The Core/Support Split

Andrew P. Black
black@cse.ogi.edu

Department of Computer Science & Engineering
Oregon Graduate Institute of Science & Technology
Beaverton, Oregon, USA

A Pattern Description Submitted to EuroPLoP 2000

Context

You are writing several concrete classes that all implement the same interface. Each of the concrete classes uses a different representation for the same abstraction. As the program evolves, the interface must often be extended to include additional messages that the implementing classes should understand. These extensions are typically "utility" or "support" methods that provide functionality that has proved to be useful to a number of clients. In addition, sometimes new concrete classes are added to provide new implementations of the interface.

The need to add new operations and new classes may arise either because there is a change in requirements that demands additional functionality, or because new implementation technology (a new device, a new network protocol) becomes available to better meet the existing requirements.

Problem

With each extension of the interface, all of the concrete classes that implement this interface must also be extended. A new method must be added to each of them; all of these methods are conceptually similar.

However, because the methods are in unrelated classes, this conceptual similarity is not explicitly represented in the structure of the program. Code is duplicated, leading to maintenance, comprehension and coherence problems. The work that must be done to extend the interface with a new message is multiplied by the number of implementing classes.

Forces

- Limiting the implementation to a single class may not provide enough efficiency or functionality.
- A real strength of Object-Orientation is that it allows multiple classes to implement the same interface, and hides this multiplicity from client code. However, it can be difficult to maintain such multiple implementations—in particular, it can be difficult to maintain their consistency.
- When the interface must be extended, it is a lot of work to add the appropriate methods to all of the implementation classes.
- If the interface is *not* extended, clients code must instead replicate the same functionality.
- Spreading conceptually similar methods across multiple classes hides that similarity.

Solution

Therefore, partition the messages of the interface into two sets: the *core* messages, which provide access to *all* of the information contained in the objects, and the *support* messages, which provide useful utility functions for the clients.

Provide each of the concrete implementations with its own complement of core methods, for these methods must necessarily be intimate with the details of the objects' representation.

Create an abstract class, and install it as a superclass of all of the implementation classes. All of the support methods shall be implemented as methods on this abstract superclass. When these methods must access the information contained in the object, they shall obtain that information by sending a *core* message. The support methods shall not do anything that depends on the concrete class of the receiver.

Because of the way the messages are partitioned, new messages demanded by clients will usually be support messages, which can consequently be implemented by a single method in the abstract superclass.

If a new class implementing the interface must be added, then only the minimal set of core methods must be implemented in that class. All the other functionality—the support methods—can be inherited from the abstract superclass.

Code Samples

We are implementing lists, with the traditional interface, which we represent as the abstract class AbsList (see Figure 1).

Initially, there are three concrete classes that implement this interface: EmptyList (see Figure 2), which has the obvious semantics, ConsList (Figure 3), representing lists built from objects that contain an element and a list, in the manner of the traditional Lisp Cons cell, and FunList. FunLists (see Figure 4) require a little explanation.

```
(Abstract) Class AbsList
    instance Variables: (none)

methodsFor accessing AbsList

isEmpty
    "answers true is this list is empty, otherwise false"
    self subclassResponsibility

first
    "answers the first element of this list"
    self subclassResponsibility

rest
    "answers a list containing all of my elements except the first"
    self subclassResponsibility
```

Figure 1: The Abstract Class AbsList

```
Class EmptyList subclass of AbsList
    instance Variables: (none)
    class Variables: theUniqueEmptyList

methodsFor accessing EmptyList

isEmpty
    ^true

first
    ^self error: 'an Empty list has no first element'

rest
    ^self error: 'an Empty list has no rest'

class methods for instance creation

new
    "answer the unique EmptyList"

    theUniqueEmptyList isNil
        ifTrue: [ theUniqueEmptyList := self basicNew ].
    ^ theUniqueEmptyList
```

Figure 2: The Concrete Class EmptyList

A FunList represents a list that can be defined as the fixed point of a function from lists to lists. A new FunList is created by providing FunList new: with an argument that is a block that represents such a function. The resulting FunList is a list that is a fixed point of that function, that is, a list that will be unchanged by the function. For example, the function that prepends 1 and 2 onto its argument has the infinite list [1 2 1 2 1 2 1 2 1 2 1 2 …] as a fixed point.

A short aside on FunLists. In general, the functions that can be used as the kernel of a FunList might or might not have fixed points. Constant functions $\lambda l.\ c$ will always have *c* as a fixed point, and constructive functions that extend their argument will have an infinite list as their fixed point. How can we determine what first and rest should do on such a list? If *lst* is a funlist with kernel function *f*, then *lst* is a fixed point of *f*, in other words, *f(lst) = lst.* So *first(lst) = first (f (lst))*. This transformation is called *unrolling*. Provided that *f* adds information to its argument, this is sufficient to define first; see the code in Figure 4. [**end of aside**]

```
Class ConsList subclass of AbsList
    instance variables: head tail

private methods

head: anElement tail: aList
    "initialize the instance variables"

    head := anElement.
    tail := aList
    ^self

methods for accessing ConsList

isEmpty
    ^false

first
    ^head

rest
    ^tail

class methods for instance creation

new
    "cancel this method"

    ^self shouldNotImplement

new: anElement onto: aList
    "create a new cons list

    ^super new head: anElement tail: aList
```

Figure 3: The Concrete Class ConsList

Each of these three concrete classes defines the core methods first, rest and isEmpty in ways that depend on the details of their representation. But support methods, like printOn: need not be defined once for each concrete class; they can be defined once and for all in the abstract superclass, as shown in Figure 5.

Note that the printElementsOn: method in AbsList is *Pure Behaviour*[1]. It does not do anything that depends on the details of any of the concrete representation subclasses. Indeed, the compiler would not allow this method to access the instance variables head and tail of ConsList, because they are out of scope. However, the compiler will allow us to write tail isKindOf: EmptyList rather than tail isEmpty. *This should be avoided*. Why? Because such code makes the assumption that all empty lists will be a subclass of EmptyList, an assumption that is likely to be violated if new representations of list are added as new subclasses of AbsList. Indeed, since FunLists can also be empty, this assumption has already been violated! (Consider FunList new: [:lst | EmptyList new].)

Other support methods can easily be added to AbsList. For example, we might define do: and add: on AbsList, similar to the way in which they are defined on Collection

1. A method is said to be *Pure Behaviour* if its body does not access any instance or class variables, but accomplishes its objective only by sending messages.

```
Class FunList subclass of AbsList
    instanceVariables: listFunction

private methods

function: aBlock
    "aBlock should be a function from lists to lists"

    listFunction := aBlock

methods for accessing

isEmpty
    "unroll the definition once and see if it is empty"

    ^ (listFunction value: self) isEmpty

first
    "unroll the definition once and take the first element"

    ^ (listFunction value: self) first

rest
    "unroll the definition once and take the rest"

    ^ (listFunction value: self) rest

Class methods for instance creation

new: aBlock
    "aBock represents a function from lists to lists"

    ^super new function: aBlock
```

Figure 4: The Concrete class FunList

```
Class AbsList subclass of Object
    instance Variables: (none)

methodsFor printing AbsList

printOn: aStream
    "Append to the argument aStream a sequence of characters
    that describes the receiver."

    aStream nextPut: $[.
    self printElementsOn: aStream.
    aStream nextPut: $].

printElementsOn: aStream
    "writes to aStream a representation of my early elements.
    Not more than 15 elements will be represented"

    | tail |
    tail := self.
    15 timesRepeat:
            [ tail isEmpty ifTrue: [^self].
              aStream nextPut: $ .          "put a space"
              tail first printOn: aStream.  "put the first element"
              tail := tail rest].
    tail isEmpty ifFalse:[ aStream nextPutAll: ' ...'].
```

Figure 5: Methods for Printing AbsLists

```
Class AbsList subclass of Object
    instance Variables: (none)

methodsFor accessing AbsList

add: aNewElement
    "answers a list containing all of my elements, and aNewElement"

    ^ConsList new: aNewElement onto: self.

methodsFor enumerating AbsList

do: aBlock
    "Evaluate aBlock with each of the my elements as the argument."

    | tail |
    tail := self.
    [tail isEmpty]
        whileFalse:
            [aBlock value: tail first.
            tail := tail rest]
```

Figure 6: do: and add: for AbsLists

(see Figure 6). Note that although the add: method is pure behaviour, it also uses the fact that ConsList is a kind of AbsList. However, this fact is likely to be robust to change: it is much more likely that new kinds of list will be *added* to the program than it is that ConsList will be *removed*. The abstract superclass AbsList should avoid using the fact that FunList, ConsList and EmptyList are its *only* subclasses; it is permissible to use the fact that they are subclasses and that they provide implementations of its abstraction.

Resulting Context

The application of this pattern produces code that makes maximal reuse of the methods in the abstract superclass, and makes minimal demands on the implementor of a new subclass. That is, the set of core methods that a new subclass *must* implement is as small as possible.

This may have consequences for efficiency. The various subclasses may be able to respond to the same messages as the superclass in much more efficient ways. A trivial example is the method for printElementsOn: in AbsList. Clearly, if EmptyList implemented printElementsOn: directly, the method would do no work at all, whereas the generic method in AbsList sets up a loop and tests self for emptiness.

If these efficiency problems prove to be significant in the application, they can be avoided by providing additional, more efficient versions of the support methods in the subclasses. Such additional implementations do not compromise correctness or extensibility, because they are additions, not replacements. The generic method in the abstract superclass is still available for reuse by those concrete classes for which it is adequate.

The most important step in applying this pattern is the initial partitioning of the messages into core and support. Support methods may rely on core methods, but core

methods may not rely on support methods. It is easy to generate circular definitions if this partitioning is not maintained and documented.

Known Uses

In most Smalltalk systems, the abstract class Magnitude is the superclass of all of the classes that represent totally ordered values. This includes not just the numeric classes, but also dates and characters. Magnitude designates <, = and hash as the core methods that must be overridden by subclasses, and defines many useful support methods, including max:, min:, between:and: as well as >, >= and <=. Some of these methods (like >) are re-implemented in subclasses for efficiency.

Another known use is Smalltalk class Stream. This is the abstract superclass for a large (and growing) collection of different kinds of stream, *e.g.*, Squeak now supports a ZipEncoder stream that writes compressed files. The core methods for the Stream classes are next, contents, atEnd and nextPut:. The Stream class implements additional support methods terms of these core methods: there are a dozen such methods in Squeak 2.7 and over 30 in VisualWave 2.0.

The class SequenceableCollection in Squeak provide something of a counter-example, because it is not clear which of the many (about 90) methods are core. SequenceableCollection does not define any methods as subclassResponsibility. It inherits three subclassResponsibility methods from Collection: do:, add: and remove:ifAbsent:. However, it defines only do: as a useful method. remove:ifAbsent: is defined as self shouldNotImplement, and add: is not defined at all. What are the core methods for SequenceableCollection? I believe that the ability to index by an integer is core. This ability is provided by at: and at: put:. Should these methods be implemented by all of SequenceableCollection's subclasses? Which of at: and at:ifAbsent: is core and which support?

Acknowledgements

The description of this pattern evolved from a paper on encapsulation and the rows and columns problem that was drafted with the help of Ewan Tempero.