-**Observer** : ajouter et supprimer comme bon nous semble l’un ne dépend pas de l’autre (=> plusieurs traitements en même temps). (TIPS : notifyAll)

-**Template Methode/Stratégie** : Permet d’éviter la redondance de codes.

Dans la stratégie on va donner à la classe **Context**, la stratégie de comment effectuer l’action.

Dans la template méthode, on définit une classe abstraite avec le code générale et une méthode abstraite qui va être appelé dans le code général de la classe abstraite, la méthode abstraite contient donc ce qui diffère entre les deux ou plusieurs appels. Ensuite ont faite une classe qui va redéfinir cette méthode méthode abstraite qui permettra de définir le code spécifique. La méthode « normale » dans la classe abstraite est appelé template method.

-**Abstract Facory** : Stratégie qui crée des objets(new)

-**Factory** **Method** : Template method qui crée des objets (new)

-**Composite** : combiner des objets similaire (par exemple 2 stratégies)

-**Decorator** : Permet d’ajouter du comportement à une classe sans toucher à la classe en question. Pour cela le decorator doit avoir un objet en attribut de la même class qu’il implémente (TIPS : suite de new dans le main Decorator(decorator(component))). Pour différencier avec le composite, dans le décorator on rajoute une ou des méthodes alors que pas dans le composite

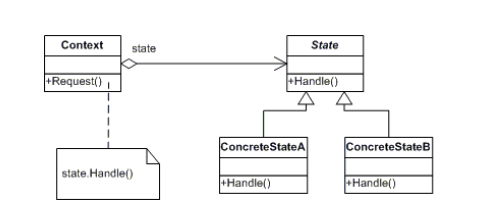
-**Adaptor** : L’adaptor à un Objet d’une autre classe, si on appel les méthodes de l’adaptor ça appel les méthodes de l’adaptee. Le client appelera les méthodes de l’adaptor.

-**State** : Machine a état.. remplace le code pourrie remplie de if, else

-> Enuméré => De base il y a le code général à tous les états de l’Enum qui contiendra le comportement par défaut qui tout état exécutera

-> Pour chaque état de l’énum on redéfinit le comportement de la méthode, si pour un état la méthode n’est pas redéfinie, ça sera la méthode de base qui sera appelé

-> Les méthodes qui doivent obligatoirement être définit dans chaque état contiennent le mot clé abstract



Public enum State {

State1, State2 {

@override

Method1()

}

Method1()

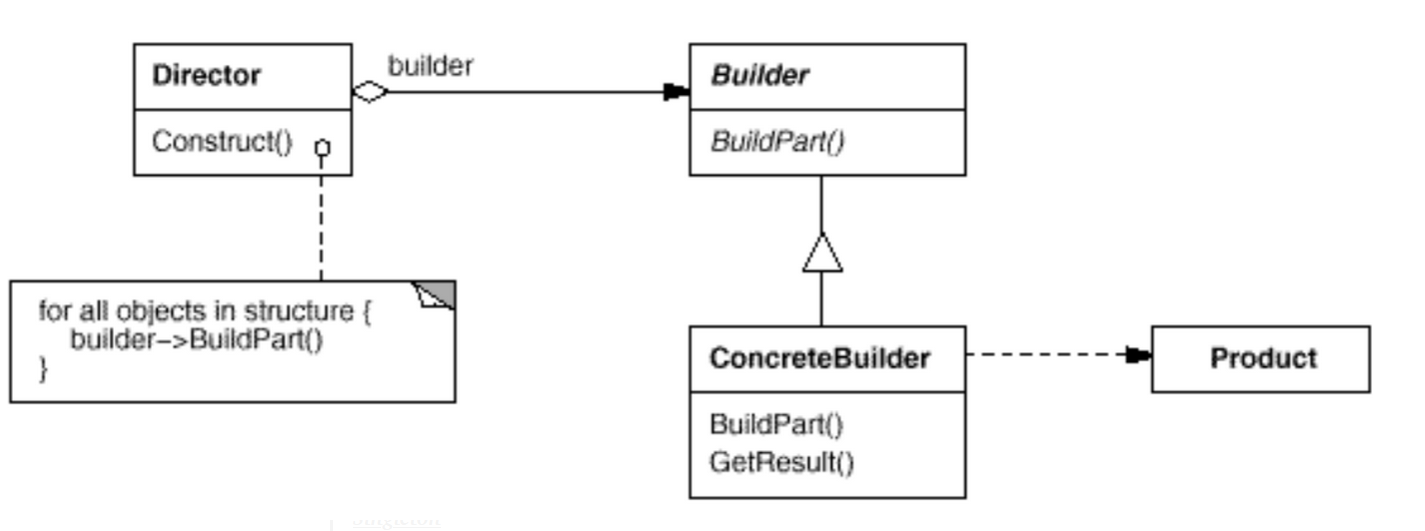
}

!!! passé aux méthodes dans l’enum l’objet (this). Toujours mettre une référence vers l’objet de départ.

-**Builder** : Permet de construire des objets d’une classe ayant certains paramètres obligatoires et d’autres optionnels (important : attributs finaux dans la classe et dans le builder uniquement ceux qui sont obligatoire, constructeur private, le build ne reçoit que les attributs requis dans son constructeur)

Les méthodes du builder s’écrive de tel manière : public NameBuilder name(string name) {this.name = name ; return this ;} // Il faut retourner le builder

La méthode build dans le builder permet de créer l’objet : public Name build() {return new Name(this) ;} // Le constructeur de la classe prend le builder comme ça il récupère les attributs du builder et modifie ces variables. Le classe de base ne contient que des getters.



-**Facade**: Cacher la complexité du code en utilisant qu’une seule classe dans la main (client) par exemple. Et cette clase (facade) va faire appel à toutes les autres classes.

-**Visitor :** Le client va avoir le Visitor et l’Element, il va permettre de passer à l’Element le visitor en question.

-**Command Pattern** : Permet de faire exécuter une action plus tard. Le client crée l’Invoker et le Receiver.

-**Flyweight**: permet d’éviter de recréer des objets qu’on utilise souvent et donc un gain de performance.

-**ChaineOfResponsaiblity** : La classe abstraite, donc le Handler va être appelé quand une classe dérivée n’arrive pas à gérer le nombre, elle va donc appeler la méthode pour que son successeur le fasse.