

Algorithms and Object-Oriented Programming: Bridging the Gap

Paolo Bucci, Wayne Heym, Timothy J. Long, Bruce W. Weide
Department of Computer & Information Science
The Ohio State University
Columbus, Ohio, 43210
{bucci,hey,m,long,weide}@cis.ohio-state.edu

Abstract

Currently there is a serious conceptual and technical gap between ideas emphasized in object-oriented programming and ideas as taught in algorithms courses. We suggest both a paradigm for “re-expressing” algorithms in terms of classes and objects, and a set of criteria relative to which the quality of such re-expressions can be evaluated. A detailed example is provided for re-expressing the algorithmic idea of sorting.

1. The Problem

In the paper “Design Patterns for Sorting”, Nguyen and Wong present an object-oriented approach to sorting [5]. Generally speaking, object-oriented programming encourages us to think about and to organize software around data. On the other hand, typical presentations of sorting encourage us to think in terms of algorithms and to organize sorting software around procedures. Thus, for the problem of sorting, Nguyen and Wong have presented an abstract class that bridges the gap between object-oriented programming and what is traditionally an algorithmically-oriented topic.

It is not too hard to see that sorting represents just one instance of a larger “gap” phenomenon [7,9]. Typical algorithms textbooks are teeming with clever algorithms in the form of procedures, such as procedures to find shortest paths, find connected components, find stable marriages, fill knapsacks, do range searches, etc [2]. Evidently, if we were to develop software as presented in algorithms textbooks, our software would be procedure-oriented. On the other hand, many CS1/CS2 instructors teach their students, starting from CS1, that software ought to be object-oriented. How are students supposed to bridge this gap between what is preached in programming courses and what is taught in algorithms courses? Do we, as instructors, know how to bridge this gap?

In this paper we present a paradigm that we have used, in a number of instances, to successfully bridge the gap between OOP and algorithms, starting in CS1. The paradigm results in object-oriented software components that feature a number of highly desirable technical properties, and that sport user-friendly APIs. In fact, the APIs are so user friendly that even CS1 students have little or no difficulty using the described components. We like to refer to our paradigm as the “machine” paradigm, and we like to think of it as a method for “recasting” or “re-expressing” algorithms in terms of classes and their instances [8].

In Section 2 we describe certain standards by which the quality of proposed algorithm “re-expressions” can be evaluated. In Section 3 we present a machine-based interface for sorting, followed in Section 4 by an assessment of the interface relative to the standards from Section 2. Section 5 briefly describes other examples of machine-based interfaces, while concluding comments appear in Section 6.

2. Setting Objective Standards

In considering the general problem of re-expressing algorithmic notions in terms of classes encapsulating both data and operations, it is helpful to keep in mind desirable properties for the interfaces to be developed. Two such properties arise from traditional object-oriented thinking:

- P1:** hide data representations (the interface does not allow direct coupling between client code and internal data representations)
- P2:** hide algorithms used to implement operations (the interface does not allow direct coupling between client code and internal algorithmic detail).

For interfaces conforming to these two properties, software developers/maintainers have the freedom to substitute one implementation of an interface for another with no effect on clients of the interface, except for possible differences in performance (efficiency). Designing for this type of substitutability appears to be widely accepted as good engineering practice.

There is another “hiding” property that is similar in spirit to properties P1 and P2:

P3: hide when specific actions are executed (the interface does not allow direct coupling between client code and *when execution of internal algorithmic detail occurs*).

We can think of property P3 as “information hiding in the time dimension” [6]. For interfaces conforming to this property, software developers/maintainers have the freedom to substitute one implementation of an interface for another with no effect on clients of the interface, except for *intended* improvement in efficiency, possibly related to client-specific applications. (Reader-friendly examples of this idea appear in Sections 3 and 4.) Said in other words, interfaces that conform to property P3 offer classes, which implement the interface, freedom in when “hard computational work” takes place. In turn, this allows interface clients a broader range of choices with respect to performance. It appears to us that property P3 does not enjoy the renown of properties P1 and P2. Even so, it has a major impact on the design of interfaces arising from the re-expression of algorithms in terms of OOP.

Properties P1 through P3 are concerned with not making certain information explicit in interfaces. The next two properties are concerned with making other information explicit, in interfaces, through abstraction. Specifically, the following should be explicit in interfaces:

P4: a *mathematical model* describing precisely the state set (value set) of objects for classes that implement the interface

P5: precise pre-/post-conditions, stated in terms of the mathematical models, for each interface operation.

We can think of interfaces that conform to properties P4 and P5 as having precisely defined “cover stories” (abstract descriptions) through which clients metaphorically understand hidden information. Software, especially as it is typically presented in CS1/CS2, frequently does not conform to properties P4 and P5 [1]. English descriptions and “wishful naming” are often used in place of mathematical models and precisely stated pre-/post-conditions.

3. An Example: the Sorting Machine Interface

In this section we describe in detail a machine-based interface for sorting. By way of a mental model, it is nice to think of a sorting machine object as a box with two doors, an input door and an output door, and a “change-of-phase” button. Initially, the sorting-machine box is empty, the input door is open, the output door is closed, and the machine is in “insertion” phase. Clients insert data items to be sorted, one at a time, through the input door using a series of Insert operations. Inserted items accumulate in the box. When all data has been presented to the machine, the client pushes the “change-of-phase” button. This

causes the input door to close, the output door to open, and the machine to enter “extraction” phase. Next the client extracts the data from the sorting machine box, one item at a time and in sorted order, through a series of Remove_First operations. The following pseudo-code summarizes how a client would use a sorting machine to sort data. (Assume that *m* is an object of type Sorting_Machine and *x* is an object of type Item; see the appendix.)

```
while (“there are more items to sort”) do
    “insert next item by Insert(m,x)”
end;
Change_To_Extraction_Phase(m);
while (Size(m) > 0) do
    “remove next item in order
    by Remove_First(m,x)” ;
end
```

The entire sorting machine interface appears in the appendix. Here we discuss selected pieces of the interface in detail.¹ Sorting_Machine_Template is parameterized by type Item and by an operation, Are_In_Order, to compare two values of type Item:

```
Sorting_Machine_Template (
    type Item,
    operation Are_In_Order
)
```

This allows clients to customize sorting machine objects, through instantiation, to the type of items to be sorted and to how the items are to be ordered. Notice that the restriction on Are_In_Order imposes a total ordering on the type Item.

The mathematical model of the state set for type Sorting_Machine is a 2-tuple (a pair):

```
type Sorting_Machine is modeled by
(inserting: Boolean
contents: finite multiset of Item)
```

The first entry, named inserting, is a mathematical Boolean value that models whether a sorting machine object, say *m*, is in insertion phase (*m.inserting* = true) or in extraction phase (*m.inserting* = false). The second entry, named contents, is a mathematical multiset that models the collection of items in a sorting machine box. Use of multisets allows for duplicate item values. The initialization ensures clause specifies that every new sorting

¹ Code appearing in this paper is intended to be language neutral. A stylized C++ version of the sorting machine is available at http://www.cis.ohio-state.edu/~weide/now/sce/rcpp/RESOLVE_Catalog-HTML/Sorting_Machine_A.html [3,4].

machine object will be in insertion phase with an empty multiset of items:

```

exemplar m
initialization ensures
  m.inserting = true and
  m.contents = empty_multiset

```

Now let's consider the pre-/post-conditions for the Insert operation.

```

operation Insert (
  alters    m: Sorting Machine,
  consumes x: Item
)
requires
  m.inserting = true
ensures
  m.inserting = true and
  m.contents = #m.contents union {#x}

```

The precondition requires that m is in insertion phase. The post condition ensures that m remains in insertion phase and that the incoming value of x has been added to the content multiset of m .

As a last example, let's consider the pre-/post-conditions for the Remove_First operation.

```

operation Remove_First (
  alters    m: Sorting_Machine,
  produces x: Item
)
requires
  m.inserting = false and
  m.contents /= empty_multiset
ensures
  m.inserting = false and
  x is in #m.contents and
  m.contents = #m.contents - {x} and
  for all y:Item where
    (y is in #m.contents)
    (Are_In_Order(x,y))

```

The pre-condition requires that m is in "extraction" phase and that there are more items in m 's multiset. The post-condition ensures that m remains in extraction phase, that the outgoing value of x is the first in order with respect to the client supplied ordering, and that the only change to the contents of m is the removal of item x .

Specifications of the other operations are similar.

4. The Standards: How Did We Do?

Let's take a look at how the sorting-machine interface measures up to properties P1 through P 5.

P1: *hide data representations*

The sorting-machine interface conforms to this property. Classes that implement this interface are free to use an

internal container of choice, such as arrays (the traditional choice), binary search trees, queues, lists, etc. The use of a mathematical multiset in the cover story (mathematical model) does not dictate the type of container used in the implementation.

The significance of property P1 may not be readily apparent for sorting, where one might argue that arrays are always the container of choice. (We happen to not agree with this, however.) On the other hand, if we consider the sophisticated data structures involved in something like Kruskal's minimum spanning tree algorithm, it is clear that requiring client code to know about and respect such a sophisticated data structure represents extremely poor component-interface engineering.

P2: *hide algorithms used to implement operations*

The sorting-machine interface conforms to this property. Classes that implement this interface are free to use a sorting algorithm of choice, such as quick sort, insertion sort, selection sort, heap sort, etc.

P3: *hide when specific actions are executed*

It's an interesting exercise to think about the possibilities for implementing the sorting-machine interface with respect to this property. Table 1 summarizes some possibilities.

As we can see, the sorting machine interface allows for a wide range of possibilities for when the "hard work" of sorting takes place. Why is this such a desirable property? Consider the following client code that extracts only the k first items in sorted order:

```

while ("there are more items to sort") do
  "insert next item by Insert(m,x)"
end;
Change_To_Extraction_Phase(m);
number_extracted := 0;
while ((number_extracted < k) and
  (Size(m) > 0)) do
  "remove next item in order
  by Remove_First(m,x)";
  number_extracted++;
end

```

For this client code, if k is a small constant or at least considerably smaller than the number of data items, a selection sort implementation would most likely best serve the client. On the other hand, if k is just about the same as the number of data items, quick sort might better serve the client. The important point is that property P3 provides clients with this type of performance flexibility.

P4: *a mathematical model describing precisely the state set (value set) of objects for classes that implement the interface*

Underlying Sorting Algorithm	Suggestions for What Happens During Operation . . .		
	Insert	Change_To_Extraction_Phase	Remove_First
insertion sort	insert in order into efficient linear data structure	change the Boolean phase flag	simple remove from linear data structure
selection sort	simple insert into efficient linear data structure	change the Boolean phase flag	search for and remove “first” item
quick sort, merge sort	simple insert into efficient linear data structure	change the Boolean phase flag and quick (merge) sort all data items	simple remove from linear data structure
heap sort	simple insert into efficient linear data structure	change the Boolean phase flag and construct a heap	remove “first” from a heap
tree sort	insert in order into binary search tree	change the Boolean phase flag	remove “first” from binary search tree

Table 1

P5: *precise pre-/post-conditions, stated in terms of the mathematical models, for each interface operation*

The sorting-machine interface clearly conforms to both of these properties. More importantly, the formal specifications are CS1/CS2-student friendly! Students at this level *can read and understand* these formal specifications (if you teach them how to [1]) and *can readily write client code* using the sorting-machine API.

It is interesting to compare the sorting-machine interface to the abstract class for sorting presented in [5]. By our understanding, the latter conforms to P2, could be considered to conform to P1 (although we suspect this was not intended), and does not conform to P3 through P5.

5. The Machine Paradigm in CS1/CS2

We have found the machine paradigm to be quite useful in CS1/CS2. The sorting machine interface is introduced and used in CS1-programming assignments. The interface is implemented in CS2. Other machine-based components that we have used/are using include:

Least_Cost_Path_Machine —

Clients insert edges, one at a time, to construct a directed, weighted graph. Clients extract, one at a time, copies of edges making up least cost paths. The interface is introduced and used in CS1. For example, using the interface and in a single 48-minute closed lab, students implement an operation that lists the costs of shortest paths between all pairs of cities in a set of cities. For a one-week programming assignment and using the interface, students list the sequences of edges to visit, in order, along shortest paths from source to destination cities.

Tokenizing_Machine —

Clients insert streams of characters, one character at a time. Clients extract, one at a time, tokens making up the corresponding character stream. The interface is introduced, used, and later implemented in CS2.

Assertion_And_Query_Machine —

Clients insert “assertions” or “facts”, one at a time. Clients extract, one at a time, instantiations satisfying and/or queries relative to the entered assertions. (This is similar to a simplified version of logic programming.) The interface has been introduced, used, and later implemented in CS2.

6. Concluding Remarks

Currently there is a serious conceptual and technical gap between ideas emphasized in object-oriented programming and ideas as generally taught in algorithms courses. The former leads to “data-oriented” software artifacts while the latter leads to “procedure-oriented” software artifacts. Attempts to rectify this situation by re-expressing algorithms in terms of classes and objects need to be grounded in sound fundamentals, less the cure be worse than the ailment. We suggest that properties P1 through P5 are grounded on sound software-engineering fundamentals and that they represent an appropriate initial standard against which such re-expressions should be measured.

The machine paradigm represents one approach to the challenge of re-expressing algorithms in terms of classes and objects. Based on our experience to date, it results in interfaces that conform to P1 through P5 and that have CS1/CS2-friendly interfaces, as well.

7. Acknowledgments

We gratefully acknowledge financial support from the National Science Foundation under grants DUE-9555062 and CDA-9634425 and from the Fund for the Improvement of Post-Secondary Education under project number P116B60717.

References

- [1] Bucci, P., T. Long, and B. Weide, Do We Really Teach Abstraction?, *Proc. 2001 ACM SIGCSE Technical Symp.*, ACM, February 2001, pp. 26-30.
- [2] Cormen, T., C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, 2001.
- [3] Hollingsworth, J., L. Blankenship, and B. Weide, Experience Report: Using RESOLVE/C++ for Commercial Software, *Proc. ACM SIGSOFT 8th Int. Symp. Foundations of Software Engineering*, ACM Press, 2000, pp. 11-19.
- [4] Long, T.J., B. Weide, P. Bucci, D. Gibson, M. Sitaraman, J. Hollingsworth, and S. Edwards, Providing

Appendix

```
Sorting_Machine_Template (  
    type Item,  
    operation Are_In_Order  
)  
restriction  
for all x,y:Item (Are_In_Order(x,y) or  
                Are_In_Order(y,x)) and  
for all x,y,z:Item  
(Are_In_Order(x,y) and Are_In_Order(y,z)  
 implies Are_In_Order (x,z))
```

interface specification

```
type Sorting_Machine is modeled by (  
    inserting: Boolean  
    contents: finite multiset of Item  
)  
exemplar m  
initialization ensures  
    m.inserting = true and  
    m.contents = empty_multiset  
operation Insert (  
    alters    m: Sorting Machine,  
    consumes  x: Item  
)  
requires  
    m.inserting = true  
ensures  
    m.inserting = true and  
    m.contents = #m.contents union {#x}  
operation Change_To_Extraction_Phase (  
)  
requires  
    m.inserting = true
```

Intellectual Focus To CS1/CS2", *Proc. ACM SIGCSE Technical Symp.*, ACM Press, 1998, pp. 252-256.

- [5] Nguyen, D. and S. Wong, Design Patterns for Sorting, *Proc. 2001 ACM SIGCSE Technical Symp.*, ACM, February 2001, pp. 263-267.
- [6] Parnas, D., On the Criteria to be Used in Decomposing Systems into Modules, *Comm. ACM*, Dec. 1972, pp. 1,053-1,058.
- [7] Sitaraman, M., B. Weide, T. Long, W. Ogden, A Data Abstraction Alternative to Data Structure/Algorithm Modularization, *LNCS*, 1766, Jazayeri, Loos, Musser (Eds.), Springer, 2000, pp. 102-113.
- [8] Weide, B., W. Ogden, and M. Sitaraman, Recasting Algorithms to Encourage Reuse, *IEEE Software*, IEEE Computer Society, 11(5), September 1994, pp. 80-88.
- [9] Weihe, K., Reuse of Algorithms: Still a Challenge to Object-Oriented Programming, *ACM SIGPLAN OOPSLA Proceedings*, 1997, 34-46.

```
ensures  
    m.inserting = false and  
    m.contents = #m.contents  
operation Remove_First (  
    alters    m: Sorting_Machine,  
    produces  x: Item  
)  
requires  
    m.inserting = false and  
    m.contents /= empty_multiset  
ensures  
    m.inserting = false and  
    x is in #m.contents and  
    m.contents = #m.contents - {x} and  
    for all y:Item where  
        (y is in #m.contents)  
            (Are_In_Order(x,y))  
operation Remove_Any (  
    alters    m: Sorting_Machine,  
    produces  x: Item  
)  
requires  
    m.contents /= empty_multiset  
ensures  
    m.inserting = #m.inserting and  
    x is in #m.contents and  
    m.contents = #m.contents - {x}  
operation Is_In_Insertion_Phase (  
    preserves m: Sorting Machine  
) : Boolean;  
ensures  
    Is_In_Insertion_Phase = m.inserting  
operation Size (  
    preserves m: Sorting Machine  
) : INTEGER;  
ensures  Size = |m.contents|
```