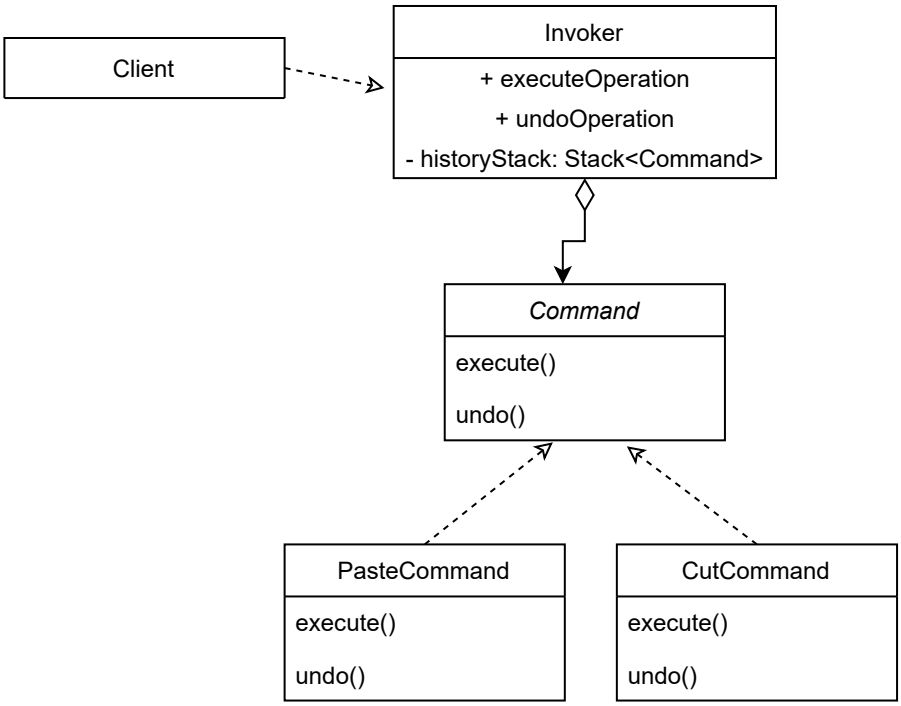
TP2 - Commande et Observateur

#DesignPattern

Commande

Le pattern commande consiste à isoler une requête ou un objet afin de gérer une file d'attente ou un historique de ces derniers sans pour autant être capable de les comprendre mais en étant capable de les annuler si les opérations sont reversibles. Le pattern commande permet également d'apporter un contexte pour l'invocation d'une méthode d'objet comme un objet à part entière. Pour cela nous allons encapsuler l'exécution d'un service seul dans un objet à part entière.




```
// l'interface commande est très simple et est applicable pour toutes
les commandes.

public interface Command {
    public void execute();
    public void undo();
}

/*************************************************************************/

public class CopyCommand implements Command {
    private String text;
```

```
    private Document document;
    private Cursor cursor;
    public CopyCommand(String text, Document document) {
        this.cursor = document.cursorPosition();
        this.document = document;
        this.text = text;
    }
    public void execute() {
        this.curseur = this.document.printStringAtCursor(this.cursor,
this.text).cursorPosition();
         // imaginons que cette methode ajoute le string  text à
l'endroit ou est notre curseur
         // et retourne la nouvelle position du curseur
    }
    public void execute() {
    this.cursor =
this.document.deleteAtCursor(this.cursor,this.text.length)
        // imaginons que cette methode supprime "x" caractères à
l'emplacement du curseur
        // et retourne la nouvelle position du curseur
    }
}
/*********************************************************************/

// on utilise un invoker qui va conserver l'historique des commandes et
va les exécuter
// c'est lui qui permet d'apporter le contexte d'exécution et c'est
avec lui que le client
// va discuter pour ajouter ses commandes et les faire exécuter.
public class Invoker {
    private final Stack<Command> historyStack = new Stack<Command>();

    public void executeOperation(Command currentCommand) {
        historyStack.push(currentCommand);
        currentCommand.execute();
    }

    public void undoOperation() throws Exception {
        if (historyStack.isEmpty()) {
            throw new Exception();
        } else {
            Command myCommand = historyStack.pop();
```

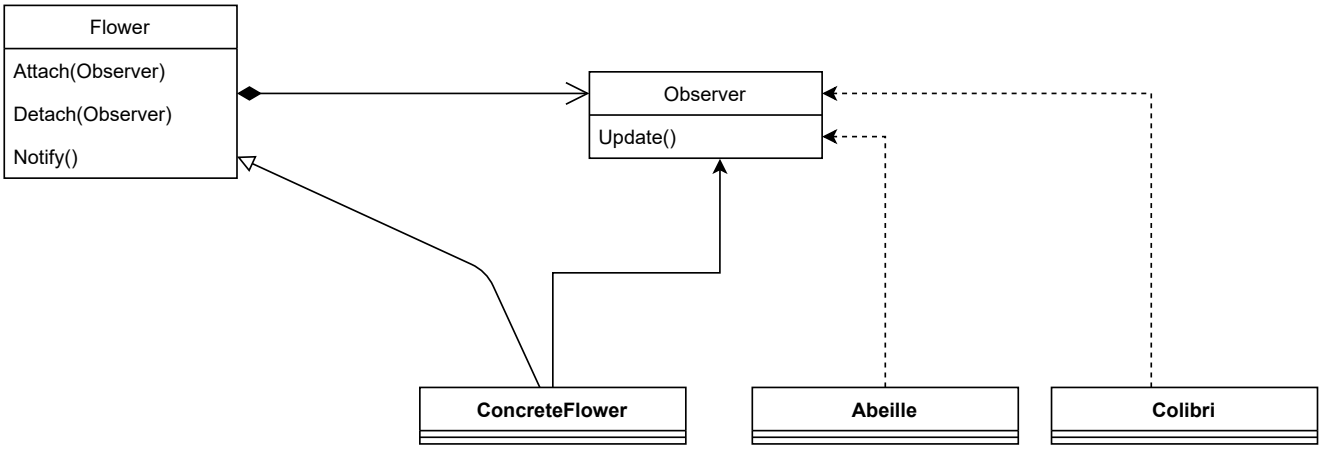
```
            myCommand.undo();
        }
    }
}

/*************************************************************************/


public class Main { // Client
    public static void main(String[] args) {
        // on récupère les variables document, cursor et
        Invoker commandHandler = new Invoker();

        commandHandler.executeOperation(new CopyCommand("bonjour",
document)); // on exec la commande
        try {
            commandHandler.undoOperation();
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println("Je ne peux pas faire undo si je n'ai
pas d'opération annulable");
        }
    }
}
```

On a alors un système de commande complètement fonctionnel, dans le cas ou l'on implémente pas de "undo" dans une commande non-annulable par exemple.

- Command déclare une interface pour exécuter une opération (en asynchrone)
- PasteCommand définit un lien entre notre objet receveur (ici un document non représenté dans l'UML mais bien existant) et une action(ici coller du texte) et implémente execute() pour invoquer les opérations du récépteur (écrire le texte au bon endroit dans le document)
- Le Main crée un object PasteCommand et assigne son récépteur (paramètres etc...)
- Invoker fait exécuter la requête à la commande et l'ajoute à son historique.
- Le document lui a subi les changements.

Observateur

L'observateur permet de généraliser la gestion d'événements à travers une seule classe. Cela permet de ne pas avoir de liens forts entre les différents objets et ainsi garantir une maintenabilité du code. Tous les événements passent par l'observateur qui informe ses sujets d'une modification sur l'élément observé.




```java
// nous allons créer une Fleur, qui sera notre objet mis à jour, il va implémenter
// Observable qui va permettre d'attacher des observateurs qui eux seront mis informés
// d'une mise à jour

import java.util.Observer;

public class Flower extends Observable {

    private boolean isBlooming;

    public Flower() {
        this.isBlooming = false;
    }

    public boolean isBlooming() {
        return this.isBlooming;
    }

    public void bloom() {
        this.isBlooming = true;
        setChanged();
        notifyObservers();
    }

    @Override
```

```
    @Override
    public void update(Observable obs, Object o) {
            this.bloom();
    }
}
/*****************************************************************************

// cette abeille va écouter les changements de la fleur et butiner si
la fleur
// se met à jour
package flower.observer;

import java.util.Observable;
import java.util.Observer;

import flower.Flower;

public class Abeille implements Observer {

    public void gatherPollen(Flower flower) {
        System.out.println("L'abeille butine une fleur.");
    }

    @Override
    public void update(Observable obs, Object o) {
        Flower flower = (Flower) obs;
        if (flower.isBlooming()) {

            this.gatherPollen(flower);
        }
    }
}
/*****************************************************************************

public class Main {
    public static void main(String[] args) {
            Abeille maya = new Abeille();
            System.out.println(">>> FLEURS <<<");
            for (int i = 0; i < 10; i++) {
                Flower f = new Flower();
                System.out.println(f);
                f.addObserver(maya);
```

```
            }
        }
}
```

- Flower connaît ses observateurs et fournit une interface pour en ajouter / retirer via l'implémentation de Observable.
- Abeille est une référence à un sujet, mémorise l'état qui doit rester cohérent avec celui du sujet et implémente l'inferface de mise à jour Observer pour assurer cette cohérence.