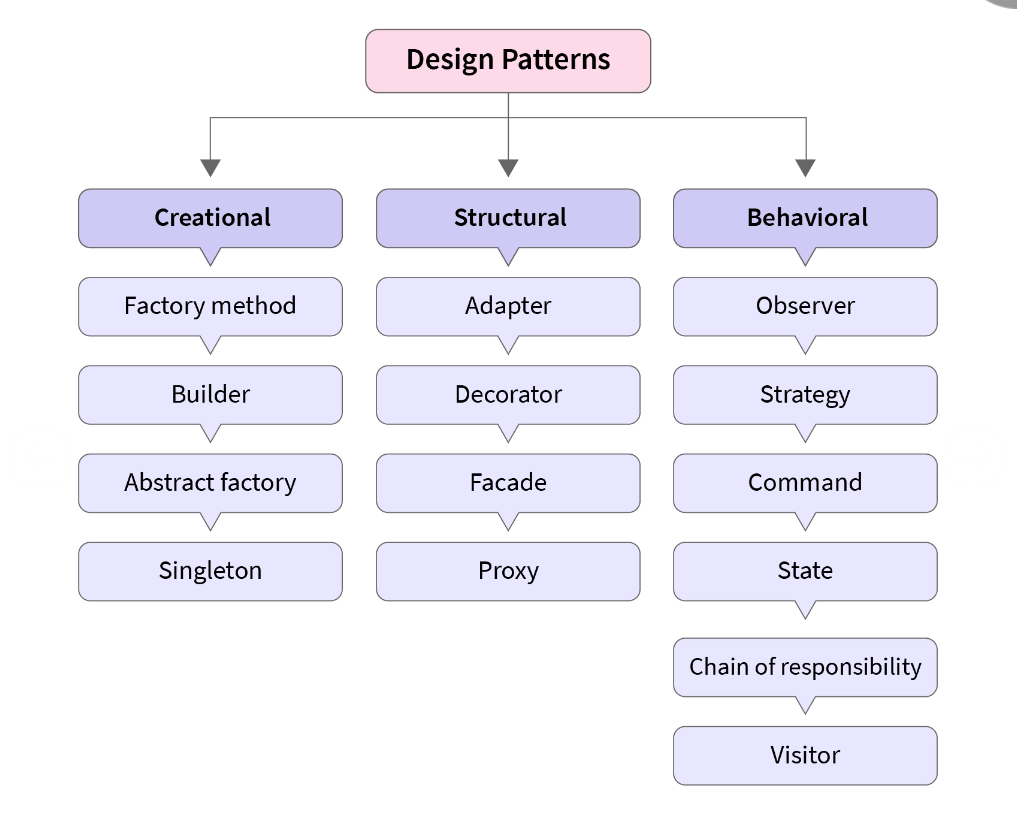
**RESUME DESIGN PATTERNS**

 les modèles de conception( ou design patterns en anglais) de logiciels sont des solutions au niveau de la conception d'un logiciel qui sont créées pour résoudre des problèmes répétitifs que les ingénieurs logiciels rencontrent souvent. **Les modèles de conception ne sont pas du code, mais plutôt des descriptions sur la résolution de certains problèmes logiciels et la conception d'une solution** . Les modèles de conception sont, en termes simples, des solutions de **problèmes récurrents** qui méritaient d'être écrits.

Notre objectif sera de comprendre l'utilisation et le but de chacun des modèles de conception, de choisir celui qui convient le mieux et de le mettre en œuvre comme solution au problème logiciel en question.

**TYPES DE DESIGN PATTERNS**



DOCUMENT : Les Différents types de design patterns

1. METHODE DE CONCEPTION CREATIONNEL

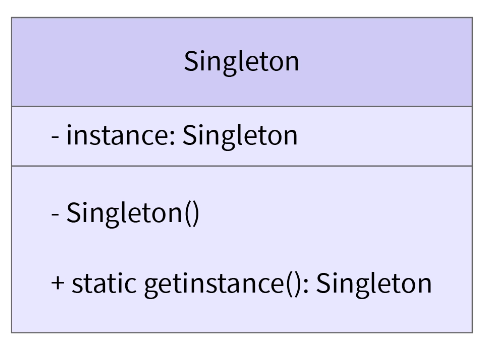
Comme leur nom l’indique, les modèles de création ont beaucoup à voir avec la création d’objets. Ces modèles sont conçus pour l'instanciation de classes. Les modèles de conception créationnels résout de nombreux problèmes de conception en côntrolant la création d’objets.

Ils sont plus utiles lorsque **des instances de plusieurs classes différentes** sont disponibles.

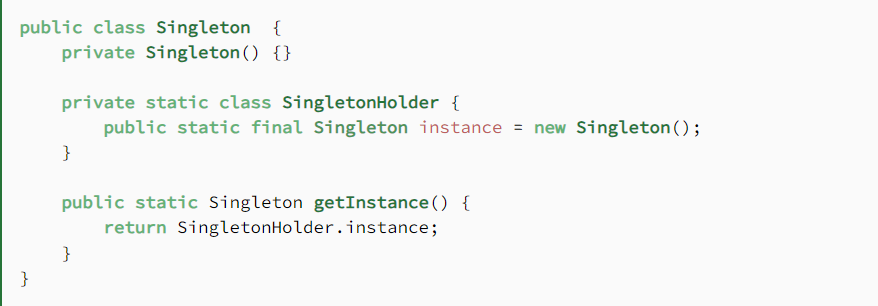
Comme exemples de ce modèle nous avons :

1. Singleton

Il est utile lorsque la coordination des actions à travers le système peut être effectuée par exactement un seul objet. Il garantit qu'une seule instance de la classe existe dans la machine virtuelle Java.



Document 1 : Exemple de classe d’un Singleton



Document 2 : Exemple de code d’un Singleton

**Quand utiliser le modèle de conception Singleton**

Pour les ressources coûteuses à créer (comme les objets de connexion à une base de données)

C'est une bonne pratique de conserver tous les enregistreurs en tant que Singletons, ce qui augmente les performances.

Classes qui donnent accès aux paramètres de configuration de l'application

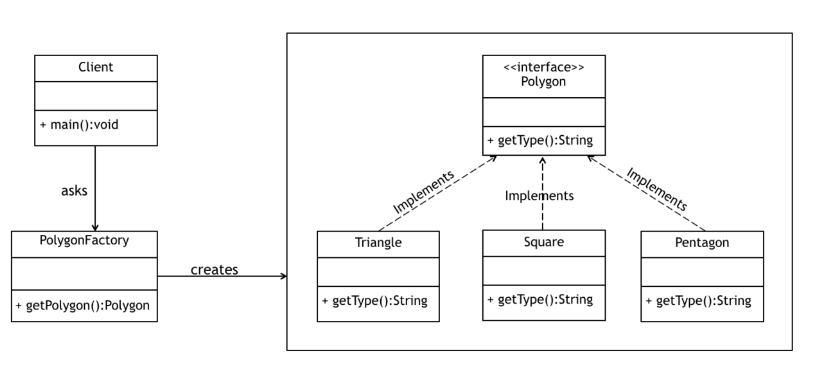
Classes contenant des ressources accessibles en mode partagé

Méthode factory ( Méthode d’Usine)

Le modèle de conception d'usine ou modèle de conception de méthode d'usine est l'un des modèles de conception les plus utilisés en Java.

Selon GoF, ce modèle « définit une interface pour créer un objet, mais laisse les sous-classes décider quelle classe instancier.La méthode Factory permet à une classe de différer l'instanciation aux sous-classes ».

Pour y parvenir, nous nous appuyons sur une usine qui nous fournit les objets, cachant les détails réels de mise en œuvre. Les objets créés sont accessibles à l'aide d'une interface commune.



Document 1 : Exemple de diagramme d’un Design Factory

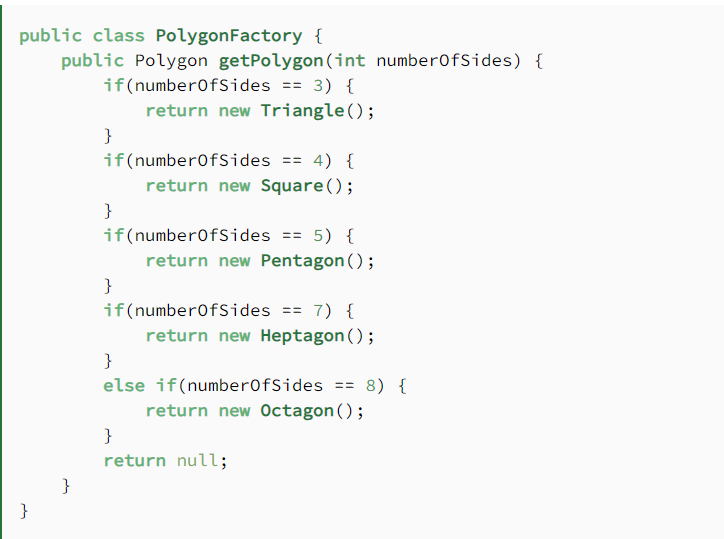
ETAPES DE CONCEPTION

1. Créons d'abord l' interface Polygon



Ensuite, nous créerons quelques implémentations comme Square , Triangle, etc. qui implémentent cette interface et renvoient un objet de type Polygon .

Nous pouvons maintenant créer une fabrique qui prend le nombre de côtés comme argument et renvoie l'implémentation appropriée de cette interface :



**Quand utiliser le modèle de conception de méthode d'usine**

Lorsque l'implémentation d'une interface ou d'une classe abstraite est censée changer fréquemment

Lorsque la mise en œuvre actuelle ne peut pas s'adapter confortablement à de nouveaux changements

Lorsque le processus d'initialisation est relativement simple et que le constructeur ne nécessite qu'une poignée de paramètres

**Abstract factory**

Dans la section précédente, nous avons vu comment le modèle de conception Factory Method pouvait être utilisé pour créer des objets liés à une seule famille.

En revanche, le modèle de conception Abstract Factory est utilisé pour créer des familles d’objets liés ou dépendants. On l'appelle aussi parfois une usine d'usines. Elle **fournit une interface pour créer des familles d’objets liés ou dépendants sans spécifier leurs classes concrètes.**

Autrement dit, ce modèle nous permet de créer des objets qui suivent un modèle général.

**Exemple de modèle de conception d'usine abstraite**

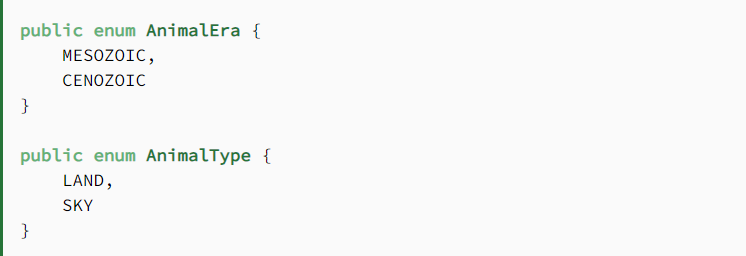
Dans cet exemple, nous allons approfondir le modèle Abstract Factory en utilisant un exemple Java impliquant des animaux préhistoriques.

Pour illustrer le modèle Abstract Factory, nous considérerons un scénario impliquant des animaux préhistoriques. Ces animaux peuvent être classés selon leur époque (Mésozoïque ou Cénozoïque) et leur type (animaux terrestres ou célestes). Nous allons créer un système qui nous permet de créer et de gérer ces animaux en utilisant le modèle Abstract Factory.

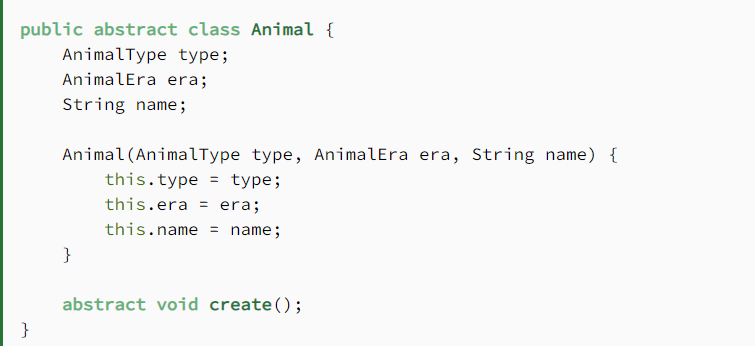
Après cela, nous gérerons leur accès à l'aide d'une fabrique abstraite AnimalAbstractFactory :

Composants du modèle d'usine abstraite

1.Nous commençons par définir deux énumérations, AnimalEra et AnimalType, pour représenter les époques (Mésozoïque et Cénozoïque) et les types d'animaux (terre et ciel).

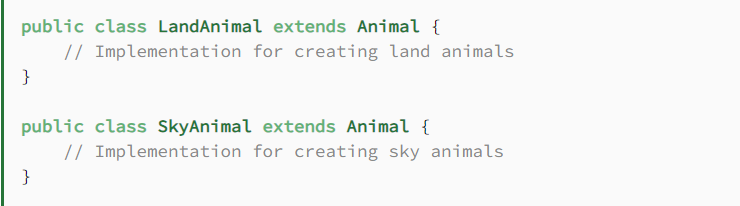


2.Ensuite, nous créons une classe abstraite Animal pour définir les attributs communs de nos animaux : leur époque, leur type et leur nom. Il comprend également une méthode abstraite create(), qui sera implémentée par des sous-classes concrètes.



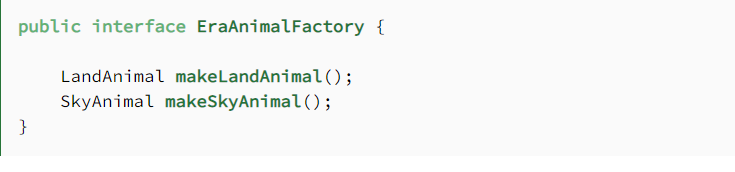
**Classes de produits en béton : LandAnimal et SkyAnimal**

Nous créons deux sous-classes concrètes d'Animal : LandAnimal et SkyAnimal. Ces classes fournissent des implémentations spécifiques pour créer respectivement des animaux terrestres et célestes.



**Interface d'usine EraAnimalFactory**

**4.Nous définissons une interface, EraAnimalFactory, qui déclare les méthodes de création d'animaux terrestres et célestes par époque. Les classes d'usine concrètes implémenteront cette interface pour créer des animaux d'époques spécifiques.**



**Classes d'usine à béton : CenozoicAnimalFactory et MesozoicAnimalFactory**

Nous créons deux classes d'usine concrètes, CenozoicAnimalFactory et MesozoicAnimalFactory, qui implémentent l'interface EraAnimalFactory. Chaque usine est chargée de créer des animaux d'une époque particulière.

Public class CenozoicAnimalFactory plements EraAnimalFactory {

// Implementation for creating Cenozoic animals

**public** **class** **MesozoicAnimalFactory** **implements** **EraAnimalFactory** { // Implementation for creating Mesozoic animals }

**Usine abstraite : AnimalAbstractFactory**

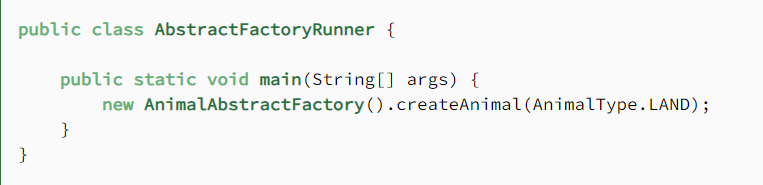
La classe AnimalAbstractFactory est chargée de créer des animaux en fonction de l’époque et du type fournis.



**Exécution de l'exemple**

Enfin, nous avons une classe client qui montre comment utiliser AnimalAbstractFactory pour créer des animaux préhistoriques en fonction de leur type. Le client ne connaît rien des usines intérieures, il demande juste un animal terrestre. Ce modèle offre un niveau supplémentaire d'abstraction.

}



## Quand utiliser le modèle d'usine abstraite****:****

Le client est indépendant de la façon dont nous créons et composons les objets dans le système

Le système se compose de plusieurs familles d'objets, et ces familles sont conçues pour être utilisées ensemble

Nous avons besoin d'une valeur d'exécution pour construire une dépendance particulière.

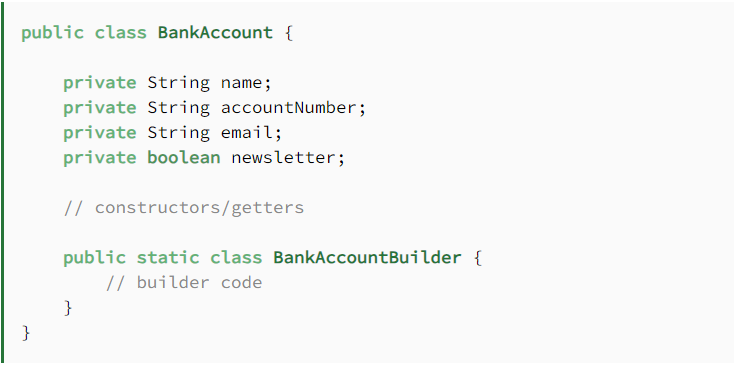
**Bien que le modèle soit idéal lors de la création d'objets prédéfinis, l'ajout de nouveaux objets peut s'avérer difficile .** Pour prendre en charge le nouveau type d'objets, il faudra modifier la classe AbstractFactory et toutes ses sous-classes.

1. **Generator objet**

Le modèle de conception Builder original introduit par GoF se concentre sur l'abstraction et est très efficace lorsqu'il s'agit d'objets complexes, cependant, la conception est un peu compliquée.

Joshua Bloch, dans son livre Effective Java, a présenté une version améliorée du modèle de construction qui est propre, hautement lisible (car il utilise une [conception fluide](https://en.wikipedia.org/wiki/Fluent_interface) ) et facile à utiliser du point de vue du client. Dans cet exemple, nous discuterons de cette version.

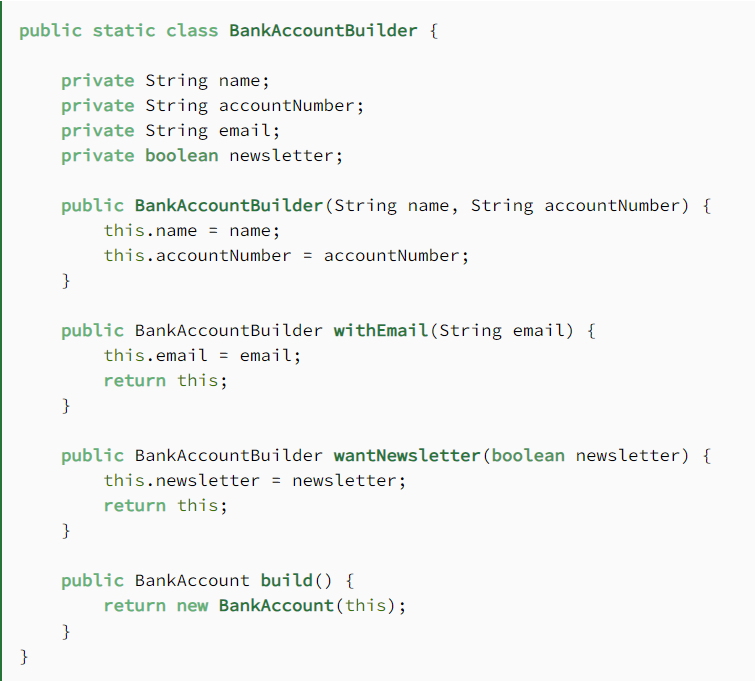
Cet exemple n'a qu'une seule classe, BankAccount , qui contient un générateur en tant que classe interne statique :



Notez que tous les modificateurs d'accès sur les champs sont déclarés privés puisque nous ne voulons pas que les objets externes y accèdent directement.

Le constructeur est également privé afin que seul le Builder affecté à cette classe puisse y accéder. Toutes les propriétés définies dans le constructeur sont extraites de l'objet constructeur que nous fournissons en argument.

Nous avons défini BankAccountBuilder dans une classe interne statique :



Enfin, la méthode build appelle le constructeur privé de la classe externe et se passe en argument. Le BankAccount renvoyé sera instancié avec les paramètres définis par BankAccountBuilder .

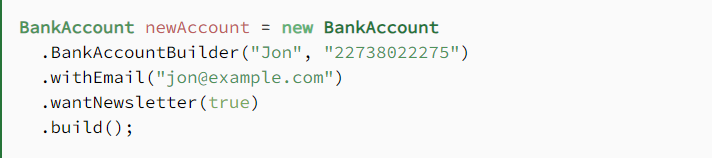
Voyons un exemple rapide du modèle de générateur en action :

### ****Quand utiliser le modèle de générateur****

Lorsque le processus de création d'un objet est extrêmement complexe, avec de nombreux paramètres obligatoires et facultatifs

Lorsqu'une augmentation du nombre de paramètres du constructeur conduit à une grande liste de constructeurs

Lorsque le client attend des représentations différentes pour l'objet construit



1. Prototype

Le modèle prototype est utilisé lorsque la Objectcréation est coûteuse et nécessite beaucoup de temps et de ressources, et que vous disposez Objectdéjà d’un modèle similaire. Ce modèle fournit donc un mécanisme pour copier l'original Objectvers un nouveau Object, puis le modifier en fonction de nos besoins.

1. Builder
2. METHODE DE CONCEPTION STRUCTURELLE

Les modèles de conception structurelle proposent différentes manières de créer une **Classstructure** (par exemple, en utilisant l'héritage et la composition pour créer un grand**Object**à partir d'un **petit Objects**).

1. Decorator Design

Le modèle de conception du décorateur est utilisé pour modifier la fonctionnalité d'un objet au moment de l'exécution.  Le modèle de conception du décorateur est l'un des modèles de conception structurels (tels qu'un modèle d'adaptateur, un modèle de pont ou un modèle composite) et utilise des classes abstraites ou une interface avec la composition à implémenter.

1. Command Design

Un modèle de conception comportementale se concentre sur la façon dont les classes et les objets communiquent entre eux. L’objectif principal du modèle de commandement est d’inculquer un degré plus élevé de couplage lâche entre les parties impliquées (lire : classes).

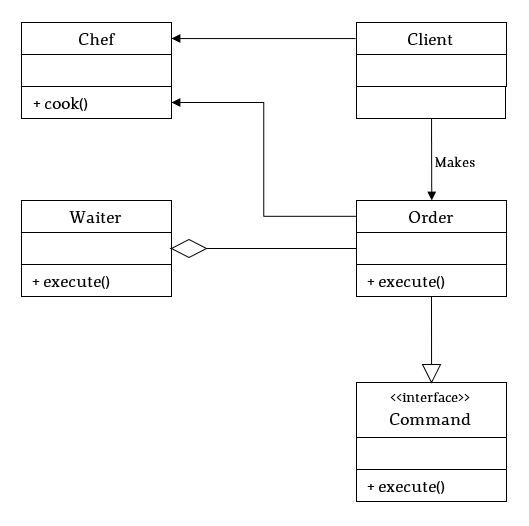
Le couplage est la manière dont deux (ou plusieurs) classes qui interagissent les unes avec les autres interagissent.Le scénario idéal dans lequel ces classes interagissent est qu’elles ne dépendent pas fortement les unes des autres. C'est un couplage lâche.

 Ainsi, une meilleure définition du couplage lâche serait celle des classes interconnectées, utilisant le moins les unes des autres.

Dans ce modèle, la classe invocatrice est découplée de la classe qui exécute réellement une action. La classe invocatrice n'a que la méthode appelable exécutée, qui exécute la commande nécessaire, lorsque le client le demande.

Prenons un exemple concret : commander un repas dans un restaurant chic. Au fur et à mesure, vous donnez votre commande (commande) au serveur (invocateur), qui la remet ensuite au chef (récepteur), afin que vous puissiez vous procurer de la nourriture. Cela peut paraître simple… mais un peu compliqué à coder.

L'idée est assez simple, mais le codage fait le tour du nez.



**Document : Diagramme de classes de modèle de conception de commande**