CentraleSupélec

IS1220 - Object Oriented Software Design

Tutorial: sorting arrays and collections

Paolo Ballarini

Exercise 1. Test Driven Development - factorial method

Q1) Following the TDD approach define a class which contains a `factorial` method which should be defined according to the following requirements:

- `factorial` takes an integer argument and return the value of the factorial $n!$
- the `factorial` should return a value of type `int`, hence represented over 32bits which means the input argument n must be $n \leq 10$ (as 11! cannot be stored as a 32bit *int* variable).

Exercise 2. Test Driven Development - calculator with String argument

Q1) Following the TDD approach define a class StringCalculator which contains an `add` method which should be defined according to the following requirements:

- The method `add` can take 0, 1 or 2 numbers, and will return their sum (for an empty string it will return 0) for example "" or "1" or "1,2"
- Allow the `add` method to handle an unknown amount of numbers
- Allow the `add` method to handle new lines between numbers (instead of commas).
- The following input is ok: "1\n2,3" (will equal 6)
- Support different delimiters
- To change a delimiter, the beginning of the string will contain a separate line that looks like this: "[delimiter]\n[numbers]" for example ";\n1;2" should return three where the default delimiter is ; .
- The first line is optional. All existing scenarios should still be supported

Exercise 3. Visitor and Composite Design Pattern

Description. Consider the following computer architecture:

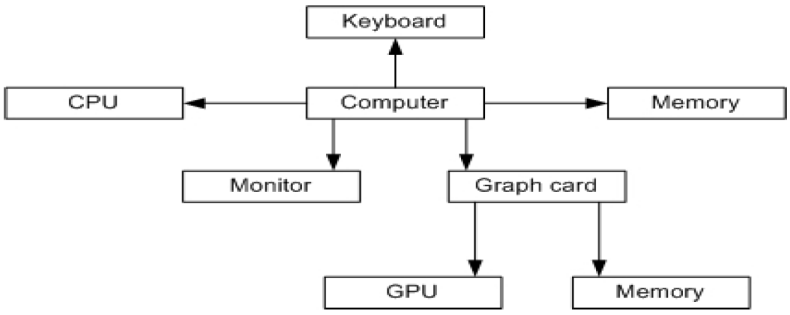


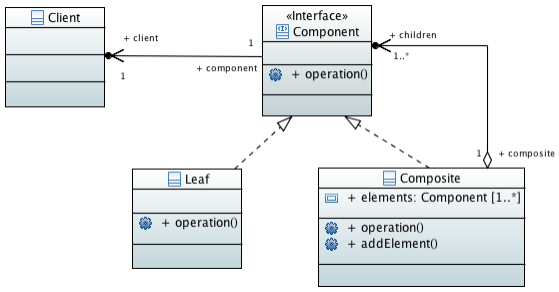
Figure 1: A computer architecture's components

In the graph, CPU, GPU, memory, monitor and keyboard are atomic components, while graph card and computer are composites. Every atomic component has a price associated with it. The price of a composite part is the sum of all its components. For example, the price of the computer is the sum of the price of the monitor, CPU, keyboard, memory and the graph card, and the price of the graph card is the sum of the price of the GPU and the memory.

Q1) Use composite pattern to model the computer architecture. Equip the Composite pattern with a `display()` method which allows for displaying the state of each node in a Composite (tree) object. Then write a visitor using the visitor pattern to calculate the price of all the memory components (the price of other kinds of components are ignored) in computer architecture.

Composite pattern
The composite pattern allows a client program to perform an operation on a hierarchical collection of "primitive" and "composite" objects transparently, that is, without having to distinguish whether an object is primitive (a leaf of a tree-structure) or a composite (an internal node of a tree-structure).

The UML class-diagram of the composite pattern is as follows:



where `operation()` is the operation the client has to perform on object of the composite (being them composite or primitive). Let us look at an example of application of the composite pattern for representing the personnel of a company where we have to distinguish between *managers* (composite nodes) and *developers* (leaf nodes). This boils down to declaration of an `Employee` interface realised by a `Developer` and a `Manager` class.

```
public interface Employee {
    public void print();
}
```

The operation that has to be *transparent* is `print()` whose goal is to display the name and salary values for each employee independent of its position in the company hierarchy (i.e. independently of whether the employee is a developer or a manaager).

```
public class Manager implements Employee{

 private String name;
 private double salary;
 private List<Employee> employees = new ArrayList<Employee>();

 public Manager(String name,double salary){
  this.name = name;
  this.salary = salary;
 }


 public void add(Employee employee) { employees.add(employee); }
 public void remove(Employee employee) {employees.remove(employee);}
 public String getName() { return name; }
 public double getSalary() {return salary; }

 public void print() {
  System.out.println("----------------");
  System.out.println("Name ="+getName());
  System.out.println("Salary ="+getSalary());
  System.out.println("----------------");
  for(Employee e: employees)
    e.print();
 }
```

```
}
```

```
public class Developer implements Employee{

  private String name;
  private double salary;

  public Developer(String name,double salary){
    this.name = name;
    this.salary = salary;
  }

  public String getName() {return name;}
  public double getSalary() {return salary;}

  public void print() {
    System.out.println("---------------");
    System.out.println("Name ="+getName());
    System.out.println("Salary ="+getSalary());
    System.out.println("---------------");
  }
}
```

```
public class MyCompany {

 public static void main(String[] args) {
  Employee emp1=new Developer("John", 10000);
  Employee emp2=new Developer("David", 15000);
  Employee manager1=new Manager("Daniel",25000);
  manager1.add(emp1);
  manager1.add(emp2);
  Employee emp3=new Developer("Michael", 20000);
  Manager generalManager=new Manager("Mark", 50000);
  generalManager.add(emp3);
  generalManager.add(manager1);
  generalManager.print();
 }
}
```

Reminder.

- **Composite pattern**. Composite is a pattern used for representing tree-like relationship between classes (can you identify a tree-like structure in diagram of Figure 2). In practice a composite pattern is realised by defining a class called for example `Node` which contains a collection of `Node` elements representing the successors of the node.

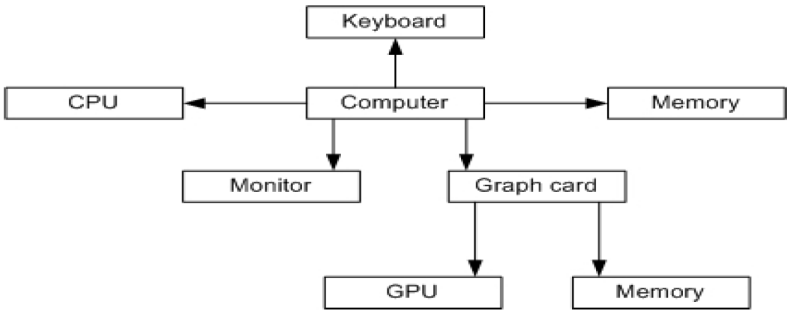


Figure 2: A computer architecture's components

- **Visitor pattern**. Visitor is a pattern used for defining specific implementation of an operation to be applied to object of different type. The implementation of the actual `visit()` operation depends on the

Q2) Import the solution file `Tutorial8new_solution.zip` available on Claroline. It contains a possible solution to the task given in Q1. The client program is given in file `Computer.java` where a composite ComputerComponent object is created according to the architecture of Figure 2 while the price of the computer is computed through a `ComputerPriceVisitor` object which indeed matches the requirement (only the price of memory components is accounted for). Extend such solution in the following manner:

- add a new implementation of a "pricing visitor" (called `ComputerPriceVisitor2`) which accounts for the price of every component in the computer architecture (not only Memory components)
- add a new implementation of a "pricing visitor" (called `ComputerPriceVisitor3`) which accounts for the price of every component and also the price of assembling each composite component.
- in `Computer.java` recalculate the price of the computer using both newly defined pricing visitors (i.e. `ComputerPriceVisitor2` and `ComputerPriceVisitor3`).