A New Design Pattern for Sorting Algorithms

Md. Khalid Imam Rahmani¹, Sherjung¹, Shailendra Narayan Singh², Dr. Mohd. Qasim Rafiq³

M. Tech. Scholars¹
Prof & Head²
Department of Computer Science and Engineering
Al-Falah School of Engineering and Technology,
M.D. University
Faridabad, INDIA.
kirahmani@rediffmail.com, sherjung2005@yahoo.co.in

Prof & Chairman³, Deptt. of Computer Engineering,
Zakir Hussian College of Engineering & Technology,
A.M.U., Aligarh(U.P), INDIA.

Abstract. In this paper we have proposed an object oriented framework for implementing and visualization of comparison based sorting algorithms and linear gradient function based sorting algorithm. We model comparison based sorting as an abstract class with an abstract method to perform the arrangement of elements. Splitting and combining of arrays are done by the concrete subclasses. Performance measurements and visualizations can be added without modifying any code by using the decorator design pattern. Our design not only provides a concrete way of unifying different sorting algorithms but also help to analyze them at the proper level of abstraction. The framework can also be adapted to implement a gradient function based sorting algorithm.

Keywords. Object-oriented design; design patterns; linear gradient function; performance measurement and visualization, abstract class, concrete class, final class, software reuse.

1 Introduction

There are two direct applications of sorting: first as an aid for searching and second as a tool to match entries in files [1]. It is estimated that more than 25% of all computing time is consumed by sorting only. Some implementations spending even more than 50% of their computing time in sorting files [1]. Bubble sort, insertion sort, selection sort and exchange sort are applicable for input data of small to medium size whereas quick sort, merge sort and heap sort are applicable for an application expecting large to huge data size. These sorting algorithms are comparison based and hence can be no faster than $O(N \log N)$ [2, 7]. There are a few algorithms claiming to run in linear time but only for specialized cases of input data [7].
Object oriented programming has emerged as a superior programming methodology to procedure oriented programming methodology owing to the salient features of object oriented programming that facilitates reduced code size, less coding effort, code reusability, code abstraction, data abstraction, polymorphism, more interoperability, easy modifiability and multi-dimensional extensibility. Design patterns are time tested and well proven frameworks which are inevitable for large software systems. Object oriented design patterns are needed to be explored for an easy and efficient implementation of sorting procedures.

2 Background and Related Work

2.1 Software design patterns

Software design patterns are inspired from the architectural patterns of Christopher Alexander. Alexander says patterns repeat themselves, since they are a generic solution to a given system of forces. The object movement appears to the real world, so it seems reasonable that software design patterns should be repeated in real world objects.

The roots of the patterns movement are found in the writings of architect Christopher Alexander, who describes a pattern as a generic solution to a given system of forces in the world [22]. Alexander's patterns can be observed in everyday structures.

Unfortunately examples of software design patterns are not as abundant as Alexandrian patterns. Details of most elegant designs are not easily available as they are of proprietary nature.

In software, real world problems are either modeled entirely, or real world objects are transformed into hardware and software to produce real world results [23]. All 23 Gang of Four Patterns have been grouped into three categories, namely, creational patterns, structural patterns and behavioral patterns as depicted in Fig. 1.

Creational Patterns. Five creational patterns can be found in manufacturing, fast food industry, biology and political organizations.

Abstract Factory. The Abstract Factory is used to provide an interface for creating related objects, without creating their concrete classes. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right doors, left doors, right front fenders, left front fenders, hoods etc. for different cars.

Builder. The Builder pattern separates the building of a complex object from its representation, so that the same building process can create different representations. In a fast food restaurant, there can be variation in the contents of the children's meal, but the construction process is the same. This same process is used at competing restaurants.

Factory Method. The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding process
demonstrates this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. **Prototype.** The **Prototype** pattern specifies the kind of objects to create using a prototypical instance. Prototypes of new products are often built prior to full production, but in this example, the prototype is passive, and does not participate in copying itself. The mitotic division of a cell, resulting in two identical cells, is an example of a prototype that plays an active role in copying itself and thus, demonstrates the **Prototype** pattern. When a cell splits, two cells of identical genotype result. In other words, the cell clones itself.

![Design Patterns](image)

**Singleton.** The **Singleton** pattern ensures that a class has only one instance, and provides a global point of access to that instance. The **Singleton** pattern is named after the singleton set, which is defined to be a set containing one element. The office of the President of the United States is a **Singleton.** The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president.
at any given time. Regardless of the personal identity of the active president, the title, “The President of the United States” is a global point of access that identifies the person in the office.

**Structural Patterns.** Seven structural patterns have been documented by the Gang of Four.

*Adapter.* The *Adapter* pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients. Socket wrenches provide an example of the *Adapter.* A socket attaches to a ratchet, provided that the size of the drive is the same.

*Bridge.* The *Bridge* pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the *Bridge.* The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, a simple two position switch, or a variety of dimmer switches.

*Composite.* The *Composite* composes objects into tree structures, and lets clients treat individual objects and compositions uniformly. Although the example is abstract, arithmetic expressions are *Composites.* An arithmetic expression consists of an operand, an operator (+ - * /), and another operand. The operand can be a number, or another arithmetic expression. Thus, 4 + 2 and (1 + 2) + (3 * 2) are both valid expressions.

*Decorator.* The *Decorator* attaches additional responsibilities to an object dynamically. Although paintings can be hung on a wall with or without frames, frames are often added, and it is the frame which is actually hung on the wall. Prior to hanging, the paintings may be matted and framed, with the painting, matting, and frame forming a single visual component.

*Facade.* The *Facade* defines a unified, higher level interface to a subsystem, that makes it easier to use. Consumers encounter a *Facade* when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a *Facade,* providing an interface to the order fulfillment department, the billing department, and the shipping department.

*Flyweight.* The *Flyweight* uses sharing to support large numbers of objects efficiently. The public switched telephone network is an example of a *Flyweight.* There are several resources such as dial tone generators, ringing generators, and digit receivers that must be shared between all subscribers. A subscriber is unaware of how many resources are in the pool when he or she lifts the hand set to make a call. All that matters to subscribers is that dial tone is provided, digits are received, and the call is completed.

*Proxy.* The *Proxy* provides a surrogate or place holder to provide access to an object. A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.

**Behavioral Patterns.** Examples of these eleven patterns can be found in coin sorting banks, restaurant orders, music, transportation, auto repair, vending machines, and home construction.

*Chain of Responsibility.* The *Chain of Responsibility* pattern avoids coupling the sender of a request to the receiver, by giving more than one object a chance to handle the request. Mechanical coin sorting banks use the *Chain of Responsibility.* Rather
than having a separate slot for each coin denomination coupled with receptacle for the denomination, a single slot is used. When the coin is dropped, the coin is routed to the appropriate receptacle by the mechanical mechanisms within the bank.

Command. The Command pattern allows requests to be encapsulated as objects, thereby allowing clients to be parameterized with different requests. The "check" at a diner is an example of a Command pattern. The waiter or waitress takes an order, or command from a customer, and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of "checks" used by different diners is not dependent on the menu, and therefore they can support commands to cook many different items.

Interpreter. The Interpreter pattern defines a grammatical representation for a language and an interpreter to interpret the grammar. Musicians are examples of Interpreters. The pitch of a sound and its duration can be represented in musical notation on a staff. This notation provides the language of music [14]. Musicians playing the music from the score are able to reproduce the original pitch and duration of each sound represented.

Iterator. The Iterator provides ways to access elements of an aggregate object sequentially without exposing the underlying structure of the object. On early television sets, a dial was used to change channels. When channel surfing, the viewer was required to move the dial through each channel position, regardless of whether or not that channel had reception. On modern television sets, a next and previous button are used. When the viewer selects the "next" button, the next tuned channel will be displayed. Consider watching television in a hotel room in a strange city. When surfing through channels, the channel number is not important, but the programming is. If the programming on one channel is not of interest, the viewer can request the next channel, without knowing its number.

Mediator. The Mediator defines an object that controls how a set of objects interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the Mediator, rather than with each other. The control tower at a controlled airport demonstrates this pattern very well. The pilots of the planes approaching or departing the terminal area communicate with the tower, rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area.

Memento. The Memento captures and externalizes an object's internal state, so the object can be restored to that state later. This pattern is common among do-it-yourself mechanics repairing drum brakes on their cars. The drums are removed from both sides, exposing both the right and left brakes. Only one side is disassembled, and the other side serves as a Memento of how the brake parts fit together [8]. Only after the job has been completed on one side is the other side disassembled. When the second side is disassembled, the first side acts as the Memento.

Observer. The Observer defines a one-to-many relationship, so that when one object changes state, the others are notified and updated automatically. Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price, which is broadcast to all of the bidders in the form of a new bid.
State. The State pattern allows an object to change its behavior when internal state of a system changes. This pattern can be observed in a vending machine. Vending machines have states based on the inventory, amount of currency deposited, the ability to make changes, the item selected, etc. When currency is deposited and a selection is made, a vending machine will deliver a product and no change, deliver a product and change, deliver no product due to insufficient currency on deposit, or deliver no product due to inventory depletion.

Strategy. A Strategy defines a set of algorithms that can be used interchangeably. Modes of transportation to an airport is an example of a Strategy. Several options exist, such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on tradeoffs between cost, convenience, and time.

Template Method. The Template Method defines a skeleton of an algorithm in an operation, and defers some steps to subclasses. Home builders use the Template Method when developing a new subdivision. A typical subdivision consists of a limited number of floor plans, with different variations available for each floor plan. Within a floor plan, the foundation, framing, plumbing, and wiring will be identical for each house. Variation is introduced in the latter stages of construction to produce a wider variety of models.

Visitor. The Visitor pattern represents an operation to be performed on the elements of an object structure, without changing the classes on which it operates. This pattern can be observed in the operation of a taxi company. When a person calls a taxi company he or she becomes part of the company's list of customers. The company then dispatches a cab to the customer (accepting a visitor). Upon entering the taxi, or Visitor, the customer is no longer in control of his or her own transportation, the taxi (driver) is.

2.2 Advantage of using patterns

Alexander had hoped that true patterns would enter a common language that all could share [24]. Within the software design community, patterns are seen as a way to develop a set of languages to streamline communication between colleagues [25, 26]. Patterns are expected to provide a vocabulary for discussing structures larger than modules, procedures, or objects [27]. If software design patterns are to become a common language among programmers, shared meaning is essential. If design decisions are communicated, but not understood, designers are forced to make missing assumptions to complete the job [28]. Commonplace examples facilitate understanding, because in order to understand anything, people must find the closest item in memory to which it relates [29].
2.3 Sorting

**Basic Concepts.** Sorting is a process of rearranging the available data items into an ordered sequence [3]. A sorting algorithm is a set of steps arranged in a particular sequence that puts the available data items into a certain order. The well-known ordered sequences have been increasing order, decreasing order, non-increasing order, non-decreasing order and lexicographic order [2]. An efficient sorting mechanism is important to optimizing the design of other algorithms that require sorted data items to work correctly.

If the items to be sorted all fit into the computer's internal memory, then it is known as an internal sorting algorithm. Due to the growing power of computers, external storage devices become less frequent in sorting. If all the items cannot be stored in the internal memory at one time, different techniques have to be used. The underlying idea is to sort as many items as the internal memory can handle at a time and then merge the results into external storage devices.

Sorting algorithms are also classified by their computational complexity and ease of implementation. For a typical sorting algorithm ideal behavior is \(O(n)\), good behavior is \(O(n \log n)\) and bad behavior is \(O(n^2)\). The lower bound of time complexity of sorting algorithms, which only use key comparison operation, is \(O(n \log n)\).

Comparison based sorting algorithms rearrange the input data items by comparing the key values of the adjacent items or that of one item with that of another item.

Some sorting algorithms are in place, such that only \(O(1)\) or \(O(\log n)\) memory is needed in addition to the items being sorted, while others are not in place that is they require auxiliary memory locations for data to be temporarily stored.

Stable sorting algorithms maintain the relative order of items with equal key values that is; a sorting algorithm is called stable if whenever there are two items \(p \) and \(q \) with the same key value and \(p \) appears before \(q \) in the input data, \(p \) will appear before \(q \) in the sorted data [3].

**Bubble Sort.** Bubble sort is considered the simplest sorting algorithm. The basic idea behind bubble sort is that items with higher key values are heavy and therefore must be bubbled up to the right. We make \(n-1\) passes from left to right. During a pass, if two adjacent items are out of order i.e. if the greater one is before, we exchange them. The effect of this operation is that at the end of first pass the largest item reaches the rightmost position. At the end of second pass, the second largest item reaches the second rightmost position, and so on.

**Insertion Sort.** We consider the data structure, which is holding the items, of having two parts. Initially, the first part is holding the first item and the second part is holding the remaining \(n-1\) items. The first part of the data structure is sorted since only a single item is trivially sorted. Now take one item at a time from the beginning of the second part and put it into a proper position in the first part such that at each stage, the first part of the data structure is sorted.

**Selection Sort.** Find the smallest of the given \(n\) items; swap it with the item at the first position. Then find the smallest of the remaining \(n-1\) items, swap it with the item
at the second position. Then find the smallest of the remaining n-2 items, swap it with the item at the third position and so on until the largest item, automatically, is left at the last position.

**Interchange Sort.** If two items in the sequence are found to be out of order, exchange them so that they are in order now.

**Enumeration Sort.** If we come to know that there are k items, which are smaller than the item we are currently considering then its final position will be k+1.

**Heap Sort.** As a sequential representation of a binary tree, the parent of the node at location i is at \([i/2]\) the left child at 2i and the right child at 2i + 1. If 2i or 2i + 1 is greater than n (the number of nodes), then the corresponding children do not exist. Heap sort can be regarded as a two stage algorithm. First the input data items are converted into a heap. A heap is defined to be a complete binary tree with the property that the value of each node is at least as small as the value of its children nodes (if they exist) (i.e., \([j/2]\) \leq K_i for \(1 \leq [j/2] < j \leq n\)). The root of the heap has the smallest item. In the second stage the sorted sequence is generated in increasing order by successively deleting the root and restructuring the remaining tree into a heap.

**Merge Sort.** The Merge sort follows the divide-and-conquer approach as follows.

- **Divide:** Divides the unsorted sequence of n items into two subsequences of \(n/2\) items each.
- **Conquer:** Applies Merge sort recursively on the two subsequences.
- **Combine:** Merges the two sorted subsequences into a single sorted sequence.

The recursion stops when the sequence to be sorted has only one item, since every single item is already sorted. The algorithm performs the main job of merging the two sorted sequences in the combine step.

**Quick Sort.** Quick sort is a well-known sorting algorithm which, on average, makes \(O(n \log n)\) comparisons to sort \(n\) items. However, in the worst case, it makes \(O(n^2)\) comparisons. Quick sort is significantly faster in practice than other \(O(n \log n)\) algorithms, because its inner loop can be efficiently implemented on most architectures, and in most real-world data it is possible to make design choices which minimize the probability of requiring quadratic time. Quick sort is a comparison sort and, in efficient implementations, is not a stable sort. Quick sort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

The steps are:

a) Pick an element, called a pivot, from the list.
b) Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
c) Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

### 2.4 Performance of conventional sorting algorithms

Here we give a table comparing the performance of the above algorithms when operate on a C arrays. These numbers are only of small significance, since they can vary from one architecture to other architecture.
Proceedings of NCSOFT 2010

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Technique</th>
<th>DS used</th>
<th>Average</th>
<th>Worst</th>
<th>Stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>Exchange</td>
<td>Array</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection</td>
<td>Selection</td>
<td>Array</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>No</td>
</tr>
<tr>
<td>Insertion</td>
<td>Insertion</td>
<td>Array/List</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Heap</td>
<td>Selection</td>
<td>Array</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
<td>No</td>
</tr>
<tr>
<td>Quick</td>
<td>Partition</td>
<td>Array</td>
<td>$O(n\log n)$</td>
<td>$O(n^2)$</td>
<td>No</td>
</tr>
<tr>
<td>Merge</td>
<td>Merge</td>
<td>Array/List</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
<td>Yes</td>
</tr>
</tbody>
</table>

3 Object Oriented Design Patterns for Sorting

In a conventional way of analyzing sorting, we start with easy algorithms such as bubble sort and insertion sort and then move to more complicated ones, such as merge sort, quick sort and heap sort. In this way, we generally lost into the coding details of such algorithms.

Merritt proposed the top down approach of studying sorting algorithms [8]. At the top of her sorting taxonomy is an abstract divide-and-conquer algorithm: break the array to be sorted into two sub-arrays, sort the sub-arrays recursively, and combine the sorted sub-arrays 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 break and combine procedures: easy break/hard combine and hard break/easy combine. At the top of the groups easy break/hard combine and hard break/easy combine are quick sort and merge sort, respectively, and below them will fit all other well-known, more "low-level" sorting 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 work is a very powerful method for studying and understanding sorting. Merritt's abstract characterization of sorting represents object-oriented methodology and can be represented using of OOP concepts.

We present in this paper our OOP formulation and implementation of Merritt's taxonomy. We have explained how the abstract concept of sorting is appropriately formulated in terms of standard design patterns [9].
3.1 Template Method Design Pattern

At the center of Merritt's taxonomy is the thesis that all comparison-based sorts can be expressed as a divide-and-conquer algorithm. She has exemplified it by merge sort and quick sort.

Modeling an invariant behavior that is made of variant behaviors is a simple use of the template method design pattern [9] as shown in the following UML diagram using syntax of Java.

In Fig 1, the abstract class, ASorter, solves the general sorting principles. Its sort() 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.

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

Figure 2 below shows the recursive call tree for the sort() method of a hypothetical sort algorithm.

![Recursive Call Tree](image)

**Figure 2** Hypothetical Recursion Tree for Sortine.

### 3.2 The strategy pattern for object comparison

The strict order comparison provides the implementation of stable sort algorithms. Following is the Java code for \textit{AOrder}.

```java
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.
3.3 Concrete Sort Examples

Listing 2 in the next column shows insertion sort is implemented in our 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) {
        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.

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.

4 Conclusion

The Object oriented sorting framework presented here reflects our research work. Object-orientation and the language of patterns provide a clean and concise way of formulating and implementing sorting based on this principle.

We are in the process of developing a clean framework to implement sorting algorithms. Object oriented design patterns for sorting which we have proposed will improve the efficiency and ease of programming such algorithms.
5 Future work

This paper is based on our ongoing research of designing a new sorting algorithm which will run in $O(N)$ time in best case and in $O(N\log_2 N)$ in worst case. Since the research is going on, we have not included the implementation of linear gradient function based sorting algorithm in our framework.

Acknowledgment

The authors would like to thank Honorable Chairman, Mr. Jawwad Ahmad Siddiqui, Al-Falah School of Engineering and Technology, Faridabad, Haryana, INDIA, for providing us the entire necessary infrastructure for higher technical education and valuable support for research & development in the college. Without the facilities at the college, this work would have not been possible.

References