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:

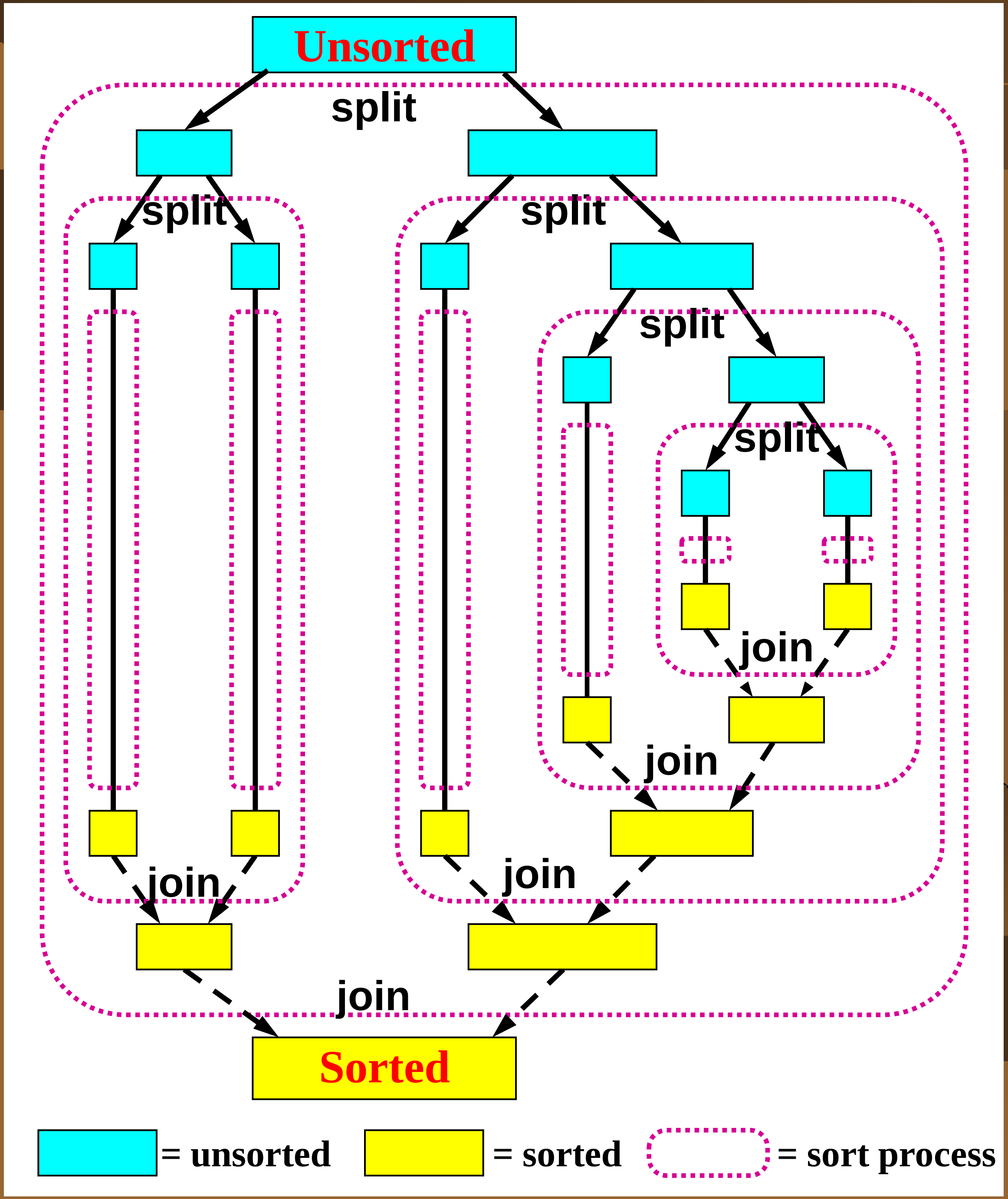
- Selection sort
- Insertion sort

- *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

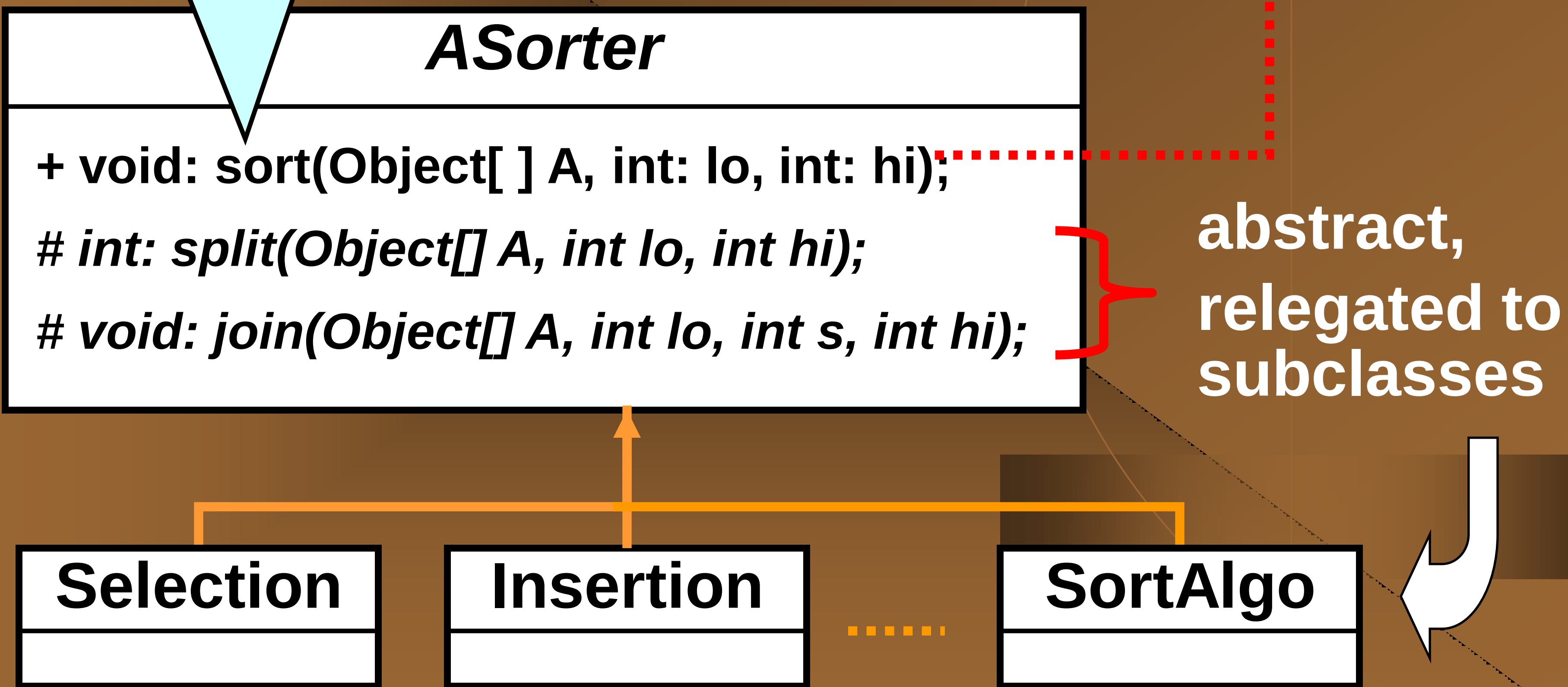


- Divide and Conquer!
- How can we capture this abstraction?

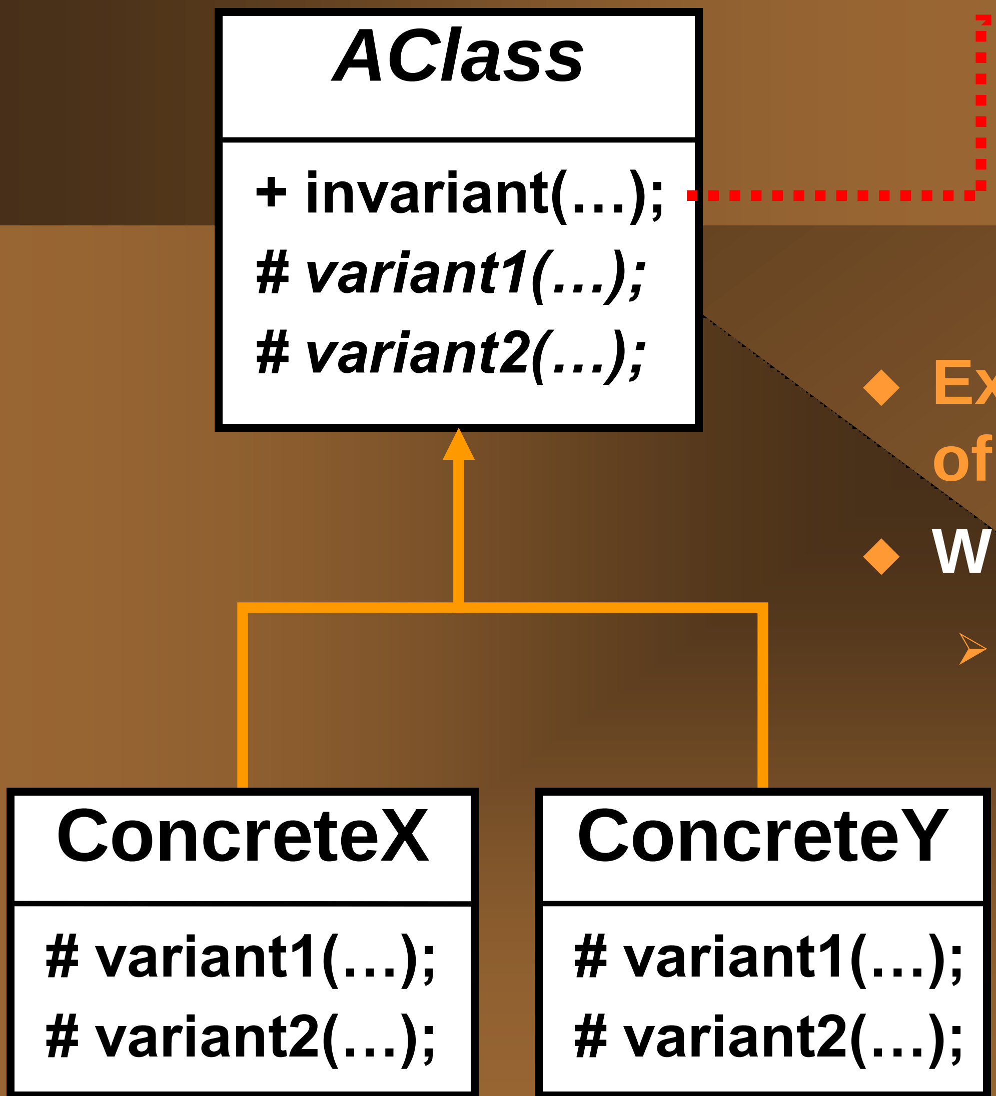
Abstract Sorter Class

Concrete
“Template Method”

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



Template Method Pattern



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

Sort Framework

```
void sort (Object A[ ], int lo, int hi) {
    if (lo < hi) {
        int s = split (A, lo, hi);
 // A[lo:s-1], A[s:hi] form a proper partition of A[lo:hi].
        sort (A, lo, s-1);
  // A[lo:s-1] is sorted.
        sort (A, s, hi);
  // A[s:hi] is sorted.
        join (A, lo, s, hi);
  // A[lo:hi] is sorted.
    } // else if (hi <= lo) do nothing!
}
```

Recursive case

"Free"

Focus points

Base case is trivial

Insertion Sort

- int split(Object[] A, int lo, int hi) {
  - return hi;
    // A splits into A[lo:hi-1] and A[hi:hi]
    }
- void join(Object[] A, int lo, int s, int hi) {

  // Pre: A[lo:hi-1] is sorted, s = hi.

  - Object key = A[hi];

```
int j;
for (j = hi; lo < j && aOrder.lt(key, A[j-1]); j- -)
    A[j] = A[j-1];
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

- Reduces to selecting a minimum value in the array.
- Simplifies proof of correctness

```
◆ 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);
    // A[lo] = min value in A[lo:hi]
    • return lo + 1;
    // A splits into A[lo:lo] and A[lo+1:hi]
      }
◆ void join(Object[] A, int lo, int s, int hi)  { }
```

Do Nothing!

Time Complexity

- void sort (Object A[ ], int l, int h) { → *T(l, h)*
  - if (l < h) { → C
    - int s = *split* (A, l, h); → *S(l, h)*
    - sort (A, l, s-1); → *T(l, s-1)*
    - sort (A, s, h); → *T(s, h)*
    - *join* (A, l, s, h);}} → *J(l, s, h)*

T(l, h) =

- C if h <= l
- C + *S(l, h)* + *T(l, s-1)* + *T(s, h)* + *J(l, s, h)* if l < h

Insertion Sort Complexity

- ```
  int split (Object[ ] A, int l, int h) {
      return h;
  } // O(1)
  ```
- ```
  void join (Object[ ] A, int l, int s, int h) {
      Object key = A[h]; int j;
      for (j = h; l < j && aOrder.lt(key, A[j-1]); j- -)
          A[j] = A[j-1];   A[j] = key;
  } // O(h-l)
  ```
- *T(l, h)* =
  - C if h<= 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 <= n

Sorting as a Framework

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

ASorter

+ void: sort(Object[ ] A, int: lo, int: hi);

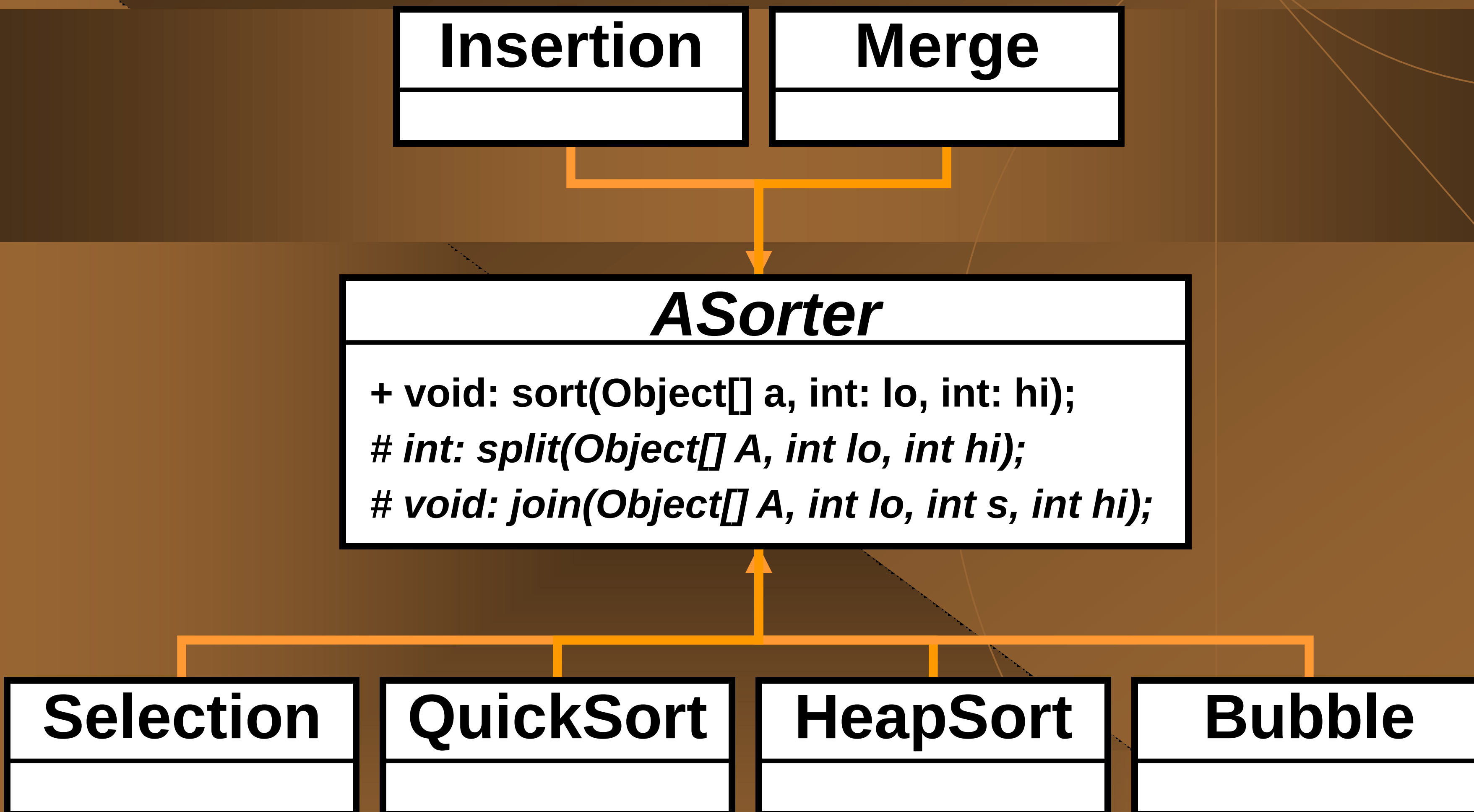
 # int: split(Object[] A, int lo, int hi);

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

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

Classifications

Easy split/Hard join



Hard split/Easy join

It's more than just sorting…

It's all about abstraction…

Abstraction teaches software engineering

Not Just Buzzwords…

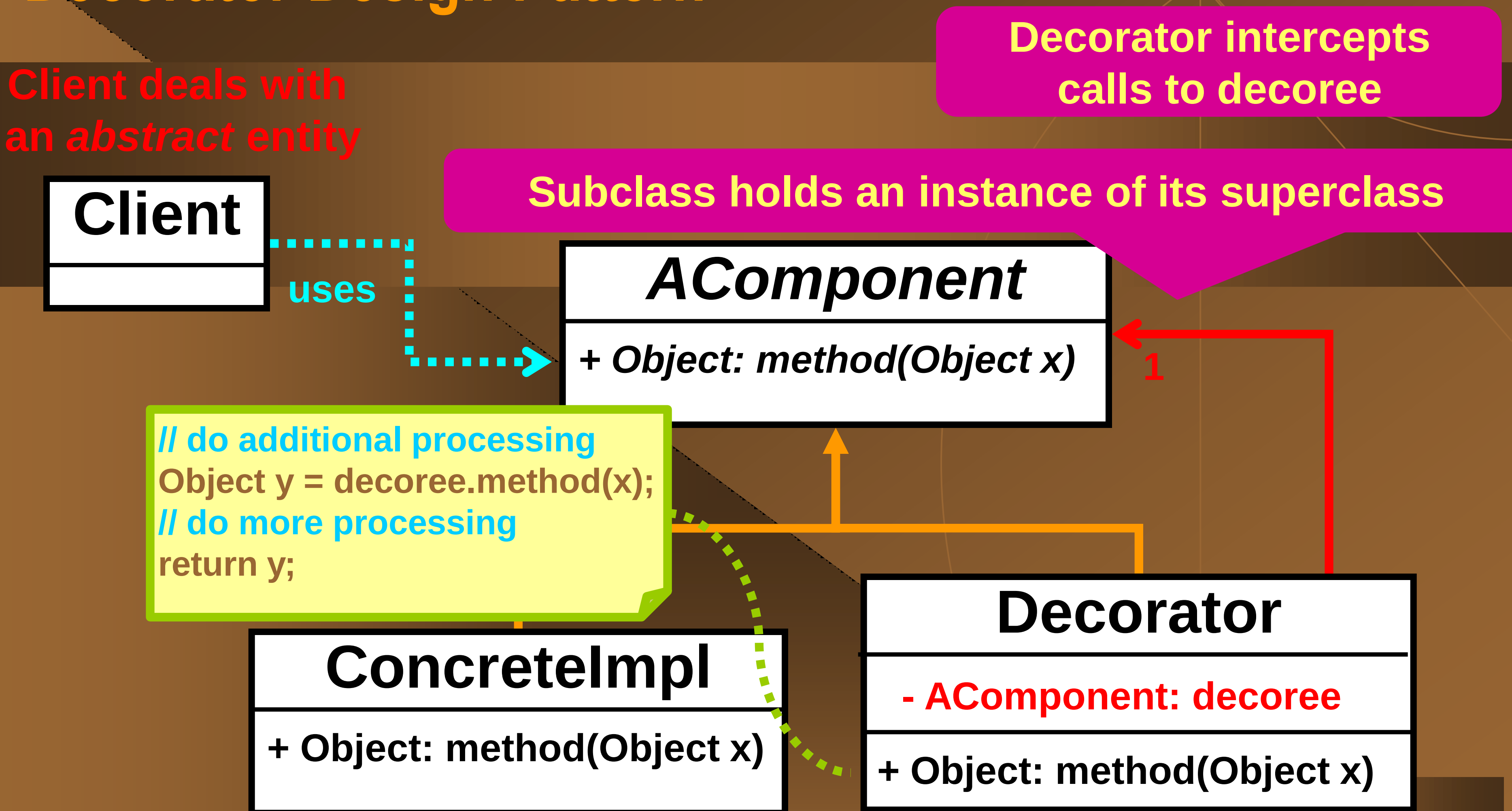
- **Reusability**: write once/use many
  - Reuse the invariant: the framework
- **Flexibility**: change the variants
  - Add new sorters
  - Change sort order
- **Extensibility**: add new capabilities
  - Visualization
  - Performance measurements

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



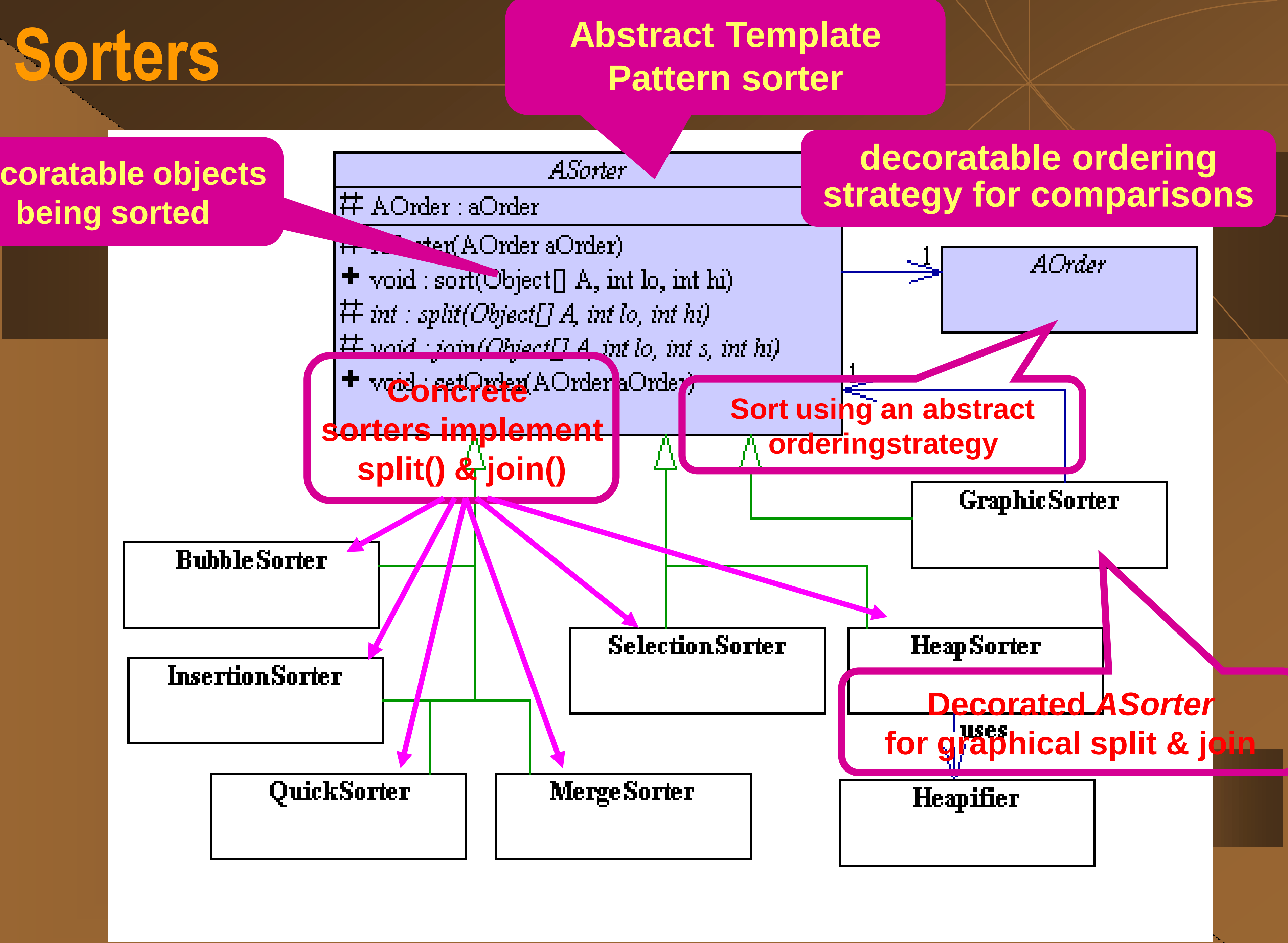
Decorator performs additional processing

Decorators can be layered on top of each other

Decorator is abstractly equivalent to the decoree

Client doesn't know the decoration exists!

Sorters



Abstract Template Pattern sorter
coratable objects being sorted
decoratable ordering strategy for comparisons
ASorter
AOrder : aOrder
+ void : sort(Object[] A, int lo, int hi)
# int : split(Object[] A, int lo, int hi)
+ void : setOrder(AOrder aOrder)
AOrder
1
1
Concrete sorters implement split() & join()
Sort using an abstract orderingstrategy
GraphicSorter
BubbleSorter
InsertionSorter
SelectionSorter
HeapSorter
QuickSorter
MergeSorter
Heapifier
uses
Decorated *ASorter* for graphical split & join

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.

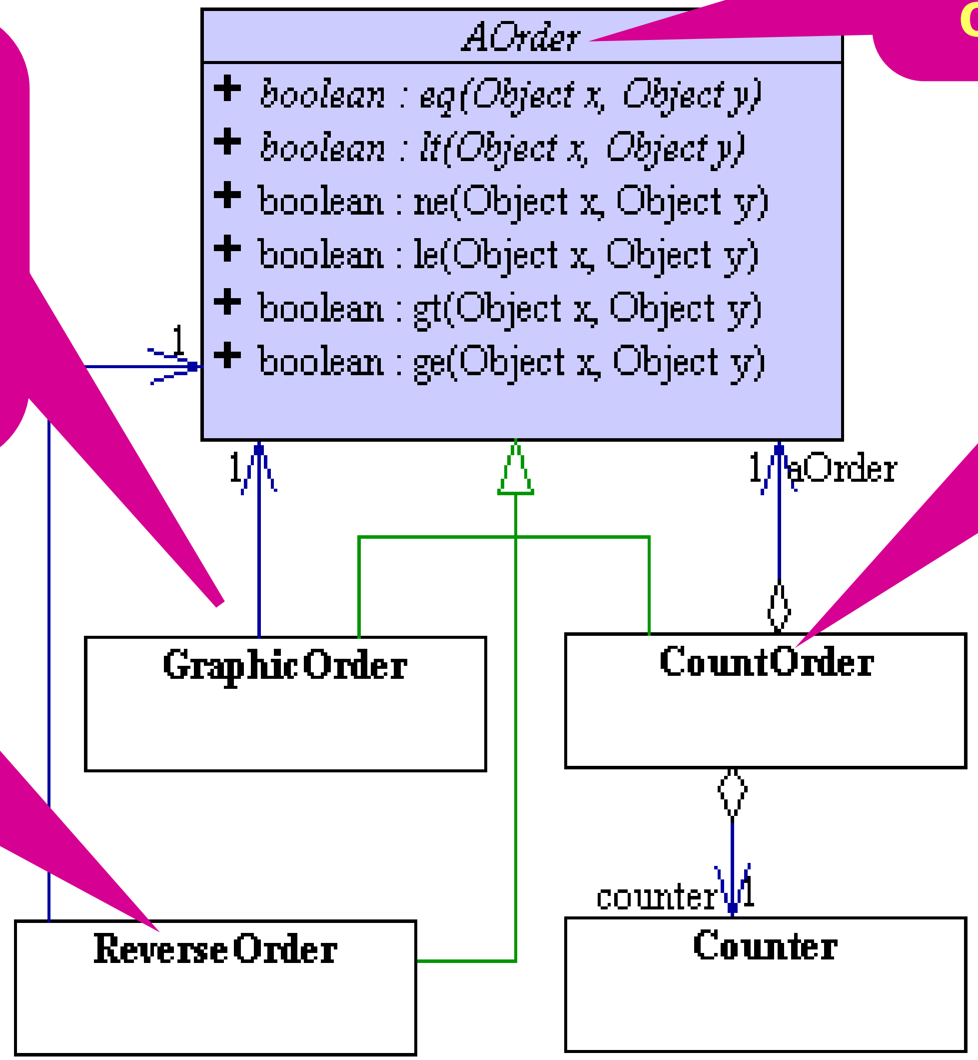
Comparison Strategies

Abstract comparison operator

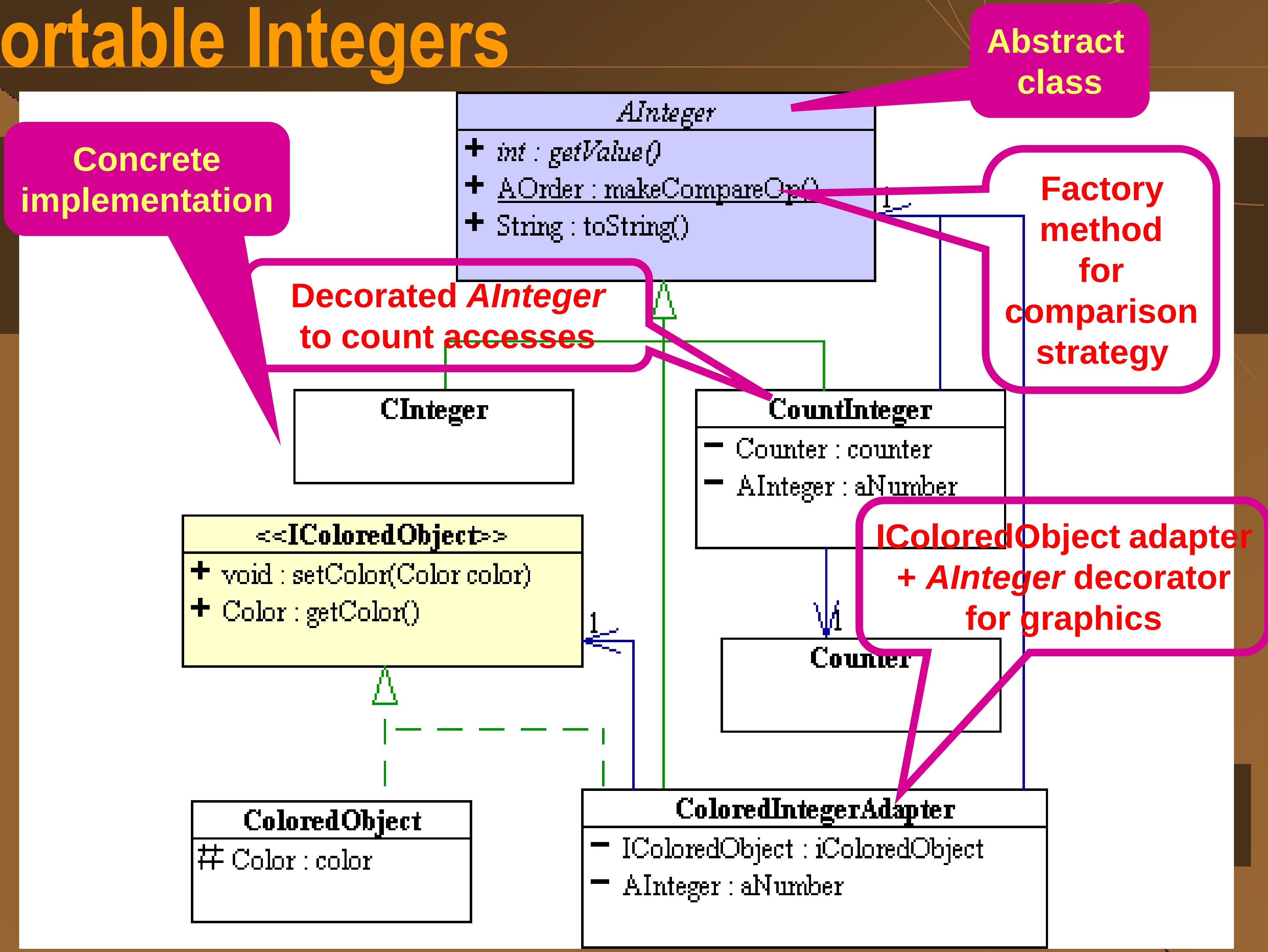
Decorated *AOrder* to graphically show comparisons

Decorated *AOrder* to reverse sort order

Decorated *AOrder* to count comparisons



Sortable Integers



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

Download the paper!

http://exciton.cs.rice.edu/research/SIGCSE01