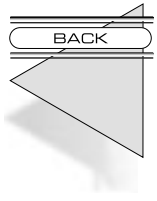
```
Stream>>print: anObject
        anObject printOn: self
Point>>printOn: aStream
        aStream
                print: x;
                nextPutAll: ' @ ';
                print: y
```

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*