L3 Info – INF5A1
Jour 7
le 14 octobre 2024

Design Patterns
(suite)

Types de Patterns

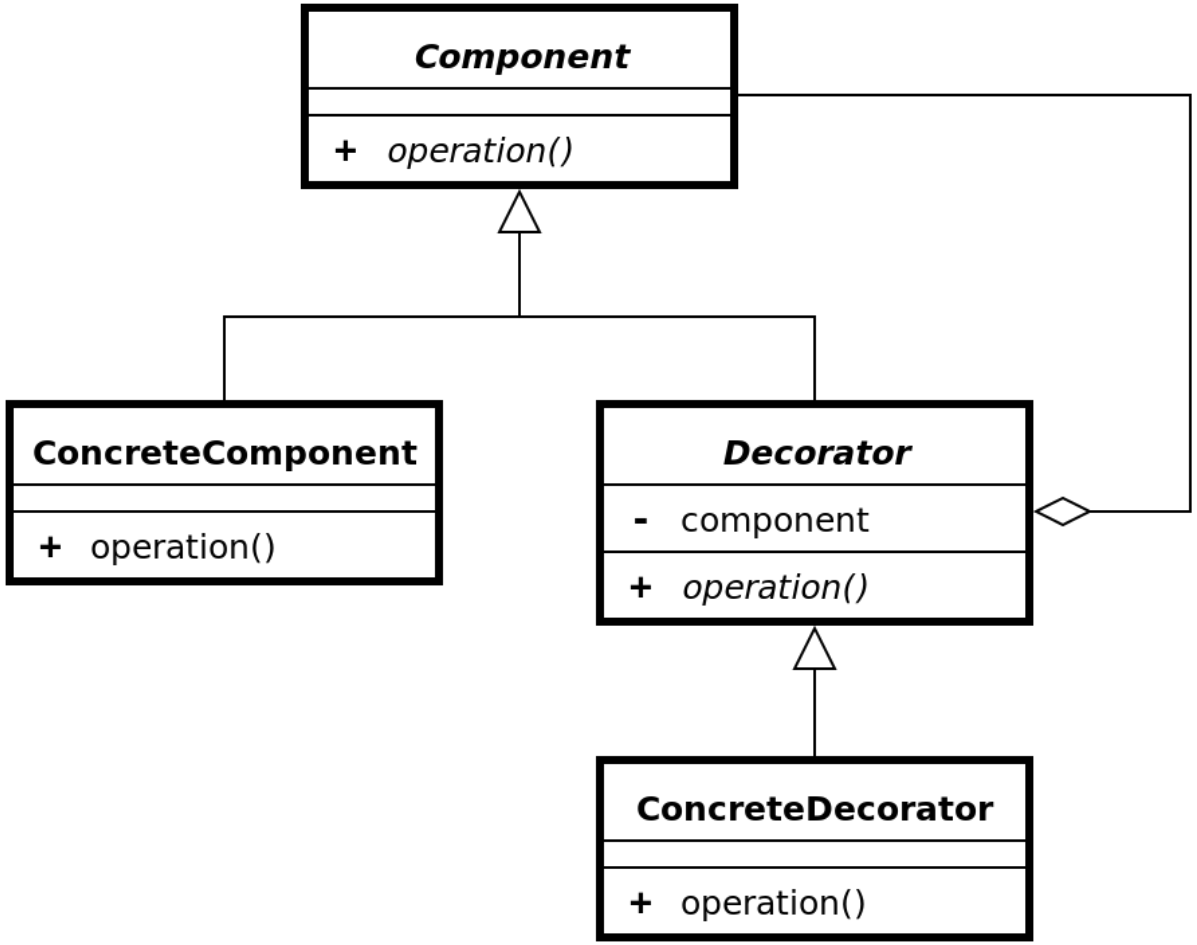
- De **construction** (Singleton, Abstract Factory), permettant de déléguer la construction d'instances à un acteur particulier.
- De **structuration** (Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy)
- De **comportement** (Chain of Responsibility, Command, Interpreter, Iterator, Memento, Observer, State, Strategy, Template Method)

Decorator

- Problème :
  - on souhaite pouvoir donner dynamiquement des fonctionnalités ou des caractéristiques à un objet, plutôt que des les figer dans des classes
  - il s'agit en particulier d'éviter l'explosion du nombre de classes et la redondance associée lorsque l'on souhaite disposer de familles d'objets ayant des fonctionnalités diverses
- Solution :
  - créer un objet B qui se fait passer pour un objet A tout en enrichissant ses fonctionnalités.

Diagramme de Decorator

crédit : wikipedia



Principe général de Decorator

- Le type à décorer est idéalement une interface (pour autoriser n'importe quelle implémentation), mais il peut s'agir d'une classe, abstraite ou non.
- Ce type peut disposer de plusieurs sous-types concrets
- La classe Decorator est à la fois :
  - un **sous-type** du type à décorer, pour pouvoir se substituer à n'importe quel élément à décorer, par polymorphisme
  - en **association** avec le type à décorer, pour l'interroger et ainsi pouvoir le représenter tout en l'enrichissant
- Tout élément du type à décorer peut alors être passé au constructeur d'une instance de Decorator, qui est alors une instance plus riche que l'instance de base.

Exemple

- L’interface Decorable définit la méthode String getDescription().
- Elle dispose de 2 implémentations Decorable1 et Decorable2, dont la méthode renvoie respectivement "décorable1" et "décorable2".
- On crée un décorator DecoratorA :
  - il implémente aussi Decorable
  - il prend en paramètre de constructeur un Decorable dont il enregistre une référence dans l’attribut monDecorable.
  - sa méthode getDescription() renvoie monDecorable.getDescription()+ " décoré par A".
- On utilise Decorator ainsi :
  - Decorable d1=new Decorable1();
  - Decorable dDecore=new DecoratorA(d1);
  - System.out.println(d1.getDescription()); // décorable1
  - System.out.println(dDecore.getDescription()); // décorable1 décoré par A
- On peut faire la même chose avec une instance de Decorable2, et on obtiendra « décorable2 décoré par A »

Les poupées russes

- Comme on le voit dans le diagramme de classes, un Decorator est lui-même un Decorable
- Il est donc possible de chaîner les décorations si l'on dispose de plusieurs types de Decorator
- Exemple :
  - On crée un Decorator DecoratorB dont la méthode getDescription() renvoie monDecorable.getDescription()+ " décoré par B"
  - On ajoute à notre programme :
    - Decorable decoratorB = new DecoratorB(decoratorA);
    - System.out.println(decoratorB.getDescription()); // decorable1 décoré par A décoré par B

Exemple 2 : interfaces graphiques

- On suppose que l'on dispose d'un type Component qui est le super-type des composants graphiques
- On crée le Decorator DecorationScrollVertical qui ajoute un ascenseur automatiquement dès que le contenu du panneau excède la hauteur
- On crée le Decorator DecorationScrollHorizontal
- On peut à présent, au choix, rendre n'importe quel composant scrollable horizontalement ou verticalement ou les deux à la fois.
- Sans se pattern, on aurait dû créer une multitude de classes (ex. Bouton, BoutonScrollHorizontal, BoutonScrollVertical, BoutonScfrollHorizontalEtVertical, puis la même chose pour la classe Label, etc.)
- Cf la classe JScrollPane de Swing.

Remarques complémentaires

- La classe abstraite Decorable, super-classe de tous les décorateurs, est optionnelle (utile seulement si l'on crée plusieurs decorators)
- Le decorator peut éventuellement ajouter de nouvelles méthodes par rapport au type décoré (cf. BufferedReader avec readLine() qui n'existe pas dans Reader), mais celles-ci ne pourront être invoquées que si l'on dispose d'un référence du type du Decorator. Si on reçoit cet objet sous le type général Decorable, ces méthodes additionnelles ne sont pas utilisables donc inutiles.

Le pattern Proxy

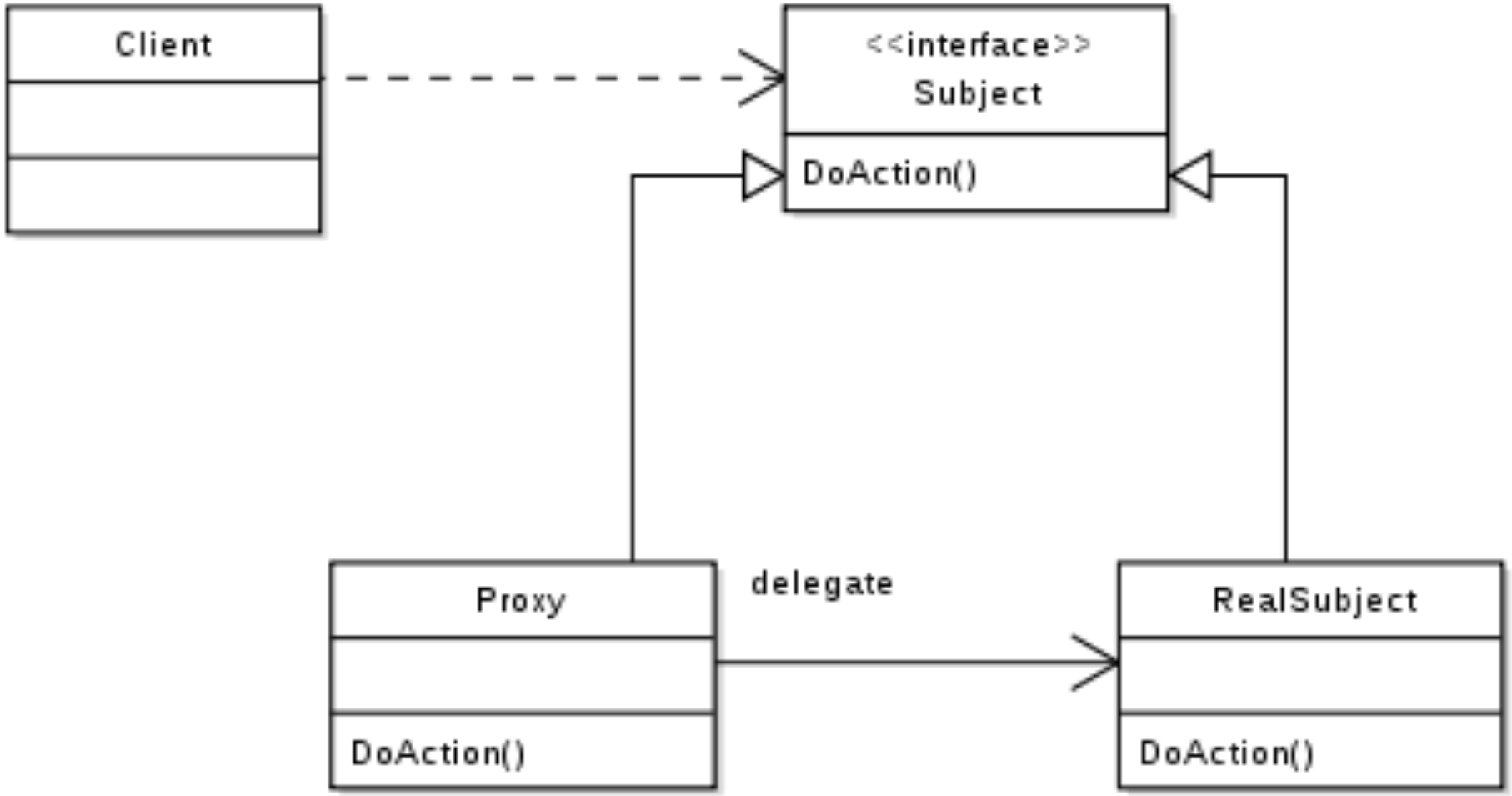
- Définition générale d'un proxy (hors patterns): « Un **proxy** est un composant logiciel informatique qui joue le rôle d'intermédiaire en se plaçant entre deux hôtes pour faciliter ou surveiller leurs échanges. » (wikipedia)
- Un proxy peut être créé en programmation objet au niveau des instances

Proxy (suite)

- Problème : on veut pouvoir contrôler, modifier ou enrichir l'accès à un objet (d'un type qui ne permet pas initialement ce contrôle)
- Solution : on crée une instance supplémentaire, le proxy, va servir d'intermédiaire entre l'objet à contrôler et celui qui veut y accéder
- On va passer au client non pas l'objet à contrôler, mais le proxy qui en contrôle l'accès

Diagramme de Proxy

crédit : wikipedia



Exemple de proxy

- L'interface Personne dispose des méthodes :
  - public String getNom()
  - public int getAnneeNaissance()
- On en crée (au moins) une implémentation PersonneConcrete
- Certaines personnes ne souhaitent pas communiquer leur date de naissance, mais veulent bien que l'on accède à leur nom. On crée donc le proxy PersonneNaissanceSecrete qui a un attribut Personne monInstance :
  - public String getNom() { return monInstance.getNom();} // délégation
  - public int getAnneeNaissance() { throw new RuntimeException("méthode indélicate bloquée par le proxy")}; // contrôle strict : interdiction et levée d'une exception
- Usage :
  - Personne p = new PersonneConcrete("Johnny",1943);
  - PersonneNaissanceSecrete proxy =new PersonneNaissanceSecrete(p);
  - System.out.println(proxy.getNom()); // Johnny
  - System.out.println(proxy.getDateNaissance()); // déclenche une exception
  - Si l'on passe par p plutôt que par le proxy, les deux méthodes restent bien sûr accessibles...
- Cet exemple est radical puisqu'il déclenche une exception, mais il pourrait renvoyer null, ou une date factice...

Quelques types de proxy

- Contrôle d'accès : accès partiel aux données, par exemple en vertu de droits d'accès
- Cache (buffer) : améliorer les performances en enregistrant les réponses déjà faites pour ne plus les recalculer
- Objet distant (Remote Object) : accès à un objet situé sur une autre machine virtuelle, via le réseau

Proxy = Decorator ?

- Du point de vue architectural, les patterns sont proches, mais :
  - Proxy est en association avec un type particulier
  - Decorator est en association avec le type général
  - Par conséquent, Proxy peut accéder à des méthodes spécifiques (non présente dans le super-type), mais ne permet pas de jouer aux poupées russes.
- Du point de vue conceptuel, la finalité est différente : Proxy = gérer/contrôler l'accès, Decorator = enrichir
- Mais il est clair que Decorator peut faire office de Proxy. Après tout, la gestion d'accès n'est-elle pas une décoration parmi d'autres ?

Proxy dans le cadre du CC

- Chaque joueur doit avoir une vue partielle sur les données (certaines bombes ou mines ne lui sont pas montrées)
- On peut malgré tout disposer d'une seule instance du contenu du plateau, et mettre en place Proxy, en donnant une instance de proxy à chaque joueur
- Attention, chaque instance de proxy doit être paramétrée pour son joueur particulier, chaque joueur voyant uniquement ses propres bombes et mines.