Introduction

Les Design patterns ou Patrons de Conception sont des solutions à des problèmes dans un contexte pour la POO.
Ce sont des architectures de classes ayant été testées et approuvées par de nombreux développeurs, faisant la part belle aux principes de développement orienté objet : faible couplage, fermée à la modification/ouvert à l'extension, encapsulation des données etc.
Suit ici une liste de design patterns non exhaustive, leur objectif, leur fonctionnement et leur diagramme UML.
Les définitions volées proviennent de l'ouvrage Tête la première sur les Design Patterns, très didactique, que je conseil.

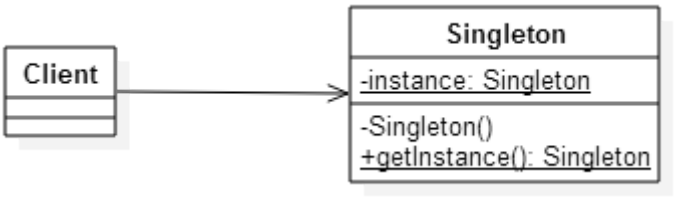
Singleton

Objectif

Le but du singleton est la création d'une classe dont une seule et même instance sera utilisée à travers tout le code.
C'est un pattern qui peut être utilisé pour les gestions de connexions.

Fonctionnement

Le singleton possède un constructeur privé, un attribut privé static contenant son instance et une méthode static qui renvoie l'instance si elle n'est pas null ou qui la créer si elle l'est.



Adapter (adaptateur)

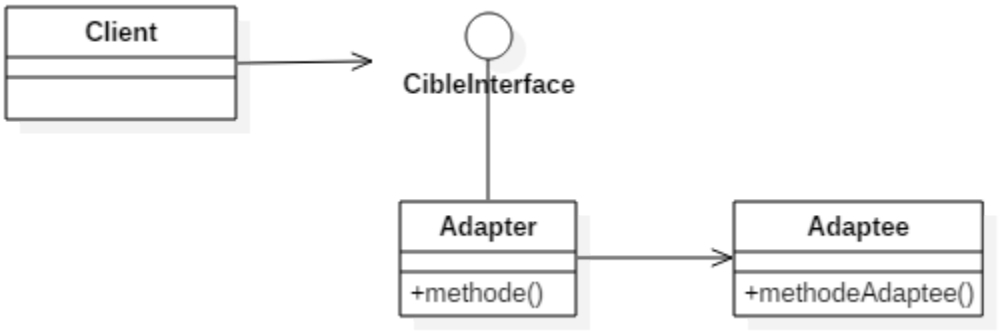
Objectif

Le but de l'Adapter est de permettre la modification de comportement d'un objet sans modifier l'objet en lui même.
C'est un pattern utile pour la création de patch et l'adaptation d'un code existant pour de nouveaux besoin.

Fonctionnement

L'adapter sera une classe implémentant éventuellement une interface similaire à l'adapté, et possédant l'adapté en attribut.
Sa ou ses méthodes consisteront à faire le lien et les modifications entre la classe adaptée et le nouveau comportement attendu.

définition volée : convertit l'interface d'une classe en une autre conforme à celle du client. L'Adaptateur permet à des classes de collaborer, alors qu'elles n'auraient pas pu le faire du fait d'interfaces incompatibles.



Simple Factory (simple fabrique)

Objectif

La simple fabrique n'est pas un design pattern à proprement parler, mais d'avantage un concept de programmation. Le but est de déléguer l'instanciation d'une classe à une autre Classe/méthode fabrique, afin de se passer du mot clef "new" et de pouvoir ainsi travailler avec des interfaces plutôt qu'avec des implémentations.

Fonctionnement

La simple factory possède une méthode, parfois static, qui instancie le type d'objet désiré, selon un argument ou autre.

En programmation orientée objet, il est dit qu'il vaut mieux travailler avec des interfaces qu'avec des implémentations. Ainsi, on pourrait dire que le mot clef "new" serait à limiter, car qui dit "new", dit implémentation, dit couplage etc.
Bien sûr, il faudra forcément instancier sa classe à un moment ou à un autre, car sans instance, pas de programme, mais c'est plus l'idée qu'il faut retenir qu'autre chose.
Nous allons donc voir les patterns fabriques qui permettent de se passer de "new" dans une certaine mesure.

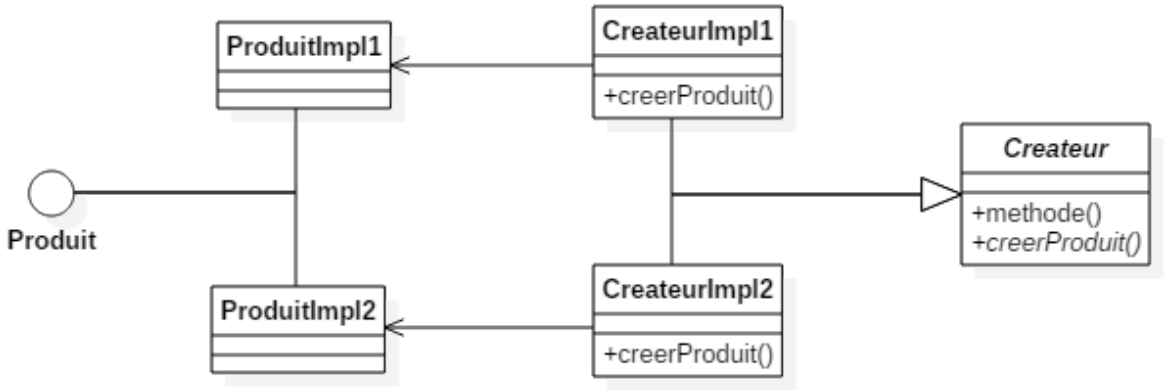
Factory Method (fabrication)

Objectif

Le but de la factory method est pour une classe créatrice de laisser l'instanciation d'une classe utilisée à ses sous classes

Fonctionnement
Une classe définit des méthode de manipulation d'un objet par son interface et possède une méthode abstraite permettant d'instancier l'objet en question. Les sous classes définiront quel type d'objet instancier.

définition volée : définit une interface pour la création d'un objet, mais en laissant aux sous-classes le choix des classes à instancier. Fabrication permet à une classe de déléguer l'instanciation à des sous-classes.

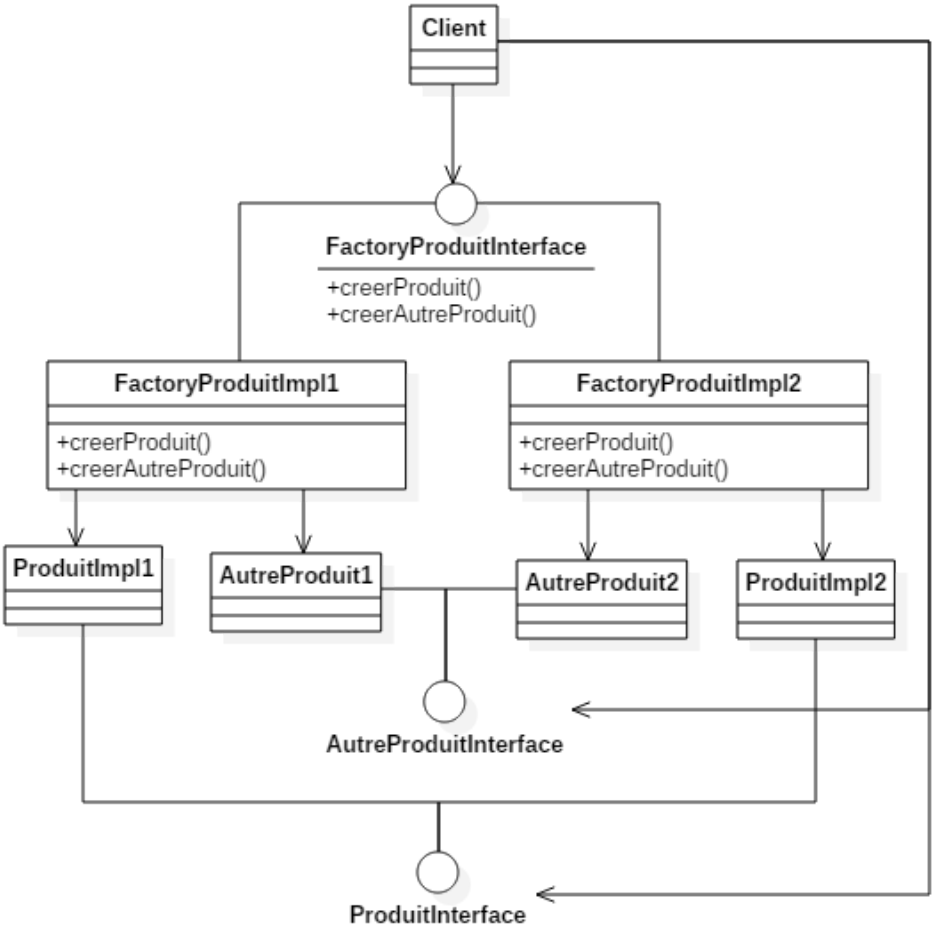


Abstract Factory (fabrique abstraite)

Objectif
La fabrique abstraite regroupe un ensemble de méthodes de créations pour des objets d'une même famille. Le client n'aura alors qu'à manipuler l'interface des objets en question et utilisera par exemple une Fabrication pour implémenter la fabrique requise.

Fonctionnement
L'abstract factory manipule des factory method. C'est une interface qu'implémenteront les factory concrètes utilisées.

définition volée : fournit une interface pour créer des familles d'objets apparentés ou dépendants sans avoir à spécifier leurs classes concrètes..



Decorator (Décorateur)

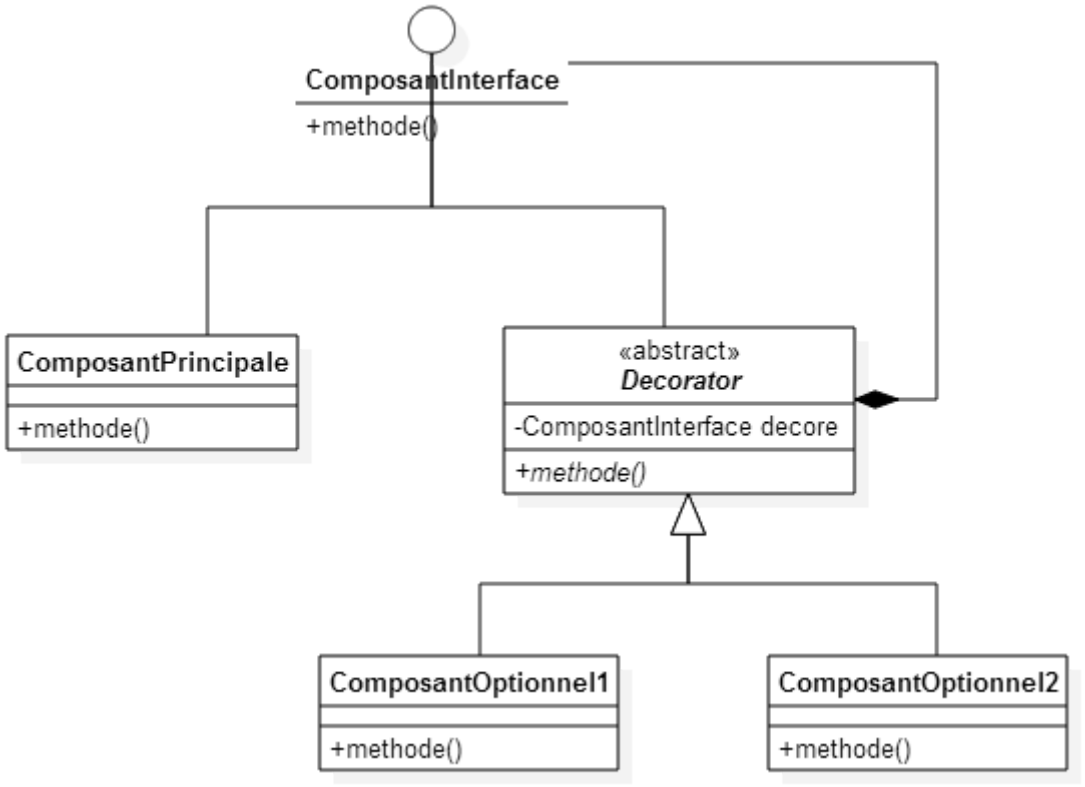
Objectif

Le decorator a pour objectif d'ajouter dynamiquement des comportement supplémentaire à un objet sans recourir à un héritage.

Fonctionnement

Le decorator ressemble dans sa conception à l'adapter mais est différent dans son objectif. Il possède le décoré en attribut et implémente la même interface/supertype.

définition volée : attache dynamiquement des responsabilités supplémentaires à un objet. Il fournit une alternative souple à la dérivation, pour étendre les fonctionnalités.



Facade

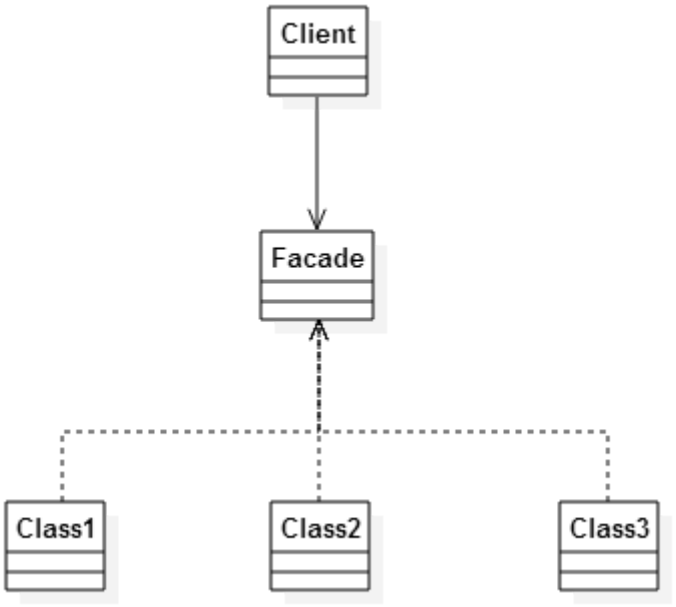
Objectif

La facade a pour but de fournir au client une interface simple à utiliser englobant un ensemble de sous classes.

Fonctionnement

La facade est une classe dépendant des classes qu'elle utilise comme attribut, et éventuellement d'une sous facade si cela s'avère pertinent.

définition volée : fournit une interface unifiée à l'ensemble des interfaces d'un sous-système. La façade fournit une interface de plus haut niveau qui rend le soussystème plus facile à utiliser.



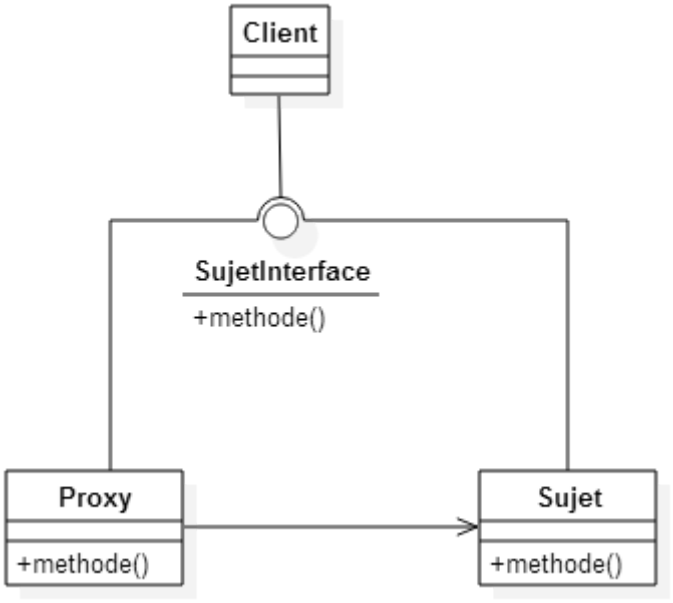
Proxy

Objectif

Le proxy a pour but de se substituer à une autre classe afin d'en contrôler l'accès de manière transparente pour le client.

Fonctionnement

Le proxy implémente la même interface que l'objet à substituer à qui il déléguera le traitement de l'opération.



Composite

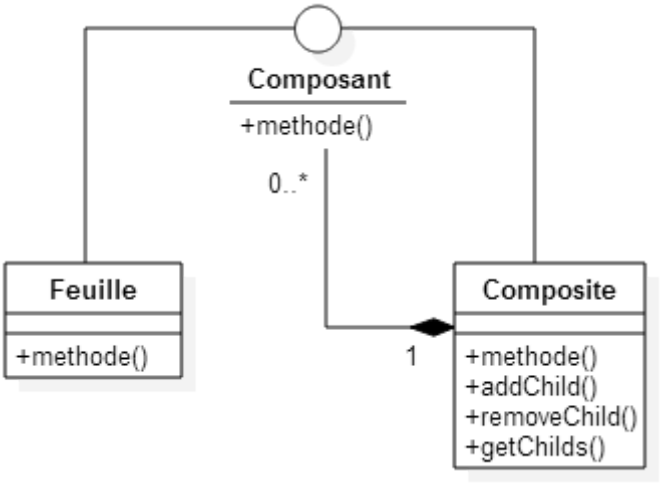
Objectif

Le pattern Composite permet d'avoir un ensemble récursif d'objets au comportement similaire qui pourront être traités par le client comme un seul objet.

Fonctionnement

Tous les objets du pattern implémentent une même interface/classe abstraite, les objets peuvent être conteneur ou non. Un objet conteneur aura une relation one-to-many avec son interface.

définition volée : compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet aux clients de traiter de la même façon les objets individuels et les combinaisons de ceux-ci.



State (état)

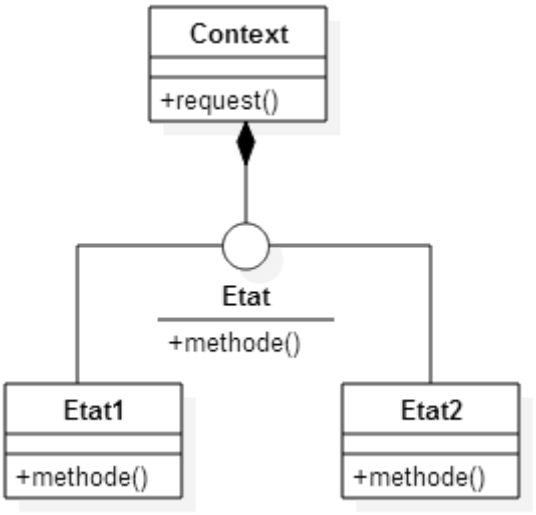
Objectif

Le pattern State permet de faire varier la comportement d'un objet selon le contexte dans lequel il est appelé.
C'est un pattern qui peut être utilisé dans les automates.

Fonctionnement

Une classe Contexte possède en attribut un objet de type Etat. L'interface / classe abstraite Etat est implémentée par les différents objets qui seront appelés par Contexte sous certaines conditions.
Le Contexte fera appel à son objet Etat à l'intérieur d'une ou plusieurs de ses méthodes, l'objet Etat appelé n'étant pas le même selon l'etat dans lequel se trouve le Contexte.

Définition volée : permet à un objet de modifier son comportement quand son état interne change. Tout se passera comme si l'objet changeait de classe.



Strategy (stratégie)

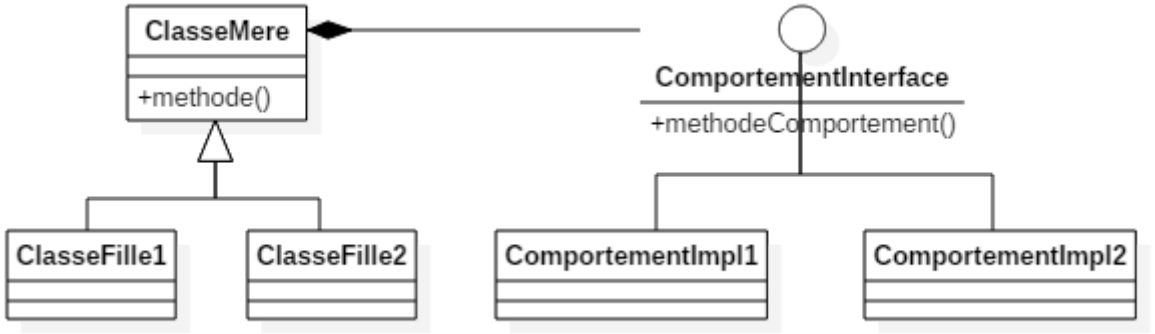
Objectif

Le but du pattern stratégie est de séparer les fonctionnalités des implémentations afin d'offrir un meilleure maintenabilité et souplesse que l'héritage tout en en conservant la réutilisabilité

Fonctionnement

Nous avons une classe dont héritent un ensemble de sous classes ayant donc des comportement commun mais une implémentation éventuellement différente de ces comportements. Plutôt que de définir les méthodes du comportement dans la classe mère, on lui fera une composition d'une interface ClasseComportement et l'on pourra ainsi définir l'implémentation voulu de cette interface selon la classe fille via le polymorphisme.

Définition volée : Le pattern Stratégie définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. Stratégie permet à l'algorithme de varier indépendamment des clients qui l'utilisent.



Oberver (Observateur)

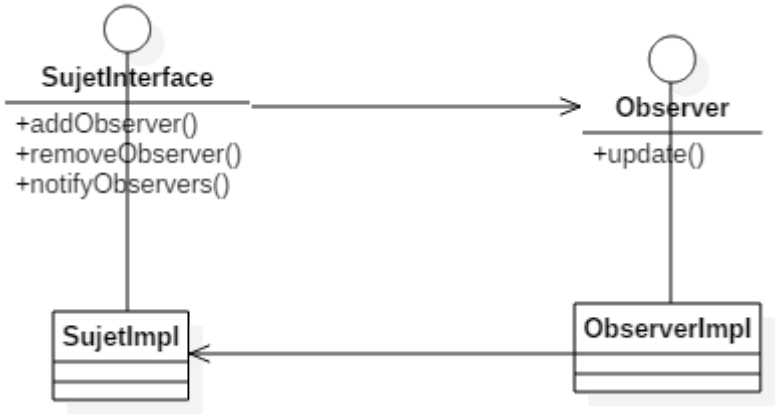
Objectif

Le but du pattern Observateur est de permettre à un ou plusieurs objet Observateur d'être notifiés / actualisés lors d'un changement d'un objet Sujet.

Fonctionnement

On définit une interface Sujet possédant les méthode ajouter/supprimer/notifier Observateur qu'implémentera notre classe Sujet (observée). On définit également une interface Observateur possédant uniquement la méthode actualiser(). Les implémentations d'Observateur seront composé d'un Sujet (afin qu'ils puissent se supprimer eux même de la liste des Observateurs si besoin).
L'implémentation de Sujet pourra alors faire appelle à sa méthode notifierObservateurs() au moment opportun, celle-ci pourra appeler la méthode actualiser() de chaque Observateur.

définition volée : définit une relation entre objets de type un-à-plusieurs, de façon que, lorsque un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.



Command (Commande)

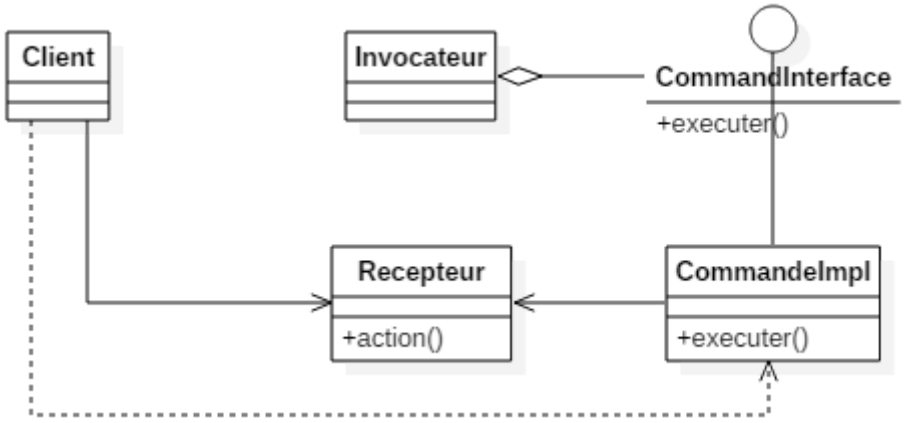
Objectif

Le but du pattern Command est d'encapsuler des requêtes comme un objet, permettant ainsi un faible couplage entre un appelant et un appelé. L'appelant n'aura pas à connaître les méthodes de l'appelé pour pouvoir y faire appel.

Fonctionnement

On crée d'abord une interface Command possédant une méthode executer(), lorsqu'elle est implémenter, cette méthode se chargera de faire appel à ou aux méthodes spécifiques de la classe appelée, ou Recepteur. Une classe Appelant est composée d'une ou plusieures Command. Le client se chargera de lié Recepteur et Command et de l'ajouter à l'Invocateur

définition volée : encapsule une requête comme un objet, autorisant ainsi le paramétrage des clients par différentes requêtes, files d'attente et récapitulatifs de requêtes, et de plus, permettant la réversibilité des opérations.



Template method (Patron de Méthode)

Objectif

Le but du patron de méthode est de créer une structure algorithmique dont certains éléments pourront être modifiés par des sous classe.

Fonctionnement

On créer une classe abstraite comportement une méthode qui contiendra la structure algorithmique voulue, chaque action de cet algorithme sera une méthode de la classe, qui pourra être abstraite ou non.

définition volée : définit le squelette d'un algorithme dans une méthode, en déléguant certaines étapes aux sous-classes. Patron de méthode permet aux sous-classes de redéfinir certaines étapes d'un algorithme sans modifier la structure de celui-ci.