Lecture note:

Software Construction

Algorithm Patterns

Finn E. Nordbjerg
**Contents**

Contents .......................................................................................................................................... 2

Algorithm Patterns......................................................................................................................... 3
  Introduction............................................................................................................................... 3
  On Notation............................................................................................................................ 4
  Sweep Algorithms.................................................................................................................. 4
    Sweep algorithms on sequences of integers......................................................................... 5
  Search Algorithms ................................................................................................................ 7
    Specification of the abstract operations .............................................................................. 8
    Concretisation of the search pattern on sequences of integers ........................................ 8
    Binary search: A "smart" realisation of the search pattern on a sorted sequence ............... 9
  Merge Algorithms .................................................................................................................. 11
    The Abstract Merge Algorithm........................................................................................... 12
    Example: Total Merge of two Arrays of Integers............................................................... 13
Algorithm Patterns

Introduction
In this lecture note we will introduce the concept of an algorithm pattern, and we take a closer look at the first two of a series of abstract algorithms or what we will call algorithm patterns.

The concept of patterns originates from architecture (Christopher Alexander, 1977\(^1\)):

\[ \textit{“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”} \]

In software engineering the concept of patterns is well known. Especially Object-Oriented Design Patterns\(^2\) is a very important concept in software development. Later on in this course on Software Construction we shall look into design patterns in more details.

There are several benefits associated with the use of patterns:

- A pattern captures a proven good design:
  - A pattern is based on experience
  - A pattern is discovered – not invented
- It introduces a new (and higher) level of abstraction, which makes it easier:
  - To talk and reason about design on a higher level
  - To document and communicate design
- One doesn’t have to reinvent solutions over and over again
- Patterns facilitate reuse not only of code fragments, but also of ideas.

The use of patterns also makes it easier to learn how to design and develop good software. It is often said that good skills in software construction require experience and talent and neither can be taught or learned at school! But luckily patterns capture experience (and talent) in a way that is communicable and comprehensible and hence can be learned. So you should put a lot of effort into understanding and learning about software patterns.

Let us now be more specific and turn to the group of software patterns, which we will call algorithm patterns:

---


\(^2\) Erich Gamma e. a.: ”Design Patterns. Elements of Reusable Object-Oriented Software”. Addison-Wesley. 1995
The notion of algorithm patterns comes from the observation that many different problems from many different problem domains may be solved by algorithms that possess a common structure – or a common pattern. By abstracting and formalising this structure it becomes a reusable pattern with all the desired properties (as mentioned above) of patterns. Patterns have names – within the field of algorithms the following – among others – can be identified:

- Sweep Pattern
- Search Pattern
- Merge Pattern
- Divide and Conquer Pattern
- Greedy Pattern
- Backtracking Pattern
- Dynamic Programming Pattern
- And many others…

The term “Algorithm pattern” is not commonly used in literature, but the concept is well known. Terms used in standard textbooks on algorithms and data structures are “Algorithm paradigms”, “Algorithm (or solution) strategies”, “Methodologies for developing algorithms” and the like. (I have only come across one single textbook using the term “Algorithmic Pattern”).

In the following we shall study two of the most common and simplest algorithm patterns: the Sweep Pattern and the Search Pattern.

**On Notation**

We will use a Java pseudo language to write algorithms. Abstract operations – i.e. operations to be concretised in a specific application of the pattern - are written inside sharp brackets (“<...>”). The abstract operations are to be concretised by statements in a concrete programming language, say Java. The concretisation depends on the actual problem to be solved by the algorithm and the representation of the data, on which the algorithm works.

**Sweep Algorithms**

A wide range of algorithms work by senselessly sweeping (hence the name sweep algorithms) through a collection of elements, looking at each element and taking some action depending on the characteristics of the actual element. For instance counting the number of students older than 25 years of age in of list of students, increasing the value of a discount percentage by 10 on all elements with a balance of more than DKK 10,000 in a set of customers, or calculating the average number of words per sentence in a text etc. etc.

---

This class of algorithms is called *sweep algorithms* and can be formulated abstractly in the *sweep pattern* as follows:

It is assumed that the elements to be processed are stored in some collection (*US*: unvisited set) and the elements in some way are removed from this set as the algorithm processes them.

< DO_INIT: initialisation necessary due to the DO operation >;
< INIT: initialise unvisited set, US >;
while < ! DONE (id. US is not empty)> {
    < SELECT actual element from US >;
    < DO something according to actual element >;
    < REMOVE actual element from US>
} // end while

The realisation of the operations INIT, DONE, SELECT and REMOVE depends on the data representation, that is: how is the collection of elements that the algorithm is sweeping represented. The realisation of DO_INIT and DO depends on the concrete task to be accomplished by the algorithm as well as the data representation.

**Sweep algorithms on sequences of integers**

If for instance the collection is a sequence of integers, say an array, the operations INIT, DONE, SELECT and REMOVE may be concretised by a counter, *i* that indicates the position of the first element in the unvisited set, see figure 1 below:

```
Figure 1: The collection is an array and the unvisited set (US) is concretised by a counter *i*.
```

Assuming the declarations:

```c
int i;
int a[];
```

and some initialisation of the array *a*, the abstract operations can be concretised as follows:

```
INIT: i = 0
```
DONE: i >= a.Length  
SELECT:  a[i]  
REMOVE:  i++

Substituting this into the abstract algorithm yields:

< DO_INIT >;  
int i = 0;  
while ( i < a.Length ) {  
    < DO something to a[i] >;  
    i++;  
} // end while

In Java a counter controlled loop may be written more simply using the for-statement:

< DO_INIT >;  
for (int i = 0; i < a.Length; i++) {  
    < DO something to a[i] >;  
} // end for

Having completed the algorithm on a given data representation all we have to do is to concretise the operations specific to the problem (DO_INIT and DO) according to the problem at hand.

If the problem is to count the number of zeros in the array, the concretisation could be as follows:

DO_INIT:  
int count= 0;  
DO:  
if (a[i] == 0) count++;  

Substituting this into the pattern we have the complete algorithm:

int count = 0;  
for (int i = 0; i < a.Length; i++){  
    if (a[i] == 0)  
        count++;  
} // end for

Note how this pattern applies to all algorithms that sweep arrays of integers. If, for instance, we instead want to increase all elements by one, all we have to do is to concretise DO_INIT and DO according to the new problem: DO_INIT is very easy, since no concretising is needed. The DO-operation is to increase every element in a by one. This can be done by the Java-statement a[i]++.

So the algorithm becomes:
for (int i = 0; i < a.Length; i++) {
    a[i]++;
} // end for

Please note how the operations INIT, DONE, SELECT and REMOVE only depend on the data representation, and hence loop control are separated from the actual work (the DO operation).

**Search Algorithms**

Search algorithms probe a collection of elements (called the *search collection*, $S$) for an element with some specific properties. For instance:

- Searching a collection of customers for a customer with balance greater than DKK 10,000
- Searching a collection of students for a student older than 30
- Searching for the word “algorithm” in a text.
- Searching an array of integers for a non-negative value
- Etc.

The element sought for is called the *target element* ($t$). The algorithms work by picking an element (the *candidate element*, $c$) from the part of the search domain not yet investigated (called the *candidate collection*, $CC$). The candidate element is checked: is it the element sought the search terminates, and we are done. If $c$ does not equal $t$, the candidate set is split into a part where $t$ cannot be and a new candidate set, and the search continues in the new candidate collection.

The abstract search algorithm can be formulated as follows:

```
< INITIALISE CC >;
boolean found= false;
while ( ! found && < CC ≠ Ø > ) {
    < SELECT c from CC >;
    if ( r==c )
        found = true;
    else {
        < SPLIT CC with regard to c and t >
    } // end else
} // end while
```

The realisation of the abstract operations INITIALISE, $CC ≠ Ø$, SELECT and SPLIT depends on the data representation of the collection.
Please note that in contrast to sweep algorithms search algorithms may terminate before the entire collection is inspected.

**Specification of the abstract operations**

**INITIALISE:**

- **PRE**: none
- **POST**: \( CC' = S \)

After INITIALISE, \( CC \) must equal the entire search set. Nothing may be removed before the search starts. (\( CC' \) denotes the state of \( CC \) after the operation has completed).

**CC ≠ Ø:**

- **PRE**: none
- **POST**: returns true, if the candidate set is empty, otherwise it returns false.

**SELECT:**

- **PRE**: \( CC ≠ Ø \)
- **POST**: \( c' ∈ CC \)

It is only valid to apply SELECT if \( CC \) is not empty. SELECT must return an element from the candidate set.

**SPLIT:**

- **PRE**: \( c ≠ t && CC ≠ Ø && c ∈ CC \)
- **POST**: \( c ∉ CC' && if t ∈ CC then t ∈ CC' \)

The precondition states that it is only valid to call SPLIT, if \( t \) has not been found yet, if there are more candidates and if \( c \) is actually a member of \( CC \). The post condition states that at least \( c \) must be removed from the set of candidates, and \( t \) must not be removed (if it was included in the first place). It would be fine, if SPLIT is able to remove more elements than just \( c \).

**Concretisation of the search pattern on sequences of integers**

If for instance the collection is a sequence of integers, say an array, the operations INITIALISE, \( CC ≠ Ø \), SELECT and SPLIT may be concretised by a counter, \( i \) that indicates the boundary of the candidate set, see figure 2 below:

![Diagram](https://via.placeholder.com/150)

**Figure 2**: The collection is an array and the candidate set (CC) is concretised by a counter \( i \).
Assuming the declarations:

```java
int t;
int a[];
```

and some initialisation of the array \( a \) and the target \( t \), the abstract operations can be concretised as follows:

- **INITIALISE:** \( i = 0 \)
- **CC** \( \neq \emptyset \): \( i < a\.Length \)
- **SELECT:** \( a[i] \)
- **SPLIT:** \( i++ \)

(You may want to use a minute or two showing that this realisation actually meets the specification of the abstract operations).

Substituting this into the abstract algorithm yields:

```java
int c;
int i= 0;
boolean found= false;
while ( !found && i<a\.Length )
{
    c = a[i];
    if (c == t)
        found= true;
    else
        i ++;
} // end while
```

**Binary search: A "smart" realisation of the search pattern on a sorted sequence**

Briefly, the strategy of binary search is as follows:

- Select an element in the middle of the candidate set:
- If this is the element we are looking for – we are done
- If the target comes after the middle element, then look in the upper part (remember the collection is sorted)
• If the target comes before the middle element, then look in the lower part (again remember the collection is sorted)
• Repeat this until the target has been found or there are no more candidates

The search set again being an array of integers the candidate set can be represented by two indices low and high which serve as boundaries for the part of the array that constitutes the candidate set (see figure 3 below):

```
Figure 3: The collection is an array and the candidate set (CC) is concretised by two indices.
```

Given this data representation the abstract operations may be concretised as follows:

**INITIALISE:** int low = 0; int high = a.Length-1;

**SELECT:**
```
middle = (high + low) / 2;
//compute the middle index between low and high
```
```
c = a[middle]
```

**CC ≠ Ø:** low <= high

**SPLIT:** if (c<t) low= middle + 1;
```
else high= middle – 1;
```

Please note how this realisation of SPLIT relies heavily on the precondition that the array is sorted.

The realisation of SELECT requires that the data representation provides random access to elements as for instance arrays do. Binary search is not to be applied otherwise (don’t ever use it on linked lists).

Again it should be shown that this realisation of the abstract operations meets the requirements of the specification. We will leave this as an exercise.

Again assuming the declarations:
int t;
int a[];

Substituting this into the abstract algorithm yields:

int low = 0;
int high = a.Length -1;
int c , middle;
boolean found = false;
while ( !found && low<=high ) {
    middle = (high + low) / 2;
    c= a[middle];
    if (c == t)
        found:= true;
    else
        if ( c<t )
            low = middle+1;
        else
            high= middle-1;
} // end while

SPLIT bisects the size of the candidate set. This leads to a very efficient implementation of the search. In a matter of fact the order of magnitude of the execution time for binary search is \( \log n \) where \( n \) is the size of the array to be searched. Logarithmic growth means that execution time increases very slowly. For instance if the array contains 1024 \( (2^{10}) \) elements, binary search will examine at most 10 elements (the number of times 1024 can be bisected). If the number of elements is doubled, binary search looks at only one more element. (You may want to consult your math professor in order to know why.)

**Merge Algorithms**

The Merge Pattern deals with the problem of joining two sorted sequences into one sorted sequence. The following example illustrates merging two sequences \( s \) and \( t \) into a third sequence \( r \):

Let

\[
    s = [1, 3, 6, 7, 9] \text{ and } t = [1, 2, 5, 7, 11, 17]
\]

Then

\[
    r = [1, 1, 2, 3, 5, 6, 7, 7, 9, 11, 17]
\]
The problem of merging two sequences has many occurrences:

- Updating a transaction file with transaction from another file with new transactions since last update (assuming transactions in both files are sorted on time stamp).
- Updating an inventory when new supplies arrive
- Finding all elements in both sequences (intersection). Union and set difference may also be found using the Merge Pattern.
- In MergeSort
- And many others

The Abstract Merge Algorithm

We will assume that the following operations are defined on a sequence:

\[
\begin{align*}
    s\text{.getCurrent}() & \quad \text{-- gets current element in the sequence} \\
    s\text{.hasMore}() & \quad \text{-- has current reached the end of the sequence} \\
    s\text{.reset}() & \quad \text{-- sets current to the head of the sequence} \\
    s\text{.next}() & \quad \text{-- moves current to the next element in the sequence}
\end{align*}
\]

Merging sequences \( s \) and \( t \):

\[
\begin{align*}
    &s\text{.reset}(); \ t\text{.reset}(); \\
    \text{while}(s\text{.hasMore}() \ \&\& \ t\text{.hasMore}()) \{ \\
    &\quad \text{if}(s\text{.getCurrent}() < t\text{.getCurrent}()) \{ \\
    &\quad \quad \langle \text{Move } s \rangle \\
    &\quad \quad s\text{.next}(); \\
    &\quad \} \\
    &\quad \text{else} \{ \\
    &\quad \quad \text{if}(s\text{.getCurrent}() > t\text{.getCurrent}()) \{ \\
    &\quad \quad \quad \langle \text{Move } t \rangle \\
    &\quad \quad \quad t\text{.next}(); \\
    &\quad \quad \} \\
    &\quad \text{else} \{ \\
    &\quad \quad \langle \text{Move } s,t \rangle \\
    &\quad \quad s\text{.next}(); \\
    &\quad \quad t\text{.next}(); \\
    &\quad \} \\
    &\quad \} \\
    &\quad \text{if}(s\text{.hasMore}()) \langle \text{MoveTail } s \rangle \ \text{else} \langle \text{MoveTail } t \rangle
\end{align*}
\]
The abstract operations <Move s>, <Move t>, <Move st>, <MoveTail s> and <MoveTail t> depends on problem at hand. If we for instance want to do the total merge of s and t into r as in the opening example, then the move operations may be concretised as follows:

- <Move s> -- add s.getCurrent() to r
- <Move t> -- add t.getCurrent() to r
- <Move st> -- add s.getCurrent() and t.getCurrent() to r
- <MoveTail s> -- add the rest of s to r
- <MoveTail t> -- add the rest of t to r

**Example: Total Merge of two Arrays of Integers**

If the sequences are arrays of integers the following code fragment (in C#) shows the total merge of two arrays:

```csharp
public static int[] Merge(int[] s, int[] t)
{
    int i = 0; // index in s
    int j = 0; // index in t
    int k = 0; // index in r
    int[] r = new int[s.Length + t.Length];
    while (i < s.Length && j < t.Length)
    {
        if (s[i] < t[j]) // <Move s>
        {
            r[k] = s[i];
            i++; k++;
        }
        else
        {
            if (s[i] > t[j]) // <Move t>
            {
                r[k] = t[j];
                j++; k++;
            }
            else // <Move st>
            {
                r[k] = s[i];
                i++; k++;
                r[k] = t[j];
                j++; k++;
            }
        }
    }
    while (i < s.Length) // <MoveTail s>
    {
        r[k] = s[i];
        i++; k++;
    }
    while (j < t.Length) // <MoveTail t>
    {
        r[k] = t[j];
        j++; k++;
    }
    return r;
}
```
It is left as an exercise to give a precise formulation of the concretesation of the abstract operations in the above example.