# Design Patterns for Sorting

**Dung ("Zung") Nguyen**
**Dept. of Computer Science**
**Rice University**
**Houston, TX 77251**
**dxnguyen@cs.rice.edu**

**Stephen B. Wong**
**Computer Science Program**
**Oberlin College**
**Oberlin, OH 44074**
**stephen.wong@oberlin.edu**

### Abstract

Drawing on Merritt's divide-and-conquer sorting taxonomy [1], we model comparison-based sorting as an abstract class with a template method to perform the sort by relegating the splitting and joining of arrays to its concrete subclasses. Comparison on objects is carried out via an abstract ordering strategy. This reduces code complexity and simplifies the analyses of the various concrete sorting algorithms. Performance measurements and visualizations can be added without modifying any code by utilizing the decorator design pattern. This object-oriented design not only provides the student a concrete way of unifying seemingly disparate sorting algorithms but also help him/her differentiate them at the proper level of abstraction.

## 1 Introduction

A widely accepted way of teaching sorting is to start with elementary algorithms such as insertion sort and selection sort and then progress to more sophisticated ones, such as quick sort and merge sort. Sorting presented in this manner often appears to students as a bewildering array of differing techniques and analyses. Often, they get lost in the nitty-gritty code details, never gaining an overall understanding of the topic.

An alternative approach would be to study sorting from the top down as proposed by Merritt [1]. At the top of her sorting taxonomy is an abstract divide-and-conquer algorithm: split the array to be sorted into two subarrays, (recursively) sort the subarrays, and join the sorted subarrays to form a sorted array. Merritt considers all comparison-based algorithms as simply specializations of this abstraction and partitions them into two groups based on the complexity of the split and join procedures: *easy split/hard join* and *hard split/easy join.* At the top of the groups *easy split/hard join* and *hard split/easy join* are merge sort and quick sort, respectively, and below them will fit all other well-known, more "low-level" algorithms. For example, splitting off only one element at each pass in merge sort results in insertion sort. Thus insertion sort can be viewed as a special case of merge sort.

Merritt's thesis is potentially a very powerful method for studying and understanding sorting. However, to our knowledge, no systematic treatment exists that truly reflects its theoretical underpinnings. Merritt's abstract characterization of sorting exhibits much object-oriented (OO) flavor and can be described in terms of OO concepts.

We present in this paper our OO formulation and implementation of Merritt's taxonomy, one that we have successfully taught to our first year computer science students. We will explain how the abstract concept of sorting is appropriately captured in terms of standard design patterns [2]. We will also seek to illustrate the many benefits of this particular OO view of sorting. Effective use of polymorphism properly relegates specific tasks to subclasses and minimizes flow control, resulting in less code and reduced code complexity. Unifying sorting under the single abstract principle of divide-and-conquer leads to an across the board simplification of running time analysis. Treating all sort algorithms uniformly at the same abstract level provides ways to extend and add more capabilities to existing code without modification, facilitating code reuse. As students digest fundamental algorithmic techniques, they also get to learn and appreciate sound software engineering principles built on key OO design considerations.

Section 2 describes how the template method pattern is applied to capture the abstract sort procedure. Section 3 explains how the strategy pattern is used to decouple the task of sorting from the comparison operation. Section 4 details a few of the common sort algorithms and shows how they fit into the abstract hierarchy. Section 5 discusses how the new formulation simplifies and unifies the complexity analysis for all the different algorithms. Section 6 demonstrates the flexibility and extensibility of the design by applying the decorator design pattern to add performance measurements and visualization such as animation without touching any existing code.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE 2001 2/01 Charlotte, NC USA
Copyright 2000 ACM 1-58113-329-4/01/0002 ... $5.00

## 2 Template Method Design Pattern

At the heart of Merritt's taxonomy is the thesis that all comparison-based sorts can be expressed as a divide-and-conquer algorithm exemplified by merge sort and quick sort. What distinguishes one sort from another is the way the original array is split off into subarrays and the way the sorted subarrays are reassembled to form a sorted array. For example, selection sort splits the array at the low index after selecting and moving the minimum element there, (recursively) sorts the two subarrays, and then rejoins the sorted subarrays by doing nothing!

Thus the *invariant* in the above abstraction of sorting is the divide-and-conquer algorithm whose split and join steps are allowed to *vary*. Modeling an invariant behavior that is composed of variant behaviors is a simple application of the template method design pattern [2] as shown in the following UML diagram using Java syntax.

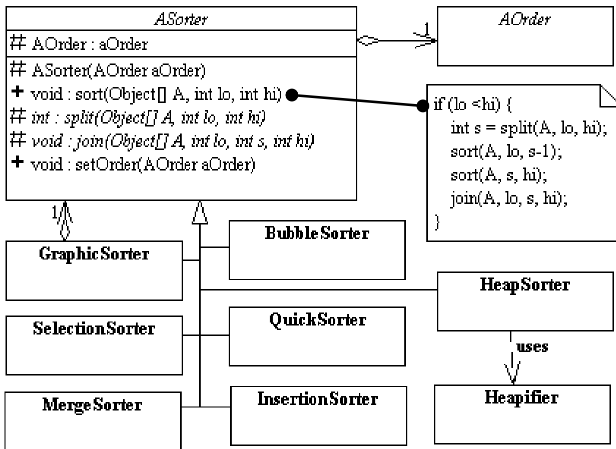


**Figure 1: The Template Method Pattern for Sorting**

In **Figure 1**, the abstract class, *ASorter*, embodies the general sorting principles under study. Its sort() method, the "template method", sorts an array indexed from lo to hi by calling the abstract *split()* and *join()* methods, deferring these tasks to a concrete subclass. By making sort() final, we guarantee its invariance throughout the complete class hierarchy.

The *split()* method rearranges the elements in the array in some specific fashion and returns the index where the array is divided into two subarrays. After the two subarrays are (recursively) sorted, the *join()* method appropriately combines them into a sorted array. Each concrete subclass of *ASorter* represents a concrete sorting algorithm. It inherits and reuses the sort() method and only overrides *split()* and *join()*. This is an example of sensible code reuse: reuse only that which is *invariant* and override that which *varies*.

As **Figure 1** indicates, the OO taxonomy tree presented here is only one level deep. And unlike Merritt's proposed hierarchy, we see no need for capturing the complexity of the *split()/join()* pair in our design. The simplicity of the template pattern and the flatness of the taxonomy tree enable students see the set of sorting algorithms as unified and interrelated whole and not as disparate low-level manipulations. **Figure 2** below shows the recursive call tree for the sort() method of a hypothetical sort algorithm.

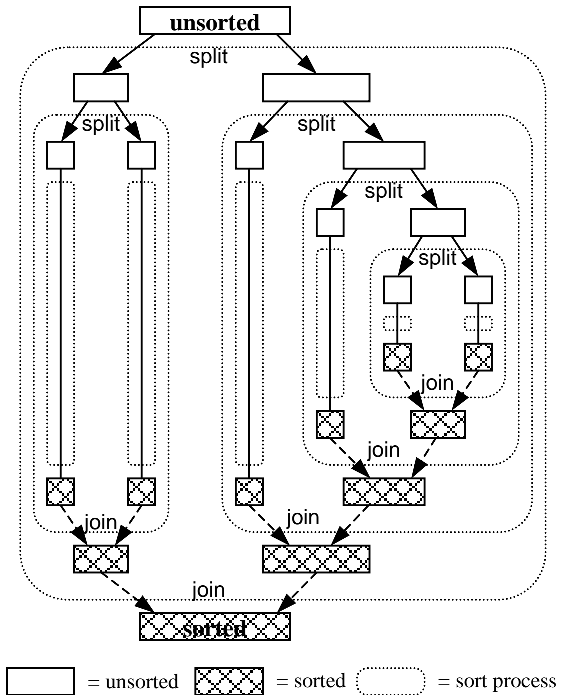


**Figure 2: Hypothetical Sort Recursion Tree.**

## 3 The Strategy Pattern for Object Comparison

Sorting requires a total order relation on the data. We hold the view that it is the user of the data that arbitrarily imposes an order relation for sorting and not that the data intrinsically "know" their ordering. And thus, we seek to decouple the ordering of data from both the data and the sorting of data.

We can define a total order in terms of two abstract comparison operations that take two Object instances as input and returns a boolean as output: one for a strict ordering[1] and one for equality, and encapsulate them in an abstract class, *AOrder*. These two operations must be implemented in a way that their disjunction defines a total order on the domain they operate on. The strict order comparison facilitates the implementation of stable sort algorithms. Below is the Java code listing for *AOrder*.

[1] A "strict" order is a relation that is non-reflexive, anti-symmetric, and transitive.

```
public abstract class AOrder {
    public abstract boolean lt(Object x, Object y);
    // defines a "less than" strict ordering.

    public abstract boolean eq(Object x, Object y);
    // defines equality.

    public boolean ne(Object x, Object y)  {return !eq(x, y);}
    public boolean le(Object x, Object y)  {return lt(x, y) || eq(x, y);}
    public boolean gt(Object x, Object y)  {return !le(x, y)};
    public boolean ge(Object x, Object y) {return !lt(x, y);}
}
```

**Listing 1: Abstract Total Order Relation.**

As a convenience, we include the other "standard" comparison operations in the specification of *AOrder*. These operations are defined in terms of the two primary abstract comparison operations and are not abstract. Any concrete subclass may override them if so desired.

*ASorter* maintains a reference to a concrete instance of *AOrder*. When a concrete sorter is instantiated, it must be given a concrete order to be used in its (concrete) split() and join() methods. The client can change the ordering in the sort by simply calling the setOrder() method on the sorter. This loose coupling between *ASorter* and *AOrder* is called the *strategy pattern*, where *AOrder* is said to be the order strategy for *ASorter*. It provides the flexibility for different sort algorithms to share the same order and for different order relations to be used by the same sorter.

## 4 Concrete Sort Examples

Our OO sort formulation clearly and cleanly delineates the task of sorting into two subtasks, making it easier to understand and verify its correctness. Writing a sort algorithm reduces to subclassing *ASorter* and overriding the *split()* and *join()* methods. The recursive method *ASorter*.sort() correctly sorts the base case where the array size is less or equal to one. It also sets up the loop variant and makes the proper recursive calls. Its correctness thus solely depends on the correctness of the concrete split() and join().

**Listing 2** below shows how insertion sort is implemented in our sort framework.

```
public class InsertionSorter extends ASorter {
    // Constructor omitted.
    protected int split(Object[] A, int lo, int hi)  {
        return hi;
    }

    protected void join(Object[] A, int lo, int s, int hi)  {
    // Pre: A[lo:hi-1] is sorted.
    // Algo: Inserts A[hi] in order into A[lo:hi-1].
    // Post: A[lo:hi] is sorted.
        int j, key = A[hi];
        for (j = hi; lo < j && aOrder.lt(key,A[j-1]); j--)  A[j] = A[j-1];
        A[j] = key;
    }
}
```

**Listing 2: Insertion Sort.**

In **Listing 2**, we see that the split() method returns the index of the last element of the array, and, in effect, partitions the array A[lo:hi] into two subarrays A[lo:hi-1] and A[hi]. The join() method simply uses the order strategy, aOrder, to search A[lo:hi-1] for proper place to insert the last element. Conspicuously missing is the familiar nested loop construct in the common procedural implementation of insertion sort. The sort() method inherited without modification from *ASorter* plays the role of the traditional outer loop. The problem of sorting of an array of data is transformed into the problem of inserting one element in order into the array. There is less code to write and fewer control constructs. Verifying the correctness of insertion sort reduces to verifying correctness of inserting A[hi] in order into A[lo:hi-1].

The quick sort implementation shown in **Listing 3** below is the opposite of the insertion sorter as the split method is more complex while the join method is trivial.

```
public class QuickSorter extends ASorter {
    // Constructor omitted.
    protected int split(Object[] A, int lo, int hi)  {
        // Select a pivot element p and rearrange A in such a way
        // that all elements to the left of p are less than p and all
        // elements to the right of p are greater or equal to p.
        // Return the index of p.
    }

    protected void join(Object[] A, int lo, int s, int hi)  {// do nothing }
}
```

**Listing 3: Quick Sort.**

In this case, the students can concentrate on understanding the algorithm for locating the pivot point without being distracted by the surrounding control structures.

**Table 1** below summarizes the split/join operations of a few common sort algorithms.

| Sort | Split operation | Join operation |
|---|---|---|
| *Insertion* | Return *hi* | Insert *A*[*hi*] into proper location. |
| *Merge* | Return midpoint index. | Merge subarrays. |
| *Quick* | Find and return pivot point index | Do nothing. |
| *Selection* | Swap extremum with *A*[*hi*] and return *hi* | Do nothing. |
| *Bubble* | Bubble up extremum to *A*[*hi*] and return *hi* | Do nothing. |
| *Heap* | Swap extremum (*A*[*lo*]) and *A*[*hi*], reheapify *A*[*lo*, *hi*-1], and return *hi*. | Do nothing. |

**Table 1: Concrete split/join Operations**

From **Table 1**, we can see that selection sort, bubble sort, and heap sort[2] are essentially identical processes, though they have different algorithmic complexities: they all pull out the extremum from the array and split it off. A trivial no-op join then follows this. Quick sort is similar to the

[2] Heap sort heapifies the array only once at construction time.

selection/bubble/heap genera except that it pulls off a set of one or more extrema values.

On the flip side of the coin, we see that insertion sort and merge sort are similar in that their split operations are trivial while their join operations are more complex. Insertion splits off one element at a time while merge sort splits the array in half each time. One can think of the join() method in insertion sort as merging a sorted array with a one-element array (which is obviously sorted).

## 5 Complexity Analysis

Our formulation and implementation of sorting helps students develop the mathematical thinking and techniques in analyzing the complexity of an algorithm. On one hand, the sort template method engenders a recursion tree (see **Figure 2**), which provides some heuristics on the sort complexity. On the other hand, it leads to a canonical recurrence relation that serves as a common starting point for students to make the first step in their analysis of each of the concrete sort algorithms.

It is easy to see from **Figure 2** that the total running time of a sort is equal to the sum of the running time of each level of the recursion sort tree. If the running time at each level is uniformly bounded by some function $f(n)$, then the total running time is bounded by $f(n)$ times the height of the sort tree. Sketching the sort tree for a few of the concrete algorithms helps students develop some intuition on their complexity analyses.

A formal treatment of complexity involves deriving a recurrence relation for $T(lo, hi)$, the running time to sort an array $A[lo..hi]$ indexed from $lo$ to $hi$ with $lo <= hi$. The code for sort() in **Figure 1** clearly indicates that

**R1**: $$T(l, h) = \begin{cases} c & \text{if } l = h \\ c + S(l,h) + T(l,s\text{-}1) + T(s,h) + J(l,s,h) & \text{if } l < h \end{cases}$$

where $c$ is the constant running time to compare $lo$ with $hi$, $S(lo, hi)$ is the running time to split $A$ into two subarrays, $A[lo..s\text{-}1]$ and $A[s, hi]$, and $J[lo, s, hi]$ is the running time for joining the two sorted subarrays $A[lo..s\text{-}1]$ and $A[s..hi]$ to form the sorted array $A[lo..hi]$.

It is necessary to examine the code for the specific split() and join() methods of a particular sort algorithm to compute $S[lo, hi]$ and $J[lo, s, hi]$ in order to solve R1. Let $n$ denote the size of the array. The steps in the computation of $T(n)$ are identical for all of the sort algorithms: start with the canonical relation **R1**, plug in the values for $s$, $S[lo, hi]$, and $J[lo, s, hi]$, and simplify. Note that the functional form of $s$ may depend on whether one sorts from $lo$ to $hi$ or $hi$ to $lo$. The simplification will lead to one of the following two recurrence relations:

**R2**: $$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ T(n\text{-}1) + O(f(n)) & \text{if } n > 1 \end{cases}$$

**R3**: $$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ aT(n/b) + O(f(n)) & \text{if } n > 1 \end{cases}$$

**R2** and **R3** can then be solved using the same standard discrete mathematics technique yielding the results shown in **Table 2** below.

| Sort | *s* | *S*[*lo*, *hi*] | *J*[*lo*, *s*, *hi*] | *T*(*n*) |
|---|---|---|---|---|
| Insertion | *hi* | O(1) | O(*hi*-*lo*) | $O(n^2)$ |
| Merge | (*lo*+*hi*+1)/2 | O(1) | O(*hi*-*lo*) | O(*n* log *n*) |
| Quick | varies | O(*hi*-*lo*) | O(1) | $O(n^2)$ worst case |
| Selection | *lo* + 1 | O(*hi*-*lo*) | O(1) | $O(n^2)$ |
| Bubble | *hi* | O(*hi*-*lo*) | O(1) | $O(n^2)$ |
| Heap | *hi* | O(log(*hi*-*lo*)) | O(1) | O(*n* log *n*) |

**Table 2: Running Time for Sorting**

This uniform treatment of sorting simplifies the analysis process and thus facilitates students' learning and reinforces their understanding of the subject. Casting sorting in terms of a divide-and-conquer template with an ordering strategy does not change the fundamental algorithms. Performance measurements and analysis of the algorithms show that while some extra dispatching overhead is incurred with this formulation, the additive nature of the template pattern does not affect the net running time complexity.

## 6 Performance Measurements and Visualization

Programming sorting at the level of abstraction characterized by the two abstract classes *ASorter* and *AOrder* enables an open-ended extension to our OO sort framework. For example, we can add performance measurements and animation to any sort algorithm without modification of any of the existing code. Performance evaluation and graphics are not fundamentally part of the sorting process, so students should learn that their sorting code should not be torn apart and re-written to add these capabilities. While a full discussion of the code is beyond the scope of this paper, we will briefly outline the fundamental principles involved.

The key design solution to such extensions is the decorator pattern [2]. This pattern enables one to use a "decorator" object to intercept methods calls to another object (the "decoree") in the same abstract class hierarchy and perform auxiliary functions in addition to dispatching the original call to the decoree. Since the decorator is abstractly equivalent to the decoree, the caller is completely unaware of its existence.

A common performance metric for sorting is the comparison count. To count comparisons, all we have to do is decorate the specific *AOrder* object by intercepting all comparison operator calls as shown in **Figure 3** below.

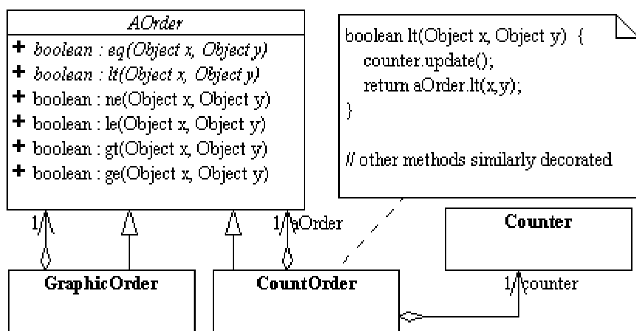


**Figure 3: Decorated ordering strategies.**

In **Figure 3**, counter is an object that "knows" how to count. Thus, CountOrder can be used to build a package that counts the number of comparisons of any concrete sorter algorithm, with any concrete *AOrder* strategy on any set of data. All the client has to do is to specify the sort algorithm, the order strategy, and the data array. This is the case where composition is clearly more flexible than inheritance.

To help visualize a sort algorithm, we arbitrarily choose to sort integers and apply the adapter pattern to add graphics capabilities that allow us to paint them in some specific ways on the screen. As we sort the data array, we highlight the two data objects that are being compared in the sort and paint the data array at each split and each join. Such an animation program can be based on the Model-View-Controller (MVC) pattern [2]. The following is a brief description of our implementation in Java.

The model consists of the array of graphical objects, GraphicSorter (see **Figure 1**), a decorator for *ASorter*, and GraphicOrder (see **Figure 3**), a decorator for *AOrder*. **Listing 5** below shows how GraphicSorter decorates *ASorter* by intercepting the split() and join() methods, carrying out graphical operations, and pausing momentarily for the view to repaint before returning.

```
public class GraphicSorter extends ASorter {
    private ASorter sorter; // decoree
    // Constructor and utility methods omitted.

    protected int split(Object[] A, int lo, int hi) {
        int s = sorter.split(A,lo, hi); // forward to decoree
        // performs graphical operations and pauses momentarily.
        return s;
    }
    // join() method is similarly decorated.
}
```

**Listing 5: Decorating *ASorter* for Animation.**

The view contains various graphics components, one of which is a JPanel where the data array is painted. The controller maintains a Timer object that periodically calls on the view to repaint the data array on a separate thread from the sorting process. As the sorting algorithm moves the data elements in the array, the view's display algorithm independently updates their position on the screen. Thus the animation of the sorting process occurs completely separate from the sorting algorithm itself.

## 7 Conclusion

The OO sort model presented here reflects our general approach of teaching principles in lieu of disparate facts and techniques. Our model draws on Merritt's "inverted taxonomy", which in turn is founded on the principle of divide-and-conquer. Object-orientation together with the language of patterns provides a clean and concise way of formulating and implementing sorting based on this principle. *Sorting is modeled as an abstract class with a template method to perform the sorting. This method relegates the splitting and joining of arrays to the concrete subclasses, which use an abstract ordering strategy to perform comparisons on objects.*

The similarities and dissimilarities of comparison-based sort algorithms can be explained in terms of the concrete split and join operations. The code in the concrete and specific sort subclasses now deals only with the small portion of the overall algorithm particular to its type. Such a transformation not only reduces code complexity but also simplifies and unifies the analysis of the various concrete sorting algorithms. The OO design of our sort model thus provides the student a concrete way of unifying and inter-relating seemingly disparate sorting algorithms.

By adhering to the general principle of programming to the abstraction, our sort model also provides the flexibility to add new capabilities without modification of existing code. For example, by using the decorator pattern, we can add performance measurements and visualizations without even knowing what sort algorithm is being used.

We teach sorting early not only as an essential programming tool but also as a means to develop students' algorithmic and mathematical thinking skill. In addition to helping achieve these goals, our OO sort model also exposes the student to key OO design and OO programming concepts in a small enough setting that is easily comprehended yet that demonstrates its large-scale advantages. This work is part of our overall effort to introduce object-orientation early into the computer science curriculum.

## References

[1] Merritt, S. An Inverted Taxonomy of Sorting Algorithms, *Comm. ACM, 28, 1* (Jan. 1985), 96-99.

[2] Gamma, E, Helm, R, Johnson, R, and Vlissides, J. Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.