures, conditionals and loops, are implemented in terms of messages. For the most part, explicit conditional logic plays a much smaller role in a Smalltalk program than a procedural program. Messages do most of the work.

- *Send a named message and let the receiving object decide what to do with it.*

Since everything in Smalltalk happens as a result of a message, it's tough to pick out one or two examples. #size is a message you can send to any object to get the number of elements (exclusive of named variables) it contains.

FORWARD

Use Delegation (p. 64) to get another object to do work for you. A Choosing Message (p. 45) invokes one of several alternatives. A Decomposing Message (p. 47) documents intent and provides for later refinement. An Intention Revealing Message (p. 48) maps intention to implementation. Use Super (p. 59) to invoke behavior in a superclass.

Choosing Message

You are using a Message (p. 43).

- How do you execute one of several alternatives?

The long term health of a system is all about managing themes and variations. When you first write a program, you have a particular theme in mind. Setting the program free in the world inevitably suggests all sorts of variations on what you first thought was a simple task.

Procedural programs implement variations with conditional logic, either if-then-else or case statements. Two problems arise from such hard-coded logic. First, you cannot add a new variation without modifying the logic. This can be a ticklish operation, getting the new variation in without disturbing the existing variations. Second, such logic tends to propagate. You do not have to account for the variation in one place, you have to account for it in several. Adding a new variation means tickling all of the places where the logic lives.

Messages provide a disciplined way to handle theme-and-variation programming. Because the variations live in different objects, they have much less opportunity to interfere with each other than just putting the variations in different parts of the same routine. The client, the object invoking the variations, is also isolated from what variation is currently active.

Adding a new variation is as simple as adding a new object that provides the same set of messages the other variations provide and introducing it to the object that wants to invoke the variations.

Sometimes, even when beginners have several kinds of objects, they still resort to conditional logic:

```
responsible := (anEntry isKindOf: Film)
    ifTrue: [anEntry producer]
    ifFalse: [anEntry author]
```

Code like this can always be transformed into communicative, flexible code by using a Choosing Message:

```
Film>>responsible
	^self producer
Entry>>responsible
	^self author
```

Now you can write:

```
responsible := anEntry responsible
```

But you probably don't need the Explaining Temporary Variable any more.

- *Send a message to one of several different kinds of objects, each of which executes one alternative.*

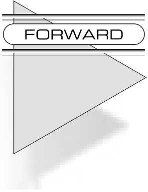
When you begin a program, you won't be able to anticipate the variations. As your program matures, you will see explicit conditional logic creep in. When you can see the same logic repeated in several places, it is time to find a way to represent the alternatives as objects and invoke them with a choosing message.

Here are some examples of choosing messages:

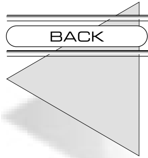
Message	Alternatives
Number>>+ aNumber	Different code will be invoked depending on what kind of Number the receiver is. Floats add differently than Integers, which add differently than Fractions.
Object>>printOn: aStream	Every object has the opportunity to change how it is represented to the programmer as a String.
Collection>>includes:	Different collections implement this very differently. The default implementation takes time proportional to the size of the collection. Others take

constant time.

If a Choosing Message is sent to self, it is done so in anticipation of future refinement by inheritance.



Give the message an Intention Revealing Selector (p. 49). Look at the section on Methods (p. 20) for examples of the kind of code that can be invoked as variations.



Decomposing Message

You are using a Message (p. 43) to break a computation into parts.

- How do you invoke parts of a computation?

A Choosing Message gets work done. It is the equivalent of a case statement in procedural languages. Depending on the circumstance, different code is invoked.

Another way messages are used is to break a computation down into pieces. As you are writing the code, you don't think about possible variations. A method is getting too big and you need to break it into parts so you can understand it better. Alternatively, you may have noticed that two or more methods have similar parts and you'd like to put the parts in a single method.

This is very similar to the way subroutines are used in procedural programming. You take a big routine and break it into pieces.

Smalltalk code reveals a much more aggressive attitude towards decomposing code than other languages. Most style guides say, "Keep the code for a routine on one page." Most good Smalltalk methods fit into a few lines, certainly less than ten and often three or four.

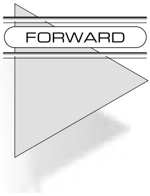
Partly this is possible because the abstractions Smalltalk provides are higher level than what you find in most languages. You don't spend three or four lines expressing iteration, you spend one word. Partly, it is possible because Smalltalk's programming tools let you manage smaller pieces easily.

- *Send several messages to "self."*

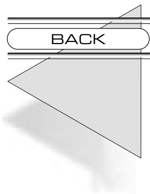
The classic example of this from the original Smalltalk image was:

```
Controller>>controlActivity
    self
        controlInitialize;
        controlLoop;
        controlTerminate
```

Later, these messages all became Choosing Messages because they were all overridden a hundred different ways.



Use Composed Method (p. 21) to break the method into pieces. Give each method an Intention Revealing Selector (p. 49). Use Intention Revealing Messages (p. 48) to communicate intent separate from implementation.



Intention Revealing Message

You are using a Message (p. 43) to invoke a computation. You may be hiding the use of Pluggable Behavior (p. 69).

- How do you communicate your intent when the implementation is simple?

These messages have to be the most frustrating part of learning Smalltalk. You see a message like “highlight:” and you think, “This has to be something interesting.” Instead, you see:

```
ParagraphEditor>>highlight: aRectangle
    self reverse: aRectangle
```

What’s going on?

Communication. Most importantly, one line methods are there to communicate. If I have the above method, the rest of the code in the object can be written in terms of highlighting. I want to highlight an area, so I send highlight. Makes sense.

I could mechanically replace all the invocations of highlight with invocations of reverse. The code would run the same. However, all the invoking code reveals the implementation—"I highlight by reversing a Rectangle."

The other advantage of code written to reveal intention and conceal implementation is that it is much easier to refine by inheritance. If I want a ParagraphEditor that highlights in color, I can make a subclass of ParagraphEditor and override a single method—highlight:.

Intention Revealing Messages are the most extreme case of writing for readers instead of the computer. As far as the computer is concerned, both versions are fine. The one that separates intention (what you want done) from implementation (how it is done) communicates better to a person.

- *Send a message to "self." Name the message so it communicates what is to be done rather than how it is to be done. Code a simple method for the message.*

Here are some examples of Intention Revealing Messages and their implementation:

```
Collection>>isEmpty
	^self size = 0
Number>>reciprocal
	^1 / self
Object>>= anObject
	^self == anObject
```

Give the message an Intention Revealing Selector (p. 49).

Intention Revealing Selector

You may be naming a method: a Constructor Method (p. 23), Conversion Method (p. 28), Converter Constructor Method (p. 26), or Execute Around Method (p. 37). You may be naming a message: Decomposing Message (p. 47), Choosing Message (p. 45), or Intention Revealing Message (p. 48). You may be implementing Double Dispatch (p. 55).

- What do you name a method?

You have two options in naming methods. The first is to name the method after how it accomplishes its task. Thus, searching methods would be called:

```
Array>>linearSearchFor:
Set>>hashedSearchFor:
BTree>>treeSearchFor:
```

The most important argument against this style of naming is that it doesn't communicate well. If I have code that invokes three other objects, I have to read and understand three different pieces of implementation before I can understand the code.

Also, naming methods this way results in code that knows what kind of object it is dealing with. If I have code that works with an Array, I can't substitute a BTree or a Set.

The second option is to name a method after what it is supposed to accomplish and leave "how" to the various method bodies. This is hard work, especially when you only have a single implementation. Your mind is filled with how you are about to accomplish the task, so it's natural that the name follow "how." The effort of moving the names of method from "how" to "what" is worth it, both long term and short term. The resulting code will be easier to read and more flexible.

- *Name methods after what they accomplish.*

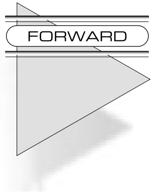
Applying this to the example above, we would name all of the messages "searchFor:."

```
Collection>>searchFor:
```

Really, though, searching is a way of implementing a more general concept, inclusion. Trying to name the message after this more general "what" leads us to "includes:" as a selector.

```
Collection>>includes:
```

Here's a simple exercise that will help you generalize names of messages with a single implementation. Imagine a second, very different implementation. Then, ask yourself if you'd give that method the same name. If so, you've probably abstracted the name as much as you know how to at the moment.



Once you name a method, write its body using Composed Method (p. 21). Format the selector in the method with an Inline Message Pattern (p. 172). Add a Collecting Parameter (p. 75)if necessary to collect results.

Dispatched Interpretation

- How can two objects cooperate when one wishes to conceal its representation?

Encoding is inevitable in programming. At some point you say, "Here is some information. How am I going to represent it?" This decision to encode information happens a hundred times a day.

Back in the days when data was separated from computation, and seldom the twain should meet, encoding decisions were critical. Any encoding decision you made was propagated to many different parts of the computation. If you got the encoding wrong, the cost of change was enormous. The longer it took to find the mistake, the more ridiculous the bill.

Objects change all this. How you distribute responsibility among objects is the critical decision, encoding is a distant second. For the most part, in well factored programs, only a single object is interested in a piece of information. That object directly references the information and privately performs all the needed encoding and decoding.

Sometimes, however, information in one object must influence the behavior of another. When the uses of the information are simple, or the possible choices based on the information limited, it is sufficient to send a message to the encoded object. Thus, the fact that boolean values are represented as instances of one of two classes, True and False, is hidden behind the message #ifTrue:ifFalse:.

```
True>>ifTrue: trueBlock ifFalse: falseBlock
    ^trueBlock value
False>>ifTrue: trueBlock ifFalse: falseBlock
    ^falseBlock value
```

We could encode boolean values some other way, and as long as we provided the same protocol, no client would be the wiser.

Sets interact with their elements like this. Regardless of how an object is represented, as long it can respond to #= and #hash, it can be put in a Set.

Sometimes, encoding decisions can be hidden behind intermediate objects. An ASCII String encoded as eight-bit bytes hides that fact by conversing with the outside world in terms of Characters:

```
String>>at: anInteger
    ^Character asciiValue: (self basicAt: anInteger)
```

When there are many different types of information to be encoded, and the behavior of clients changes based on the information, these simple strategies won't work. The problem is that you don't want each of a hundred clients to explicitly record in a case statement what all the types of information are.

For example, consider a graphical Shape represented by a sequence of line, curve, stroke, and fill commands. Regardless of how the Shape is represented internally, it can provide a message #commandAt: anInteger that returns a Symbol representing the command and #argumentsAt: anInteger that returns an array of arguments. We could use these messages to write a PostScriptShapePrinter that would convert a Shape to PostScript:

```
PostScriptShapePrinter>>display: aShape
    1 to: aShape size do:
        [:each || command arguments |
        command := aShape commandAt: each.
        arguments := aShape argumentsAt: each.
        command = #line ifTrue:
            [self
                printPoint: (arguments at: 1);
                space;
                printPoint: (arguments at: 2);
                space;
                nextPutAll: 'line'].
        command = #curve...
        ...]
```

Every client that wanted to make decisions based on what commands were in a Shape would have to have the same case statement, violating the "once and only once" rule. We need a solution where the case statement is hidden inside of the encoded object.

- *Have the client send a message to the encoded object. Pass a parameter to which the encoded object will send decoded messages.*

The simplest example of this is Collection>>do:. No matter what kind of collection you have, you can always send it #do:. By passing a one argument Block (or any other object that responds to #value:), you are assured that the code will work, no matter whether the Collection is encoded as a linear list, an array, a hash table, or a balanced tree.

This is a simplified case of Dispatched Interpretation because there is only a single message coming back. For the most part, there will be several messages. For example, we can use this pattern with the Shape example. Rather than have a case statement for every command, we have a method in PostScriptShapePrinter for every command. For example:

```
PostScriptShapePrinter>>lineFrom: fromPoint to: toPoint
	self
		printPoint: fromPoint;
		space;
		printPoint: toPoint;
		space;
		nextPutAll: 'line'
```

Rather than Shapes providing #commandAt: and #argumentsAt:, they provide #sendCommandAt: anInteger to: anObject, where #lineFrom:to: is one of the messages that could be sent back. Then, the original display code could read:

```
PostScriptShapePrinter>>display: aShape
	1 to: aShape size do:
		[:each |
		aShape
			sendCommandAt: each
			to: self]
```

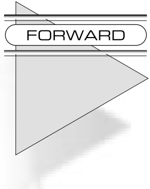
This could be further simplified by giving Shapes the responsibility to iterate over themselves:

```
Shape>>sendCommandsTo: anObject
	1 to: self size do:
		[:each |
		self
			sendCommandAt: each
			to: anObject]
```

With this, the original display code becomes:

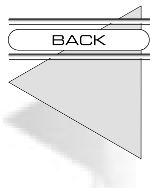
```
PostScriptShapePrinter>>display: aShape
	aShape sendCommandsTo: self
```

The name "dispatched interpretation" comes from the distribution of responsibility. The encoded object "dispatches" a message to the client. The client "interprets" the message. Thus, the Shape dispatches messages like #lineFrom:to: and #curveFrom:mid:to:. It's up to the clients to interpret the messages, with the PostScriptShapePrinter creating PostScript and the ShapeDisplayer displaying on the screen.



You will have to design a Mediating Protocol (p. 57) of messages to be sent back. Computations where both objects have decoding to do need Double Dispatch (p. 55).

Double Dispatch



You have a Dispatched Interpretation (p. 51) between two families of objects. You may be implementing a complex Equality Method (p. 124).

- How can you code a computation that has many cases, the cross product of two families of classes?

This pattern helps manage another of Smalltalk's engineering compromises—method dispatch. When you send a message to an object, and you include an argument, only the class of the receiver is taken into account when looking for a corresponding method. Ninety-nine percent of the time this causes you no trouble. There are a few cases, though, where the logic to be invoked really depends not just on the class of the receiver, but the class of one of the arguments as well. In fact, which object is the receiver and which is the argument may be entirely arbitrary.

		Argument	
		C	**D**
Receiver	**A**	Method 1	Method 2
	B	Method 3	Method 4

One classic example where this relationship exists is arithmetic. When you add an Integer to an Integer you want one method, when you add a Float

to a Float you want another, an Integer and a Float another, and a Float and an Integer another.

The procedural solution to this situation is to have a big case statement. Like all explicit case logic, this is difficult to maintain and extend, even though it has the advantage of putting all the program logic in one place.

The solution is adding a layer of messages that get both objects involved in the computation. As with Self Delegation, this causes you to create more messages, but the additional complexity is worth it.

- *Send a message to the argument. Append the class name of the receiver to the selector. Pass the receiver as an argument.*

The arithmetic example can be coded as follows. Integer and Float both Double Dispatch to the argument:

```
Integer>>+ aNumber
	^aNumber addInteger: self
Float>>+ aNumber
	^aNumber addFloat: self
```

Integer and Float both have to implement both flavors of addition. The Integer-Integer and Float-Float cases are handled as primitives.

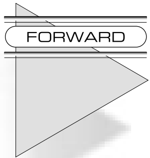
```
Integer>>addInteger: anInteger
	<primitive: 1>
Float>>addFloat: aFloat
	<primitive: 2>
```

When you have one number of each class, you have to convert the Integer to a Float and start over:

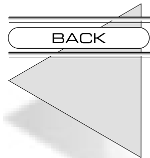
```
Integer>>addFloat: aFloat
	^self asFloat addFloat: aFloat
Float>>addInteger: anInteger
	^self addFloat: anInteger asFloat
```

In the worst case, Double Dispatch can lead to N × M methods, where N is the number of classes of the original receiver and M is the number of classes of the original argument. Practically speaking, the receiver classes are usually related by inheritance, as are the argument classes, so many common implementations can be factored out.

A reviewer suggested another good use for Double Dispatch—implementing drag-and-drop operations. You want to execute different code depending on what kind of object is being dragged over what kind of receiver. The simplest and most flexible way to implemement this is with Double Dispatch.



Create a Mediating Protocol (p. 57) with which the objects communicate. Type Suggesting Parameter Names (p. 174) are important for keeping track of how much you know at any stage of the process.



Mediating Protocol

You are implementing Dispatched Interpretation (p. 51) or Double Dispatch (p. 55).

- How do you code the interaction between two objects that need to remain independent?

For the most part, when you write a program that involves the cooperation of two objects, you create methods as needed. The dialog grows organically. When you finish, the two objects work together, but you don't necessarily have a strong sense of all the messages flowing back and forth.

Most of the time, this sort of ad hoc interaction doesn't cost you much. The changes you need to make involve changing one object or the other and occasionally adding to the messages going between them.

You need to make the protocol between the objects more visible when you decide to replace one or the other of them. The important question for you then becomes, "Exactly what messages flow between these two objects?"

When you find the answer, a list of message selectors, you will probably have some work to do. First, you need to look at the words in the selectors and see if they form a coherent system. Protocols that grow piecemeal tend to accumulate little inconsistencies. Sometimes, you will not have consistent opposites in the messages, as in #show being the opposite of #makeInvisible. Sometimes, you will not have consistently made selectors plural, as in #addEmployees: being the opposite of #removeAllEmployees:.

Because you are finding it necessary to replace one of the objects in the interaction, it is likely that others will have to create other replacements in the future. If the words in the protocol are consistent and clearly presented, they will be able to quickly create their replacements, using your code as examples.

- *Refine the protocol between the objects so the words used are consistent.*

In VisualWorks, #value and #value: is the Mediating Protocol between the user interface components and the application model.

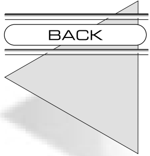
In the Double Dispatch example, the Mediating Protocol is #addFloat: and #addInteger:. Of course, if we finished mixed mode arithmetic the protocol would be much larger.

I worked with Smalltalk/V for the Macintosh for a couple of years. One of the exercises I tried was replacing the Smalltalk-programmed TextPane with a wrapper around the native Macintosh text editor. Smalltalk/V used a Model/Pane/Dispatcher-based user interface framework, so there was a TextDispatcher associated with the pane. Its purpose was to interpret user input and pass along meaningful messages to the pane.

For a while, I tried just sticking the new pane in and debugging my way to health. It didn't take long before I realized there was no way that was going to work. There was just too much going on between the TextPane and the TextDispatcher. So I sat down with the code and recorded every message that went from the pane to the dis-

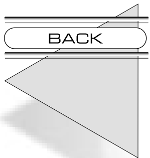
patcher and vice versa. In the end, I had only a handful of messages going from the pane to the dispatcher, but the dispatcher was sending the pane 56 different messages.

With this Mediating Protocol of 56 messages in place, I could sit down and design the new pane to support those messages. When I got them all implemented, I knew I was done.



Put all the methods to support a Mediating Protocol in a single method protocol, so they are easy to find and duplicate.

Examine each message to make sure it has an Intention Revealing Selector (p. 49).



Super

You are sending a Message (p. 43).

- How can you invoke superclass behavior?

An object executes in a rich context of state and behavior, created by composing together the contexts of its class and all of its class' superclasses. Most of the time, code in the class can be written as if the entire universe of methods it has available is flat. That is, take the union of all the methods up the superclass chain and that's what you have to work with.

Working this way has many advantages. It minimizes any given method's reliance on inheritance structure. If a method invokes another method on self, as long as that method is implemented somewhere in the chain, the invoking method is happy. This gives you great freedom to refactor code without having to make massive changes to methods that assume the location of some method.

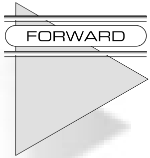
There are important exceptions to this model. In particular, inheritance makes it possible to override a method in a superclass. What if the subclass method wants some aspect of the superclass method? Good style boils down to one rule: say things once and only once. If the subclass method were to contain a copy of the code from the superclass method, the result would no longer be easy to maintain. We would have to remember to update both or (potentially) many copies at once. How can we resolve the tension between the need to override, the need to retain the illusion of a flat space of methods, and the need to factor code completely?

- *Invoke code in a superclass explicitly by sending a message to "super" instead of "self." The method corresponding to the message will be found in the superclass of the class implementing the*

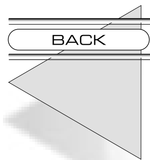
sending method.

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:

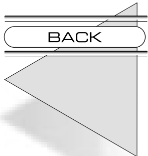
```
Sub>>initialize
    super initialize.
    b := self defaultB
```

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
```

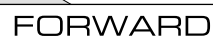
However, sometimes you have to work with superclasses that are not completely factored (i.e. the superclass does not implement #defaultSum). You are faced with the choice of either copying code or using super and accepting the costs of tighter subclass/superclass coupling. Most of the time, the addi-

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?

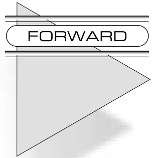
Inheritance is the primary built-in mechanism for sharing implementation in Smalltalk. However, inheritance in Smalltalk is limited to a single superclass. What if you want to implement a new object like A but also like B? Also, inheritance carries with it potentially staggering long-term costs. Code 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?

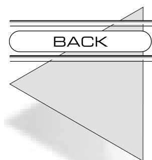
- *Pass part of its work on to another object.*

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.*

The typical example of this is an object that acts like a Collection (at least a little) but has lots of other protocol. Rather than waste inheritance by subclassing one of the collection classes, your object refers to a Collection. From a client's perspective, though, you respond to protocol like do: or at:put:.

The Collection doesn't care who invoked it. No state from the delegating object is required. The identity of the delegating object is irrelevant.

Here's an example—a Vector that holds only Numbers. We could implement it by subclassing Collection, but there are likely to be many messages that don't make sense for a Vector. Rather than subclass Collection and block out scads of messages, we can subclass object and delegate only those messages we want.

```
Vector
	superclass: Object
	instance variables: elements
```

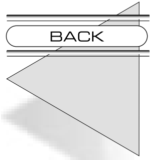
We create a Vector with a given number of elements:

```
Vector class>>new: anInteger
	^self new setElements: (Array new: anInteger)
Vector>>setElements: aCollection
	elements := aCollection
```

We'll ignore the arithmetic nature of Vectors and focus on how it delegates. Sometimes, clients want to treat a Vector as a Collection of Numbers. When someone iterates over a Vector, it delegates to its "elements" instance variable:

```
Vector>>do: aBlock
	elements do: aBlock
```

This is an example of Simple Delegation. You can imagine implementing at:, at:put:, size, etc. the same way.



Self Delegation

You are using Delegation (p. 64).

- How do you implement delegation to an object that needs reference to the delegating object?

The issues are the same for Self Delegation as for Simple Delegation. Do you need the identity of the original delegating object? Do you need state from the delegating object?

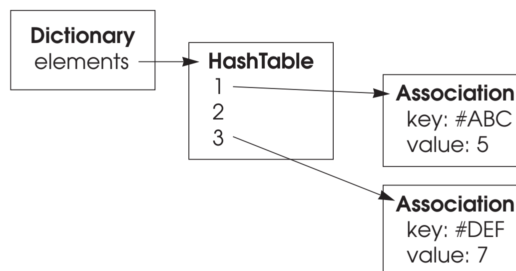
If the answer to either of these questions is "yes," Simple Delegation won't work. Somehow, the delegate needs access to the delegating object.

One way to give the delegate access is to include a reference from the delegate back to the delegating object. This approach has a number of drawbacks. The backwards reference introduces additional programming complexity. Every time the delegate changes, the reference in the old delegate has to be destroyed and the reference in the new delegate set. More importantly, each delegate can only be used by one delegating object at a time. If creating multiple copies of the delegate is expensive or impossible, this simply won't work.

The other approach, the one suggested here, is to pass the delegating object along as an additional parameter. This introduces a variant of the original method, which isn't great, but the additional flexibility of this approach is worth the cost.

- *Pass along the delegating object (i.e. "self") in an additional parameter called "for:"*

The Digitalk Visual Smalltalk 3.0 image has an excellent example of Self Delegation. The implementation of hashed collections, like Dictionaries, is divided into two parts. The first is the Dictionary, the second is a HashTable. There are variants of HashTables that are efficient in different circumstances. The same collection might delegate to different HashTables at different times, depending on its



characteristics (how big, how full, etc.)

The hash value of an object is implemented differently for different kinds of Collections. Dictionaries compute hash by sending "hash." IdentityDictionaries compute it by sending "basicHash." This is implemented using Self Delegation. When the Collection sends a message to the HashTable to add an element, it passes itself along:

```
Dictionary>>at: keyObject put: valueObject
	self hashTable
		at: keyObject
		put: valueObject
		for: self
```

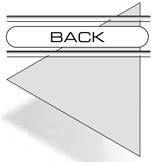
The HashTable computes the hash value by sending back a message to the Collection:

```
HashTable>>at: keyObject put: valueObject for: aCollection
	| hash |
	hash := aCollection hashOf: keyObject.
	...
```

Dictionaries and IdentityDictionaries implement this message differently:

```
Dictionary>>hashOf: anObject
	^anObject hash
IdentityDictionary>>hashOf: anObject
	^anObject basicHash
```

Self Delegation allows the hierarchy of hashed Collections to be



independent of the hierarchy of HashTables.

If the delegate needs different logic depending on who is delegating, use Double Dispatch (p. 55).

Pluggable Behavior

- How do you parameterize the behavior of an object?

The conventional model of objects is that different instances of the same class have different state and the same behavior. Every Point can have different values for x and y, but they all use the same logic to compute "translatedBy:." When you want different logic, you use a different class.

Using classes to specify behavior is simple. The programming tools are set up to help readers understand the behavior of your system statically, without necessarily having to run the code.

This model works for 90 percent of the objects you will create. Creating classes comes at a cost, though, and sometimes different classes don't effectively communicate how you think about a problem.

Classes are an opportunity. Each one will be useful to instantiate and/or specialize. However, each class you create places a burden on you, as the writer, to communicate its purpose and implementation to future readers. A system with hundreds or thousands of classes will intimidate a reader. Managing a namespace across many classes is expensive. You would like to invoke the costs of a new class only when there is a reasonable payoff. A large family of classes with only a single method each is unlikely to be valuable.

The other problem with specializing behavior only through classes is that classes are not flexible. Once you have created an object of a certain class, you cannot change that object's class without completely ruining the ability to understand the code statically. Only watching carefully while single stepping will give you insight into how such code runs. Smalltalk's single inheritance also does not allow specialization along several different axes at the same time.

If you are going to use Pluggable Behavior, here are the issues you need to consider:

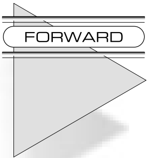
- How much flexibility do you need?
- How many methods will need to vary dynamically?

- How hard is it to follow the code?
- Will clients need to specify the behavior to be plugged, or can it be hidden within the plugged object?

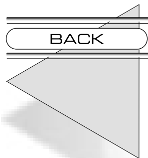
How can you specify different logic in different instances when creating lots of little classes or changing classes at run time won't work?

- *Add a variable that will be used to trigger different behavior.*

Typical examples of pluggable behavior are objects that have to interface with a variety of other objects, like user interface components that have to display the contents of many different objects. Using Pluggable Behavior is a much better solution than creating a hundred different subclasses, each differing from each other in only one or two methods.



For simple behavior changes, use a Pluggable Selector (p. 70). A Pluggable Block (p. 73) gives you more flexibility. Hide the implementation of pluggability behind an Intention Revealing Message (p. 48).



Pluggable Selector

You need simple Pluggable Behavior (p. 69).

- How do you code simple instance specific behavior?

The simplest way to implement Pluggable Behavior is to store a selector to be performed.

Let's say we have implemented a ListPane. We create a method that takes one of the elements of the collection to be displayed and returns a String:

```
ListPane>>printElement: anObject
	^anObject printString
```

After awhile, we notice that there are many subclasses of ListPane that only override this one method:

```
DollarListPane>>printElement: anObject
    ^anObject asDollarFormatString
DescriptionListPane>>printElement: anObject
    ^anObject description
```

It hardly seems worth the cost of all these subclasses if all they are going to do is override one method. A simpler solution is to make ListPane itself a little more flexible, so different instances send different messages to their elements. We add a variable called "printMessage" and modify #printElement:

```
ListPane>>printElement: anObject
    ^anObject perform: printMessage
```

To preserve the previous behavior, we would have to initialize the printMessage:

```
ListPane>>initialize
    printMessage := #printString
```

Pluggable Selector meets the Pluggable Behavior criteria as follows:

Readability—Pluggable Selector is harder to follow than simple class-based behavior. By looking at an object with an inspector, you can tell how it will behave. You don't necessarily have to single step through the code.

Flexibility—The methods for the Pluggable Selectors must be implemented in the receiving object. The set of possible methods to be invoked should change at the same rate as the rest of the object.

Extent—Pluggable selectors should be used no more than twice per object. Any more than that and you risk obscuring the intent of the program. Use State Object if you need more dimensions of variability.

- *Add a variable that contains a selector to be performed. Append "Message" to the Role Suggesting Instance Variable Name. Create a Composed Method that simply performs the selector.*

Pluggable Selector is also useful for a simple kind of constraint. For example, if you wanted to locate one visual component relative to some part of another, we could use Pluggable Selector to create a RelativePoint:

```
Class: RelativePoint
	superclass: Object
	instance variables: figure locationMessage
```

Here is the Constructor Method:

```
RelativePoint class>>centered: aFigure
	^self new
		setFigure: aFigure
		message: #center
RelativePoint>>setFigure: aFigure message: aSymbol
	figure := aFigure.
	locationMessage := aSymbol
```

To use a RelativePoint, you send it messages like #x and #y, just like a regular Point.

```
RelativePoint>>asPoint
	^figure perform: locationMessage
RelativePoint>>x
	^self asPoint x
```

Once you have this, you can go crazy duplicating all the necessary Point protocol, re-engineering for performance, etc. As an example of Pluggable Selector, however, the interesting observation is that you don't have to make a subclass for CenteredRelativePoint, TopLeftRelativePoint, etc.; you can capture the variability in a single selector.

Pluggable Block

You need complex Pluggable Behavior (p. 69) that is not implemented by the plugged object.

- How do you code complex Pluggable Behavior that is not quite worth its own class?

Pluggable Selector works when the behavior to be invoked lives within the plugged object. Sometimes, though, the behavior can't live within the plugged object either because it is complex and not related to the plugged object's other responsibilities, because it is already implemented in another object not easily accessible to the plugged object, or because the range of behavior to be plugged was not known when the object was created.

The common solution in this case, particularly when the behavior is already implemented, is to plug in a Block to be evaluated rather than a selector to be performed. The block can be created anywhere, can access objects otherwise inaccessible to the plugged object through the use of a Block Closure, and can involve arbitrary amounts of logic.

Blocks used in such a general way come at enormous cost. You can never statically analyze the code to understand the flow of control. Even inspecting the plugged object is unlikely to unearth its secrets. Only by single stepping through the invocation of the block will the reader understand what is going on.

Blocks are also more difficult to store on external media than Symbols. Some Object Streams and object databases cannot store and retrieve Blocks.

- *Add an instance variable to store a Block. Append "Block" to the Role Suggesting Instance Variable Name. Create a Composed Method to evaluate the Block to invoke the Pluggable Behavior.*

The VisualWorks object PluggableAdaptor is a good example of a Pluggable Block. All the objects in the ValueModel family, of which

PluggableAdaptor is one, provide the protocol #value and #value:. PluggableAdaptor implements these messages with a PluggableBlock. Here is a simplified implementation:

```
Class: PluggableAdaptor
	superclass: ValueModel
	instance variables: getBlock setBlock
```

The Constructor Method sets the blocks:

```
PluggableAdaptor class>>getBlock: getBlock setBlock:
setBlock
	^self new
		setGetBlock: getBlock
		setBlock: setBlock
```

Notice that the Constructor Parameter Method has to use a variant of Type Suggesting Parameter Name because the obvious parameter names are already used for instance variable names.

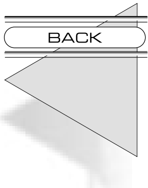
```
PluggableAdaptor>>setGetBlock: gBlock setBlock: sBlock
	getBlock := gBlock.
	setBlock := sBlock
```

We can implement #value and #value: by invoking the Pluggable Blocks.

```
PluggableAdaptor>>value
    ^getBlock value
PluggableAdaptor>>value: anObject
    putBlock value: anObject
```

Now we can connect any object that expects #value and #value: to any other object:

```
Car>>speedAdaptor
    ^PluggableAdaptor
        getBlock: [self speed]
        putBlock: [:newSpeed | self speed: newSpeed]
```



Collecting Parameter

You have written an Intention Revealing Selector (p. 49).

- How do you return a collection that is the collaborative result of several methods?

One of the downsides of Composed Method is that it occasionally creates problems because of linkages between the small methods. A state that would have been stored in a temporary variable now has to be shared between methods.

The simplest solution to this problem is to leave all the code in a single method and use temporary variables to communicate between the parts of the method. All the benefits you expect from Composed Method vanish if you take this approach. The code is less revealing, more difficult to reuse and refine, and harder to modify.

Another solution is to add an instance variable to the object that is shared only between the methods. This variable is very different than the other variables in the object. It is only valid while the methods are executing, not for the lifetime of the object. Instance variables should exist to communicate and store only state that must go together.

We can solve the problem by adding an additional parameter that is passed to all the methods. I hesitate to add layers of methods like this, except when they do useful work. In this case, because the other solutions aren't valid, this is the right solution.

- *Add a parameter that collects their results to all of the submethods.*

Here's an example. The following code extracts all the married men and unmarried women from a collection of people:

```
marriedMenAndUnmarriedWomen
	| result |
	result := OrderedCollection new.
	self people do: [:each | each isMarried & each isMan
ifTrue: [result add: each]].
	self people do: [:each | each isUnmarried & each
isWoman ifTrue: [result add: each]].
	^result
```

Using Composed Method, we put each iteration into its own method:

```
marriedMen
	| result |
	result := OrderedCollection new.
	self people do: [:each | each isMarried & each isMan
ifTrue: [result add: each]].
	^result
unmarriedWomen
	| result |
	result := OrderedCollection new.
	self people do: [:each | each isUnmarried & each
isWoman ifTrue: [result add: each]].
	^result
```

Now the question is how to compose the two methods. For an example this simple, I would probably use Concatenation to write:

```
marriedMenAndUnmarriedWomen
	^self marriedMen , self unmarriedWomen
```

but that doesn't demonstrate this pattern very well. If several layers of methods, or several objects, are involved, it is more clear to modify the submethods. Instead of returning a Collection, each adds its objects to a Collection. The code then becomes:

```
marriedMenAndUnmarriedWomen
	| result |
	result := OrderedCollection new.
	self addMarriedMenTo: result.
	self addUnmarriedWomenTo: result.
	^result
addMarriedMenTo: aCollection
	self people do: [:each | each isMarried & each isMan
ifTrue: [aCollection add: each]]
addUnmarriedWomenTo: aCollection
	self people do: [:each | each isUnmarried & each
isWoman ifTrue: [aCollection add: each]]
```

This code contains fewer lines and is more direct than the original. (If this were production code, I would probably continue factoring via Composed Method to concentrate the similarities between addMarriedMenTo: and addUnmarriedWomenTo:.)

In general, use an OrderedCollection (p. 116) as the Collecting Parameter. You may use a Concatenating Stream (p. 165) as the Collecting Parameter if the objects to be collected are bytes or Characters. Use a Set (p. 119) if you want to avoid duplicates.

Index

#addFloat: Mediating Protocol 58

#addInteger Mediating Protocol 58

Array pattern 133–135
  example 191–201

arrays
  duplicate elements 118–119
  numbers in a range 135–137
  numbers in sequence 137–138

become method 8

behavior, definition 19

best practice, defined 1

blocks, formatting 177–178

Boolean Property Setting Method pattern 100–101

ByteArray pattern 135–137

Caching Temporary Variable pattern 106–108

Cascade pattern 183–185
  last message doesn't return receiver 186–188

Choosing Message pattern 45–47

Choosing Method pattern, example
191–201

classes
Array pattern 133–135
arrays
duplicate elements 118–119
numbers in a range 135–137
numbers in sequence 137–138
ByteArray pattern 135–137
Collection pattern 115–116
collections
duplicate elements 118–119
fixed number of elements 133–135
sorting 131–132
undetermined size, coding 116–117
unique elements 119–124
Dictionary pattern 128–131
Equality Method pattern 124–126
Hashing Method pattern 126–128
Interval pattern 137–138
objects
coding equality 124–126
mapping one to another 128–131
working with hashed collections
126–128
one-to-many relationships 115–116
OrderedCollection pattern 116–117
Qualified Subclass Name pattern
169–170
RunArray pattern 118–119
Set pattern 119–124
Simple Superclass Name pattern
168–169
SortedCollection pattern 131–132
subclasses, naming 169–170
superclasses, naming 168–169

Collect pattern 147–149

Collecting Parameter pattern 75–77

Collecting Temporary Variable pattern
105–106

Collection Accessor Method pattern
96–99

collection idioms
collections
concatenating 165–166
removing duplicates 154–155
sorting temporarily 155–156
Concatenating Stream pattern
165–166
Detect loops, optimizing 161–163
Duplicate Removing Set pattern
154–155
literals, searching for 159–161
Lookup Cache pattern 161–163
parser, writing 164–165
Parsing Stream pattern 164–165
Queue pattern 158–159
queues 158–159
Searching Literal pattern 159–161
Select/Reject loops, optimizing
161–163
Stack pattern 156–158
stacks 156–158
Temporarily Sorted Collection pattern
155–156

Collection pattern 115–116

collection protocols
Collect pattern 147–149
collections
concatenating 143–144
executing code across 144–145
executing code for each element
146–147
filtering 149–150
keeping running values 152–154
results of a message sent to each
object 147–149
searching 151–152
searching for elements 141–143
testing for empty 139–141
Concatenation pattern 143–144

Detect pattern 151–152
Do pattern 146–147
Enumeration pattern 144–145
Includes pattern 141–143
Inject:into: pattern 152–154
IsEmpty pattern 139–141
Select/Reject pattern 149–150

collections
concatenating 143–144, 165–166
duplicate elements 118–119
executing code across 144–145
executing code for each element 146–147
filtering 149–150
fixed number of elements 133–135
general access to 99–100
keeping running values 152–154
removing duplicates 154–155
results of a message sent to each object 147–149
searching 141–143, 151–152
sorting 131–132
temporarily 155–156
testing for empty 139–141
undetermined size, coding 116–117
unique elements 119–124
See also classes
See also collection idioms
See also collection protocol

Common State pattern 80–81
example 191–201

Comparing Method pattern 32–33

Composed Method pattern 21–22
example 191–201

Concatenating Stream pattern 165–166

Concatenation pattern 143–144

conditional code, formatting 178–179

Conditional Expression pattern 180–182
example 191–201

conditional logic 45–47

Constant Method pattern 87–89

constants, coding 87–89

Constructor Method pattern 23–24
example 191–201

Constructor Parameter Method pattern 25–26
example 191–201

Conversion pattern 28

Converter Constructor Method pattern 29–30

Converter Method pattern 28–29

Debug Print Method pattern 39–40
example 191–201

Decomposing Message pattern 47–48

Decomposing Method pattern
example 191–201

Default Value Method pattern 86–87

Delegation pattern 64–65

Detect loops, optimizing pattern 161–163

Detect pattern 151–152

Dictionary pattern 128–131

Direct Variable Access pattern 89–91
example 191–201

Dispatched Interpretation pattern 51–55
example 191–201

Do pattern 146–147

Double Dispatch pattern 55–57
example 191–201

Duplicate Removing Set pattern 154–155

enumeration blocks, naming parameters 182–183

Enumeration Method pattern 99–100
Enumeration pattern 144–145
Equality Method pattern 124–126
Execute Around Method pattern 37–39
Explaining Temporary Variable pattern 108–109
Explicit Initialization pattern 83–85
expressions
  formatting conditional 180–182
  reusing 109–110
  saving results of 103–106
  simplifying 108–109
Extending Super pattern 60–62
formatting 171–172
  blocks 177–178
  Cascade pattern 183–188
  conditional code 178–179
  Conditional Expression pattern 180–182
  conditional expressions 180–182
  enumeration blocks, naming parameters 182–183
  Guard Clause pattern 178–179
  Indented Flow Control pattern 175–177
  Inline Message Pattern pattern 172–174
  Interesting Return Value pattern 188–189
  message patterns 172–174
  messages
    indenting 175–177
    multiple to same receiver 183–185
  methods
    parameters, naming 174–175
    returning values from 188–189
  Rectangular Block pattern 177–178
  Simple Enumeration Parameter pattern 182–183
  Type Suggesting Parameter Name pattern 174–175
  Yourself pattern 186–188
Getting Method pattern 93–95
  example 191–201
Guard Clause pattern 178–179
Hashing Method pattern 126–128
Includes pattern 141–143
Indented Control Flow pattern 175–177
  example 191–201
Indirect Variable Access pattern 91–93
Inject:into: pattern 152–154
Inline Message Pattern pattern 172–174
  example 191–201
instance variables
  accessing 93–99
  Boolean Property Setting Method pattern 100–101
  changing values 95–96
  Collection Accessor Method pattern 96–99
  collections, general access to 99–100
  Common State pattern 80–81
  Constant Method pattern 87–89
  constants, coding 87–89
  Default Value Method pattern 86–87
  Direct Variable Access pattern 89–91
  Enumeration Method pattern 99–100
  Explicit Initialization pattern 83–85
  Getting Method pattern 93–95
  getting/setting 89–93
  Indirect Variable Access pattern 91–93
  initializing defaults 83–86
  initializing values 86–87
  naming 102–103
instances
  common states 80–81
  creating 23–24
  passing parameters to 25–26

variable states 82–83
Intention Revealing Messages pattern 48–49
Intention Revealing Selector pattern 49–51
example 191–201
Interesting Return Value pattern 188–189
example 191–201
Interval pattern 137–138
IsEmpty pattern 139–141
Lazy Initialization pattern 85–86
literals, searching for 159–161
Lookup Cache pattern 161–163
Mediating Protocol pattern 57–59
Message pattern 43–44
message patterns, formatting 172–174
messages
Choosing Message pattern 45–47
Collecting Parameter pattern 75–77
conditional logic 45–47
Decomposing Message pattern 47–48
Delegation pattern 64–65
Dispatched Interpretation pattern 51–55
Double Dispatch pattern 55–57
Extending Super pattern 60–62
flow control 33–34
formatting multiple to same receiver 183–185
indenting 175–177
Intention Revealing Messages pattern 48–49
Intention Revealing Selector pattern 49–51
invoking in tandem 37–39
Mediating Protocol pattern 57–59
Message pattern 43–44
method dispatch 55–57
methods
breaking into parts 47–48
collections resulting form multiple 75–77
communicating intent 48–49
complex Pluggable Behavior 73–75
instance-specific behavior 70–73
invoking 43–44
naming 49–51
Modifying Super pattern 52–64
objects
cooperating 51–55
delegate access 67–68
interaction 57–59
invoking disinterested delegates 65–66
parameterizing behavior 69–70
sharing implementation without inheritance 64–65
Pluggable Behavior pattern 69–70
Pluggable Block pattern 73–75
Pluggable Selector pattern 70–73
Self Delegation pattern 67–68
Simple Delegation pattern 65–66
Super pattern 59–60
superclass behavior
adding to 60–62
invoking 59–60
modifying 52–64
Method Comment pattern 40–43
method dispatch 55–57
Method Object pattern 34–37
methods
become 8
breaking into parts 47–48
collections resulting form multiple 75–77
commenting 40–43
communicating intent 48–49
Comparing Method pattern 32–33

complex Pluggable Behavior 73–75
Composed Method pattern 21–22
Constructor Method pattern 23–24
Constructor Parameter Method pattern 25–26
Conversion pattern 28
Converter Constructor Method pattern 29–30
Converter Method pattern 28–29
Debug Printing Method pattern 39–40
definition 20
Execute Around Method pattern 37–39
instances
  creating 23–24
  passing parameters to 25–26
instance-specific behavior 70–73
invoking 43–44
messages
  flow control 33–34
  invoking in tandem 37–39
Method Comment pattern 40–43
Method Object pattern 34–37
naming 49–51, 95
objects
  converting 28–30
  creating, shortcut 26–27
  sorting 32–33
  testing properties of 30–32
parameters, naming 174–175
performance tuning 106–108
printing, default method 39–40
programs, dividing into methods 21–22
Query Method pattern 30–32
returning values from 188–189
Reversing Method pattern 33–34
setting boolean properties 100–101
Shortcut Constructor Method pattern 26–27
simplifying complexity 34–37

Modifying Super pattern 52–64

naming
  enumeration block parameters 182–183
  instance variables 102–103
  method parameters 174–175
  methods 49–51, 95
  subclasses 169–170
  superclasses 168–169
  temporary variables 110–111

objects
  coding equality 124–126
  converting 28–30
  cooperating 51–55
  copying 8
  creating, shortcut 26–27
  delegate access 67–68
  interaction 57–59
  invoking disinterested delegates 65–66
  mapping one to another 128–131
  parameterizing behavior 69–70
  sharing implementation without inheritance 64–65
  sorting 32–33
  testing properties of 30–32
  working with hashed collections 126–128

one-to-many relationships 115–116

OrderedCollection pattern 116–117

parser, writing 164–165

Parsing Stream pattern 164–165

patterns
  adopting 9–10
  definition 1
  examples 191–202
  learning 10–11
  reasons for 14–15
  roles of 15–16
  See also classes

See also collection idioms
See also collection protocols
See also formatting
See also instance variables
See also messages
See also methods
See also specific pattern names
See also temporary variables

performance tuning 8, 21

Pluggable Behavior pattern 69–70

Pluggable Block pattern 73–75

Pluggable Selector pattern 70–73

printing, default method 39–40

procedure calls {\i vs.} messages and methods 19–20

programming style 6–7

programs
become method 8
coding activities 1–3
copying objects 8
dividing into methods 21–22
exception handling 7–8
flow of control 21
lifecycle cost 5–6
performance tuning 8, 21
productivity 5
quality criteria 4–6
readability 21–22
risk 6
system structure 3–4
time to market 6

Qualified Subclass Name pattern 169–170

Query Method pattern 30–32

Queue pattern 158–159

queues 158–159

Rectangular Block pattern 177–178
example 191–201

Reusing Temporary Variable pattern 109–110

Reversing Method pattern 33–34

Role pattern, example 191–201

Role Suggesting Instance Variable Name pattern 102–103

Role Suggesting Temporary Variable Name pattern 110–111

RunArray pattern 118–119

Searching Literal pattern 159–161

Select/Reject loops, optimizing 161–163

Select/Reject pattern 149–150

Self Delegation pattern 67–68

Set pattern 119–124

Setting Method pattern 95–96

Shortcut Constructor Method pattern 26–27

Simple Delegation pattern 65–66

Simple Enumeration Parameter pattern 182–183
example 191–201

Simple Superclass Name pattern 168–169
example 191–201

SortedCollection pattern 131–132

Stack pattern 156–158

stacks 156–158

state
definition 19
See also instance variables
See also temporary variables

subclasses, naming 169–170

Suggesting Instance Variable Name pattern, example 191–201

Super pattern 59–60

superclass behavior
adding to 60–62
invoking 59–60
modifying 52–64

superclasses, naming 168–169

Temporarily Sorted Collection pattern 155–156

Temporary Variable pattern 103–104

temporary variables
Caching Temporary Variable pattern 106–108
Collecting Temporary Variable pattern 105–106
Explaining Temporary Variable pattern 108–109
expressions
reusing 109–110
saving results of 103–106
simplifying 108–109
methods, performance tuning 106–108
naming 110–111
Reusing Temporary Variable pattern 109–110
Role Suggesting Temporary Variable Name pattern 110–111
Temporary Variable pattern 103–104

temporary variables, naming 110–111

Type Suggesting Parameter Name pattern 174–175
example 191–201

#value: Mediating Protocol 58

#value Mediating Protocol 58

Variable State 82–83

Variable State pattern 82–83

variables, coding 87–89

Yourself pattern 186–188