Chapter

28

Eliminating Procedural Code

In procedural programming, we write a lot of code that gets information then makes a decision based on the information. In C, we see a lot of if/else if/else blocks, and a lot of switch statements. If we wrote the same type of code in Smalltalk we might see the following.

```
MyClass>>myDoLoop
   [self myProcessObject: self myGetObject] repeat

MyClass>>myProcessObject: anObject
   anObject isSuccessResponse
      ifTrue: [^self myProcessSuccess: anObject].
   anObject isFailureResponse
      ifTrue: [^self myProcessFailure: anObject].
   anObject isHeartbeat
      ifTrue: [^self myProcessHeartbeat: anObject].
   self myProcessUnknown: anObject
```

The trouble is that this code demonstrates procedural thinking. In fact, if you start writing code like this, the logical conclusion is "why is there no switch statement in Smalltalk?" At one time, we wrote a *Switch* class, but each time we used it we got into trouble. Eventually we realized that while a switch statement was seductive, it caused us to think procedurally and our objects got messed up. We finally removed the class and rewrote all the code that used it.

Tell, don't ask

How might we write the above code if we use the OO *tell, don't ask* approach. The classic solution is use polymorphism, to simply tell the object to process itself, and rely on each type of object being able to do this in its own way. Give the object responsibility and make it responsible for carrying out the action. For example,

```
MyClass>>myProcessObject: anObject
   anObject processYourself
```

Sometimes, however, this approach doesn't work because some or most of the behavior and knowledge is in MyClass. If the work will be shared between MyClass and the object it is processing, we can give the objects knowledge of each other. For example,

```
MyClass>>myProcessObject: anObject
   anObject processYourselfWith: self

OtherClass>>processYourselfWith: anObject.
   self myDoSomeStuff.
   anObject doSomeStuffWith: self.
   self myDoMore Stuff

MyClass>>doSomeStuffWith: anObject
   ....
```

If all the work will be done in MyClass, we need a mechanism to figure out what method in MyClass to execute. The classic answer to this problem is *double dispatching*, where the first object tells the second object to tell the first object what to do. In our example, MyClass doesn't know what method to execute, so it tells the other object to tell it what method to execute.

```
MyClass>>myProcessObject: anObject
   anObject processUsing: self
```

Then in each class that the object could belong to, we see the following.

```
SuccessResponse>>processUsing: anObject.
   anObject processSuccessResponse: self

FailureResponse>>processUsing: anObject.
   anObject processFailureResponse: self

Heartbeat>>processUsing: anObject.
   anObject processHeartbeat: self
```

We also add `processUsing`: to *Object* to trap all the situations where we come across an object that doesn't understand `processUsing:`. Ie, an object for which we haven't yet written `processUsing:`.

```
Object>>processUsing: anObject
   anObject processUnknown: self.
```

Back in MyClass, we implement the code that knows how to handle the different types of objects we might be processing.

```
MyClass>>processSuccess: anObject
   .. do success stuff ..

MyClass>>processFailure: anObject
   .. do failure stuff ..

MyClass>>processHeartbeat: anObject
   .. do heartbeat stuff ..
```

We also add `processUnknown:` to handle those situations where the other object inherited `processUsing:` from *Object*. Ie, where we forgot to write `processUsing:`.

```
MyClass>>processUnknown: anObject
   self error: anObject printString,
      ‘ does not understand processUsing:’.
```

Processing external objects

Another type of object we may need to deal with is an object we get from an outside source such as a socket or a serial line. For example, we might be talking with an external device and getting back status codes. Or we might be getting requests sent in over sockets, each request telling the program to do something different.

Typically we will get information in the form of a string and we'll have to figure out what to do based on the contents of the string. An obvious procedural answer is to compare the data with known strings and do different things for different data. However, we want an OO answer, so let’s look at ways to deal with this type of object without having to make explicit decisions.

perform:

One way is to use one of the `perform:` family of messages. When you write a Smalltalk method, you usually specify all the messages that it will send. So, although the arguments won't be know until runtime, all the message names are known at compile time (when you accept the method). The `perform:` family of messages give us the ability to create a symbol and use this symbol as a message name. By using `perform:`, we don't have to specify the message names when we write the method and can delay this knowledge until runtime. The `perform:` family has several members, depending on how many keywords there are in the message to be performed. For each keyword, you need one argument.

```
self perform: selector.
self perform: selector with: argument.
self perform: selector with: argument1 with: argument2.
self perform: selector with: argument1 with: argument2 with:
argument3.
self perform: selector withArguments: argumentArray.
```

Suppose that from an external device we get back status codes as numeric strings, and we take different actions for diffperen tc stus codes. We could prefix the code with a string, then perform this as a method. For example:

```
MyInterface>>handleResponse
   statusCode := self myDeviceResponse.
   selector := (‘msg’, statusCode) asSymbol.
   self perform: selector.
```

In a production system you would pass in additional informion to the perform mhod, so let’s assume that we get back a status code followed by a space followed by some real data.

```
MyInterface>>handleResponse
   response := self myDeviceResponse.
   selector := ('msg', (response copyUpTo: Character space))
asSymbol.
   self perform: selector with: response.
```

Instead of having meaningless methods such as `msg0215`, let's keep a dictionary of message selectors (alternatively, you could keep a dictionary of code blocks). The relationship between status codes and selectors could be set up in the class side `initialize` method, storing the relationships in an instance of *Dictionary*. We may have to take the same action for several status codes so this technique has the added benefit that we don't have to write several methods to do the same thing.

```
MyInterface class>>initialize
   "self initialize"
   StatusCodeDictionary := Dictionary new.
   StatusCodeDictionary
      at: '0215' put: #performDeviceOffline;
      at: '0216' put: #performInvalidParameters

MyInterface>>myStatusSelector: aString
   ^StatusCodeDictionary
      at: aString
      ifAbsent: [#performUnknownStatusCode]

MyInterface>>handleResponse
   response := self myDeviceResponse.
   selector := self myStatusSelector: (response copyUpTo: Character
space).
   self perform: selector with: response.
```

Notice that we've prefixed the method names in the dictionary with *perform*. A drawback to performing methods is that if the method name is created programatically, there will be no references to the method. If you run Class Reports (a tool in the Advanced Tools that offers some lint-like capabilities), it will tell you that the method is implemented but not sent. Unsent messages are always candidates for removal, so having a distinct prefix for performed methods reduces that likelihood that you will unknowingly remove methods that are needed.

Dictionary of classes

The technique shown above can be useful when we get some data and want to execute different methods based on the data. Sometimes, however, we will want to create a new object and have the object do the processing. For example, from a socket we get a string representing a request and the various request parameters. We want to create a request object of the right type then tell the request object to process itself.

One approach is to have a set of request classes, with a superclass of *MyRequest*. On the class side, MyRequest has a dictionary containing relationships between strings and class names. To create a request object, we ask the superclass, MyRequest, to create the appropriate type of request. An instance of InvalidRequest is created if the string does not correspond to a valid request type. This InvalidRequest should respond with error information when sent the message `processYourself`.

```
MyInterface>>handleRequest
   input := self mySocketInput.
   request := MyRequest newFrom: input.
   request processYourself

MyRequest class>>newFrom: aString
   requestType := aString copyUpTo: Character space.
   requestClass := RequestDictionary
```

```
            at: requestType
            ifAbsent: [#InvalidRequest].
        ^requestClass new initialize: aString.
```

The dictionary could be built explicitly as in the previous example. However, this has the disadvantage that every time you add a new request class, you have to remember to update the RequestDictionary. You're taking responsibility away from the objects themselves and putting it somewhere else.

Another approach is for each request class to know the string that is associated with it. In MyRequest we write an initialization method, `initializeRequests`, which asks all the subclasses of MyRequest for their string. Depending on whether you fileIn your code or keep it in the image, you could have `initialize` send `initializeRequests` or you could do `MyRequest initializeRequests` during product startup.

```
MyRequest class>>initializeRequests
   "self initializeRequests"
   RequestDictionary := Dictionary new.
   self allSubclassesDo:
      [:each | each requestName notNil
         ifTrue: [RequestDictionary
            at: each requestName
            put: each]]

MyRequest class>>requestName
   "Don't inherit the name. Make sure the request explicitly
implements it"
   ^(self class includesSelector: #myName)
      ifTrue: [self myName]
      ifFalse: [nil]

SomeRequestSubclass class>>myName
   ^'register'
```

We don't want a request class to inherit its request name from a superclass, so we have code to handle classes that don't implement `myName`. We simply don't put them in the dictionary. If we get a string for which there is no entry in the dictionary, the code in `newFrom:` returns the `InvalidRequest` class, which generates the appropriate error code when instantiated and sent the `processYourself` message.

Summary

To eliminate procedural code we should *tell rather than ask*. Ideally we use polymorphism and simply tell the object to process itself in some way. We may have to split the responsibility, in which case we may tell the object to process itself using us. Or we may use *double dispatching* to avoid a pseudo switch statement. With certain types of object we can use the `perform:` family of messages, or we can build dictionaries that store relationships between classes and another object such as a string or a symbol.