A Use for Inheritance

Andrew P. Black

Department of Computer Science & Engineering, OGI School of Science & Engineering,
Oregon Health & Science University
black@cse.ogi.edu

Abstract. There are, of course, many uses for inheritance. In general, they fall into two categories: expressing subtyping, and reusing implementation. In this paper we study one particular way of applying inheritance to the problem of reusing implementation: we us inheritance to mitigate the rows and columns dichotomy. We explain this problem, and then give, in pattern form, a detailed description of this use of inheritance.

Key words: Rows and columns, core/support split, pattern, reuse.

1 Introduction

The rows and columns dichotomy is by now a well-known issue in the programming language design community — although it is perhaps less well known in the community of practicing programmers. As commonly described, the dichotomy is presented as a win-lose battle between Object structure and Algebraic Data Types. We believe that, with proper use of inheritance, many of these battles can be turned into win-wins for Object structure.

We first describe the rows and columns dichotomy as it is usually depicted. We then go on to give a design pattern, which we have called the core/support split, that can often be used to avoid, or at least ameliorate, the consequences of choosing to organize one's program by columns, *i.e.*, using object structure.

2 The Rows and Columns dichotomy

Odersky and Wadler have described the dichotomy between Object structure and algebraic data types as follows [4]:

> Object types and inheritance make it easy to extend the set of constructors for the type, so long as the set of operations is relatively fixed. Conversely, algebraic types and [pattern] matching make it easy to add new operations over the type, so long as the set of constructors is relatively fixed. The former might be useful for building a prototype interpreter for a new programming language, where one often wants to add new language constructs, but the set of operations is small and fixed (evaluate, print). The latter might be useful for building an optimising compiler for a mature language, where one often wants to add new passes, but the set of language constructs is fixed.

Algebraic types make it easy to add new operations because each operation is represented as a single procedure. The cost of this convenience is that each of these procedures is defined by pattern matching on each of the constructors: adding a constructor therefore requires that *every* procedure be edited. If this step is forgotten, the result will be, depending on the programming language, either a compile-time "non-exhaustive match" error or, worse, a runtime failure.

Object structure makes it easy to add a new representation for an abstraction: this is accomplished by defining a new class. There is no need to modify the pre-existing classes that provide the other implementations of the abstraction. The cost of this convenience is that adding a new operation to an abstraction requires that the corresponding method be provided for *every* class that implements the abstraction. If this step is forgotten, the result will be, depending on the language, either a compile-time type error or, worse, a runtime "message not understood" error.

If one needs to extend an abstraction in both of these dimensions, one is left with the distinct impression that there is no solution: once can arrange one's data abstraction by rows (algebraic types) or by columns (classes), but clearly one can't do both.

But this description of the dichotomy ignores the rôle of inheritance. This position taken by this paper is that inheritance is a powerful tool that can often give us the best of both worlds. We can have object structure and the concomitant ability to easily add new representations while also reducing the effort required to add new operations. In fact, it will frequently be the case that a new operation can be implemented by the addition of a single method in a single class.

In the remainder of this paper we show how this feat can be accomplished by use of a programming pattern that we have dubbed the *Core/Support split*.

3 The Core/Support Split

3.1 Context

You are writing several concrete classes that all implement the same abstraction and export the same interface. Each of the concrete classes uses a different representation for this abstraction. As the program evolves, the interface must sometimes be extended to include additional messages; consequently, all of the implementing classes must also be extended so that their instances understand. these additional messages.These interface 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.

3.2 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 different 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.

3.3 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 their client. 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. This makes the implementor reluctant to undertake an extension.
- If the interface is *not* extended, clients code must instead replicate the same functionality.
- Spreading conceptually similar methods across multiple classes hides that similarity.

3.4 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.

3.5 Code Samples

We are implementing lists with the traditional interface,

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

methodsFor accessing AbsList

isEmpty
    "answers true if 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

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

```
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

```
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

```
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

FunList. FunLists (see Figure 4) require a little explanation. 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 list 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 list 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**]

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

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

```
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 not more
    than 15 of my early elements"

    | 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

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, in a way similar to that in which they are defined on Collection (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.

3.6 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.

```
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 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

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 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.

However, the additional implementations do pose a correctness problem: the additional implementations may be wrong, in that their behaviour may differ from that of the overridden abstract implementation. Moreover, as the semantics of the data abstraction evolves, the additional implementations must be modified to track the changes in semantics of the abstract ones in the superclass.

Automated testing can be used to check and maintain this coherence. All that is necessary is that the abstract superclass contain a method that provides access to the overridden method under a different name. This allows a test method to compare the results of the efficient and the inherited versions.

3.7 A Caution

The most important step in applying this pattern is the initial partitioning of the messages into the categories *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.

Which set of methods should be the core? The answer is: a complete set of observers[1]. These methods must be sufficient to extract all of the information contained in the abstraction represented by the objects of the class.

The basic idea is that an object-oriented definition of a data structure is the dual of the algebraic definition; in fact, object structure corresponds to *co*algebraic definition.

Jacobs and Rutten have written a excellent, if rather technical, tutorial on algebras and coalgebras [2]; we will not attempt to go into the details here. Instead, let us illustrate the concepts by example.

If binary trees are defined algebraically, there are typically two constructors: empty and subtreeLeft: value: right:. A complete set of observers for the corresponding coalgebraic (object) type is isEmpty, which tests to see if a tree is empty, left and right, which obtain the subtrees, and value, which extracts the data. This set of observer method is sufficient to explore any binary tree. For example, the support method leftSubtreeIfAbstent: aBlock can be defined using left and isEmpty.

Of course, the programmer must still make some choices about the way that the set of core methods is represented. For example, a single observer that returns the two subtrees and the value of the node as a triple is obviously sufficient to subsume left, right, and value. The choice between these two representations for the core may well depend more on the details of the implementation language than on any fundamental issue in the problem space. For example, if returning a tuple of results is inconvenient or expensive in the chosen language, then it will be preferable to use three separate observers. As another example: if the left right and value observers raise an exception when applied to an empty tree, there is actually no need to make isEmpty a (core) observer: it can be implemented as a support method. However, if raising and catching an exception is an order or magnitude more costly than executing a method, it would be wise to implement isEmpty as a core method anyway.

3.8 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; this is done by declaring them to be *subclassResponsibility*, roughly equivalent to declaring an abstract method in Java. Magnitude also defines many support methods, including max:, min:, between:and: as well as >, >= and <=. Some of these methods (like >) are re-implemented in subclasses for efficiency. In this case, the choice of the core methods is somewhat arbitrary; $\geq$ could have been chosen instead of < and =, and only the details would have changed.

Another known use is Smalltalk's 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 in terms of these core methods: there are 16 such methods in Squeak 3.0 and over 30 in VisualWave 2.0. Strictly, contents need not logically be part of the core, since it could be simulated by repeated calls on next, but depending on the implementation of the stream, contents may be very much more efficient.

The abstract class SequenceableCollection in Squeak provides something of a counter-example. Since the various subclasses of SequenceableCollection implement *different* abstractions, it does not fit the context of this pattern. Moreover, it is not clear which of the many (about 90) methods are core. We have noted that defining a method as *subclassResponsibility* is a clear indication that the corresponding message is core. Although SequenceableCollection does not define any methods as *subclassResponsibility*, three such methods are inherited from the abstract class Collection: these are do:, add: and remove:ifAbsent:. SequenceableCollection defines only do: as a useful method; it does so in terms of at:. The method add: is not defined at all. SequenceableCollection>>remove:ifAbsent: is defined as self shouldNotImplement; this reflects the fact that SequenceableCollection implements a more restricted abstraction than Collection.

What should the core methods be for SequenceableCollection? It would seem that the ability to index the collection by an integer is core to the idea of sequences; this ability provided by the methods for at: and at: put:. Along with these methods, we need a way of finding the range of legal indices; although includesKey: is theoretically sufficient, a method keys that returns the valid keys (*e.g.*, as an Interval) would be more useful. Finally, sequenceable collections (unlike arrays) are intended to be extensible; this requires either a method add: (or addLast:), or that collection at: n put: newValue be permitted in the special case when n = collection size + 1.

Which of at: and at:ifAbsent: should be core and which support? It is usually the case that a method like at: n is defined as at: n ifAbsent: [self errorNoSuchElement]. However, in Squeak, at:ifAbsent: is in fact defined in terms of at: It really does not matter, but successful use of this pattern does require that such a choice is made consistently and that it is documented, so that implementors of subclasses understand their responsibilities. The lack of such clarity in some of the

[1] In algebra, observers are also called *destructors*: the destructors of a co-algebra play a dual role to the constructors of an algebra. However, we will not use that term here, because of the possible confusion with an operation that deallocate storage for an object.

collection classes is probably due to their age, and to the fact that they have been extended inconsistently by different authors over a long period.

4 Related Work

The idea that an abstract class might implement an abstract algorithm in terms of abstract methods that its subclasses must take the responsibility to design is an old one. It was categorized as the *Template Pattern* by the Gang of Four [1], but Rebecca Wirfs-Brock and her co-authors discuss it at some length in their 1990 book [5].

Woolf [6] discusses the Abstract Class pattern, and indeed uses Smalltalk's Magnitude class as an illustration. This paper observes that Abstract classes containing template methods are especially useful when several concrete classes provide alternate implementations of the *same* abstraction.

Some of the same issues that motivate this pattern were also discussed by Lamping at least as far back as 1993 [3]; this paper also uses the the name "core methods" to refer to those that access the instance variables of the representation directly. The set containing the remaining methods is not named, but these methods are given a type that explicitly distinguishes the protocol that a method uses on self from the protocol that is implemented by the method's class. However, Lamping's inheritance hierarchy is the inverse of that proposed here: his support protocol inherits from the core protocol, while this pattern proposes the opposite.

One way of understanding the Core/Support split is as generalisation of Template Method to the design of entire classes rather than single methods. Certainly, this paper does not propose any new techniques; its contribution is to provide a name for a pattern and to identify the places where it is particularly applicable — multiple implementations of a single abstraction — as well as making explicit the connection to coalgebras.

5 Summary

We have introduced a pattern that can be applied to a wide variety of data abstractions. Through a careful separation of the core and the support methods, the Support methods can be reused through inheritance even thought the underlying representation of the data abstraction, and thus the implementation of the core methods, are completely different.

Used in this way, inheritance has nothing whatever to do with classification. It is a code reuse tool, enabling abstract implementations of the Support methods to be "parameterised" by the implementation of the core methods. Compared to using explicit parameterization, the use of inheritance leads to code that "feels" more concrete, and that is thus much easier to read. Such code is nevertheless highly abstract and easy to reuse.

Acknowledgements. This work was partially supported by NSF Grant CCR-9988440, by the Murdock Trust, and by Intel. We thank Ewan Tempero for his insights on the rôle of abstraction in programming and James Noble for his helpful comments on a previous description of the Core/Support Split.

References

1. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: elements of reusable object-oriented software*. Addison Wesley professionbal computing series. 1995: Addison-Wesley. iv+395.

2. Bart Jacobs and Jan Rutten, *A Tutorial on (Co)Algebras and (Co)Induction*. EATCS Bulletin, 1997. **62**: pp 222–259.

3. John Lamping. *Typing the specialization interface*. In *Proceedings Eighth ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 1993.

4. Martin Odersky and Philip Wadler. *Pizza into Java: Translating Theory into Practice*. In *24th ACM Symposium on Principles of Programming Languages*, 1997, Paris, France: ACM Press, .

5. R. Wirfs-Brock, B. Wilkerson and L. Wiener, *Designing Object-Oriented Software*. 1990, Englewood Cliffs, NJ: Prentice Hall.

6. Bobby Woolf. *The Abstract Class Pattern*. In *The 4th Pattern Languages of Programming Conference*, 1997.