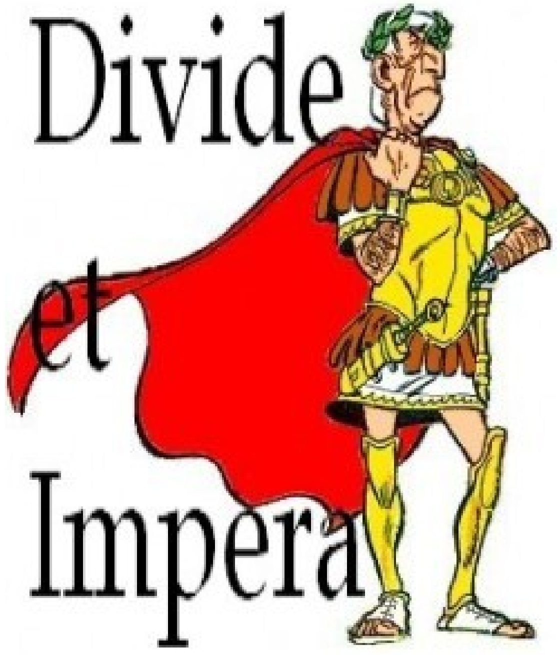
Abstractions

Giorgio Brajnik

giorgio.brajnik@uniud.it

Ott 2021



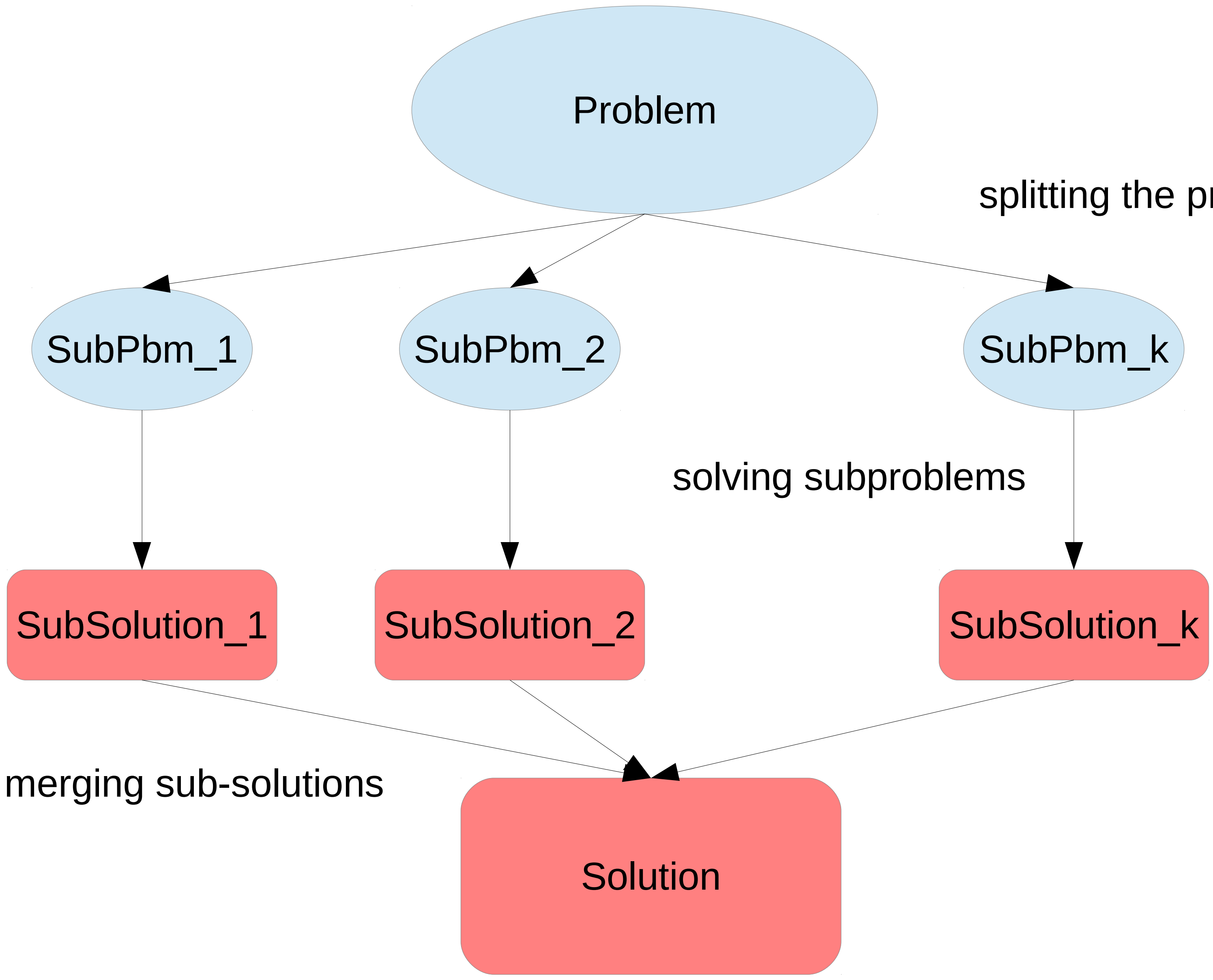
In politics and sociology, divide and rule (or divide and conquer) is gaining and maintaining power by **breaking up larger concentrations of power** into pieces that individually have less power than the one implementing the strategy.
The concept refers to a strategy that **breaks up existing power structures** and **prevents smaller power groups from linking up**.

(Wikipedia, Sept 2015)

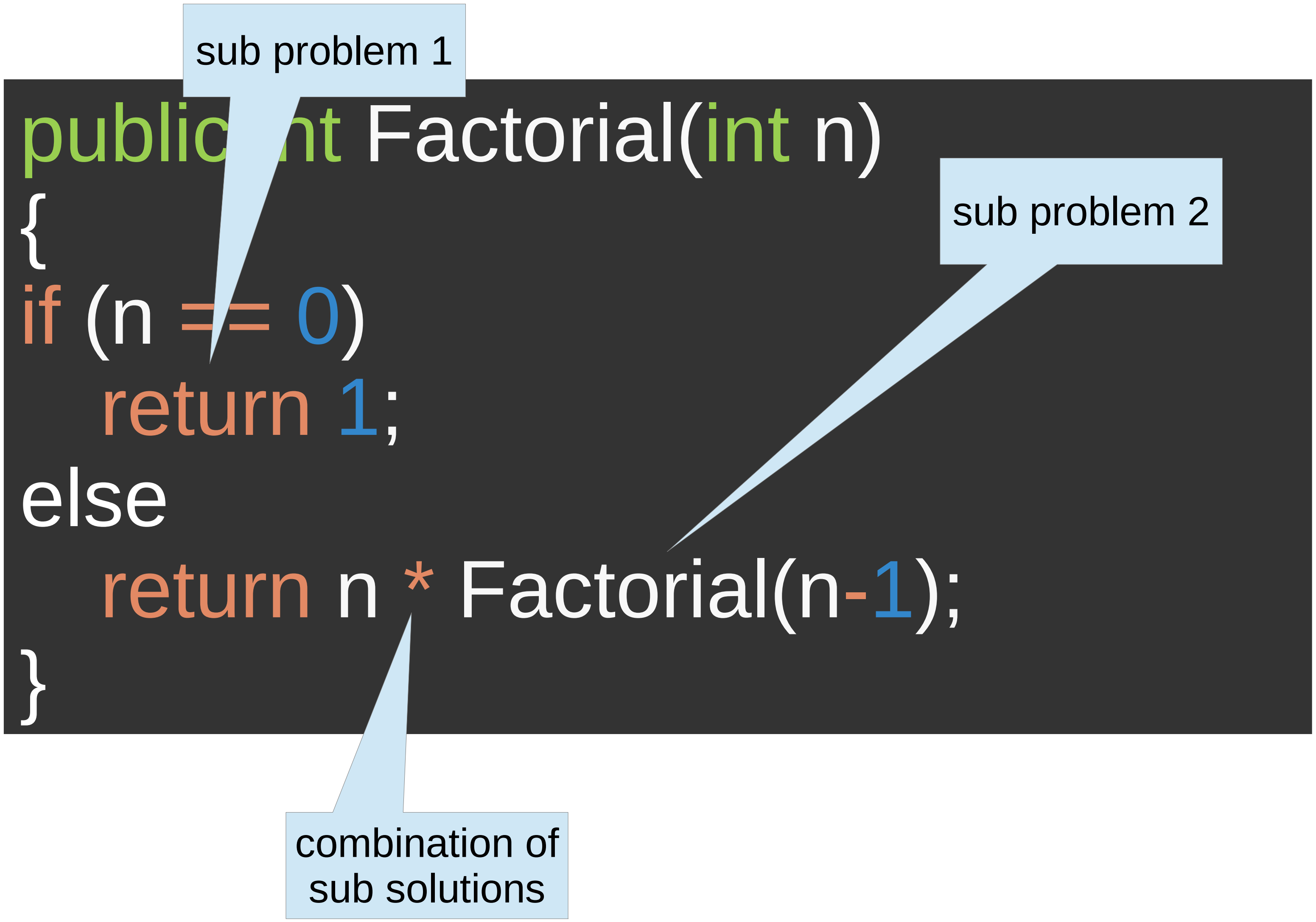
Decomposition

- Goal:
  - to create modules that **interact** with one another in **simple**, **well-defined** ways
- Decomposing a problem:
  - to factor it into separable subproblems so that:
    - same level of detail
    - can be solved independently
    - solutions of subpbms can be combined into a solution of the original pbm

Decomposition



Decomposition example



```
public int Factorial(int n)
{
if (n == 0)
    return 1;
else
    return n * Factorial(n-1);
}
```

Not always an easy task

In the context of a system for supporting logistics: shipping, journeys, cargos, payloads, ships, air-cargos, trucks, customers, delivery, routes, ...

What happens when a new concept is added: **Discounts?**

- who deals with it?
- who talks with whom?
- when?

Not always an easy task

In the context of a user interface of a complex interactive system

Can several sub-solutions be merged effectively?

- different look & feel?
- inconsistencies: different names for similar operations
- dependencies of Views wrt Model/Controller

Abstraction

- ignoring certain details with the aim of **simplifying** the original problem

- **abstraction** as a way to decompose a problem
- it assists in making a good choice of subproblems/components

Abstractions?

```
public class Sort {
    static int[] arr1 = {10,34,2,56,7,67,88,42};
    static int temp;

    public static void main(String a[]){
        for (int i = 1; i < arr1.length; i++) {
            for(int j = i ; j > 0 ; j--){
                if(arr1[j] < arr1[j-1]){
                    temp = arr1[j];
                    arr1[j] = arr1[j-1];
                    arr1[j-1] = temp;
                }
            }
        }
        for(int i = 0; i < arr1.length; i++){
            System.out.print(arr1[i]);
            System.out.print(", ");
        }
    }
```

Two fundamental mechanisms

- abstraction by **parameterization**
  - generalization
  - reuse of code with different data
- abstraction by **specification**
  - removal of implementation details (how-to)
  - definition of a *contract*:
    - I promise you something
    - if you give me something else

Parameterization

```
public class MyInsertionSort {

    public static void main(String a[]){
    int[] arr1 = {10,34,2,56,7,67,88,42};
       insertionSort(arr1);
         for(int i = 0; i < arr1.length; i++){
            System.out.print(arr1[i]);
            System.out.print(", ");
        }
    }

    private static void insertionSort(int[] a) {
        int temp;
        for (int i = 1; i < a.length; i++) {
            for(int j = i ; j > 0 ; j--){
                if(a[j] < a[j-1]){
                    temp = a[j];
                    a[j] = a[j-1];
                    a[j-1] = temp;
                }
            }
        }
    }
```

Why parameterizing?

Reuse

Specification

```
public static void main(String a[]){
    int[] source = {10,34,2,56,7,67,88,42};
    int[] results = doInsertionSort(source);
 ...
    /**
     * MODIFY the array a so that values are ordered, increasing
     * @param a items to be sorted, increasingly
     * @return the modified array
     */
    private static int[] DoInsertionSort(int[] a) {
        for (int i = 1; i < a.length; i++) {
            for(int j = i ; j > 0 ; j--){
                if(a[j] < a[j-1]){
                    swap(a, j);
                }
            }
        }
        return a;// BAD DECISION!
    }
    /**
     * swap a[j] with a[j-1]; MODIFY the array a
     * @param a, REQUIRED to have 2 or more elements
     * @param j an index of the array, REQUIRED to be a valid index and > 0.
     */
    private static void swap(int[] a, int j) {// CAN BE IMPROVED
        int temp;
        temp = a[j];
        a[j] = a[j-1];
        a[j-1] = temp;
    }
```

Specification

- precondition
  - REQUIRED
    - it implies a partial function
- postcondition
  - RETURN
  - MODIFY

NB
It is a contract definition

Specification

Meaning:

if

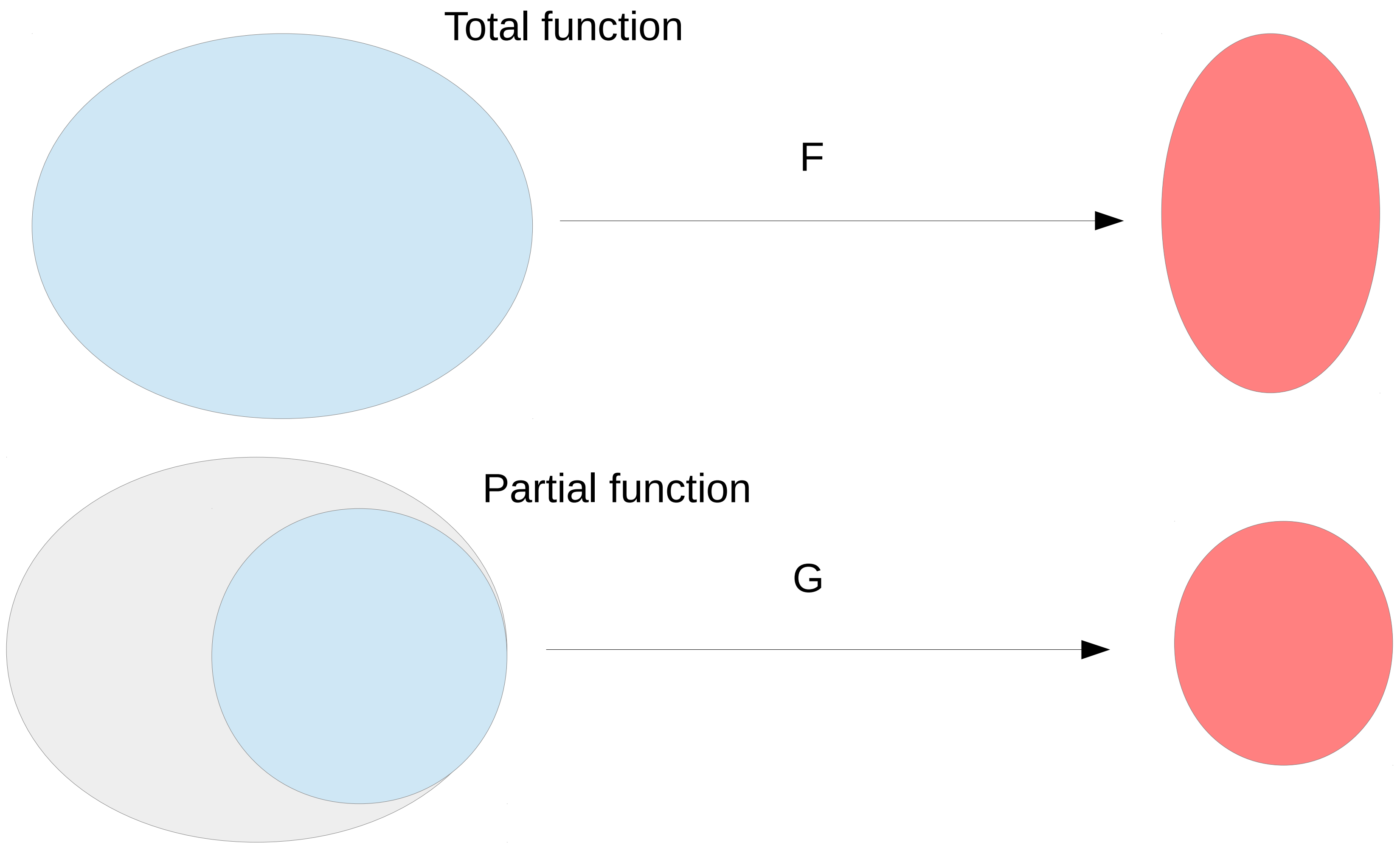
precondition is true

then

postcondition is guaranteed (after executing the method)

- NB: when precondition=false then anything can happen

Partial functions

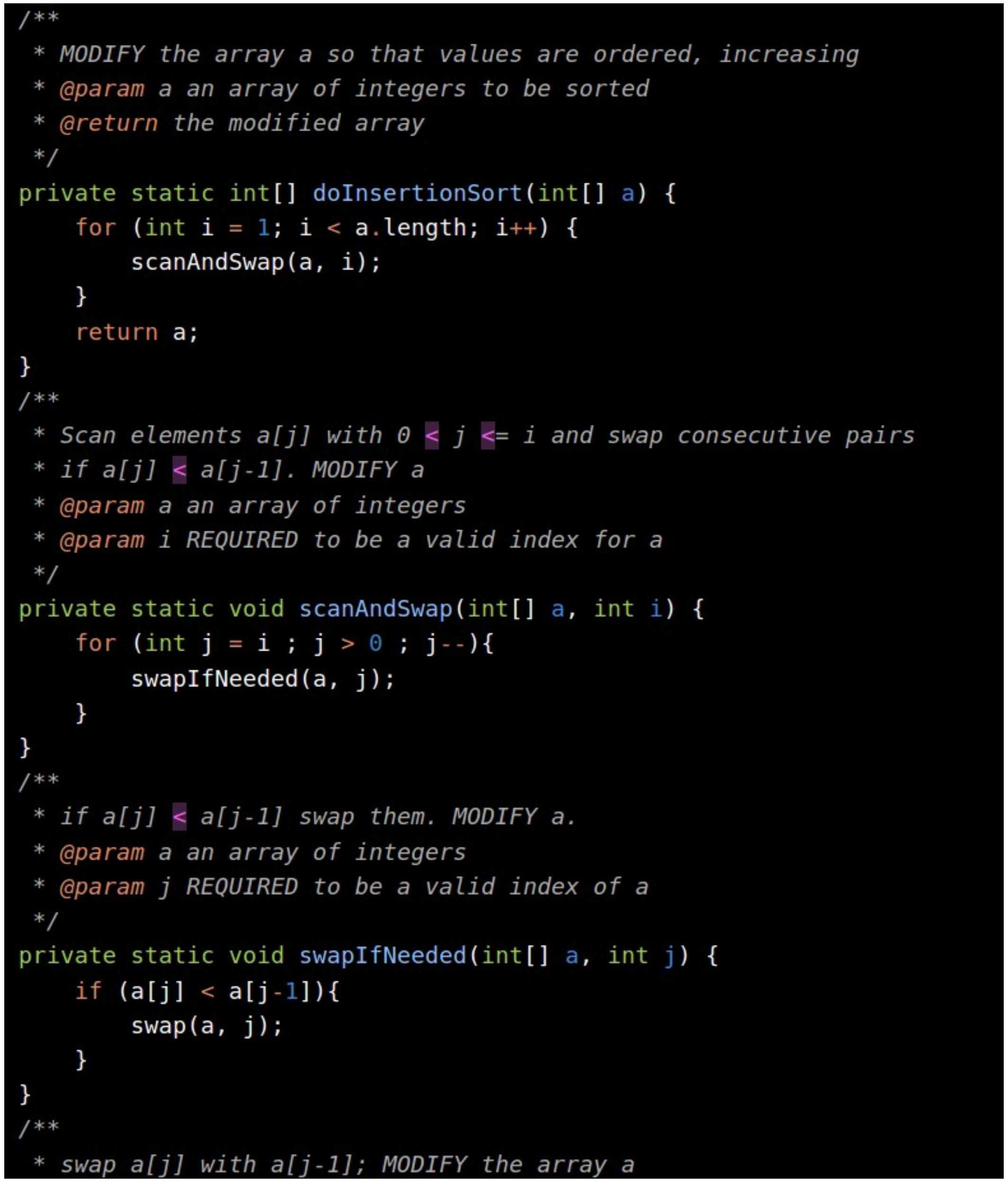


Specification

```
/**
 * swap a[j] with a[j-1]; MODIFY the array a
 * @param a, REQUIRED to have 2 or more elements
 * @param j an index of the array, REQUIRED to be a valid index and > 0.
 */
private static void swap(int[] a, int j) {
 ...
}
```

We can forget details

Even more abstraction



```
/**
 * MODIFY the array a so that values are ordered, increasing
 * @param a an array of integers to be sorted
 * @return the modified array
 */
private static int[] doInsertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        scanAndSwap(a, i);
    }
    return a;
}
/**
 * Scan elements a[j] with 0 < j <= i and swap consecutive pairs
 * if a[j] < a[j-1]. MODIFY a
 * @param a an array of integers
 * @param i REQUIRED to be a valid index for a
 */
private static void scanAndSwap(int[] a, int i) {
    for (int j = i ; j > 0 ; j--){
        swapIfNeeded(a, j);
    }
}
/**
 * if a[j] < a[j-1] swap them. MODIFY a.
 * @param a an array of integers
 * @param j REQUIRED to be a valid index of a
 */
private static void swapIfNeeded(int[] a, int j) {
    if (a[j] < a[j-1]){
        swap(a, j);
    }
}
/**
 * swap a[j] with a[j-1]; MODIFY the array a
```

Even better:

Functions should do one thing.
They should do it well.
The should do only that.

Decomposition vs specification

- decomposition/procedural abstraction
  - we do not **necessarily** hide details
  - we don't have a contract
- abstr. by specification
  - we do decompose
  - we intend to hide details
  - we specify a contract

Procedural abstraction

- we define procedures/functions
  - to extend the programming language with new **operations**
  - eg. swap(int i, int j)
- BUT
  1) they do not hide implementation details
  2) they might be interdependent but this is not clear

Data abstraction

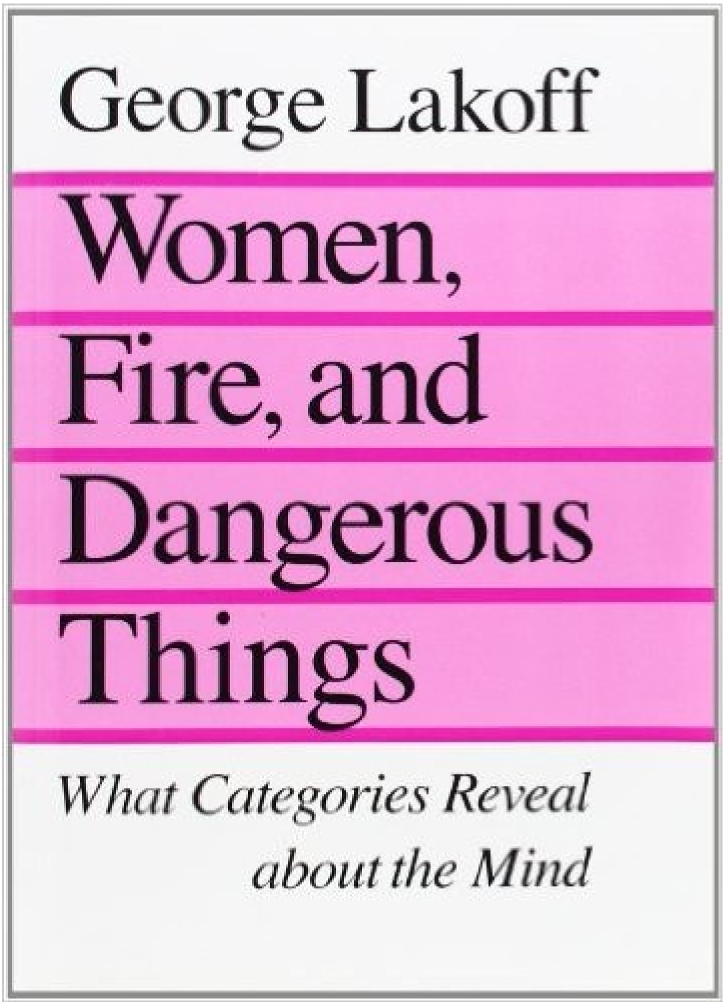
- we extend the programming languag
  - new operations
  - that do hide implementation
  - that are coordinated
- new data type
  - set of objects + set of operations
- Example: Stack
  - pop(push(s,x))=<x,s>
  - peek(push(s,x))=x
  - push({},x)={x}
  - ...

Abstract Data Type

NB
Each specification deals with 2 or more operations

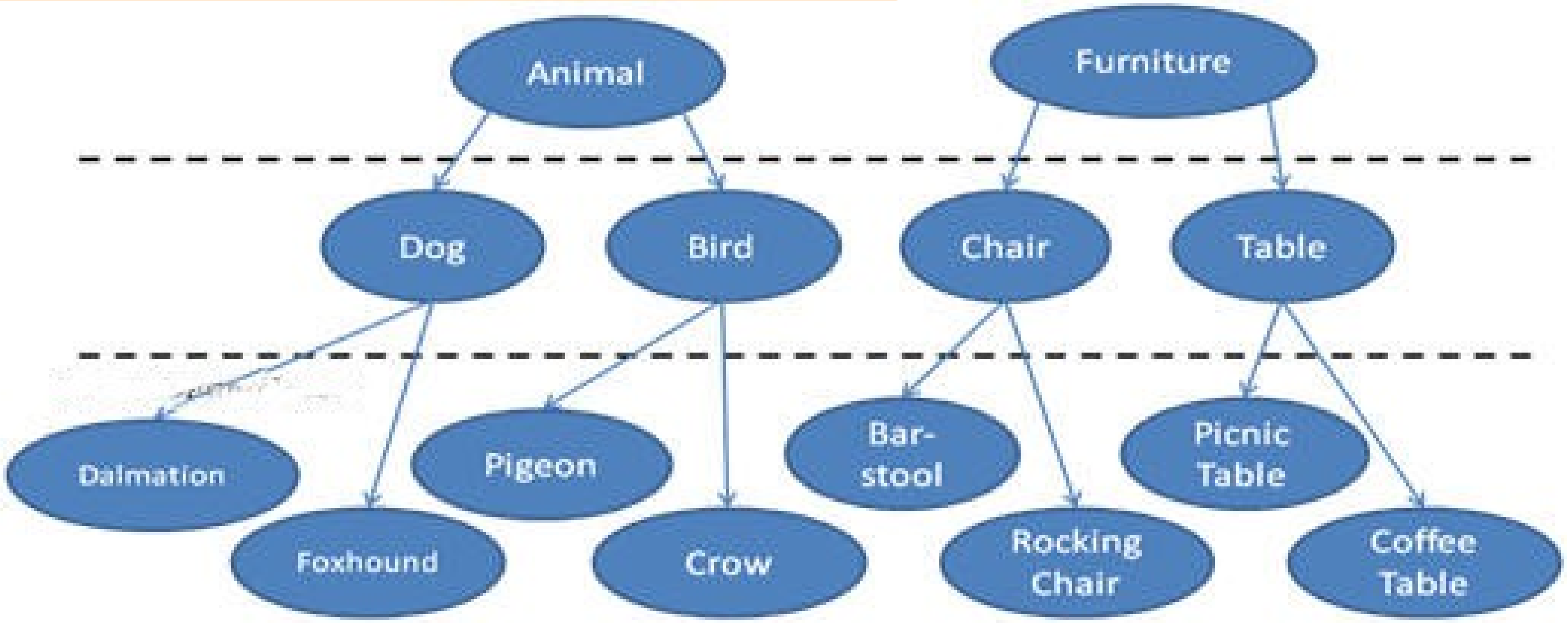
Categorization

- Category
  - set of things that share some characteristics
- Classification criterion
  - rules to decide what is in and what is out
- Result:
  - a tree or a DAG (Directed Acyclic Graph)


George Lakoff
Women, Fire, and Dangerous Things
What Categories Reveal about the Mind

Categorization

Benefit?
- inheritance of attributes
- inheritance of operations



Iteration abstraction

- how to process a collection of items
  - array, list, ordered list, set, hash table, tree, …
- WITHOUT revealing details of the implementation

```
…    StudentIterator si = createStudentIterator(PrOrOg1516);
     while (si.hasNext()){
         Student s = si.next();
         s.assignGrade(ItalianGrades.trentaELode);
     }
```

Conclusion

- Object Oriented Programming
  - it supports abstraction by specification
    - ADT
    - Categorizations
    - Iterators
  - it supports generalization
  - they are orthogonal mechanisms
- their combination = very powerful means to control complexity