Conception Logicielle

II - Pattern Design

- - - - - - - - - -

- - - - - - - - - -

ENSMA A3-S5 - période A

2023-2024

M. Richard
richardm@ensma.fr

Sommaire

I - Définition & Objectifs

1 – Problématique
2 – Définition
3 – Objectifs

↪ Point de départ . . .

- *Ce que l'on connaît de la Programmation Orientée Objet :*
  - Typage selon la donnée : les classes
  - Encapsulation
  - Abstraction
  - Héritage
  - polymorphisme
  - Typage selon les fonctionnalités : les interfaces

- *Constat :*
  1. la maîtrise des concepts de base de la POO n'est pas suffisante pour optimiser les différents critères de qualité d'un logiciel [1].
  2. les trois critères difficilement optimisables à ce stade :
     - Souplesse
     - Extensibilité
     - Facile à maintenir

[1] cf. introduction du premier cours

↪ Définition précise

- *Dans le contexte de la programmation objet, un pattern décrit une structure, un agencement de classes utilisant des interfaces.*
- En bref, il s'agit souvent d'un modèle d'architecture logiciel permettant :
  - de résoudre un problème
  - d'améliorer sensiblement la qualité du code vis-à-vis des différents critères de génie logiciel

↪ Catégories :

- Il existe des grands groupes de patterns
- généralement catégoriser selon leur·s auteur·s
- les plus connus et utilisés sont ceux présentés par le GoF (Gang Of Four, composé de Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides)
  - les patrons présentés sont répertoriés en trois catégories :
  - patrons de comportement
  - patrons de structure
  - patrons de création

↪ C'est quoi ? ? ... Ça sert à quoi ? ?

- *Qu'est-ce qu'un Pattern Design ?*
  - un patron de conception . . .
  - un modèle de conception adapté à une situation particulière
  - . . . vous le verrez en les pratiquant ! ! !

- *À quoi ça sert ... ce que l'on entend :*
  - Les patterns indiquent comment construire un logiciel de qualité . . .
    - i.e. en optimisant nos fameux critères
  - Les patterns ne sont pas inventés, ils sont découverts
  - Les patterns sont issus d'une longue expérience de la conception objet
  - Les patterns ne sont pas des morceaux de code . . .
    - mais des modèles de conception adaptés
  - L'objectif de la plupart des patterns est de faciliter l'évolutivité et la modularité des logiciels
  - La plupart des patterns *mettent en œuvre l'encapsulation de ce qui varie dans un système*

II - Patrons de comportement

↪ À utiliser pour :

- fournir des réponses aux problèmes d'interaction entre les classes et aux problèmes de comportement

- On trouve 11 patterns dans cette catégorie :
  - Chaîne de responsabilité
  - Commande
  - Interpréteur
  - Itérateur
  - Médiateur
  - Mémento
  - Observateur
  - État
  - Stratégie
  - Patron de méthode

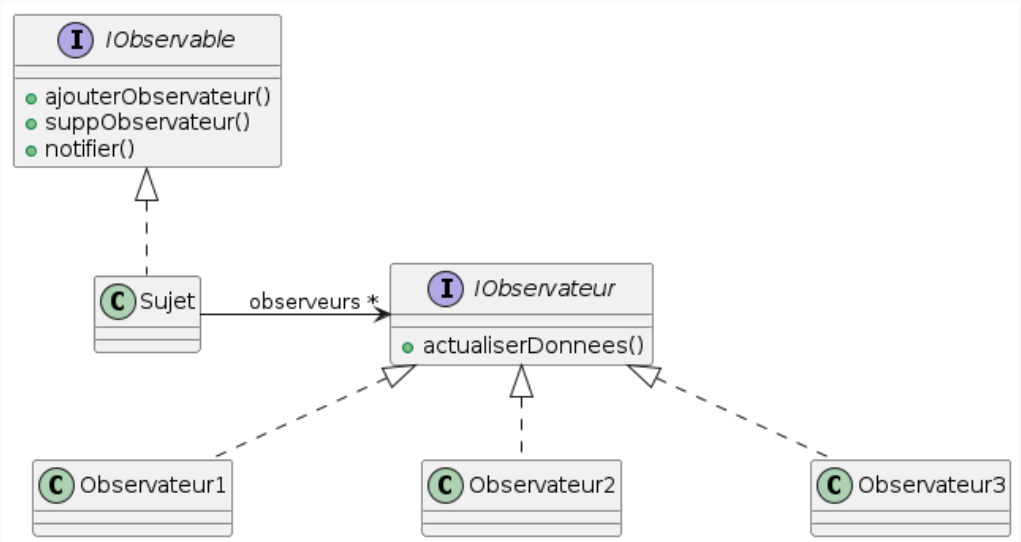
↪ Objectif

- Le pattern observateur permet de définir une relation entre instances de type un à plusieurs de telle sorte que lorsqu'un objet change d'état, tous ceux qui en dépendent soient avertis et soient mis-à-jour automatiquement.

↪ Principe

- Le sujet concret possède une liste d'IObservateur. Il implémente l'interface IObservable définissant les méthodes nécessaires à l'enregistrement ou à la suppression des Observateurs et à leur notification
- les observateurs concrets implémentent l'interface IObservateur. Cette interface définit une méthode commune permettant la mise à jour des observateurs concrets

↪ UML



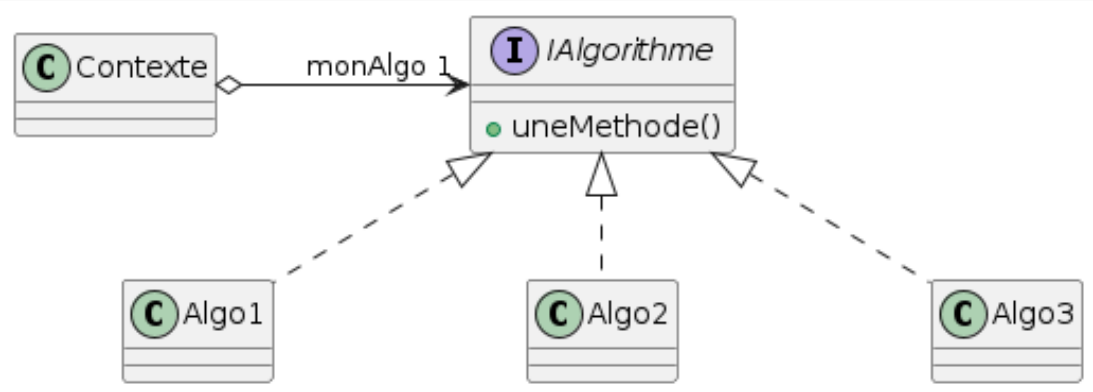
↪ Objectif

- le Pattern Stratégie (Strategy) permet de changer le comportement, i.e. l'algorithme, d'une méthode à la volée, en fonction par exemple d'un état ou de la valeur d'une variable.

↪ Principe

- Le pattern stratégie définit une famille d'algorithmes, encapsule chacun d'eux pour les rendre interchangeables. Le comportement peut ainsi varier en fonction du contexte et donc dynamiquement.

↪ UML



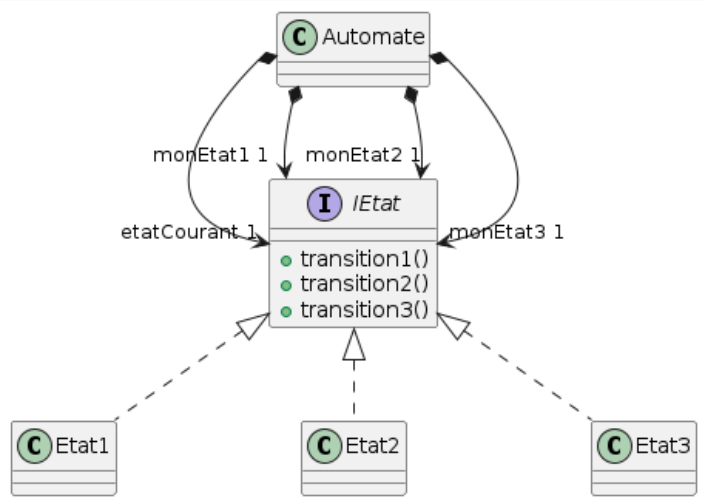
↪ Objectif

- Permet de changer le comportement de l'état d'un objet sans pour autant en changer l'instance, c'est-à-dire l'implémentation d'un automate (composé d'un ensemble d'états et de transitions)

↪ Principe

- la classe devant changer d'état possède un lien vers une interface `Etat`. Cette interface définit les différentes méthodes représentant l'ensemble des transitions présentes dans l'automate.
- chaque état concret implémente l'interface état et définit son comportement pour chacune des transitions
- chaque état concret doit être instancié en recevant en paramètre une référence de l'instance devant changer d'état

↪ UML



III - Patrons de création

1 – Objectifs
2 – Pattern Singleton
3 – Pattern Prototype
4 – Pattern Fabrique & Fabrique abstraite

↪ À utiliser pour :

- résoudre les problèmes liés à la création et/ou à la configuration des objets.

- Il existe 5 patterns différents dans cette catégorie :
  - Singleton
  - Prototype
  - Fabrique
  - Fabrique abstraite
  - Monteur

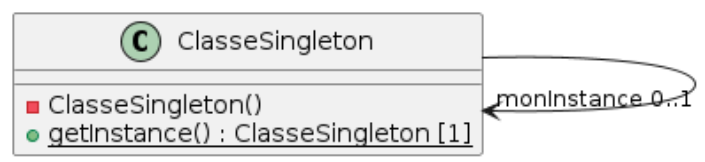
↪ Objectif

- permet de restreindre l'instanciation d'une classe à un seul objet (ou bien à quelques objets seulement).

↪ Principe

- le singleton est implémenté par une classe contenant une méthode statique qui instancie une instance uniquement s'il n'en existe pas encore.
- si l'instance existe, la méthode retourne l'objet existant déjà
- le constructeur sera privé afin qu'aucune autre classe ne puisse instancier une nouvelle instance
- nécessite un attribut pour stocker l'instance créée

↪ UML



↪ Environnement multi-thread

- *Problématique :*
  - l'implémentation classique est extrêmement dangereuse en environnement multi-threadé
    - deux threads peuvent exécuter le test simultanément et créer ainsi chacun une instance du singleton
    - Elle doit donc être absolument proscrite dans ce contexte.
- *Solutions :*
  - Utiliser la synchronisation locale
    - utilise le mot clé `synchronized` (Cf. cours App. Mobiles)
    - solution très coûteuse si utilisation intensive
  - Technique du Holder
    - repose sur l'utilisation d'une classe interne privée et stockée dans un attribut statique
    - cette classe interne réalise l'instanciation de l'instance unique du Singleton.

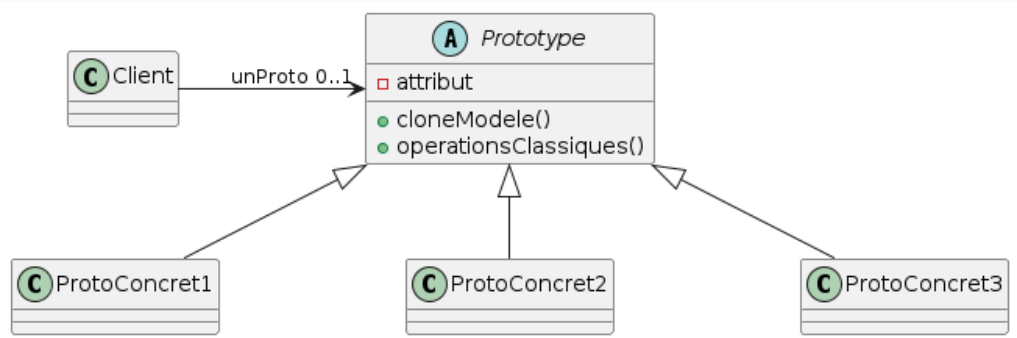
↪ Objectif

- Permet d'alléger la création d'une instance à partir d'une instance existante
- Utile lorsque l'instanciation d'une instance est coûteuse en temps
  - phase d'initialisation longue,
  - Initialisation complexe,
  - ...

↪ Principe

- Le principe est d'instancier de nouvelles instances par copie/clonage d'une autre instance et de ne modifier que les attributs nécessaires.
- Construction d'une classe abstraite possédant une méthode clone abstraite devant être redéfinie par les classes de spécialisation.
  - cette méthode clone l'instance sur laquelle elle est appelée
  - parfois lié au Pattern Fabrique

↪ UML



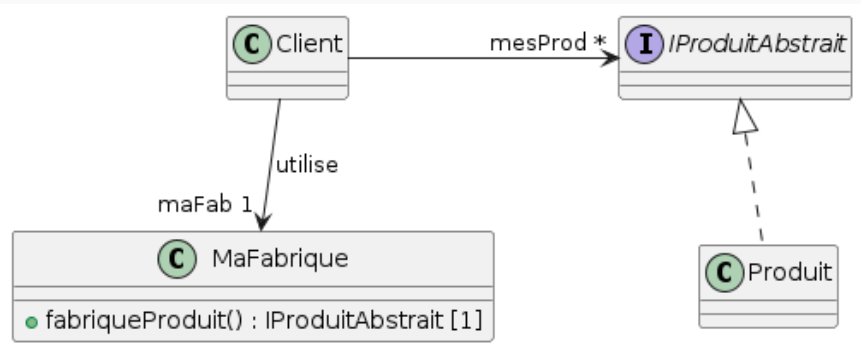
↪ Objectif

- permet la création d'instance dont le type n'est pas prédéfini
- unifie l'interface de création d'objet de même catégorie
- les instances sont crées dynamiquement en fonction des informations passées en paramètre à la fabrique

↪ Principe

- le·s paramètre·s passé·s à la fabrique lui permet d'instancier le bon objet.
  - type énuméré,
  - chaîne de caractères,
  - ...
- elle retourne alors une référence sur un IProduit contenant une instance du type voulu

↪ UML



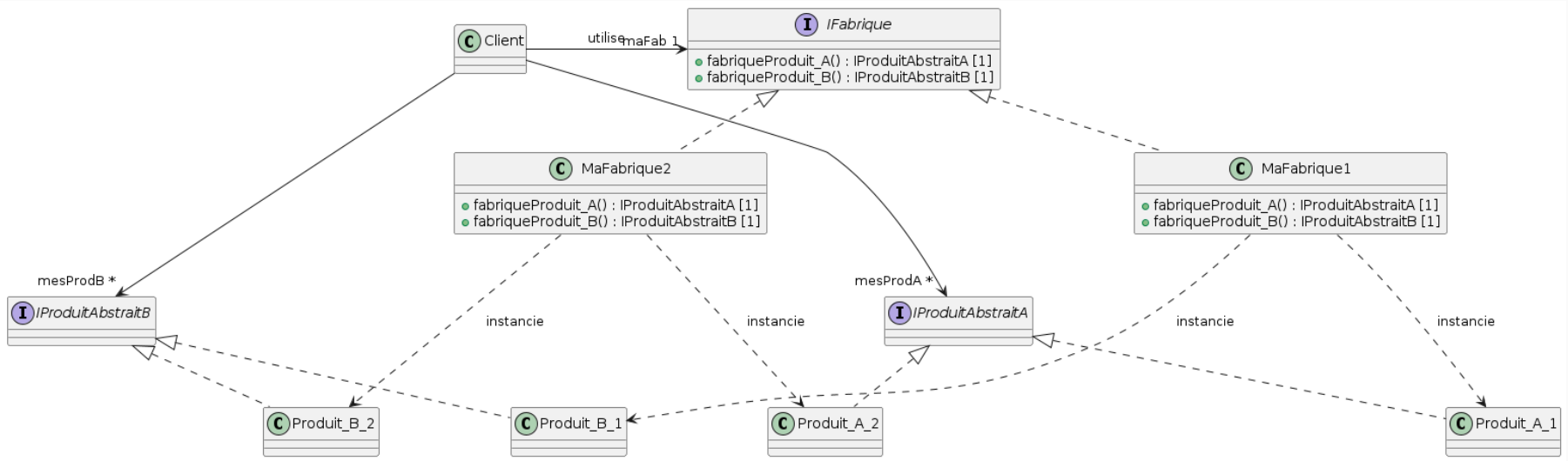
↪ Objectif

- permet la création d'instance dont le type n'est pas prédéfini pour différents types
- unifie la manipulation de plusieurs fabriques permettant ainsi la création d'instance de types différents
- les instances sont crées dynamiquement en fonction des informations passées en paramètre à l'une des fabriques

↪ Principe

- mutualise et uniformise le fonctionnement de différentes fabriques

↪ UML



IV - Patrons de structure

↪ À utiliser pour :

- permet de résoudre les problèmes liés à la structuration des classes
- il existe 7 patterns dans cette catégorie :
  - Pont
  - Façade
  - Adapteur
  - Composite
  - Proxy
  - Poids-mouche
  - Décorateur

↪ Objectif

- permet de simplifier la complexité d'une API en fournissant une interface simple du sous-système
  - La façade doit encapsuler la complexité des interactions entre les objets

↪ Principe

- la façade est réalisée en réduisant les fonctionnalités,
- mais doit fournir toutes les fonctions nécessaires à la plupart des utilisateurs.

↪ Utilisation

- rendre une bibliothèque plus facile à utiliser, comprendre et tester
- rendre une bibliothèque plus lisible
- réduire les dépendances entre les clients de la bibliothèque et le fonctionnement interne de celle-ci

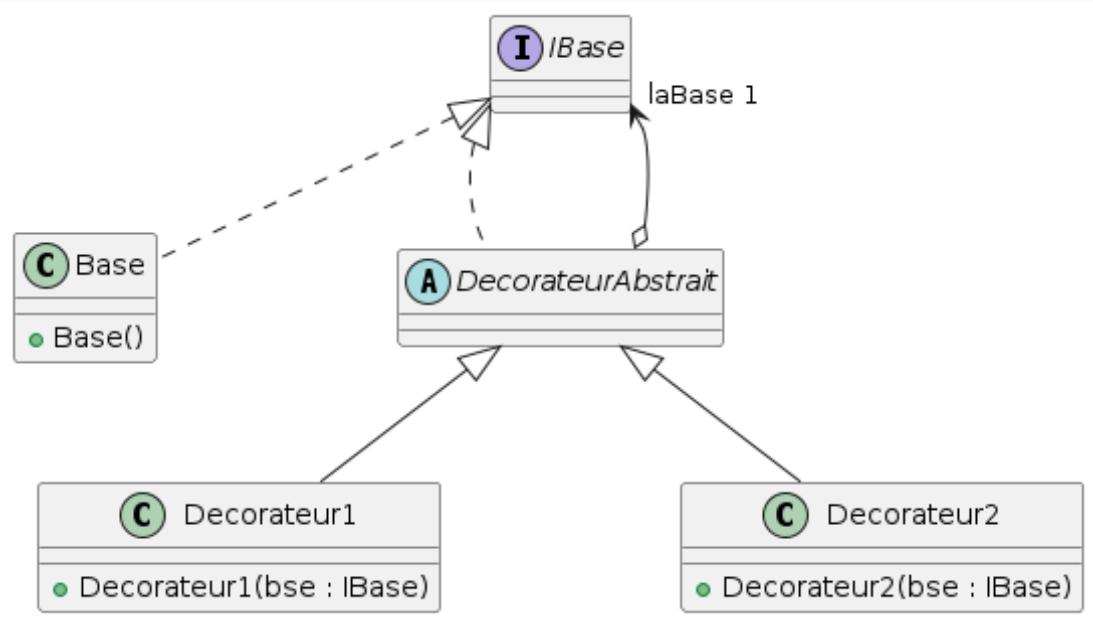
↪ Objectif

- alternative à l'héritage
- Permet d'attacher dynamiquement de nouveaux comportements aux classes
- limite la multiplication des classes de spécialisations

↪ Principe

- lors de spécialisations successives, pour couvrir tous les cas, le nombre de classes croît très rapidement
- en créant uniquement les classes modélisant les différents comportement, il est ensuite possible de construire tous les combinaisons possibles par allocation dynamique
- ces classes décorent la classe de base
- la création se fait en instanciant les décorateurs les uns en paramètres des autres ; le paramètre du dernier décorateur est la classe de base

↪ UML


IBase
laBase 1
Base
Base()
DecorateurAbstrait
Decorateur1
Decorateur1(bse : IBase)
Decorateur2
Decorateur2(bse : IBase)