Design Patterns for Sorting

something old in a new light

Dung “Zung” Nguyen, Rice University

Stephen Wong, Rice University
What is Sorting Anyway?

Some concrete examples:

- Selection sort
- Insertion sort

- Can We Abstract All Sorting Processes?
Merritt’s Thesis for Sorting

- All comparison-based sorting can be viewed as “Divide and Conquer” algorithms.
- Sort a pile
  - Split the pile into smaller piles
  - Sort each the smaller piles
  - Join the sorted smaller piles into sorted pile
Hypothetical Sort

- **Divide and Conquer!**
- **How can we capture this abstraction?**
Abstract Sorter Class

Concrete
“Template Method”

ASorter

+ void: sort(Object[ ] A, int: lo, int: hi);
# int: split(Object[ ] A, int lo, int hi);
# void: join(Object[ ] A, int lo, int s, int hi);

if (lo < hi) {
    int s = split (A, lo, hi);
    sort (A, lo, s-1);
    sort (A, s, hi);
    join (A, lo, s, hi);
}

abstract, relegated to subclasses

Selection

Insertion

SortAlgo
Template Method Pattern

- Expresses `invariant` in terms of `variants`.
- White-box Framework:
  - Extension by subclassing

```java
invariant () {
    ...
    variant1 (...);
    ...
    variant2 (...);
    ...
}
```
void sort (Object A[], int lo, int hi) {
    if (lo < hi) {
        int s = split (A, lo, hi);
        sort (A, lo, s-1);
        sort (A, s, hi);
        // A[s:hi] is sorted.
        join (A, lo, s, hi);
    } // else if (hi <= lo) do nothing!
}
Insertion Sort

- int split(Object[] A, int lo, int hi) {
  - return hi;
}

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

- Reduces to insertion of a single object into a sorted array.
- Simplifies proof of correctness.
Selection Sort

- int split(Object[] A, int lo, int hi) {
  int s = lo;
  for (int i = lo + 1; i <= hi; i++) {
    if (aOrder.lt(A[i], A[s])) s = i;
  }
  // s = index of min value
  swap (A, lo, s);
  return lo + 1;
}

- void join(Object[] A, int lo, int s, int hi) {

- Reduces to selecting a minimum value in the array.
- Simplifies proof of correctness
void sort (Object A[ ], int l, int h) {
    if (l < h) {
        int s = split (A, l, h);
        sort (A, l, s - 1);
        sort (A, s, h);
        join (A, l, s, h);
    }
}

\[ T(l, h) = \begin{cases} 
C & \text{if } h \leq l \\
C + S(l, h) + T(l, s-1) + T(s, h) + J(l, s, h) & \text{if } l < h 
\end{cases} \]
**Insertion Sort Complexity**

- \( \text{int split (Object[ ] A, int l, int h)} \) {
  \[
  \text{return h;}
  \]
} // \( \mathcal{O}(1) \)

- \( \text{void join (Object[ ] A, int l, int s, int h)} \) {
  \[
  \text{Object key = A[h]; int j;}
  \text{for (j = h; l < j && aOrder.lt(key, A[j-1]); j - -) }
  \]
} // \( \mathcal{O}(h-l) \)

- \( T(l, h) = \)
  - \( C \) if \( h \leq l \)
  - \( C + S(l, h) + T(l, h-1) + T(h, h) + J(l, h, h) \) if \( l < h \)

- **Let** \( n = h - l, T(l, h) = T(n) = \)
  - \( C \) if \( n < 1 \)
  - \( T(n-1) + O(n) = O(n^2) \) if \( 1 \leq n \)
Sorting as a Framework

- Unifies sorting under one foundational principle: Divide and Conquer!
- Reduces code complexity. Increases robustness.
- Simplifies program verification and complexity analysis.

```
if (lo < hi) {
    int s = split (A, lo, hi);
    sort (A, lo, s-1);
    sort (A, s, hi);
    join (A, lo, s, hi);
}
```

**ASorter**

```plaintext
+ void: sort(Object[ ] A, int: lo, int: hi);
# int: split(Object[ ] A, int lo, int hi);
# void: join(Object[ ] A, int lo, int s, int hi);
```
Classifications

**Insertion**  **Merge**

**ASorter**

+ void: sort(Object[] a, int lo, int hi);
# int: split(Object[] A, int lo, int hi);
# void: join(Object[] A, int lo, int s, int hi);

**Selection**  **QuickSort**  **HeapSort**  **Bubble**

Easy split/Hard join

Hard split/Easy join
Not Just Buzzwords…

- **Reusability**: write once/use many
  - Reuse the invariant: the framework
- **Flexibility**: change the variants
  - Add new sorters
  - Change sort order
- **Extensibility**: add new capabilities
  - Visualization
  - Performance measurements

**It’s more than just sorting…**

**It’s all about abstraction…**

Abstraction teaches software engineering
Extending Without Changing

The Abstract is the Invariant

○ Graphics, sort order and performance measurements are completely separate from the sorting.

○ Add functionality to the sorters, ordering operators, and sortable objects **without disturbing their abstract behavior**.

○ Wrap the sorters, operators and objects in something abstractly equivalent that adds functionality: **Decorators**
Decorator Design Pattern

Client

uses

AComponent

+ Object: method(Object x)

Decorator performs additional processing

Decorators can be layered on top of each other

Decorator is abstractly equivalent to the decoree

Decorator intercepts calls to decoree

Subclass holds an instance of its superclass

Concretempl

+ Object: method(Object x)

// do additional processing
Object y = decoree.method(x);
// do more processing
return y;

Decorator

- AComponent: decoree

+ Object: method(Object x)

Client deals with an abstract entity

Client doesn’t know the decoration exists!
Sorters

Abstract Template Pattern sorter

Concrete sorters implement split() & join()

Decorated ASorter for graphical split & join

Sortable objects being sorted

decorate ordering strategy for comparisons

Sort using an abstract ordering strategy

BubbleSorter

InsertionSorter

QuickSorter

SelectionSorter

MergeSorter

HeapSorter

Heapifier

ASorter

# AOrder : aOrder
# ASorter(AOrder aOrder)
+ void : sort(Object[] A, int lo, int hi)
# int : split(Object[] A, int lo, int hi)
+ void : join(Object[] A, int lo, int s, int hi)
+ void : setOrder(AOrder aOrder)
GraphicSorter Decorator

private ASorter sorter;

int split(Object[] A, int lo, int hi) {
    int s = sorter.split(A, lo, hi);
    // recolor split sections and pause
    return s;
}

void join(Object[] A, int lo, int s, int hi) {
    sorter.join(A, lo, s, hi);
    // recolor joined sections and pause
}

Decoree.
Delegation to the decoree.
Graphics decoration.

Identical behavior as the decoree.

Delegation to the decoree.
Graphics decoration.
Comparison Strategies

- Decorated AOrder to graphically show comparisons
- Decorated AOrder to reverse sort order
- Abstract comparison operator
- Decorated AOrder to count comparisons

Abstract:
- boolean : eq(Object x, Object y)
- boolean : lt(Object x, Object y)
- boolean : ne(Object x, Object y)
- boolean : le(Object x, Object y)
- boolean : gt(Object x, Object y)
- boolean : ge(Object x, Object y)
Sortable Integers

Concrete implementation

Decorated AInteger to count accesses

Abstract class

Factory method for comparison strategy

IColoredObject adapter + AInteger decorator for graphics
Beyond sorting…

*Design Patterns Express Abstraction*

Instead of disparate and complex

Abstraction unifies and simplifies

Instead of rigidity and limitedness

Abstraction enables flexibility and extensibility