Software Engineering Course

Chapter 8

Design patterns and their place in the development process

2022-2023

Dr. Kouah Sofia.

Plan

- Introduction to design patterns
- Concept of pattern design
- Description of a design pattern

Introduction to design patterns

Challenge of object-oriented programming

Minimize interdependencies (Coupling) and maximize cohesion

Objects (and modules) are building blocks for building these architectures since:

✓ They help maximizing cohesion by encapsulating their state.

✓ Abstract classes (and inheritance) help minimizing interdependencies by controlling the level of detail of services visible to clients of an object.

Today's topic: How should these bricks be arranged?

Introduction to design patterns

Two main difficulties of object-oriented programming :

✓The design of **reusable components** is a delicate activity.

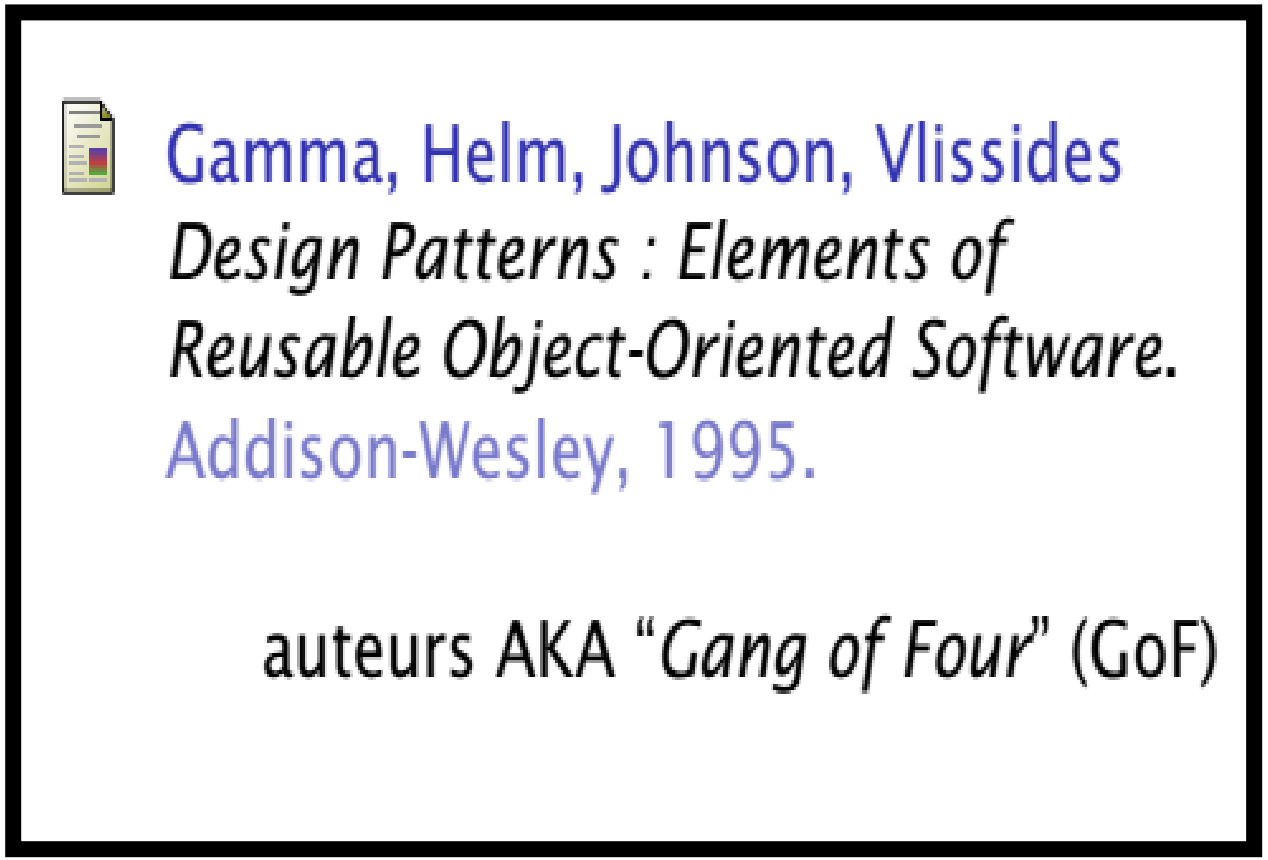
✓Object-oriented programming **has intrinsic problems**.

Introduction to design patterns

- In design, it is not uncommon to face a **problem that has already been encountered and solved by other people.**
- Reusing the solutions found by these other people **saves not only time but also quality**, provided that these solutions have been widely distributed and corrected for any errors.
- To make this idea concrete, **E. Gamma** defined the concept of **design pattern**.
- A **design pattern** is a proven solution that solves a very well identified design problem. Note that a design pattern is a **<problem/solution> pair**.
- The solution defined by a design **pattern is only interesting if we face the same problem as the one addressed by the pattern**. We must not, in any case, want to apply the solutions defined by the design patterns if we do not encounter the problems.

Introduction to design patterns

- Design Pattern Reference



Concept of pattern design

- ✓ A **Design Pattern** is a solution to a recurring problem in designing object-oriented applications.
- ✓ A **design pattern** then describes the proven solution to solve this software architecture problem. As a recurring problem we find for example the design of an application where it will be easy to add functionalities to a class without modifying it.
- ✓ Note that when **placed at the design level**, Design Patterns **<u>are independent of the programming languages used.</u>**

Definition of Design pattern
Abstract technique for solving a recurring problem

Description of a design pattern

The **description of a design pattern** follows a fixed formalism:

- ✓ **Name**
- ✓ **Problem** description of the problem to be solved
- ✓ **Solution**: the elements of the solution, with their relationships. The solution is called design pattern.
- ✓ **Consequences results:** resulting from the solution.
- ✓ **Other** known implementations, usages, etc.

Advantages of design patterns

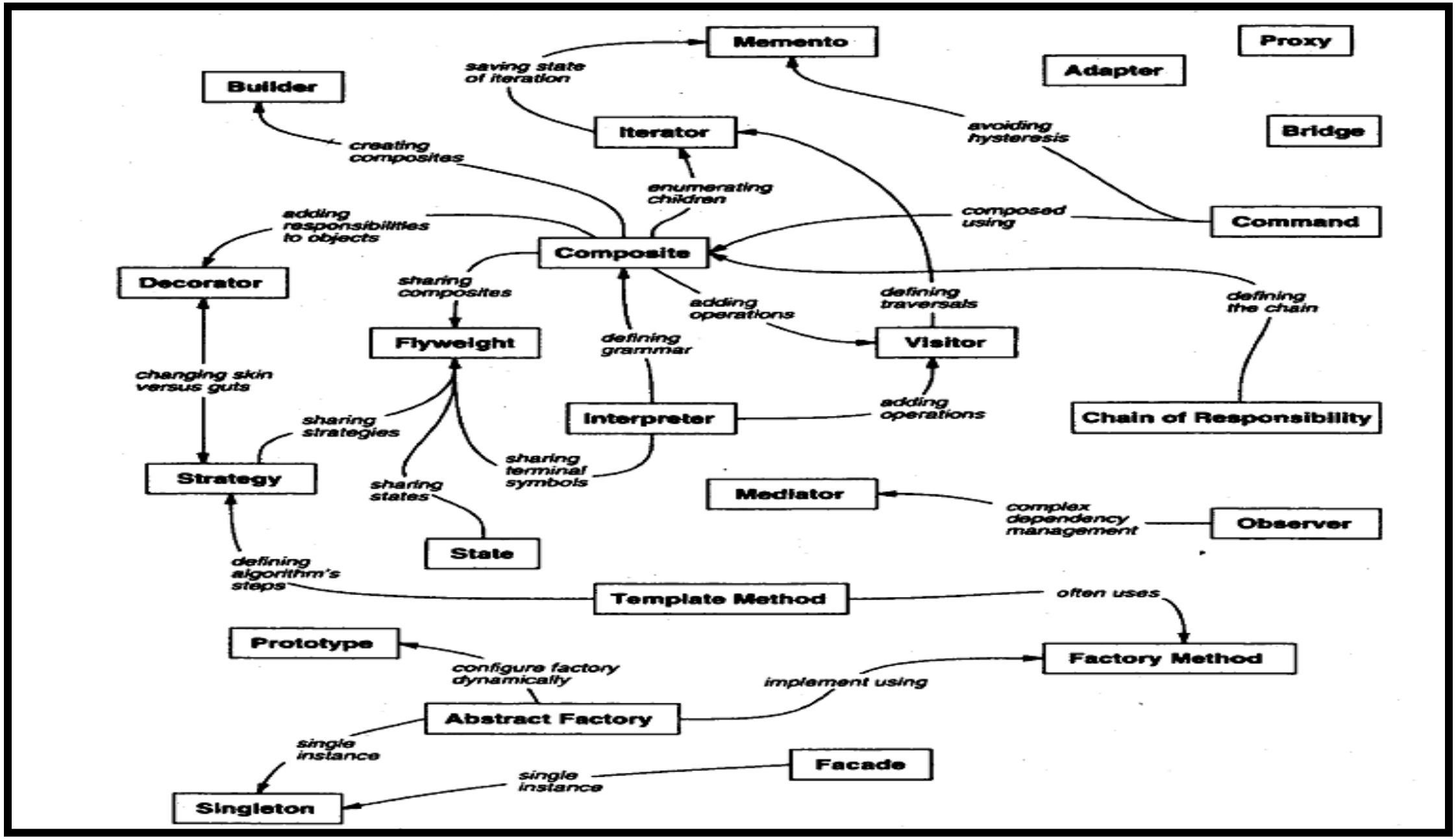
- ✓Design pattern → **Reuse**
- ✓It is easier to reuse a design solution rather than code.
- ✓Solution to:
  - ✓Describe good design practices.
  - ✓Transmit knowledge acquired through experience.
  - ✓So **don't reinvent the wheel**.
- ✓Facilitates communication between developers.

Design Pattern Categories

GoF have explained **three main classes** of patterns in their book, each one specializing in:

- ✓ **Creation Of Objects (Creational Patterns)**
- ✓ **Structure Of Relationships Between Objects (Structural Patterns)**
- ✓ **Behavior Of Objects (Behavioral Patterns**)

Design Pattern Categories



Design Pattern Scope

	Creational	Structural	Behavioral
Class	Factory	Adaptor	Interpreter Template
Object	Abstract Factory Builder Prototype Singleton	Adaptor Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Creational patterns

5 creational patterns are: Factory, AbstractFactory, Builder, Prototype, Singleton

✓**Abstraction of the creation process**.

✓**Encapsulation** of creation logic.

✓We do not **know in advance what will be created or how** it will be created.

Structural Patterns

7 structural patterns: Adapter, Bridge, Composite, Decorator, Interface, Flyweight, Proxy

- ✓**How the objects are assembled**.
- ✓**Decouple the interface from the implementation**.

Behavioral Patterns

11 behavioral patterns are: Template method, Chain of responsibility,

Command, Iterate, Mediator, Memento, Observator, State, Strategy, Visitor

✓**Communication mode between objects**

General Principle

- ✓We talk about **white-box reuse** when we **reuse code** through **inheritance**
  - ✓is dangerous, because subclasses are allowed to break encapsulation
  - ✓ is not always possible (e.g. proprietary code whose internal documentation is not available)
- ✓We speak of **black-box reuse** when we reuse code through **object composition** (association/composition/aggregation)
  - ✓is dynamic, so the compiler and the programming language do not help to impose well-formed constraints. It is more difficult to reason about software
- ✓Objects **talk to each other only through interfaces**, so encapsulation is saved.
- ✓**General principle**: Favor object composition **over** class inhereitance.

General Principle

- ✓Description of a solution to a **general and recurring design problem** in a particular context
- ✓**Description of communicating objects and classes**
- ✓**Independent of an application** or specific
- ✓Some patterns are related to **concurrency**, **distributed programming**, **real time**
- ✓All patterns aim to **reinforce cohesion and decrease coupling**

Example

Example motivating pattern designs

- ✓ **Example**: [from UML 2 by practice, P. Rocques]
- ✓ **Statement**: propose an elegant solution which makes it possible to model the following **file management system**:
  - ✓ files, shortcuts and directories are contained in directories and have a name.
  - ✓ a shortcut can concern a file or a directory.
  - ✓ within a given directory, a name can identify only one element (file, sub-directory or shortcut).

Example

Example motivating pattern designs

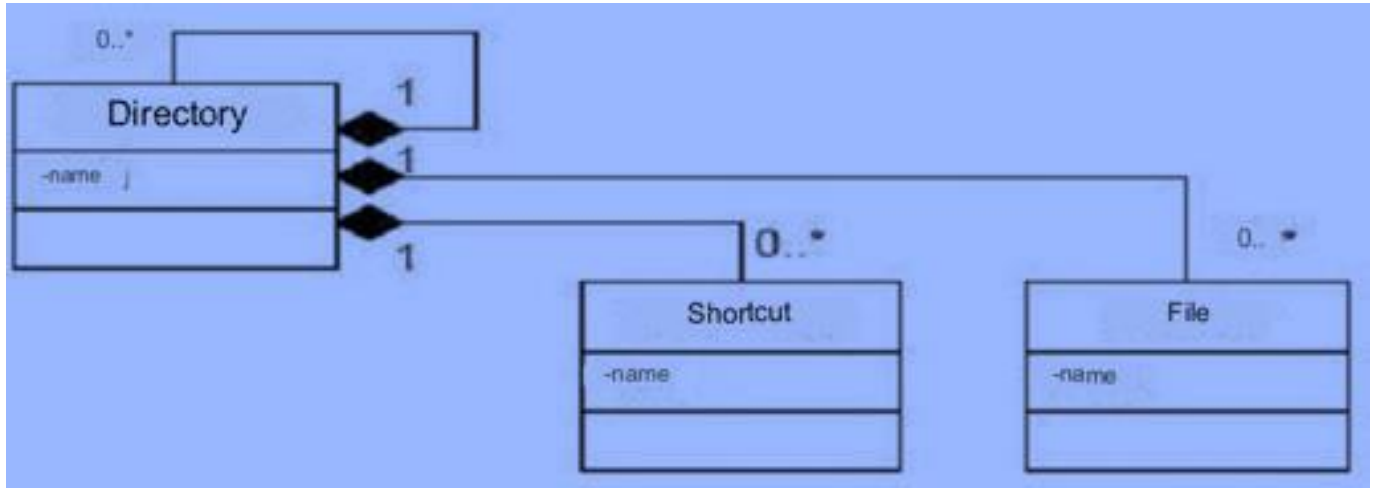
1st sentence

✓3 concepts = 3 classes

✓Inclusion is modeled by a composition.

✓Containing side multiplicity is equal to 1

✓The destruction of the directory involves the destruction of everything it contains.



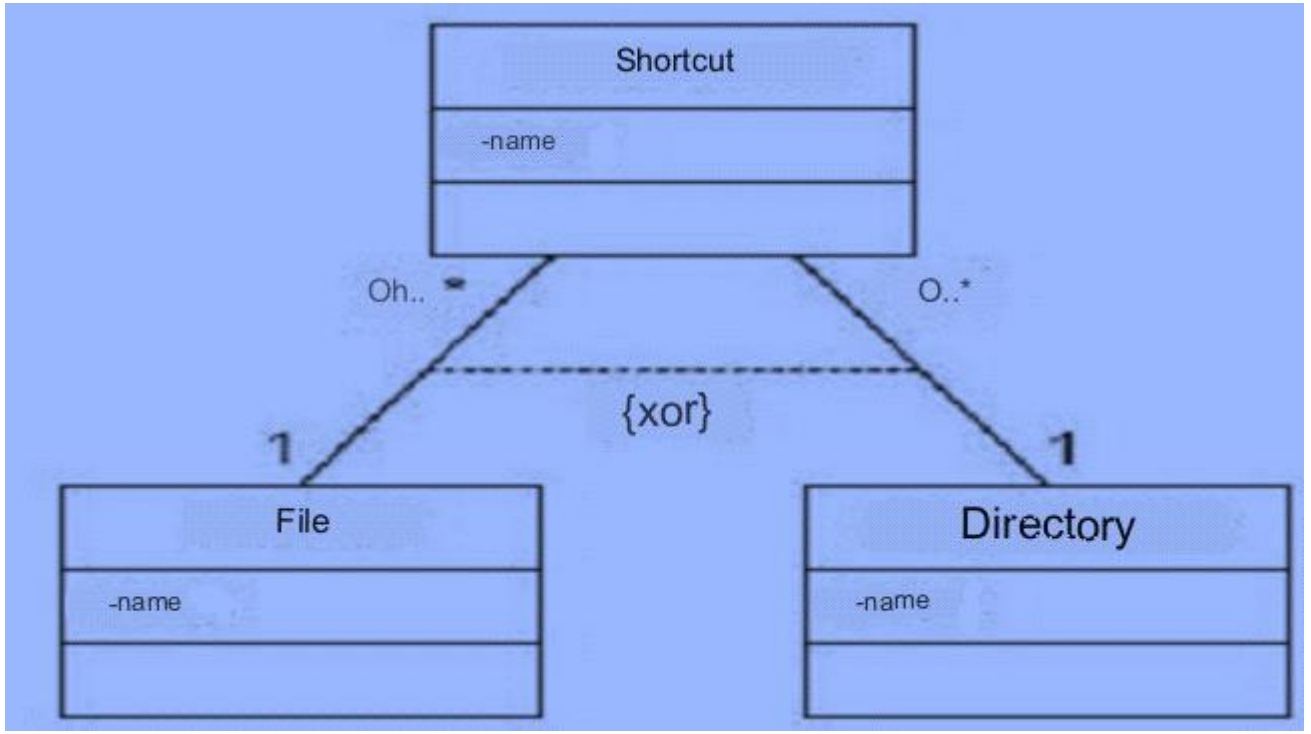
Exemple

Example motivating pattern designs

2nd sentence

✓ a shortcut can concern a file or a directory

✓ 2 associations in mutual exclusion

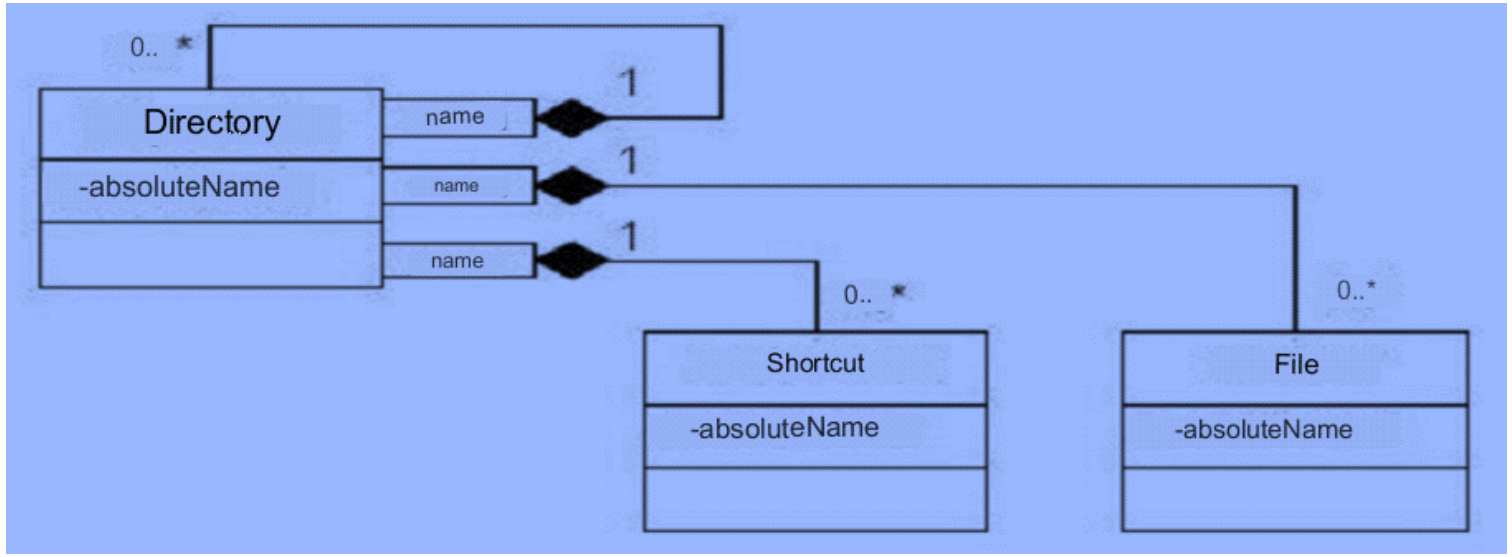


Example

Example motivating pattern designs

3rd sentence

- ✓Within a given directory, a name can identify only one element (file, subdirectory or shortcut)
- ✓**Idea** : **qualify** each of the three compositions with a name attribute



Example

Example motivating pattern designs

Problem: nothing prevents a file and a shortcut from having the same name

- ✓3 compositions and 3 names are not such a good idea.
- ✓a unique qualifier is required for each type of item contained in a directory.

Solution: importance of the element term.

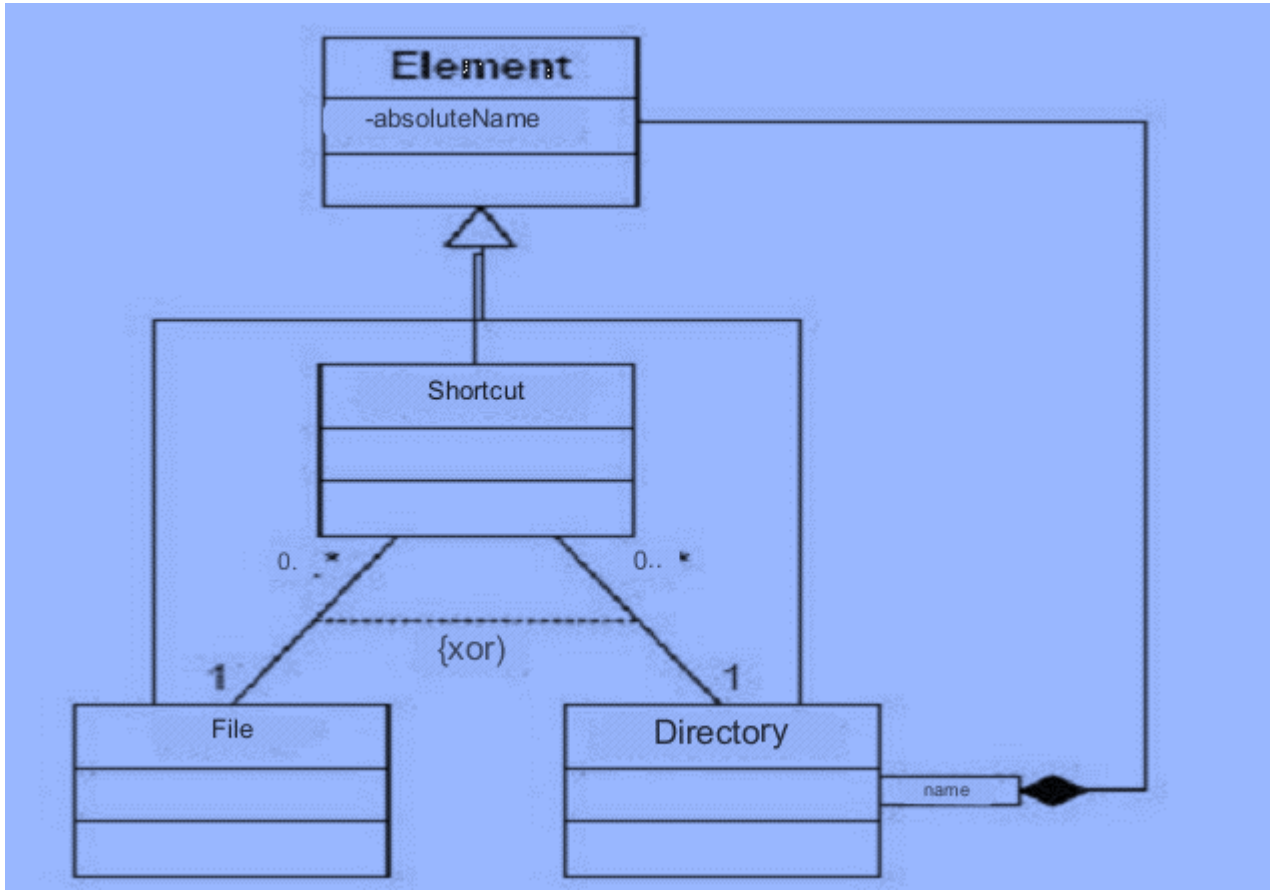
- ✓modify the model so that you only have one composition to qualify.

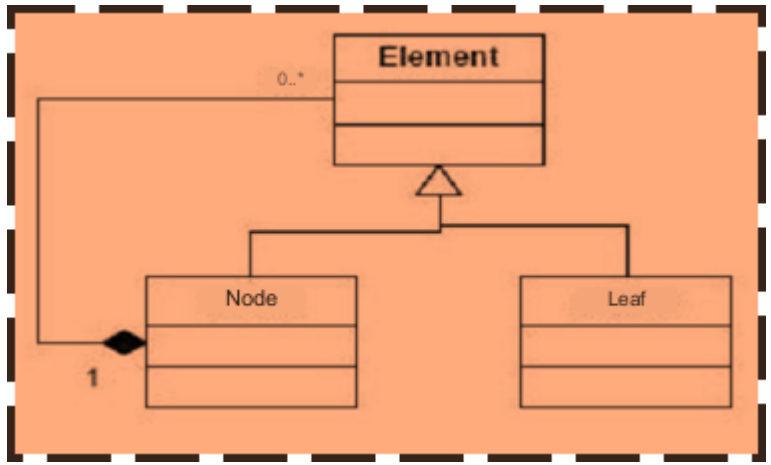
Example

Example motivating pattern designs

Solution:

- ✓ double asymmetric relationship between Directory and Element Directory contains Elements
- ✓ Directory is Element



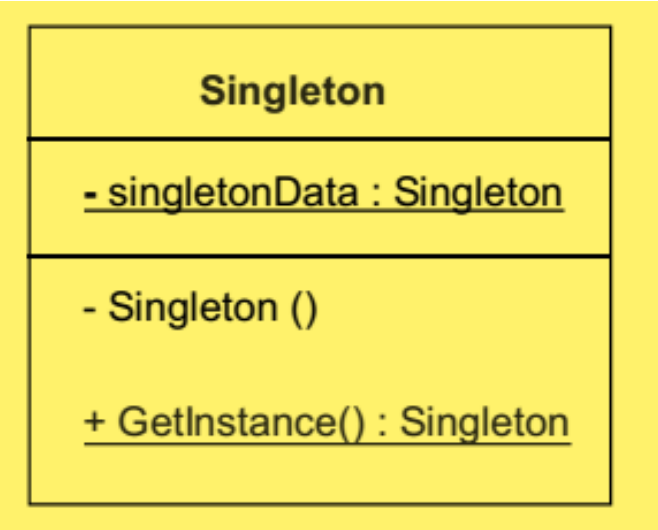


Composite Pattern

Presentation of some Design Patterns

Singleton Pattern

- ✓**Problem**: Ensuring that a class has only one instance.
- ✓**Solution**:
  - ✓The **constructor is private**
  - ✓A **method only creates an instance** if one does not already exist



- ✓The purpose of the **SINGLETON** pattern is to ensure that a class has only one instance and to provide a global access point to it.

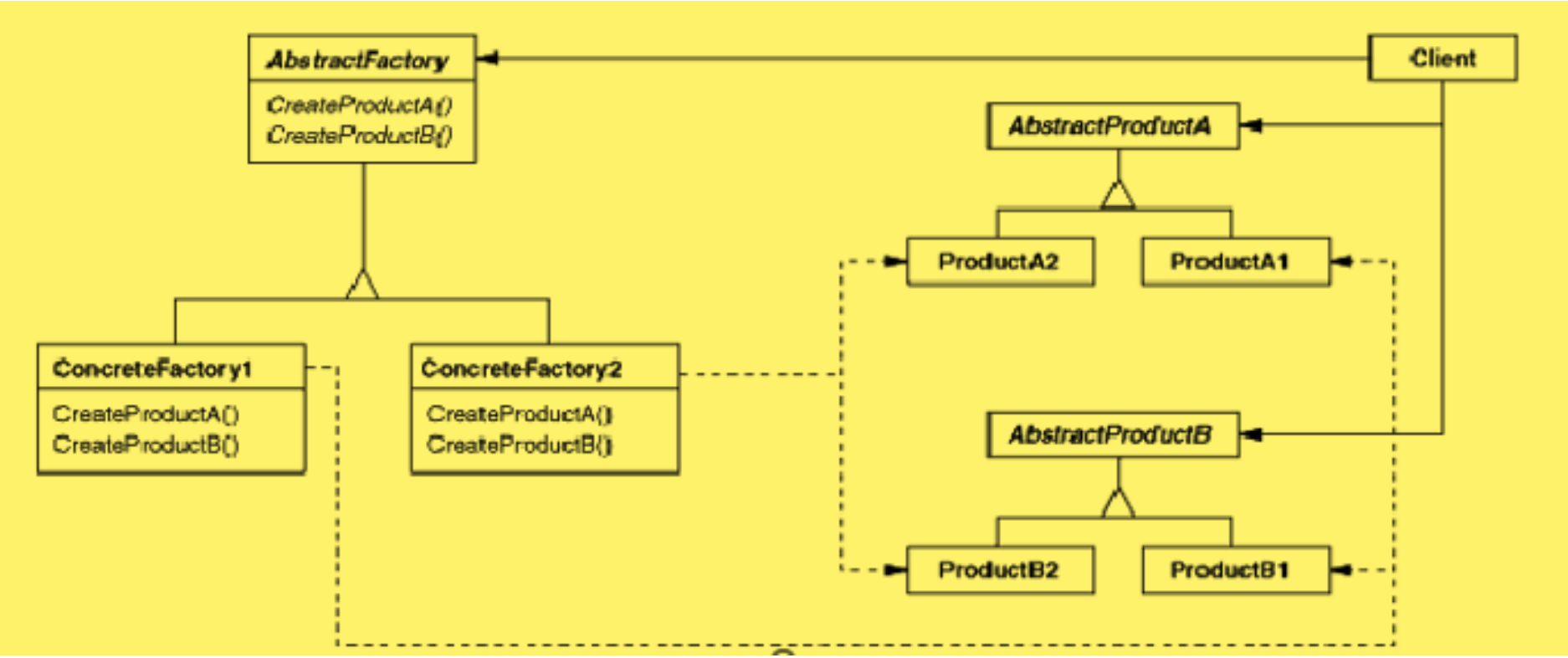
Singleton Pattern

Java implementation

```
// Only one object of this class can be created
class Singleton {
   private static Singleton instance = null;

   private Singleton() {
      ...
   }

   public static Singleton getInstance() {
      if (instance == null)
         instance = new Singleton();
      return instance;
   }
   ...
}

class Program {
   public void aMethod() {
      Singleton X = Singleton.getInstance();
   }
}
```

Abstract Factory

- ✓**Problem:** We want to create an instance of an interface → but when we write the code, we don't know what concrete type the instance will be.
  - ✓Ex: XML documents, window borders...
- ✓**Solution:** Provides an **interface for creating objects** of the same family ignoring the concrete implementing class

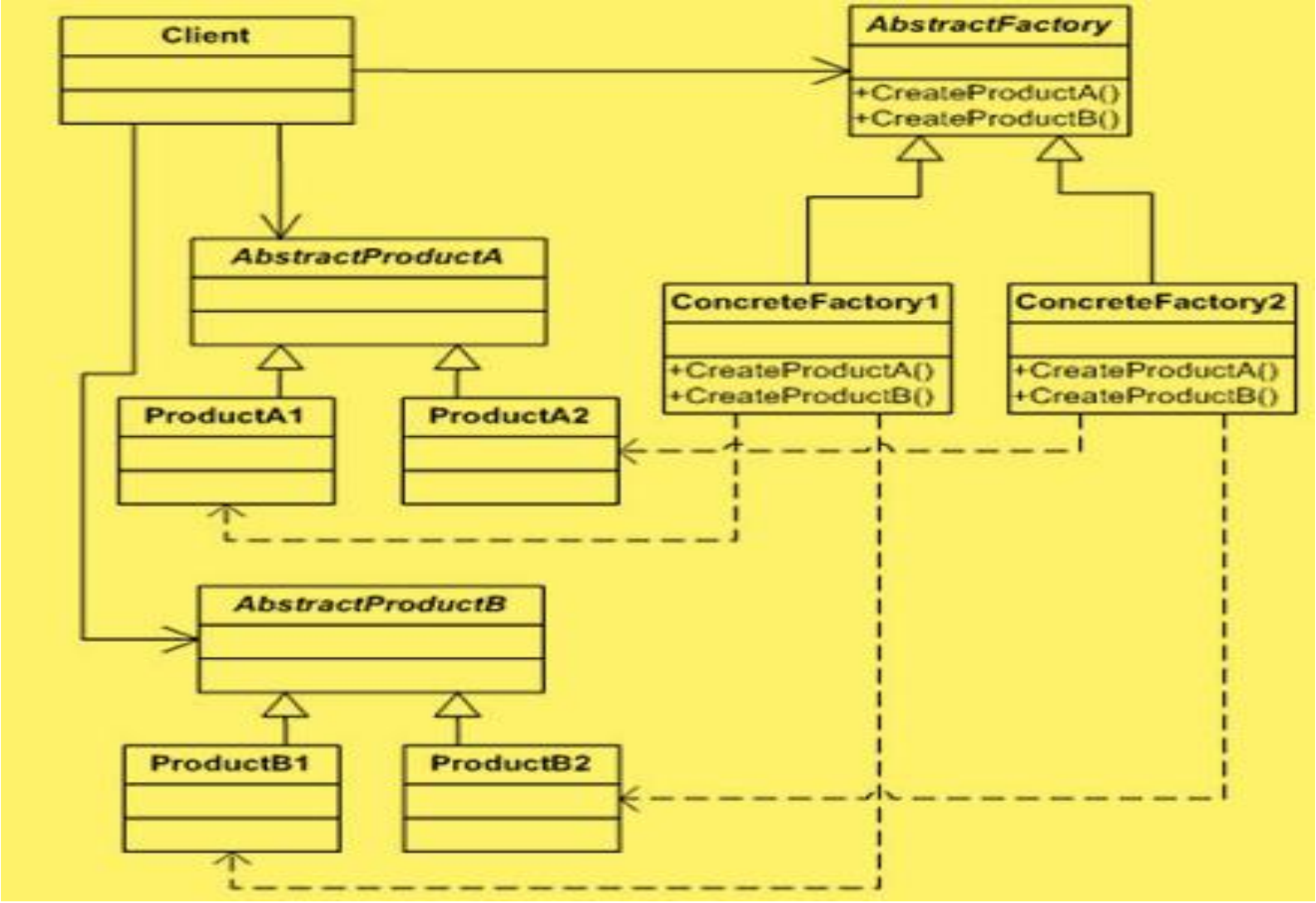


Abstract Factory

- ✓**Create a family of objects** without specifying their concrete classes
- ✓**Problem**: Create an object without worrying about its implementation.
- ✓There are several ways to construct an object
- ✓**Solution**: Factory/element separation Abstract/concrete separation
- ✓**Utilization** : Multiple platforms

Abstract Factory

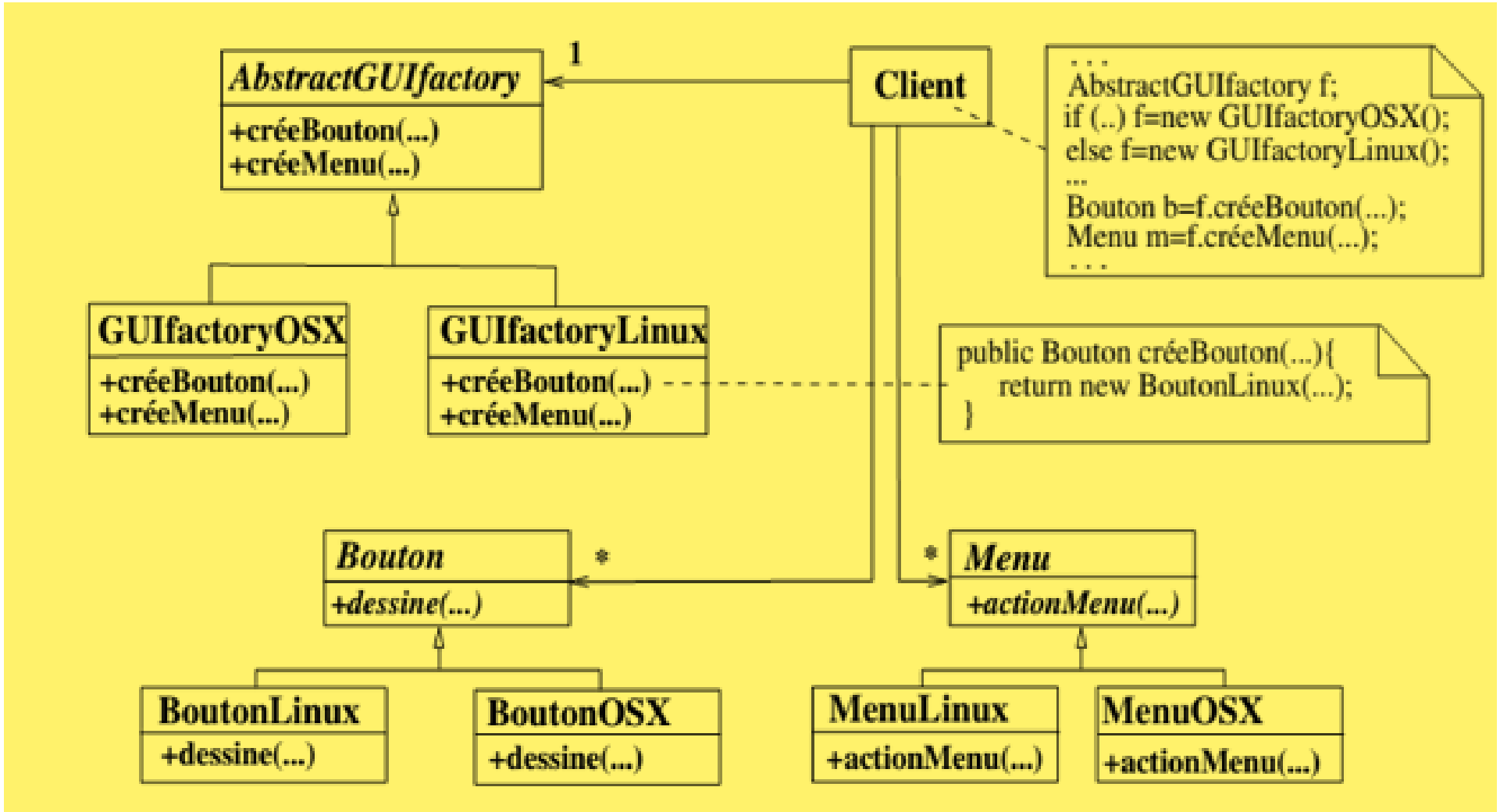
Diagram of the pattern



(Abstract Factory)Illustrative Example

✓Create a **graphical interface** with widgets (buttons, menus, ...)

✓Point of variation: OS (Linux, OSX, Windows)



Abstract Factory

Abstract Factory: it is an abstract interface or class with methods for creating Product objects. We have a creation method for each type of Product.

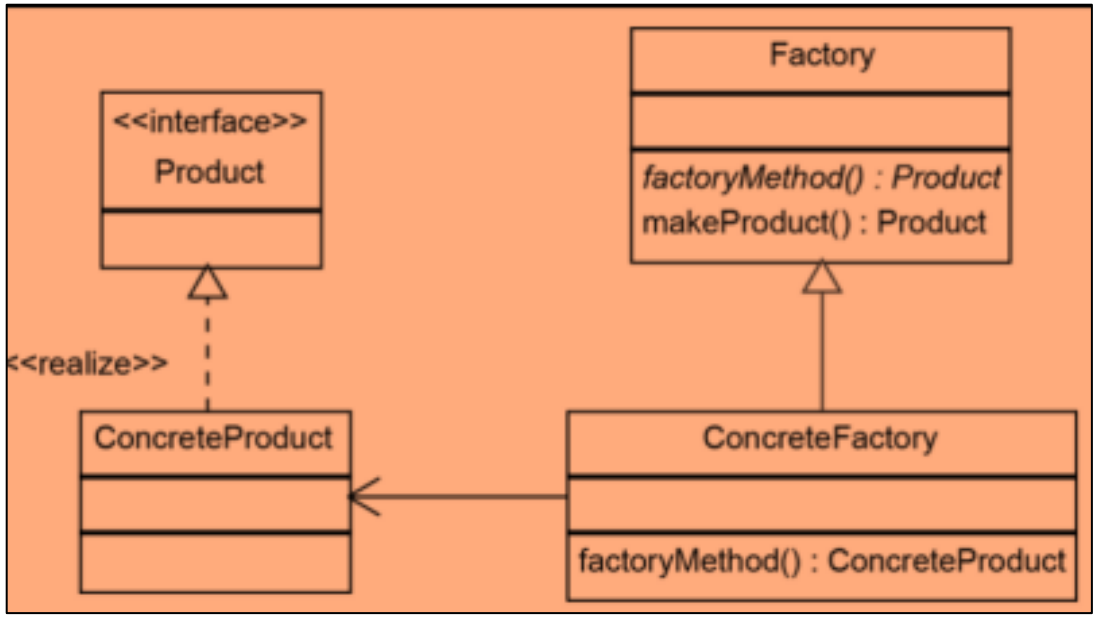
- **Concrete Factory**: It implements the Abstract Factory interface by defining the operations for creating Product objects.
- **Abstract Product**: This is an abstract interface or class relating to a Product object type. It offers a set of methods applicable to all variants of a

same Product.

- **Product**: this is a class defining a Product object that must be created by the corresponding Concrete Factory.
- **Client**: a class using the interfaces declared by Abstract Factory and Abstract Product in order to create Products

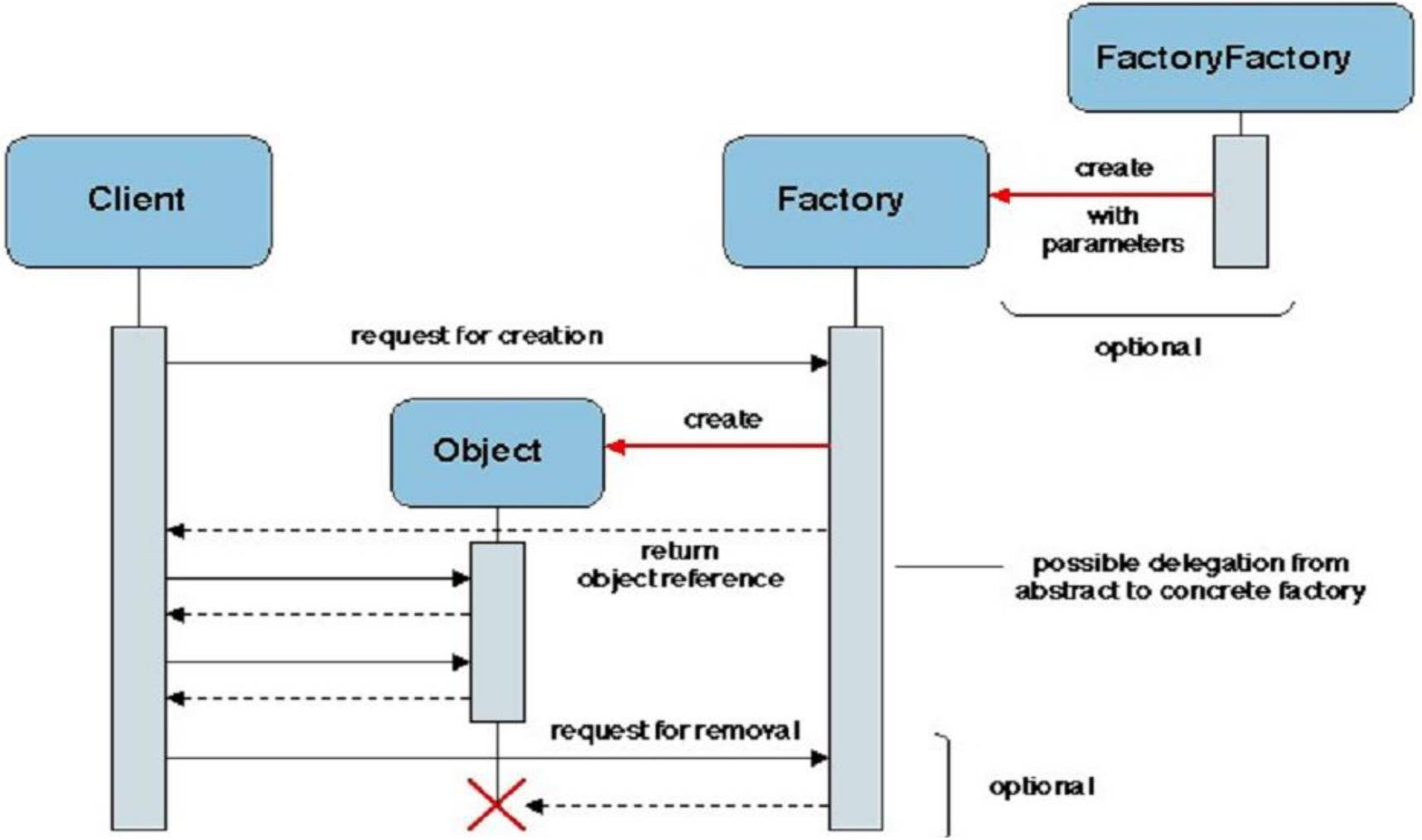
Factory Method (1)

The **Factory method** design pattern allows the **dynamic creation of objects** according to the parameters passed to a creation method called Factory, from which the name of the design pattern comes.
It is a **creation pattern** which aims to instantiate objects whose type is unknown a priori.



Factory Method (2)

Diagram of the pattern



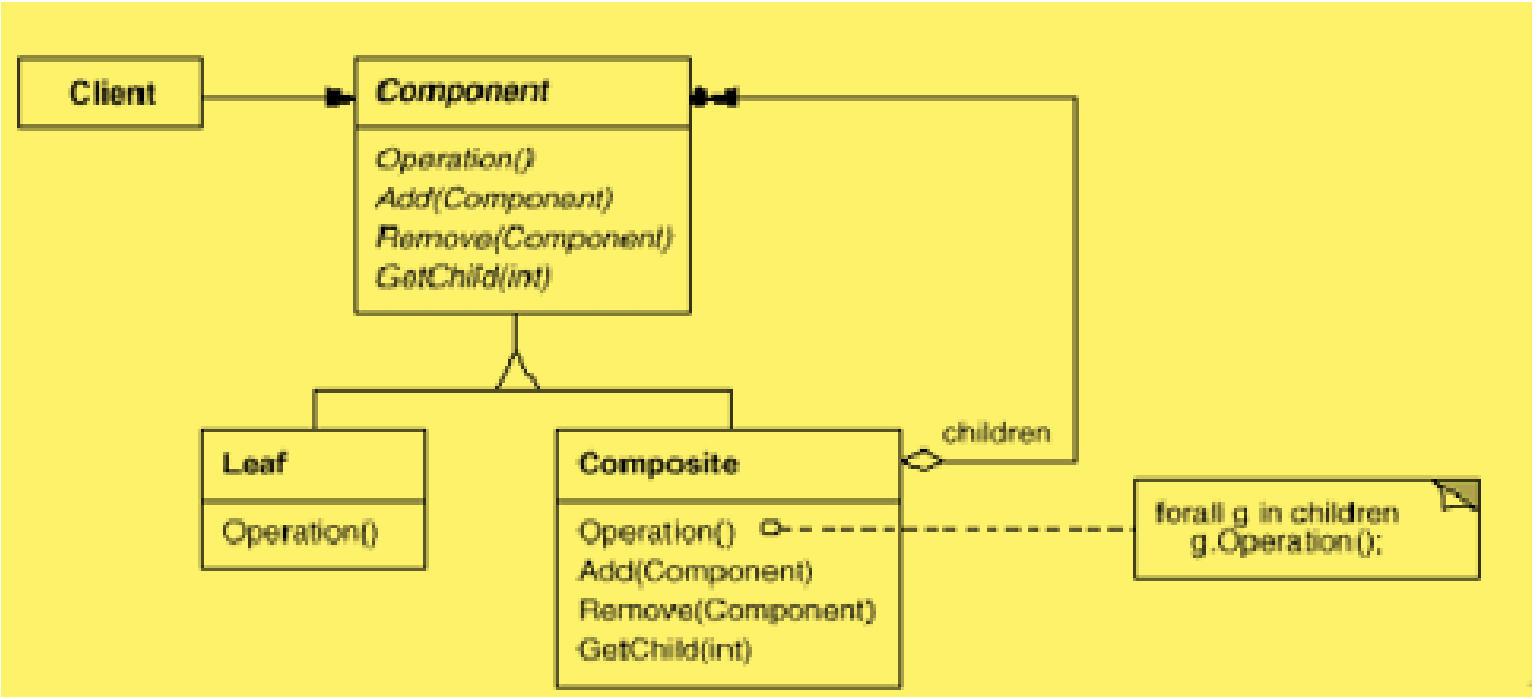
Composite (1)

Problem: We want to create objects which are made up of several other objects

Ex: drawing or cooking software

Solution: We define a component object which is an interface for all the objects forming the composition

→ We declare a **class allowing us to manage the children** (add,remove, etc.)



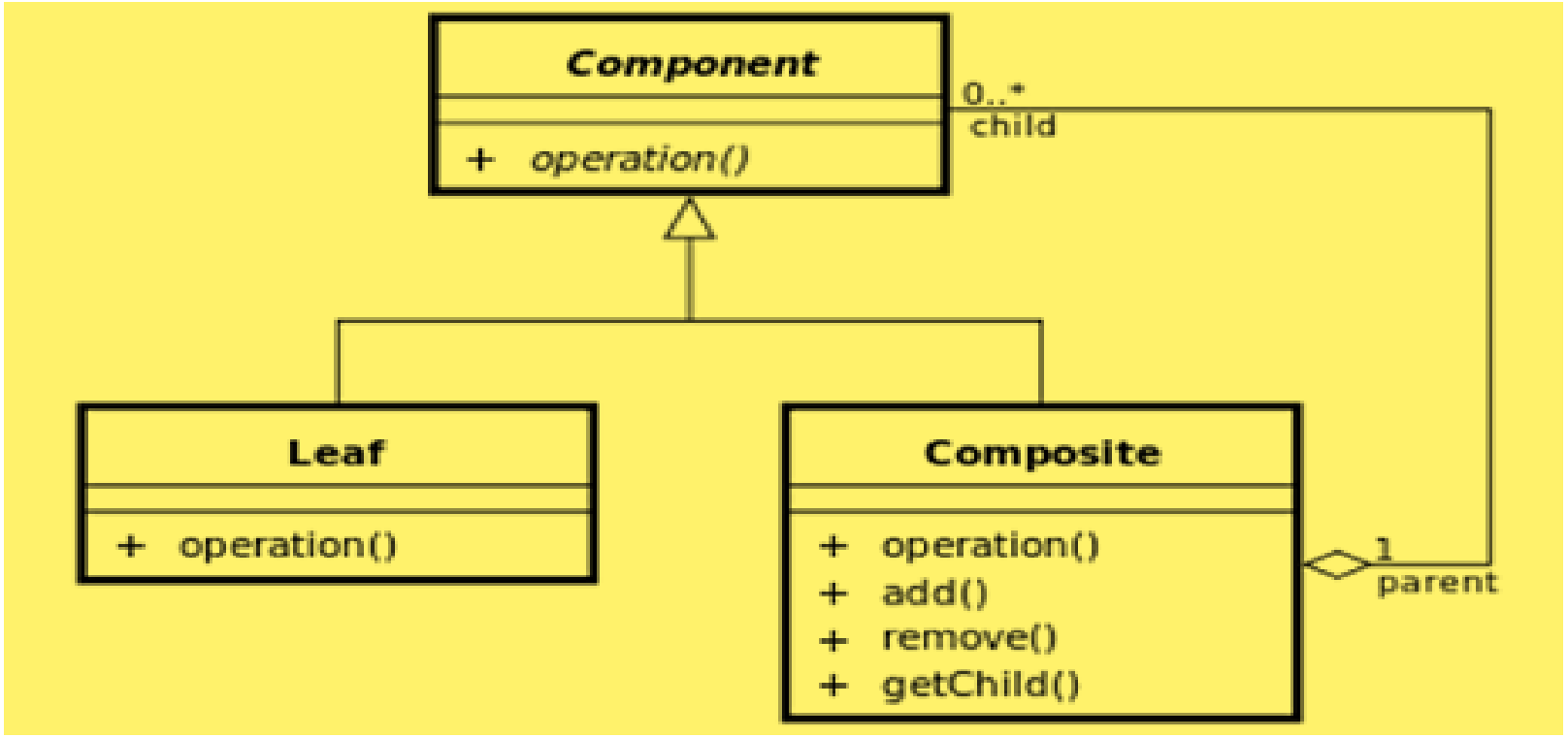
Composite (2)

Applications:

✓Graphic interface

✓Office automation

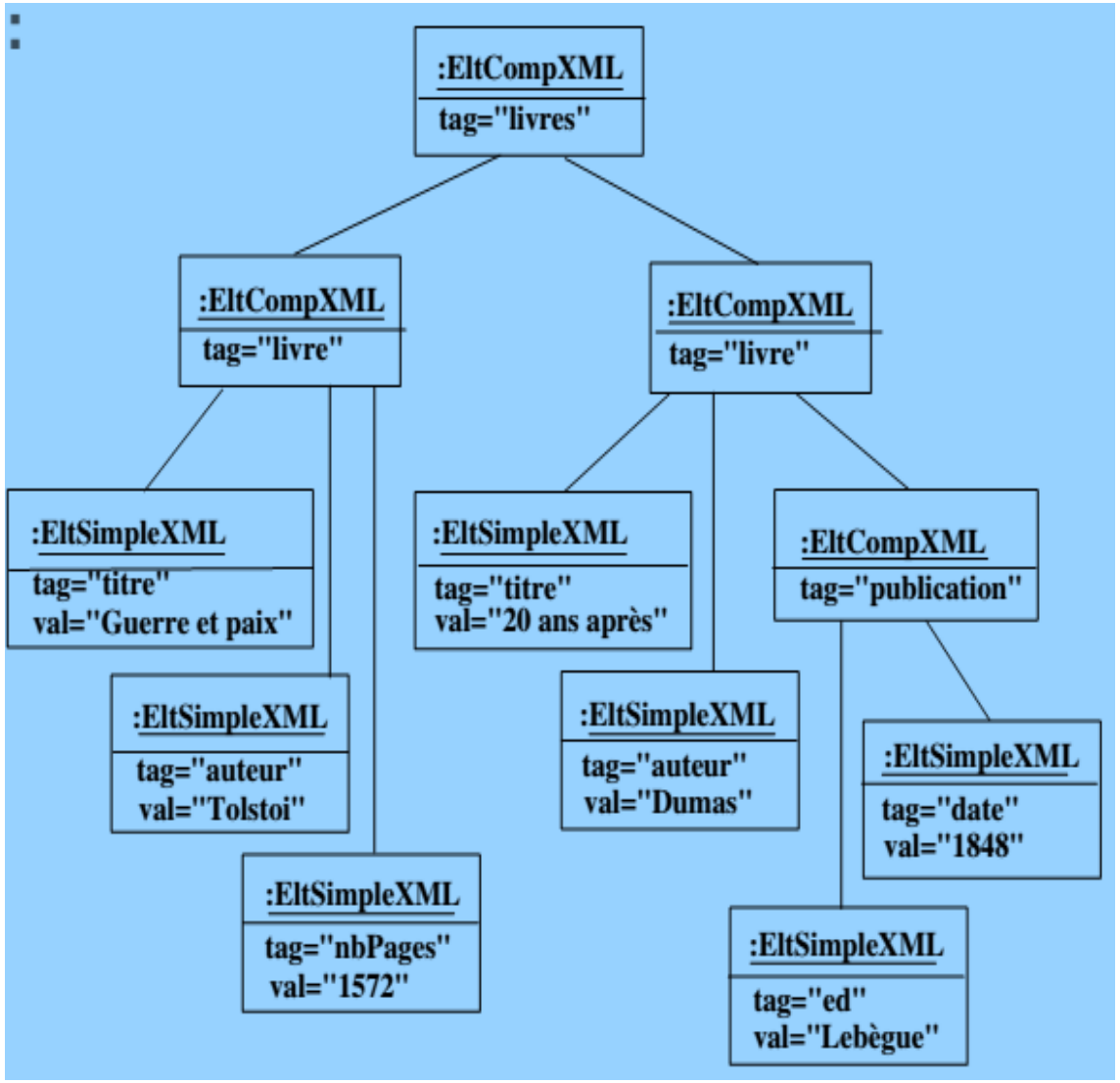
✓Often degraded versions



Composite (3)

- Example

```
<?xml version="1.0"?>
<livres>
    <livre>
        <titre>Guerre et paix</titre>
        <auteur>Tolstoï</auteur>
        <nbPages>1572</nbPages>
    </livre>
    <livre>
        <titre>20 ans après</titre>
        <auteur>Dumas</auteur>
        <publication>
            <ed>Lebègue</ed>
            <date>1848</date>
        </publication>
    </livre>
</livres>
```



Application of Composite Pattern

Statement: A simple figure can be a **point**, a **line** or a **circle**.

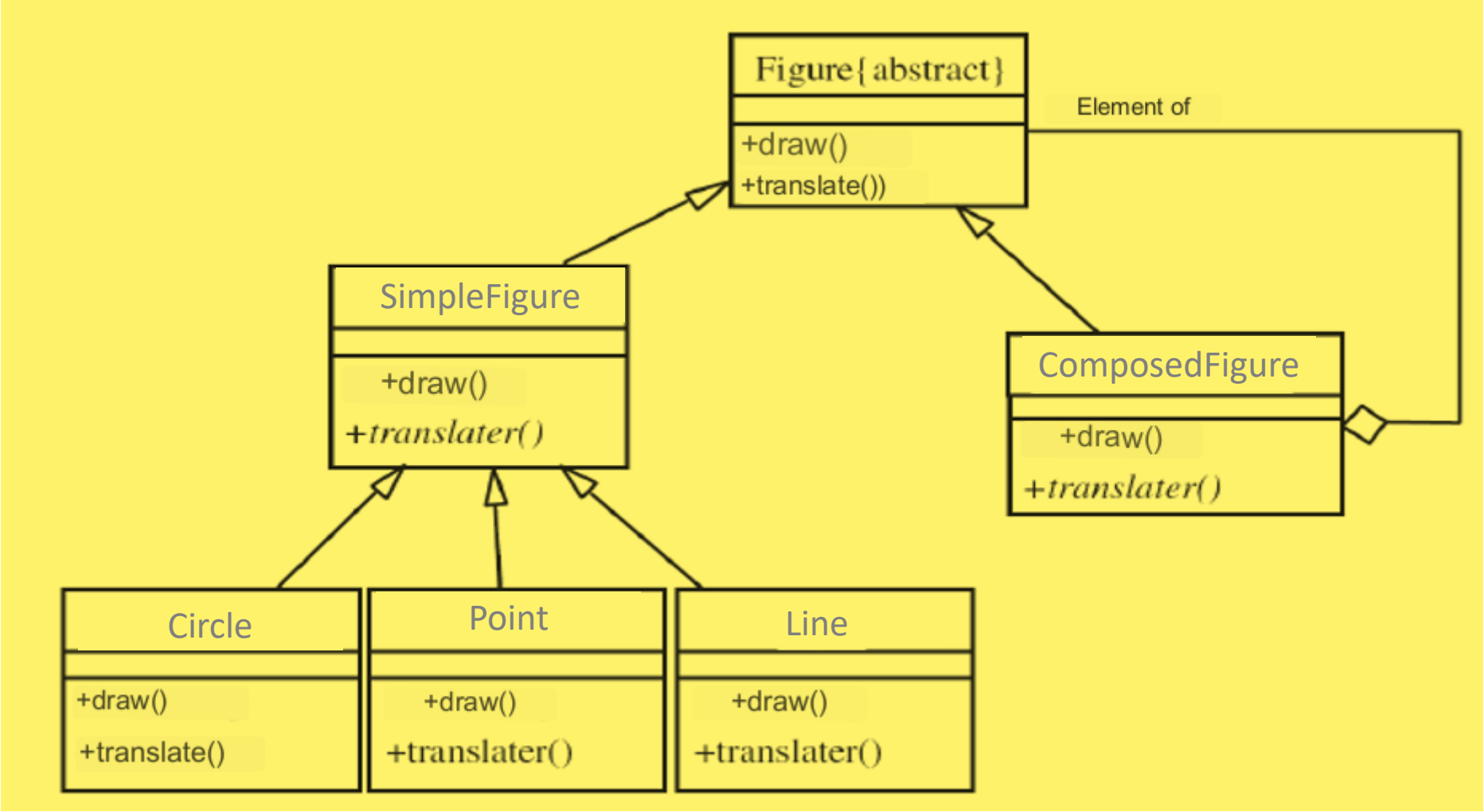
A **figure** can **be composed of other figures**, simple or themselves composed of other figures.

All figures can be **drawn** or **translated**.

Question: Use the composite pattern to construct a class diagram reflecting this hierarchy of objects.

Application of Composite Pattern

Solution



Advantages & Disadvantages of Design Pattern

- ✓ **A high level of abstraction**
  - ✓ which allows to develop software constructions of better quality
- ✓ **Reducing complexity by separating concerns**
- ✓ **Capitalization of experience in software design**
  - ✓ Solution Catalog
- ✓ **Many implementations in programming languages**
- ✓ **Learning**
  - ✓ Synthesis effort: recognize, abstract and apply.
- ✓ **Implementation experience.**
  - ✓ Choosing the “right” Design Patterns to apply
    - ✓ Degraded implementation
  - ✓ Many solutions
    - ✓ Composing patterns...
- ✓ **Code complexity and efficiency**
  - ✓ Many layers, many classes
  - ✓ Blend patterns into code