Le patron de conception Commande

Adrien CAUBEL

11 octobre 2022

Table des matières

1 Le concept du patron Commande

1.1 Définition

Le patron Commande est un patron de conception comportemental qui prend une action à effectuer et la transforme en un objet autonome qui contient tous les détails de cette action. Cette transformation permet de paramétrer des méthodes avec différentes actions, planifier leur exécution, les mettre dans une file d'attente ou d'annuler des opérations effectuées.

1.1.1 Interprétation de la définition

"Prend une action à effectuer et la transforme en un objet autonome." Traditionnellement, lorsqu'on veut réaliser une action nous appelons une méthode. Ici, au lieu d'appeler cette méthode qui est dans une classe nous allons créer une nouvelle classe `C` représentant l'action a effectuer. D'où la création d'un objet autonome de type `C`.

"Qui contient tous les détails de cette action". Les détails, c'est-à-dire que doit-on faire quand l'action est réalisée seront déplacés dans la nouvelle classe `C`.

2 Exemples

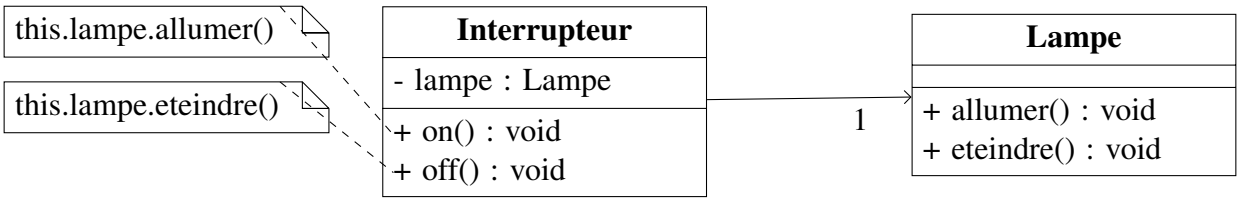
Pour mieux appréhender ce patron nous allons l'illustrer avec deux exemples. Le premier exemple permettra de poser les bases de ce patron, et le second, un peu plus complexe, montrera toute l'utilité de ce patron dans une application.

2.1 Exemple d'un interrupteur

Dans ce premier exemple, nous allons voir comment ce patron permet de simplifier l'architecture de notre application.

2.1.1 Définition du problème

Nous souhaitons réaliser un interrupteur qui allume et éteint une lampe. De cette problématique on peut déduire deux classes `Interrupteur` et `Lampe`. Nous en donnons donc une représentation UML.

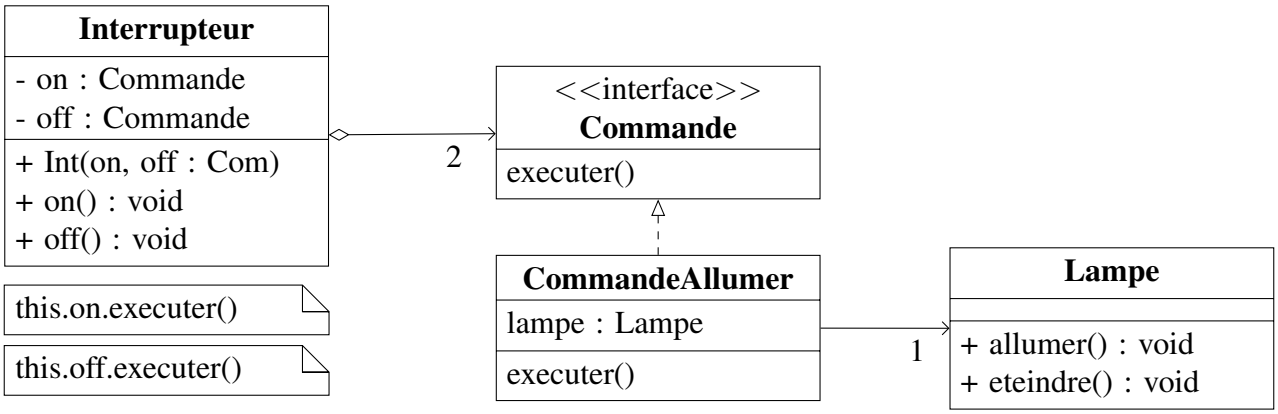


Dans cette conception notre interrupteur a connaissance du fonctionnement de la lampe.

2.1.2 Ajout du patron Commande

La définition du patron Commande nous demande de transformer une action en un objet autonome. Ici les deux actions qu'on souhaite réaliser sont `allumer()` et `éteindre()` une lampe. Les méthodes `on()` et `off()` sont les éléments déclencheurs des deux actions.

Nous représentons donc nos deux actions par une classe chacune. Puis l'interrupteur sera en charge d'appeler l'action souhaitée.



Notre conception se décompose en quatre parties :

— Les commandes qui sont représentées par les classes `CommandeAllumer` et `CommandeEteindre`. Chaque commande est associée au récepteur.

— Les commandes sont sous la responsabilité d'une interface qui définit un contrat avec seulement la méthode `executer()`. Étant donné que chaque commande est en charge d'une seule et unique action.
Cette interface nous permet également de respecter les principes d'inversion de dépendances et d'injection de dépendances
— Le déclencheur de l'action (ou invocateur) est l'`Interrupteur`. On y définit les deux commandes par injection de dépendances.
Puis nous définissons deux méthodes `on()` et `off()` qui sont en charge d'appeler la bonne commande.
— Enfin, nous avons le récepteur de notre action qui est la `Lampe`.

2.1.3 Code implémentant la solution

```
interface Commande {
   void executer();
}

class CommandeAllumer {
  private Lampe lampe;

  public CommandeAllumer(Lampe lampe) { this.lampe = lampe; }

  public executer() {
     lampe.allumer();
  }
}
```

```
class Interrupteur {
   private Commande on;
   private Commande off;

   public Interrupteur(Commande on, Commande off) {
     this.on = on; this.off = off;
   }

   public void on() { on.executer(); }
   public void off() { off.executer(); }
}
```

```
public static void main(String args[]) {
   Lampe lampe = new Lampe();

   Interrupteur i = new Interrupteur(new CommandeAllumer(lampe),
                                     new CommandeEteindre(lampe));
  ...
  i.on();
    // -> va appeler CommandeAllumer#executer()
       // -> va appeler Lampe#allumer()
}
```

2.1.4 Première interrogation

Pourquoi utiliser le patron Commande tandis qu'on a une solution architecturale plus simple sans ?

https ://stackoverflow.com/questions/32597736/why-should-i-use-the-command-design-pattern-while-i-can-easily-call-required-met/32597828

2.1.5 Conclusion de l'exemple

Il est vrai que sur cet exemple l'utilisation du patron semble moindre. Mais, au lieu de réaliser l'action dans le code initiateur de l'action (méthode `on()` ou `off()`), nous faisons un simple appel à la méthode `executer()`, ainsi s'il y a d'autres étapes à faire lorsqu'on allume ou éteint la lampe nous les définirons dans cette même méthode et nous ne "polluerons" pas les méthodes de la classe `Interrupteur`.

2.2 Exemple d'un éditeur

Ce second exemple un peu plus complexe va nous permettre découvrir la solution apportée par le patron Commande pour réduire la complexité de notre application.

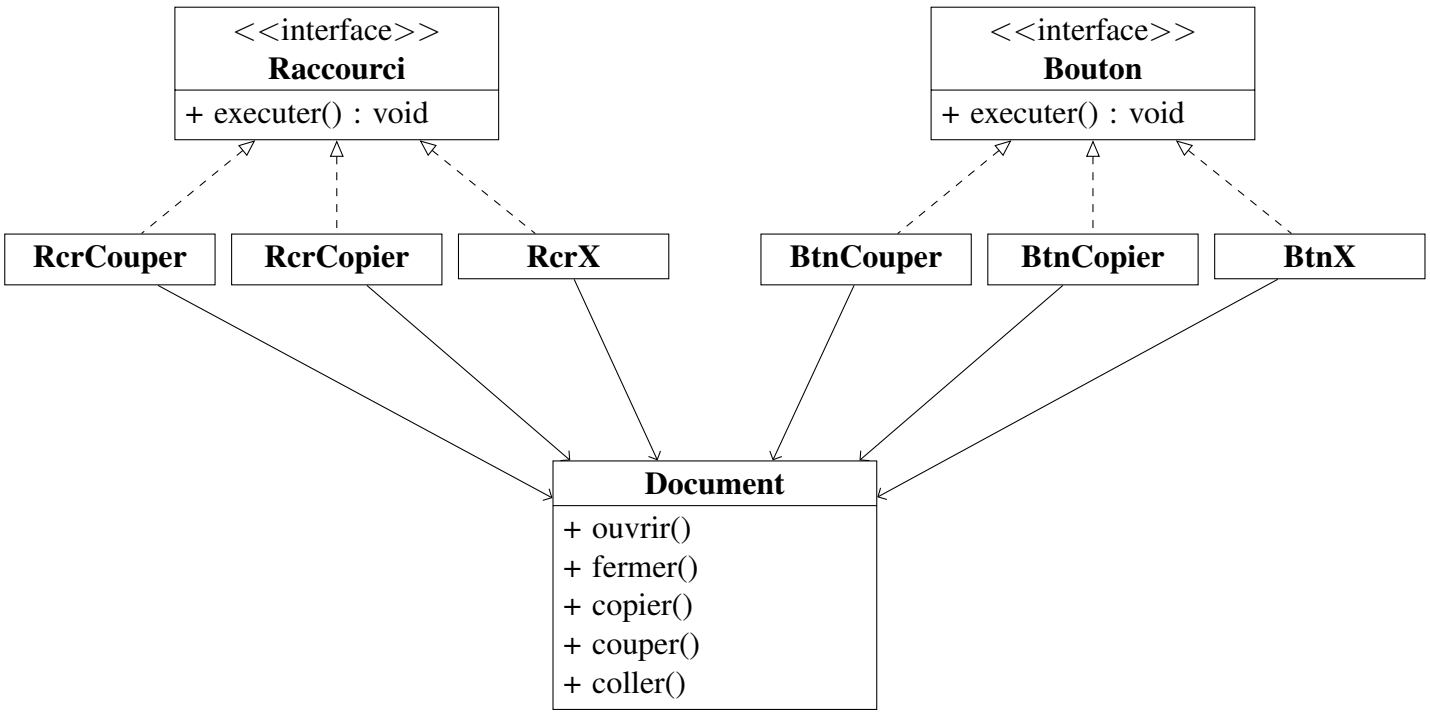
Nous souhaitons créer un éditeur de documents avec les fonctionnalités primaires copier, couper et coller une phrase dans le document. Comme dans chaque éditeur, ces actions sont réalisables à partir d'un bouton ou à partir d'un raccourci clavier.

2.2.1 Une première réponse

De cet énoncé nous pouvons déduire sept classes facilement :

- `BoutonCopier`, `BoutonCouper`, `BoutonColler`
  qu'on regroupera sous une interface commune `Bouton`
- `RaccourciCopier`, `RaccourciCouper`, `RaccourciColler`
  qu'on regroupera sous une interface commune `Raccourci`
- `Document` qui pourra être modifié par une action sur un bouton ou un raccourci

Ci-dessous nous présentons la solution envisagée



Mais cette solution possède un désavantage majeur :

- Pour chaque action (copier, couper ou coller) et pour chaque réalisateur (bouton ou raccourci) j'ai une classe concrète. Par conséquent dans un éditeur de document complet cela représente plusieurs centaines de classes à maintenir (produit cartésien de action et réalisateur).

2.2.2 Réfléchissons à une meilleure solution

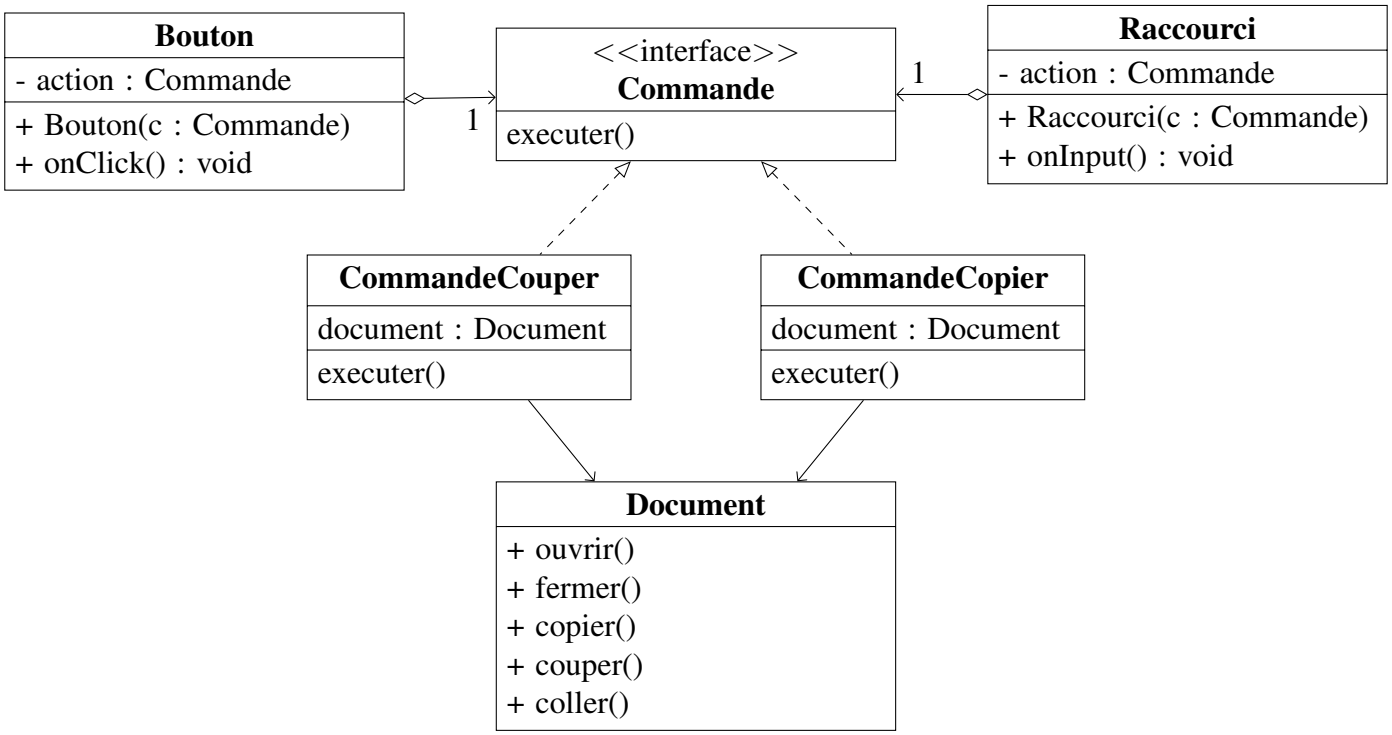
- Étant donné que l'impact de couper un mot dans le document est le même quelque soit l'invocateur (bouton ou raccourci) on pourrait factoriser le code de l'action. Une solution de factorisation est d'extraire les actions et d'en faire des classes autonomes. Ainsi pour chaque action nous associons une classe : `Couper`, `Copier` et `Coller`.

— Ces classes impactent de document, elles auront donc une association vers celui-ci.
— Et si ce sont ces classes qui impacte le document, alors nos classes `Bouton` et `Raccourci` non plus besoin de connaitre le document.

En soi, avec cette solution nous sommes en train de redéfinir le patron Commande.

2.2.3 Suggestion avec le patron Commande

— Nous rassemblons nos actions sous une même interface `Commande`.
— Le `Bouton` et le `Raccourci` connait l'action qu'il doit réalisée, car nous lui injectons la commande concrète (dans le `main()`)



Il n'y a qu'une seule classe `Bouton` et `Raccourci`. Pour dire quelle action sera effectuée par le bouton, nous devrons lui injecter une `Commande` lors de sa construction.

```
main() {
  CommandeCouper cCouper = new CommandeCouper();

  Bouton boutonCouper = new Bouton(cCouper); // injection de dépendance
  Raccourci raccourciCouper = new Raccourci(cCouper); // "..."

}
```

Le principal bénéfice de cette solution est la réduction du nombre de classes à maintenir. Maintenant nous aurons le nombre d'actions plus de nombre d'invocateurs au lieu d'en avoir le produit cartésien.

2.2.4 Code implémentant la conception

La classe BoutonCopier

```
private Commande commande;

public Bouton(Commande commande) {
   this.commande = commande;
}
public void onClick() {
   this.commande.executer()
}
```

La classe RaccourciCopier

```
private Commande commande

public RaccourciCopier(Commande commande) {
   this.commande = commande;
}
public void onInput() {
   this.commande.executer()
}
```

La classe CommandeCopier

```
private Document d;

public CommandeCopier(Document d) {
   this.document = d
}
public void executer() {
   /* Code àfaire lors d'une copie */
   document.getSelection();
   ...
}
```

Client

```
Document document = new Document();
CommandeCopier cCopier = new CommandeCopier(document);

Bouton boutonCopier = new Bouton(cc);
Raccourcir raccourciCopier = new Raccourci(cc);

/* On simule le bouton ou un CTRL+C */
boutonCopier.onClick();
raccourciCopier.onInput();
```

2.2.5 Conclusion de l'exemple

Le patron Commande permet la gestion de plusieurs invocateurs. Ainsi, si nous décidons de modifier le code d'une action nous ne modifierons que la Commande concernée sans toucher aux invocateurs. Nous respectons donc les principes *Responsabilité unique* et *Ouvert/Fermé*. En effet, il est très simple de rajouter un invocateur sans modifier les commandes.

2.3 Conclusion des exemples

Ces deux exemples ont montré plusieurs façons d'implémenter le patron Commande.

— Le premier exemple est le cas le plus classique. Mais s'il y a un grand nombre de `Commande`, il serait maladroit de toutes les passer en paramètre du constructeur.

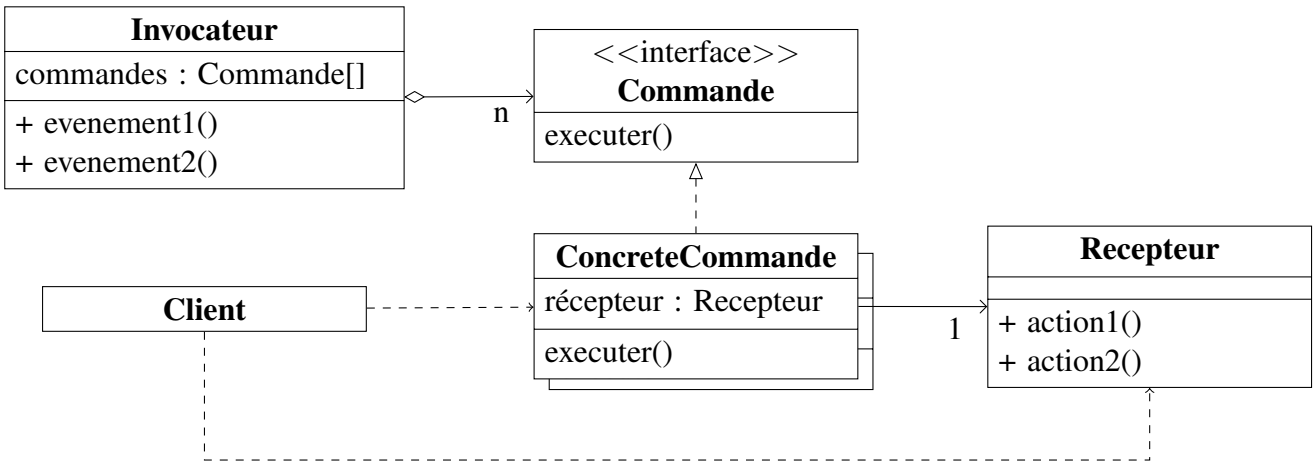
— Enfin, le dernier exemple montre qu'on peut avoir plusieurs invocateurs pour une même action. Cela permet en cas de changement du code de l'action (méthode `executer()`) de faire la modification à un seul endroit, nous y avons réduit le nombre de classes à maintenir en y appliquant le patron Commande.

3 Structure du patron Commande

Comme vous l'aurez compris à travers ces trois exemples, le patron Commande nécessite un ou plusieurs **invocateurs**, un **récepteur** et plusieurs **commandes concrètes** implémentant une interface commune.

3.1 Conception UML

Nous définissons ci-dessous la structure la plus classique. Mais comme nous l'avons vu dans les exemples précédents, elle peut être adaptée à vos besoins.



3.2 Participants

— **Commande** déclaration d'une interface pour exécuter les actions.

— **ConcreteCommande** définit une liaison entre un objet récepteur et une action. On invoque ainsi la bonne méthode `Recepteur`.

— **Client** crée les `ConcreteCommande` est fixe son `Recepteur`.

— **Invocateur** demande à la commande pour effectuer une action.

— **Récepteur** sait comment effectuer les opérations associées à l'exécution d'une requête . Toutes classes peuvent servir de récepteur.

3.2.1 Exemple d'un code Client

Ci-dessous l'exemple d'un code client avec une seule commande concrète.

```
Recepteur récepteur = new Recepteur();

ConcreteCommande cc = new ConcrereCommande(récepteur);
Invocateur invocateur = new Invocateur(cc);

invocateur.evenement1();
   --> this.commande.executer(); /* est appelé */
      --> this.recepteur.action1(); /* est appelé */
         --> /* action1() se déroule */
```