Design Patterns

Introduction aux Design Patterns

Des solutions éprouvées pour une conception logicielle intelligente.

Qu’est-ce qu’un Design Pattern ?

- **Définition simple** Une solution réutilisable à un problème courant dans la conception logicielle. C’est un modèle éprouvé, adaptable à divers contextes.
- **Définition formelle** Une solution générale à un problème récurrent, utilisée pour structurer le code de façon optimale, maintenable et évolutive.

> *"A general, reusable solution to a commonly occurring problem in many contexts in software design." – Wikipedia*

Pourquoi utiliser des Design Patterns ?

Ils sont des atouts majeurs pour la qualité et l’efficacité de vos projets logiciels :

- **Réutilisabilité** Évite de "réinventer la roue" en fournissant des solutions validées.
- **Communication** Crée un vocabulaire commun, facilitant les échanges entre développeurs.
- **Maintenance & Évolutivité** Simplifie la gestion et l’adaptation future du code.
- **Qualité de conception** Encourage les bonnes pratiques de programmation et une architecture claire.

Exemple concret : Le Pattern Singleton

Le Singleton est un pattern fondamental qui garantit :

- **Instance unique** Une classe ne possède qu'une seule instance tout au long de l'exécution.
- **Accès global** Il fournit un point d'accès unique et global à cette instance.
- **Utilité** Idéal lorsqu'une ressource partagée unique est nécessaire (ex: connexion à une base de données, gestionnaire de configuration).

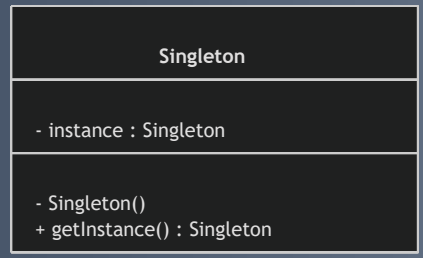
Singleton : Comment ça marche ?

Le principe est de contrôler l'instanciation de la classe.

Pseudo-code :

```
class Singleton {
    private static instance = null

    private constructor() {
        // Le constructeur est privé pour empêcher
        // toute instanciation externe directe.
    }

    public static getInstance() {
        if (instance == null) {
            instance = new Singleton()
        }
        return instance
    }
}
```

Structure illustrée (Diagramme de Classes) :



Ce diagramme montre que `Singleton` contient une instance privée statique et un constructeur privé, avec une méthode publique `getInstance()` pour accéder à l'instance unique.

Ce qu’il faut retenir & Pour aller plus loin

L’essentiel :

Un design pattern est un modèle de solution standard et éprouvé qui s’applique à un problème récurrent dans la conception logicielle. Il facilite la communication entre développeurs, améliore la qualité du code et accélère le développement.

Sources pour approfondir :

1. Wikipedia – Software design pattern
2. DZone - What Is a Design Pattern?
3. Refactoring.Guru - What’s a design pattern?

Les Design Patterns

Solutions aux Problèmes de Conception Récurrents

Comprendre l’utilité des Design Patterns

Les Design Patterns sont nés du besoin d’offrir des solutions éprouvées à des problèmes récurrents dans la conception logicielle. Sans ces solutions, le code peut devenir difficile à maintenir, peu flexible et inefficace.

Ils répondent à des défis fondamentaux :

- **Gestion des objets et contrôle de leur création**
- **Organisation et structuration des objets**
- **Communication entre objets**
- **Gestion du comportement des objets**

1. Gérer la Création des Objets

Problème : Instancier des objets de manière complexe ou coûteuse

La création d’objets peut être délicate : configuration spécifique, instance unique nécessaire, ou coût d’instanciation élevé.

Patterns associés :

- **Singleton :** Assure qu’une classe n’a qu’une seule instance.
- **Factory Method :** Définit une interface pour créer un objet, laissant les sous-classes décider de la classe à instancier.
- **Builder :** Construit des objets complexes étape par étape.

Exemple simple : Factory Method

```
interface Product {
    operation()
}

class ConcreteProductA implements Product {
    operation() { print("Produit A") }
}

class Creator {
    abstract factoryMethod() : Product

    someOperation() {
        product = factoryMethod()
        product.operation()
    }
}

class ConcreteCreatorA extends Creator {
    factoryMethod() : Product {
        return new ConcreteProductA()
    }
}
```

2. Structurer les Objets avec Flexibilité

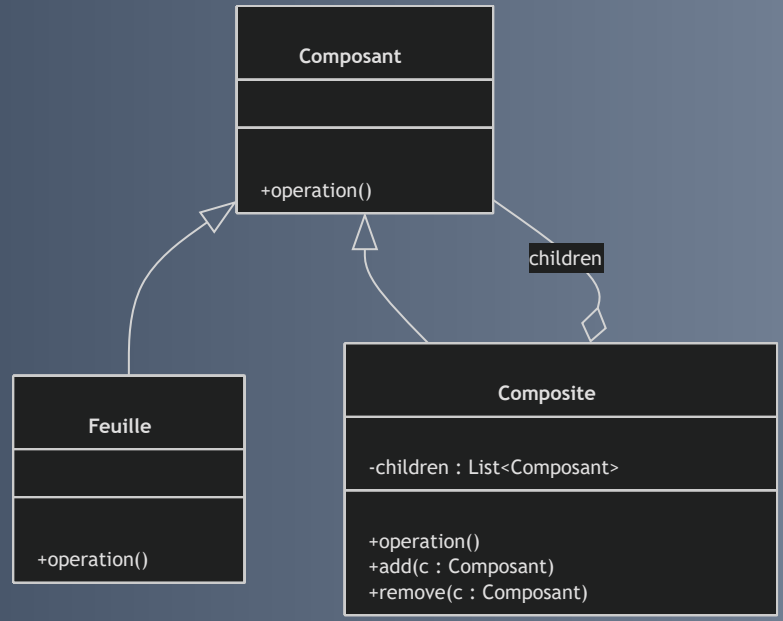
Problème : Organiser des objets en structures complexes

Il est souvent complexe de manipuler des groupes d’objets de manière uniforme sans rendre le système confus.

Patterns associés :

- **Composite :** Compose des objets en structures arborescentes pour représenter des hiérarchies partie-tout. Permet de traiter des objets individuels et des groupes d’objets de manière uniforme.
- **Decorator :** Ajoute dynamiquement des responsabilités à un objet sans modifier son interface ni sa structure de classe existante.

Diagramme – Composite :



3. Optimiser la Communication et le Comportement

Problème 1 : Réduire la dépendance entre objets (Communication)

Dans les systèmes complexes, les objets doivent communiquer sans créer de couplage fort, ce qui entraverait la maintenance et l'évolution.

- **Observer :** Un objet (sujet) notifie automatiquement ses dépendants (observateurs) en cas de changement d'état.
- **Mediator :** Encapsule et gère la communication complexe entre objets, évitant les liens directs.

Problème 2 : Gérer les algorithmes et comportements interchangeables

Changer le comportement d'un objet ou l'algorithme utilisé sans modifier le code client.

- **Strategy :** Définit une famille d'algorithmes, les encapsule individuellement et les rend interchangeables.
- **State :** Permet à un objet de modifier son comportement en fonction de son état interne.

Synthèse : Les Design Patterns en un Coup d’Œil

Problème	Patterns Associés	Objectif Clé
Gestion de la création	Singleton, Factory Method, Builder	Contrôler précisément la création des objets
Structuration des objets	Composite, Decorator	Organiser les objets en structures flexibles et étendues
Communication entre objets	Observer, Mediator	Réduire le couplage et faciliter les échanges
Gestion du comportement	Strategy, State	Interchanger des comportements sans modifier le code

Ce qu’il faut retenir & Sources

L’Essentiel sur les Design Patterns

Les Design Patterns sont des solutions éprouvées aux défis récurrents de la conception logicielle. En les utilisant, les développeurs peuvent créer des systèmes plus robustes, flexibles et maintenables. Chaque pattern adresse un problème spécifique, offrant une approche standardisée pour améliorer la qualité du code.

Sources d’approfondissement

- Refactoring.Guru – Design Patterns et problèmes résolus
- Wikipedia – Design Pattern
- DZone – Introduction to Design Patterns

Introduction aux Design Patterns

Qu’est-ce qu’un Design Pattern ?

Les Design Patterns sont des solutions génériques, testées et éprouvées à des problèmes récurrents dans la conception logicielle. Ils ne sont pas du code directement utilisable, mais des modèles pour résoudre des défis architecturaux spécifiques.

Pourquoi les utiliser ?

L’adoption des design patterns apporte plusieurs avantages essentiels à la qualité et à la robustesse du développement logiciel. Ils améliorent la manière dont le code est conçu, écrit et maintenu, en facilitant la collaboration et l’évolution des applications.

Avantages clés :

- **Réutilisabilité**
- **Maintenabilité**
- **Flexibilité**
- **Communication**

Réutilisabilité : Éviter de Réinventer la Roue

Les design patterns fournissent des solutions génériques, testées et éprouvées, qui peuvent être adaptées à différents projets et contextes. Cela évite de repartir de zéro pour chaque problème récurrent.

Exemple : Le Pattern Factory Method

Le pattern **Factory Method** permet de centraliser la création d'objets grâce à une interface commune, ce qui rend le code plus réutilisable et extensible sans modifier les classes clientes.

Permet de créer des objets sans spécifier leur classe exacte, facilitant l'évolution du système.

Maintenabilité : Code Propre et Évolutif

En standardisant les solutions, les design patterns rendent le code plus lisible et bien structuré. Ils aident à découpler les composants, ce qui facilite la correction des bugs et l'ajout de nouvelles fonctionnalités sans risque de provoquer des effets de bord.

Exemple : Le Pattern Observer

Le pattern **Observer** permet de gérer proprement la communication entre objets en évitant un couplage serré, ce qui simplifie la modification ou la suppression d'observateurs sans impacter les autres parties du système.

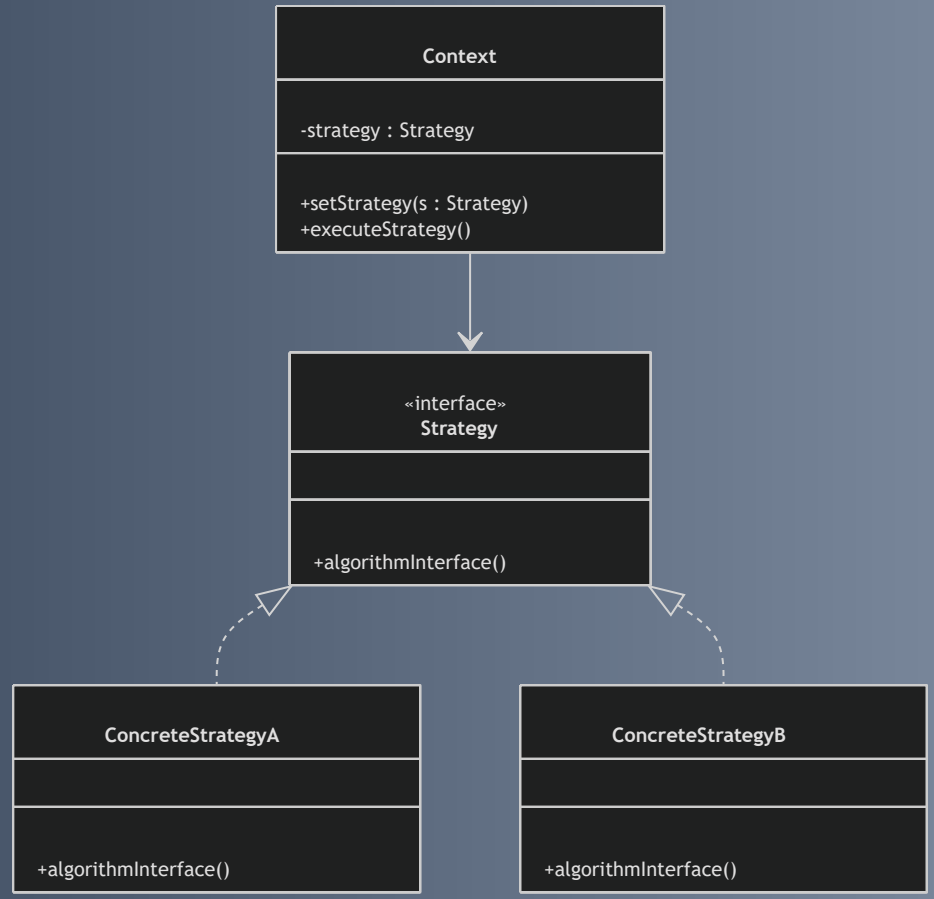
Les changements dans une partie du système n'affectent pas directement les autres, réduisant les risques d'erreurs.

Flexibilité : Architectures Modulaires et Adaptables

Les design patterns favorisent des architectures modulaires où il est facile de remplacer ou modifier des éléments sans changer l'ensemble du programme. Ils encouragent la programmation orientée interface plutôt que sur des implémentations concrètes.

Exemple : Le Pattern Strategy

Le pattern **Strategy** encapsule des algorithmes interchangeables, permettant de changer le comportement d'un objet à la volée.



Le comportement d'un objet peut être modifié dynamiquement sans altérer son code.

Communication Améliorée : Un langage commun

Utiliser un vocabulaire commun issu des design patterns facilite la communication entre développeurs, architectes et autres parties prenantes du projet. Reconnaître et nommer un pattern dans le code permet de comprendre rapidement sa structure et son comportement.

Exemple : Le Pattern Decorator

Dire qu'un système utilise le pattern **Decorator** indique immédiatement que des fonctionnalités peuvent être ajoutées dynamiquement sans modifier les classes existantes.

Un vocabulaire partagé simplifie la conception, la revue de code et la documentation.

Synthèse des Bénéfices & Ressources

Ce qu’il faut retenir

Ce tour d’horizon met en lumière l’impact direct des design patterns sur la qualité du code, son maintien dans le temps, et la collaboration entre équipes, renforçant ainsi la robustesse et la souplesse des applications développées.

Récapitulatif des avantages

Avantage	Description	Exemple de Pattern
Réutilisabilité	Solutions génériques adaptables à plusieurs contextes	Factory Method, Singleton
Maintenabilité	Code lisible, découplé, facile à modifier	Observer, Command
Flexibilité	Architecture modulaire, comportements interchangeables	Strategy, State
Communication	Vocabulaire commun, facilitation de la compréhension	Decorator, Composite

Sources

- Refactoring.Guru – Advantages of Design Patterns
- Wikipedia - Software design pattern
- DZone – Benefits of Using Design Patterns

Introduction au Gang of Four (GoF)

- **Qui sont-ils ?**
  - Un groupe de quatre informaticiens :
    - Erich Gamma
    - Richard Helm
    - Ralph Johnson
    - John Vlissides
  - Ils ont révolutionné l’architecture logicielle moderne.
- **Leur ouvrage fondateur :**
  - "Design Patterns: Elements of Reusable Object-Oriented Software"
  - Publié en 1994 aux éditions Addison-Wesley.
  - A formalisé un catalogue de solutions de conception réutilisables.

L’Ouvrage GoF : Une Contribution majeure

- **Première référence exhaustive :**
  - Décrit et explique 23 design patterns clés.
  - Résout des problèmes récurrents en programmation orientée objet.

- **Apports clés du livre :**
  - **Terminologie commune** pour décrire structures et comportements.
  - **Classification claire** selon le domaine d’action (création, structure, comportement).
  - **Exemples détaillés** en C++ et Smalltalk.
  - **Méthodologie** pour appliquer efficacement ces patterns.

Classification des Patterns GoF

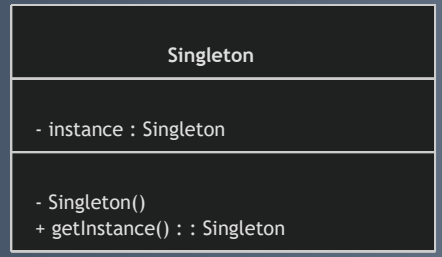
Les 23 GoF Design Patterns sont regroupés en trois catégories principales :

Catégorie	Description	Exemples
Création	Gestion de la création d’objets	Singleton, Factory Method, Builder, Prototype, Abstract Factory
Structure	Organisation des classes et objets	Adapter, Composite, Decorator, Facade, Flyweight, Proxy
Comportement	Interaction et responsabilités entre objets	Observer, Strategy, Command, State, Iterator, Mediator, Visitor, Template Method

Exemple : Le Pattern Singleton

- **Définition :**
  - Forme classique pour restreindre l'instanciation d'une classe à une seule instance.
  - Garantit un point d'accès global à cette instance unique.

- **Structure classique :**



- L'ouvrage du GoF présente également des variantes pour la création sûre dans des contextes multithread.

Impact Durable des Patterns GoF

- **Démocratisation des design patterns :**
  - A introduit une approche structurée de la conception logicielle.
- **Influence majeure sur :**
  - Le développement de nombreux frameworks et bibliothèques logicielles.
  - Les pratiques architecturales dans les langages orientés objet modernes (Java, C#, Python, etc.).
  - La pédagogie informatique, où la connaissance des patterns est devenue un socle fondamental pour les développeurs.

Ce qu’il faut retenir & Ressources

Ce qu’il faut retenir :

- L’apport du Gang of Four réside autant dans la formalisation que dans la diffusion d’un savoir-faire structurant la conception logicielle.
- Leur ouvrage reste une référence majeure pour comprendre et appliquer les design patterns au quotidien.

Sources :

- Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- Wikipedia – Design pattern
- Refactoring.Guru – GoF Design Patterns

Historique et Classification des Design Patterns

- Les **Design Patterns**, popularisés par le **Gang of Four (GoF)**, sont des solutions éprouvées à des problèmes de conception récurrents en programmation orientée objet.
- Ils sont classés en **trois catégories majeures** qui regroupent les solutions selon le type de problème qu'elles visent à résoudre :
  - **Création**
  - **Structure**
  - **Comportement**

1. Patterns de Création : Gérer l’Instanciation

- **Objectif principal** :
  - Gérer la création d’objets d’une manière flexible et adaptée au contexte.
  - Encapsuler la logique d’instanciation et réduire le couplage entre classes.

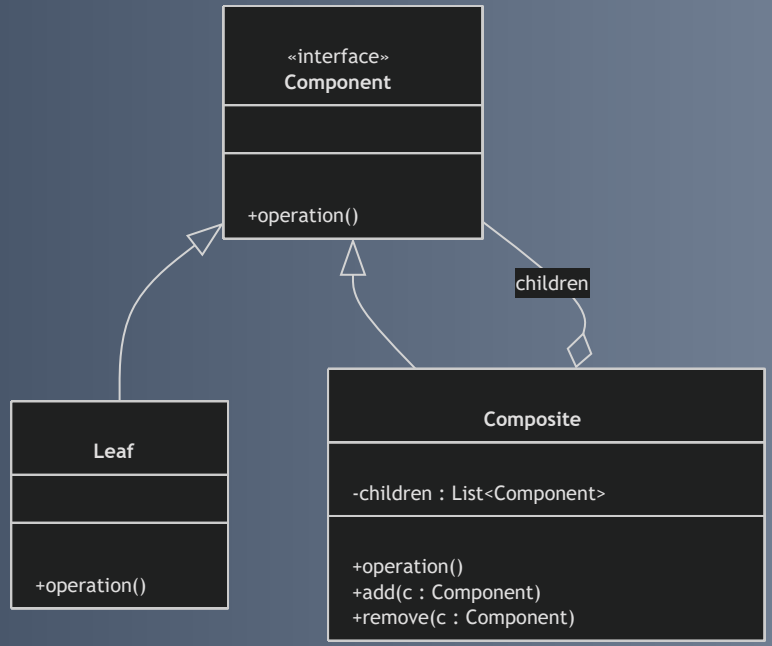
- **Exemples clés** :
  - **Singleton** : Garantit une unique instance d’une classe.
  - **Factory Method** : Définit une interface pour créer un objet, en déléguant l’instanciation.
  - **Abstract Factory** : Crée des familles d’objets liés sans spécifier leurs classes concrètes.
  - **Builder** : Sépare la construction d’un objet complexe de sa représentation.
  - **Prototype** : Initialise de nouveaux objets en copiant un prototype existant.

2. Patterns Structurels : Organiser et Composer

- **Objectif principal** :
  - Définir comment composer et organiser les classes et objets en structures plus grandes et plus flexibles.
  - Faciliter leur manipulation et leur extension.

- **Exemples clés** :
  - **Adapter** : Permet à des classes incompatibles de coopérer.
  - **Composite** : Compose des objets en structures arborescentes pour traiter uniformément des éléments.
  - **Decorator** : Ajoute dynamiquement des responsabilités à un objet sans modifier sa structure.
  - **Facade** : Fournit une interface unifiée, simplifiée, à un ensemble de classes complexes.
  - **Proxy** : Interpose un substitut pour contrôler l'accès à un autre objet.

Exemple : Composite Pattern



Le diagramme illustre comment `Composite` *et* `Leaf` *implémentent l'interface* `Component`*, permettant de traiter de manière uniforme une structure composée ou une feuille.*

3. Patterns Comportementaux : Interactions et Responsabilités

- **Objectif principal** :
  - Gérer les interactions entre objets et la distribution des responsabilités.
  - Faciliter la communication ou modifier dynamiquement les comportements.

- **Exemples clés** :
  - **Observer** : Définit une dépendance un-à-plusieurs pour la notification automatique des dépendants.
  - **Strategy** : Définit une famille d'algorithmes interchangeables, sélectionnables au moment de l'exécution.
  - **Command** : Encapsule une requête sous forme d'objet.
  - **State** : Permet à un objet de changer son comportement lorsque son état interne change.
  - **Template Method** : Définit le squelette d'un algorithme en déléguant certaines étapes aux sous-classes.

Récapitulatif : Les 3 Catégories en un Coup d’Œil

Catégorie	Objectif principal	Exemples clés
Création	Contrôler la création des objets	Singleton, Factory Method, Builder
Structure	Organiser les classes et objets en structures	Adapter, Composite, Decorator
Comportement	Gérer les interactions et responsabilités	Observer, Strategy, Command

Ce qu’il faut retenir & Pour aller plus loin

- **L’essentiel** :
  - Comprendre cette classification principale permet d’aborder les design patterns avec une vision claire de leur objectif.
  - Cela facilite leur sélection et leur application appropriée dans la construction de systèmes logiciels robustes et maintenables.

- **Sources recommandées** :
  - Refactoring.Guru – Design Patterns Classification
  - Wikipedia – Software design pattern
  - Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Design Patterns de Création (Partie 1)

Le Pattern Singleton : Maîtrise de l’Instance Unique

Le **pattern Singleton** est un design pattern de création ayant pour objectif de garantir qu’une classe ne possède qu’une seule instance, et de fournir un point d’accès global à cette instance.

Intention du pattern : Contrôler strictement la création des objets d’une classe. Une seule instance est souvent nécessaire pour assurer la cohérence d’un état ou d’un service dans une application.

Utilité : Gérer des ressources uniques partagées (connexion DB, gestionnaire de configuration, logger).

Singleton : Pourquoi une instance unique ?

Le défi que résout le Singleton est d’assurer que :

- Une instance unique est créée et contrôlée.
- Cette instance est accessible globalement.
- La création multiple accidentelle est empêchée.

Pourquoi cette unicité est-elle cruciale ? Quand une seule instance suffit, voire est nécessaire, pour maintenir la cohérence d’un état ou d’un service au sein de votre application.

Mettre en œuvre le Singleton

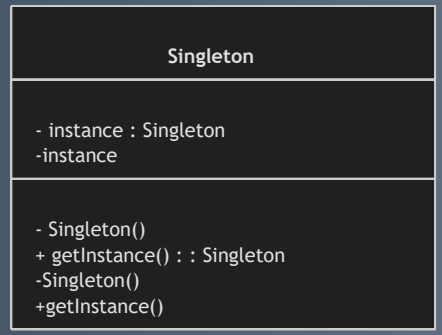
L'implémentation classique consiste à :

1. Rendre le **constructeur de la classe privé** pour empêcher l'instanciation directe.
2. Fournir une **méthode statique publique** (souvent `getInstance()`) qui contrôle l'accès à l'unique instance et la crée si elle n'existe pas encore.

Exemple en pseudo-code :

```
class Singleton {
    private static instance : Singleton = null

    // Constructeur privé pour empêcher l'instanciation extérieure
    private constructor() {}

    public static getInstance() : Singleton {
        if (instance == null) {
            instance = new Singleton()
        }
        return instance
    }
}
```

Architecture du Singleton (UML)



- `instance` est un attribut statique privé qui stocke la seule instance de la classe.
- Le constructeur `Singleton()` est privé pour empêcher l'instanciation externe.
- `getInstance()` est une méthode publique statique qui contrôle la création et l'accès à l'instance unique.

Applications et Limites du Singleton

Utilisations courantes :

- **Gestionnaires de configuration :** Un objet centralisé pour les paramètres.
- **Logger :** Un unique journal d'événements accessible partout.
- **Connexion à une base de données :** Gérer une connexion unique pour éviter le surcoût.

Limites et Précautions :

- **Problèmes en environnement multithread :** Nécessite des mécanismes de synchronisation (verrous).
- **Testabilité :** Peut compliquer les tests unitaires à cause de son état global.
- **Alternatives :** Injection de dépendances pour une gestion plus fine du cycle de vie des objets.

Le Singleton : Essentiel à retenir

Ce qu'il faut retenir : Le pattern Singleton est un outil simple mais puissant, offrant un moyen fiable de contrôler la création d'instances uniques. Son application judicieuse évite la prolifération d'objets et garantit une cohérence globale lorsque l'unicité est cruciale.

Sources :

- Refactoring.Guru – Singleton Pattern
- Wikipedia – Singleton pattern
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Design Patterns de Création (Partie 1) : Singleton

Implémentation en Java et C# (Lazy initialization, Thread-safe)

Le **Singleton** garantit qu'une classe n'est instanciée qu'une seule fois, offrant un point d'accès global à cette unique instance.

Objectifs clés d'une implémentation efficace :

- **Lazy initialization :** L'objet est créé uniquement à la première demande, optimisant les ressources.
- **Thread-safe :** L'implémentation est sécurisée et fonctionne correctement en environnement multithread.

Singleton en Java : Évolution de l'implémentation

1. Lazy Initialization simple (non thread-safe)

```java
public class Singleton {
    private static Singleton instance;
    private Singleton() { }
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

- **Limite :** Non thread-safe. Plusieurs instances peuvent être créées simultanément en environnement multithread.

2. Thread-safe avec synchronisation

```
public class Singleton {
    private static Singleton instance;
    private Singleton() { }
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

- **Avantage :** Garantit une seule instance en multithread.
- **Inconvénient :** La synchronisation constante peut entraîner un surcoût en performance.

Singleton en Java : Optimisation et Pattern Recommandé

3. Double-checked locking (Optimisé)

```java
public class Singleton {
    private static volatile Singleton instance; // volatile essentiel
    private Singleton() { }
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized(Singleton.class) { // Synchronisation
                if (instance == null) {      // Double vérification
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

- `volatile` : Assure la visibilité des modifications de l'instance entre threads.
- **Optimisation :** La synchronisation est minimale, activée uniquement lors de la première création.

4. Singleton avec Holder idiom (Java)

```java
public class Singleton {
    private Singleton() { }
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

- **Avantages :** Initialisation **lazy** et **thread-safe** sans synchronisation explicite, performance optimale.

Singleton en C# : Approches Efficaces

1. Pattern moderne thread-safe

```
public sealed class Singleton {
    private static readonly Lazy<Singleton> lazy =
        new Lazy<Singleton>(() ⇒ new Singleton());

    public static Singleton Instance { get { return lazy.Value; } }

    private Singleton() { }
}
```

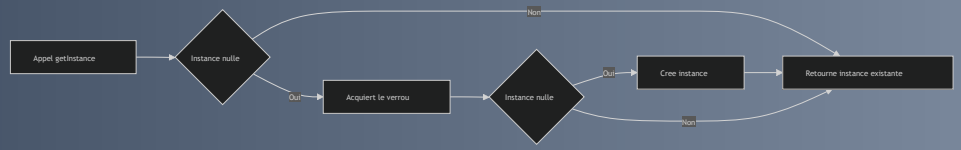
- `Lazy< T >` gère la synchronisation et la lazy initialization automatiquement.
- **Recommandé** pour sa concision et sa robustesse dans les versions modernes de C#.

2. Version simple thread-safe classique

```
public sealed class Singleton {
    private static Singleton instance = null;
    private static readonly object padlock = new object();

    Singleton() { }

    public static Singleton Instance {
        get {
            lock(padlock) {
                if (instance == null) {
                    instance = new Singleton();
                }
                return instance;
            }
        }
    }
}
```

- Utilise un verrou (`lock`) pour garantir la sécurité des threads.

Visualisation du flux : Double-Checked Locking



Ce diagramme illustre comment le mécanisme de double vérification minimise la période de verrouillage, améliorant ainsi la performance tout en garantissant la création unique de l’instance.

En Bref et Pour Aller Plus Loin

Ce qu'il faut retenir : Ces implémentations du Singleton démontrent l'importance d'allier :

- Le **contrôle précis de l'instanciation** (une seule instance).
- La **performance** (lazy initialization et minimisation des surcoûts).
- La **sécurité en environnement multithread** (garantie d'intégrité). Ce sont des conditions essentielles pour un Singleton efficace dans les applications modernes.

Sources :

- Refactoring.Guru - Singleton
- Microsoft Docs - Lazy Initialization
- Java Concurrency in Practice - Double-Checked Locking
- Wikipedia - Singleton pattern

Design Patterns de Création (Partie 1)

Singleton : Cas d’usage typiques et limites

Le pattern Singleton assure la création d’une seule instance d’une classe, accessible globalement. Son utilisation est particulièrement adaptée dans certains scénarios précis mais présente également des inconvénients qu’il convient de connaître.

Cas d’usage typiques : Gestionnaire de Configuration

Dans une application, les paramètres de configuration (fichiers, variables d’environnement, options) doivent généralement être chargés une seule fois et utilisés partout. Le Singleton permet de centraliser cette gestion.

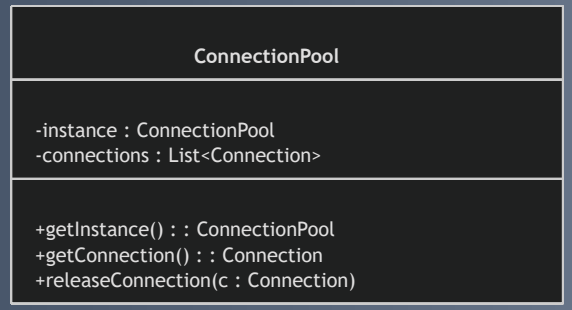
Exemple en Java :

```java
public class ConfigManager {
    private static ConfigManager instance;
    private Properties properties;

    private ConfigManager() {
        // Chargement des propriétés
        properties = new Properties();
        // code pour charger le fichier de configuration ...
    }

    public static ConfigManager getInstance() {
        if (instance == null) {
            instance = new ConfigManager();
        }
        return instance;
    }

    public String getProperty(String key) {
        return properties.getProperty(key);
    }
}
```

Cas d’usage typiques : Pool de Connexions

Le pattern est souvent utilisé pour gérer un pool limité de ressources uniques ou coûteuses, comme des connexions à une base de données. Contrôler la création et la gestion de ce pool garantit un accès ordonné et performant.

Rôle du Singleton dans un Pool de Connexions :



- La classe `ConnectionPool` est Singleton, garantissant une unique instance gérant toutes les connexions.
- Accès contrôlé aux connexions pour éviter les conflits et la surcharge.

Limites du pattern Singleton

1. **Problèmes en contexte multithread :**
    1. Sans implémentation soigneuse (ex : double-checked locking), de multiples threads peuvent créer plusieurs instances par un accès simultané.
2. **Difficulté de testabilité :**
    1. Le Singleton introduit une **dépendance globale** qui complique le testing unitaire, car il empêche d’injecter facilement des mocks ou alternatives.

Limites et Alternatives au Singleton

3. **Risque de mauvais usage : Couplage fort :**
    1. Avec un accès global, les objets peuvent dépendre directement du Singleton, ce qui diminue la modularité et accroît le couplage, rendant le code rigide et moins facilement évolutif.
4. **Cycle de vie rigide :**
    1. Un Singleton vit généralement tout au long de l'exécution de l'application, ce qui peut poser problème s'il gère des ressources consommant mémoire ou connexions réseaux non libérées.

Alternatives possibles :

- Utilisation de **services injectés** via des frameworks d'injection de dépendances (ex : Spring en Java, .NET Core DI) pour mieux contrôler le cycle de vie.
- Séparation des responsabilités en découplant la gestion d'instance unique de la logique métier.

Ce qu’il faut retenir & Sources

Conclusion : Le Singleton répond efficacement à des besoins de ressources uniques partagées globalement, comme les gestionnaires de configuration ou les pools de connexions. Cependant, ses contraintes liées à la concurrence, au testing et à la modularité invitent à en limiter l’usage et à privilégier, lorsque c’est possible, des solutions plus flexibles et testables.

Sources :

- Refactoring.Guru – Singleton Pattern Usage and Limitations
- Wikipedia – Singleton Pattern
- Martin Fowler – Inversion of Control Containers and the Dependency Injection pattern

Factory Method : Encapsuler la Création d’Objets

Qu’est-ce que le Factory Method ?

Le **Factory Method** est un *pattern de création* qui permet d’organiser la production d’objets :

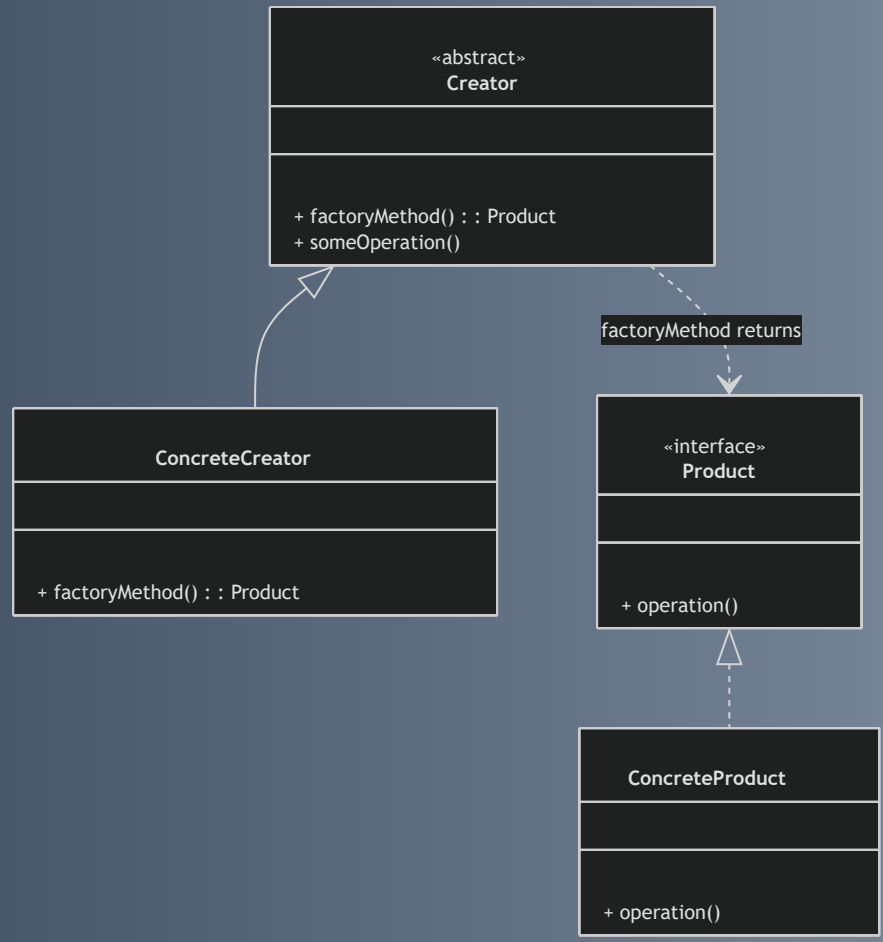
- **Encapsule la création d’objets** dans une interface commune.
- **Délègue aux sous-classes** la décision d’instancier une classe concrète spécifique.

L’objectif : Rendre votre code plus flexible, ouvert à l’extension et facile à maintenir.

Pourquoi choisir le Factory Method ?

Ce pattern répond à plusieurs besoins clés en conception logicielle :

- **Détacher la création des objets de leur utilisation :** Le code client interagit avec une abstraction, ignorant la classe exacte du produit.
- **Faciliter l'introduction de nouveaux types de produits :** Ajoutez de nouvelles classes de produits sans toucher au code existant du client.
- **Respecter le Principe Ouvert/Fermé :** Votre code est ouvert à l'extension (pour de nouveaux produits) mais fermé à la modification (pas de changements dans le code client).
- **Gérer des familles d'objets liés :** Créez des collections d'objets sans avoir à connaître leurs implémentations concrètes.


«abstract»
Creator
+ factoryMethod() : : Product
+ someOperation()
factoryMethod returns
ConcreteCreator
+ factoryMethod() : : Product
«interface»
Product
+ operation()
ConcreteProduct
+ operation()

- **Product :** Déclare l'interface que tous les produits concrets doivent implémenter.
- **ConcreteProduct :** Implémente l'interface `Product`.
- **Creator :** Déclare la `factoryMethod()` abstraite qui retourne un objet de type `Product`. Peut contenir d'autres méthodes métier.
- **ConcreteCreator :** Surcharge `factoryMethod()` pour retourner une instance de `ConcreteProduct`.

Exemple : Gestion de Documents

Imaginons un système qui gère différents types de documents (Texte, PDF).

```
// L'interface Produit
public interface Document {
    void open();
}

// Un Produit concret
public class TextDocument implements Document {
    public void open() {
        System.out.println("Ouverture document texte");
    }
}
```

```java
// Le Créateur abstrait
public abstract class Application {
    // La Factory Method
    public abstract Document createDocument();

    // Une méthode métier utilisant le produit
    public void newDocument() {
        Document doc = createDocument(); // Délégation de la création
        doc.open();
    }
}

// Un Créateur concret
public class TextApplication extends Application {
    // Implémente la Factory Method pour créer un TextDocument
    public Document createDocument() {
        return new TextDocument();
    }
}
```

Le client interagit avec `Application` pour créer des `Document`, sans connaître les classes concrètes.

Comment ça marche ?

1. **Le Client** utilise la classe abstraite `Application` et sa méthode générique `newDocument()`.
2. Lorsque `newDocument()` est appelée, elle invoque la méthode `createDocument()` du même `Application`.
3. **La sous-classe concrète** (`TextApplication`, `PdfApplication`…) fournit l'implémentation spécifique de `createDocument()`, décidant ainsi quelle instance de `Document` créer (`TextDocument`, `PdfDocument`).
4. Le client interagit toujours avec l'objet via l'interface `Document`, ignorant totalement la classe concrète qui a été instanciée.

Ceci garantit une séparation entre la logique métier et la logique de création.

Ce qu’il faut retenir & Sources

En résumé :

Le Factory Method introduit une abstraction dans le processus de création d’objets. Cette délégation de la responsabilité de création aux sous-classes augmente significativement la **flexibilité** et la **maintenabilité** de vos applications, surtout lorsque votre gamme de produits est amenée à évoluer.

Pour aller plus loin :

- Refactoring.Guru – Factory Method
- Wikipedia – Factory Method pattern
- Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Design Patterns de Création : Factory Method

Séparer la logique de création

Qu’est-ce que le Factory Method ?

- Un pattern qui définit une méthode (la "factory method") pour créer des objets.
- Laisse les classes dérivées décider quelle classe concrètement instancier.

Objectif principal :

- **Flexibilité :** Adapter la création d’objets sans modifier le code client.
- **Extensibilité :** Ajouter facilement de nouveaux types de produits.

La Structure Clé

- `Product` (Interface ou Classe Abstraite)
  - Définit l'interface commune pour les objets que la factory doit créer.
- `ConcreteProduct` (Classes Concrètes)
  - Implémentent l'interface `Product`.
  - Ce sont les objets spécifiques qui sont produits.
- `Creator` (Classe Abstraite ou Interface)
  - Déclare la **méthode factory** (`factoryMethod`), qui doit retourner un `Product`.
  - Peut contenir du code qui utilise les objets `Product`.
- `ConcreteCreator` (Classes Concrètes)
  - Implémentent la `factoryMethod()` pour créer et retourner des `ConcreteProduct` spécifiques.
- **Le Client**
  - Utilise `Creator` et interagit avec les objets via l'interface `Product`, sans se soucier de leur instanciation concrète.

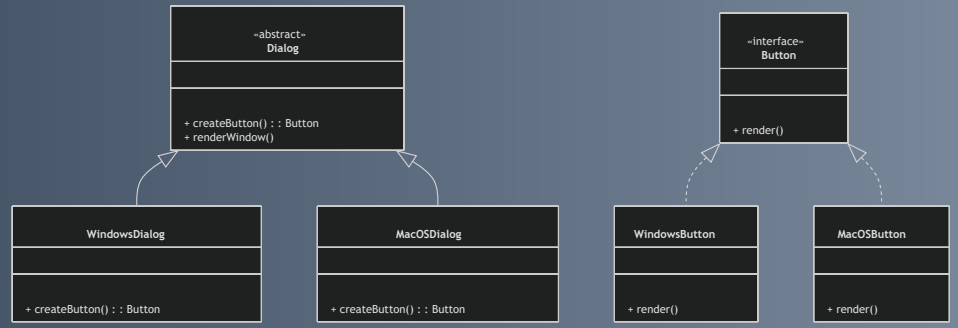
Cas Pratique : Boutons d’Interface UI

Contexte : Créer des boutons (Windows, MacOS) selon le système d’exploitation.

- **Produits :**
  - `Button` (interface) avec la méthode `render()`.
  - `WindowsButton` et `MacOSButton` (implémentations concrètes).
- **Créateurs :**
  - `Dialog` (classe abstraite)
    - Contient la logique d’affichage (`renderWindow()`).
    - Déclare la méthode abstraite `createButton() : Button`.
  - `WindowsDialog` et `MacOSDialog` (implémentations concrètes)
    - Implémentent `createButton()` pour retourner respectivement un `WindowsButton` ou un `MacOSButton`.

Fonctionnement Client : Le client sélectionne un type de `Dialog` (ex: `new WindowsDialog()`). Le `Dialog` se charge de créer le `Button` adapté (`createButton()`) et le reste de l’application interagit via l’interface `Button`.

Vue d’Ensemble : Diagramme UML



Interprétation :

- `Dialog` (le Créateur abstrait) déclare la méthode `createButton()`.
- `WindowsDialog` et `MacOSDialog` (les Créateurs concrets) déterminent quel bouton concret instancier (`WindowsButton` ou `MacOSButton`).
- Le client utilise `renderWindow()` de `Dialog`, qui fait appel à la méthode factory sans connaître les détails d’implémentation du bouton.

Points Forts de l’Implémentation

- **Méthode abstraite :** La `factoryMethod()` est généralement abstraite dans le créateur, forçant les sous-classes à définir leur logique de création.
- **Extension facilitée :** Permet d’ajouter de nouveaux types de produits (ex: un `LinuxButton` et son `LinuxDialog`) :
  - **Sans modifier** le code client.
  - **Sans modifier** la classe créatrice abstraite (`Dialog`).
- **Découplage fort :** Le client est complètement séparé de la logique d’instanciation des objets concrets. Il interagit uniquement avec les interfaces (`Product` et `Creator`).

Ce qu’il faut retenir & Sources

La force du Factory Method :

- Isole la création d’objets spécifiques dans des classes dédiées.
- Favorise une architecture plus modulaire et facile à maintenir.
- Facilite l’extension future des familles de produits et la gestion de nouvelles variantes.

Références :

- Refactoring.Guru – Factory Method Pattern
- Wikipedia – Factory Method pattern
- Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Design Patterns de Création (Partie 1)

Factory Method : Ses Avantages Clés

Le pattern **Factory Method** offre deux bénéfices fondamentaux pour la qualité du code :

- **Découplage** entre le client et la création d'objets.
- **Extensibilité** facilitée, permettant d'ajouter de nouveaux types sans modifier le client.

Découplage : Une Connexion Allégée

Problème : Couplage fort Lorsque le client instancie directement des objets concrets, il y a un couplage fort. Toute modification ou extension des objets affecte le client.

Sans Factory Method : Le client est directement lié à la classe concrète.

```
Button button = new WindowsButton();  // Couplage direct
button.render();
```

Découplage : La Solution Factory Method

Principe : Déléguer la création Le Factory Method encapsule l'instanciation dans une méthode abstraite, déléguée à des sous-classes spécialisées.

Avec Factory Method : Le client interagit avec des interfaces ou classes abstraites, ignorant la classe concrète créée.

```
Dialog dialog = new WindowsDialog();
Button button = dialog.createButton();  // Le client ne connaît pas la classe concrète
button.render();
```

Extensibilité : Accueillir de nouveaux Types facilement

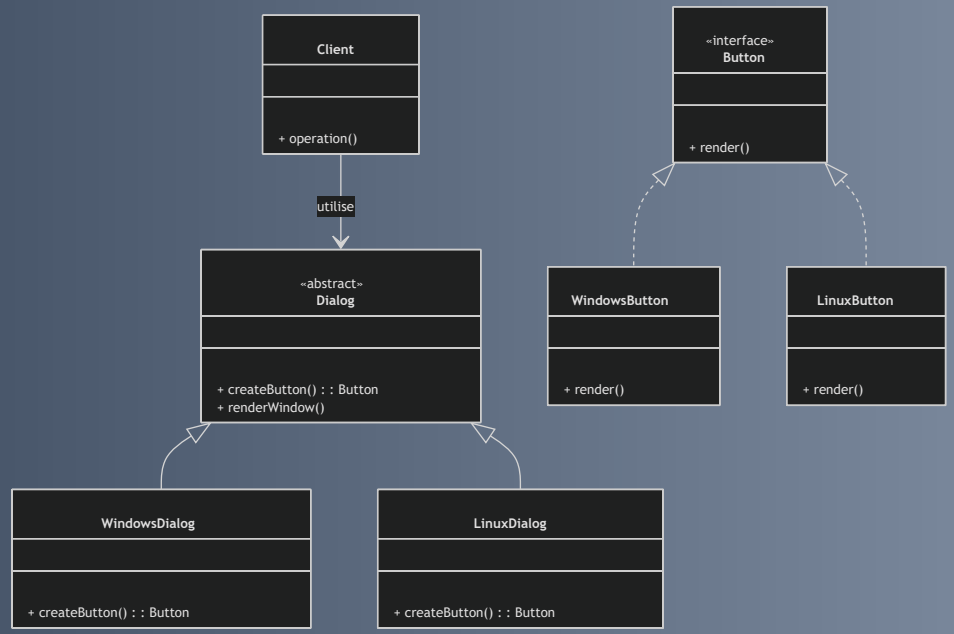
Principe : Aucune modification du client L'ajout d'un nouveau type d'objet ne nécessite pas de modifier la classe cliente ou la classe abstraite `Dialog`.

Exemple d'extension : Ajout de Linux Il suffit de créer une nouvelle sous-classe du créateur et de surcharger `createButton()`.

```
public class LinuxButton implements Button {
    public void render() {
        System.out.println("Render bouton Linux");
    }
}

public class LinuxDialog extends Dialog {
    public Button createButton() {
        return new LinuxButton();
    }
}
```

Le client peut utiliser `LinuxDialog` sans aucun changement.

Vision Globale : Découplage & Extensibilité en Action



Le client dépend uniquement des abstractions `Dialog` et `Button`, garantissant un faible couplage.

Autres Bénéfices observés :

- Favorise le principe **Single Responsibility**.
- Facilite la gestion des familles de produits liés.
- Rend le test unitaire plus aisé.

En Bref : L’Essentiel du Factory Method & Sources

Ce qu’il faut retenir : Le Factory Method assure un découplage solide entre clients et produits, et soutient l’évolution du code via une architecture ouverte à l’extension. C’est un outil clé pour concevoir des systèmes modulaires et maintenables.

Sources :

- Refactoring.Guru – Factory Method Advantages
- Wikipedia – Factory Method
- Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Design Patterns de Création (Partie 2) : Abstract Factory

L'**Abstract Factory** est un pattern de création qui fournit une **interface** pour créer des **familles d'objets apparentés ou interdépendants**, sans spécifier leurs classes concrètes.

Ce mécanisme essentiel assure la **cohérence** entre les produits d'une même famille et facilite l'**extensibilité** de votre application.

Définition & Intention du Pattern

Définition

L’Abstract Factory propose une interface regroupant plusieurs méthodes de fabrication. Chaque méthode correspond à un type d’objet, permettant de créer différentes familles complètes de produits sans exposer au client les classes concrètes utilisées.

Intention

- **Encapsuler** un ensemble d’objets liés à créer en une seule interface.
- **Garantir** que les produits d’une même famille soient compatibles.
- **Isoler** le client des détails spécifiques de l’instanciation.
- **Faciliter** le changement d’une famille de produits complète simplement en changeant l’Abstract Factory utilisée.

Illustration Conceptuelle : Interfaces Utilisateur

Imaginez une application devant gérer plusieurs styles d'interface utilisateur, par exemple **Windows** et **MacOS**.

Ces styles nécessitent des **boutons** et des **cases à cocher** spécifiques, qui doivent impérativement fonctionner ensemble visuellement et fonctionnellement.

L'Abstract Factory fournit alors :

- Une interface unique pour créer un bouton et une case à cocher.
- Des Concrete Factories (`WindowsFactory` et `MacOSFactory`) pour créer les variantes respectives de ces composants.

Diagramme de Classes : les Factories

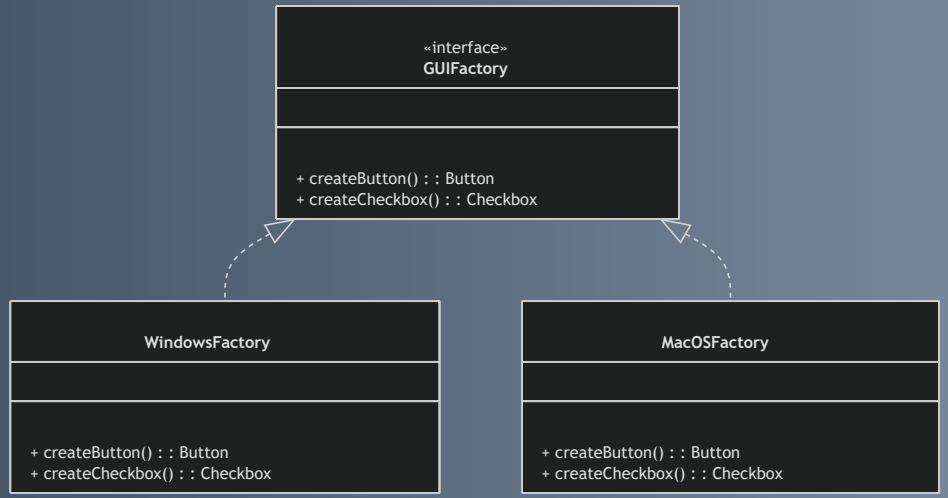
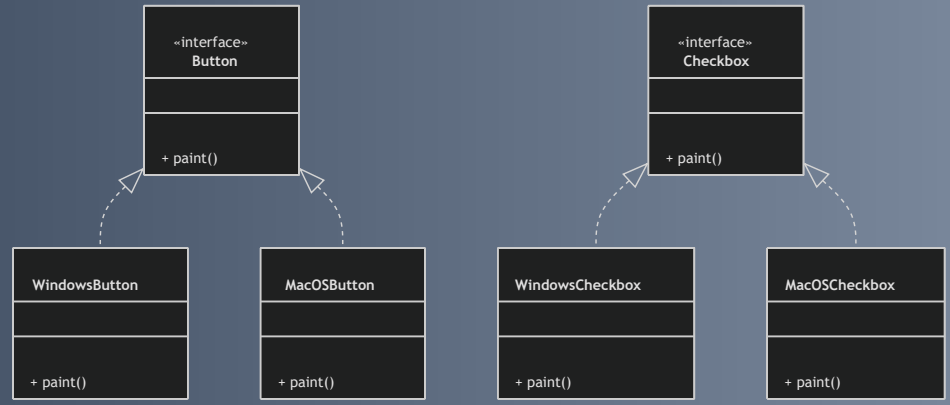


Diagramme de Classes : les produits



Exemple Concret en Java

```java
// Interfaces produits et leurs implémentations (Windows, MacOS)
public interface Button { void paint(); }
public class WindowsButton implements Button { /* ... */ }
public class MacOSButton implements Button { /* ... */ }

public interface Checkbox { void paint(); }
public class WindowsCheckbox implements Checkbox { /* ... */ }
public class MacOSCheckbox implements Checkbox { /* ... */ }

// Usine abstraite (GUIFactory) et ses implémentations concrètes
public interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

public class WindowsFactory implements GUIFactory {
    public Button createButton() { return new WindowsButton(); }
    public Checkbox createCheckbox() { return new WindowsCheckbox(); }
}

public class MacOSFactory implements GUIFactory {
    public Button createButton() { return new MacOSButton(); }
    public Checkbox createCheckbox() { return new MacOSCheckbox(); }
}
```

-> Suite ->

```
// Client (Application)
public class Application {
    private Button button;
    private Checkbox checkbox;

    public Application(GUIFactory factory) {
        button = factory.createButton();
        checkbox = factory.createCheckbox();
    }
    public void paint() {
        button.paint();
        checkbox.paint();
    }
}
```

Fonctionnement

Pour changer le style de l'interface, il suffit de remplacer l'implémentation de `GUIFactory` passée au client :

```
GUIFactory factory;
String osName = System.getProperty("os.name").toLowerCase();

if (osName.contains("windows")) {
    factory = new WindowsFactory();
} else {
    factory = new MacOSFactory();
}

Application app = new Application(factory); // Le client reçoit l'usine
app.paint(); // Il utilise les produits de cette usine
```

Le client n'a aucune connaissance des classes concrètes des produits. Il travaille uniquement avec l'abstraction.

Ce qu’il faut retenir

L’Abstract Factory se révèle particulièrement utile lorsque plusieurs types de produits liés doivent être créés ensemble de façon cohérente, tout en maintenant un couplage faible et en facilitant la maintenance et l’évolution.

Sources

- Refactoring.Guru – Abstract Factory
- Wikipedia – Abstract Factory pattern
- Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Design Patterns de Création (Partie 2)

Abstract Factory : Différence avec Factory Method

Les patterns **Abstract Factory** et **Factory Method** sont des patrons de création qui facilitent l'instanciation flexible d'objets. Bien qu'ils partagent cet objectif, ils répondent à des problématiques distinctes et opèrent à des niveaux différents.

Factory Method vs Abstract Factory

Critère	Factory Method	Abstract Factory
Objet créé	Un seul type d'objet concret	Une famille d'objets apparentés
Nombre de méthodes de création	Une seule méthode abstraite pour créer un produit	Plusieurs méthodes abstraites regroupées dans une interface
Héritage	Utilise l'héritage : sous-classes implémentent la méthode de création	Utilise la composition : une usine concrète instancie plusieurs produits
Complexité	Plus simple, pour une seule hiérarchie de produit	Plus complexe, pour gérer plusieurs variantes de plusieurs produits
But	Déléguer la création d'un objet particulier à une sous-classe	Fournir une interface pour créer des familles cohérentes d'objets

En bref :

- Le **Factory Method** délègue la création d'un *seul* produit à des sous-classes.
- L'**Abstract Factory** propose une interface pour créer *plusieurs* objets liés et compatibles (une famille de produits).

Factory Method : Création d’un Produit Unique

Le **Factory Method** définit une interface pour la création d’un seul type de produit. La sous-classe décide de la classe concrète à instancier, permettant ainsi une création flexible et découplée.

```
public abstract class Dialog {
    // Méthode abstraite que les sous-classes implémentent
    public abstract Button createButton();
    public void renderWindow() {
        Button okButton = createButton(); // Utilise la méthode de création
        okButton.render();
    }
}

public class WindowsDialog extends Dialog {
    // Implémentation concrète : crée un bouton spécifique à Windows
    public Button createButton() {
        return new WindowsButton();
    }
}

public class WindowsButton implements Button {
    public void render() {
        System.out.println("Render Windows Button");
    }
}
```

Abstract Factory : Création d'une Famille de Produits

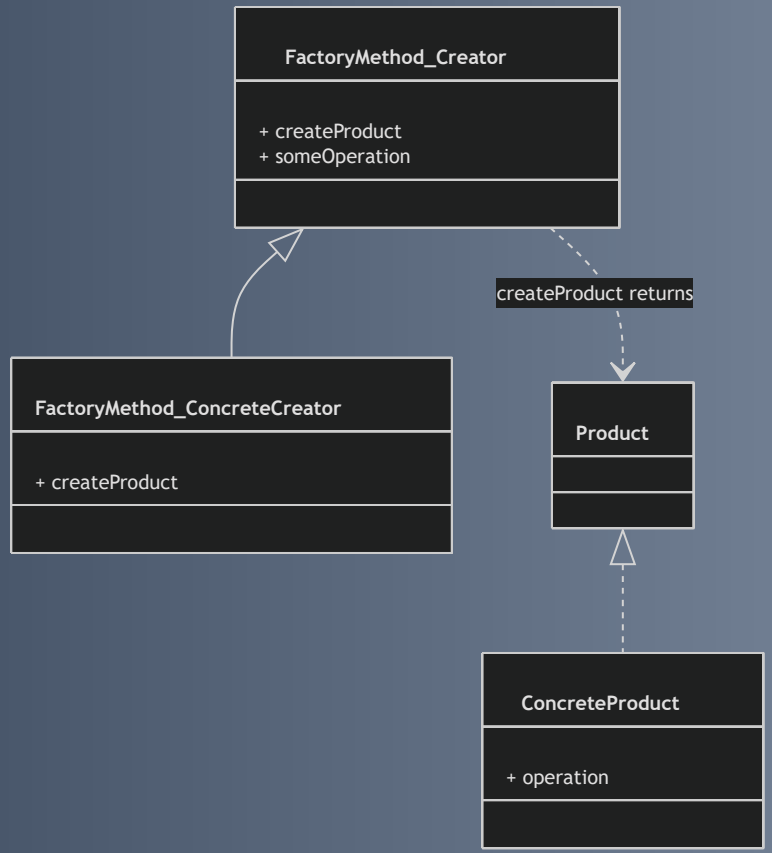
L'**Abstract Factory** définit une interface pour créer plusieurs objets liés (une famille de produits). Chaque usine concrète implémentant cette interface est responsable de produire l'ensemble des produits de cette famille, garantissant leur compatibilité.

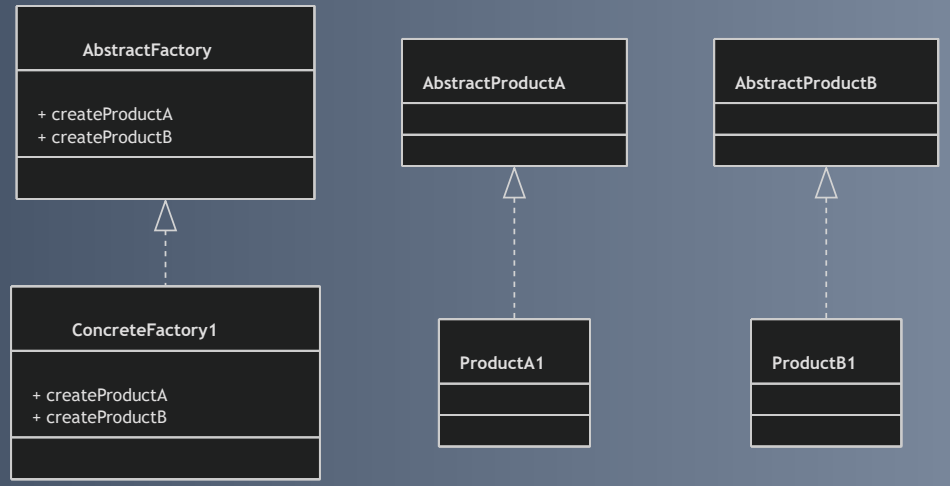
```
public interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

public class WindowsFactory implements GUIFactory {
    public Button createButton() {
        return new WindowsButton();
    }
    public Checkbox createCheckbox() {
        return new WindowsCheckbox();
    }
}
```

--> Suite -->

```java
public interface Button { void paint(); }
public interface Checkbox { void paint(); }

public class WindowsButton implements Button {
    public void paint() {
        System.out.println("Windows Button");
    }
}
public class WindowsCheckbox implements Checkbox {
    public void paint() {
        System.out.println("Windows Checkbox");
    }
}
```

Vision Globale : Factory Method vs Abstract Factory




AbstractFactory
+ createProductA
+ createProductB
ConcreteFactory1
+ createProductA
+ createProductB
AbstractProductA
ProductA1
AbstractProductB
ProductB1

Récapitulatif et Ressources

Aspect	Factory Method	Abstract Factory
Type d'objets créés	Un produit unique	Famille de produits liés
Création	Méthode abstraite dans une classe abstraite	Interface avec plusieurs méthodes
Flexibilité	Découpage via héritage	Découpage via composition d'usines
Utilisation typique	Libérer l'instanciation d'un seul type	Garantir la cohérence entre plusieurs objets

Ce qu'il faut retenir : Le choix entre Factory Method et Abstract Factory dépend de votre besoin : souhaitez-vous déléguer l'instanciation d'un produit unique, ou garantir la cohérence d'une famille complète de produits ? Comprendre leurs différences permet d'adopter la solution optimale.

Sources :

- Refactoring.Guru – Factory Method vs Abstract Factory
- Wikipedia – Factory Method
- Wikipedia – Abstract Factory
- Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Design Patterns de Création (Partie 2)

Le Pattern Builder

Le pattern **Builder** vise la construction d’objets complexes étape par étape. Il sépare la représentation d’un objet de sa construction, offrant une grande flexibilité dans le processus d’instanciation.

Objectif :

- Construire des objets complexes de manière structurée.
- Offrir une flexibilité accrue lors de l’instanciation.

Définition du Builder

Le Builder propose une approche pas à pas pour assembler les différentes parties d'un objet complexe.

Composants Clés :

- **Builder Abstrait :** Définit l'interface pour les étapes de construction (ex: `buildPartA()`, `buildPartB()`).
- **Builder Concret :** Implémente les étapes spécifiques pour créer une représentation particulière du produit.
- **Directeur (optionnel) :** Oriente la construction en invoquant les étapes du builder dans un ordre prédéfini pour produire un type d'objet standardisé.

Pourquoi Choisir le Builder ? (Intention)

Le pattern Builder répond à plusieurs besoins spécifiques :

- **Construction Séquentielle :** Bâtir un objet complexe en plusieurs étapes clairement séparées.
- **Variantes d'Objets :** Permettre différentes représentations (variantes) du même objet par différents builders.
- **Délégation de la Construction :** Transférer la logique complexe de construction au builder, laissant au client la possibilité de récupérer l'objet finalisé.
- **Simplification :** Faciliter la création d'objets :
  - **Immuables** (qui ne peuvent pas être modifiés après leur création).
  - **Sans constructeur à nombreux paramètres** (évite les "telescoping constructors").

Exemple : Construction d’une Voiture

Le Scénario : Créer différents types de voitures (produits complexes) pièce par pièce.

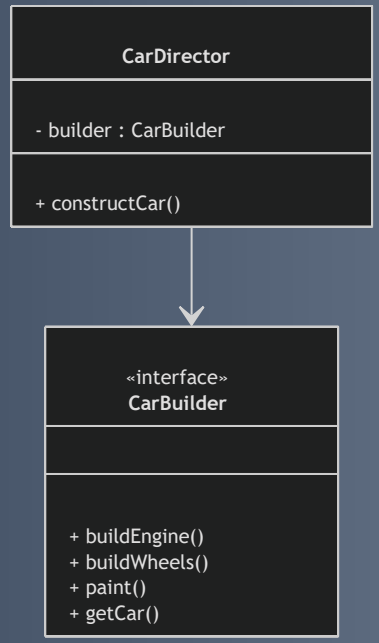
- **Produit :** `Car` (Moteur, Roues, Couleur).
- **Builder Abstrait :** `CarBuilder` (définit les étapes : `buildEngine()`, `buildWheels()`, `paint()`, `getCar()`).
- **Builder Concret :** `SportsCarBuilder` (implémente les étapes pour une voiture de sport).
- **Directeur :** `CarDirector` (orchestre la séquence de construction via `constructCar()`).

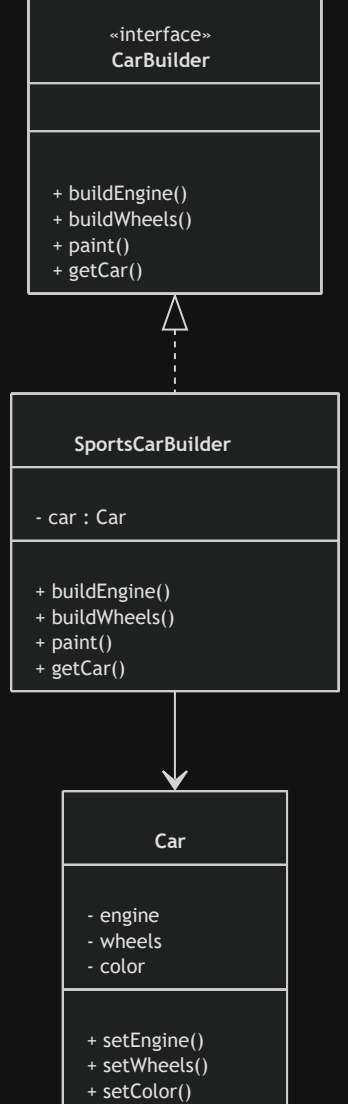
Utilisation Client Simplifiée :

```
public class Client {
    public static void main(String[] args) {
        CarBuilder builder = new SportsCarBuilder(); // Je veux une voiture de sport
        CarDirector director = new CarDirector(builder); // Le directeur sait comment faire
        director.constructCar(); // Le directeur ordonne la construction
        Car car = builder.getCar(); // Le client récupère la voiture assemblée
        System.out.println(car);
        // Affiche : Car [engine=V8 Engine, wheels=18 inch Alloy Wheels, color=Red]
    }
}
```

Structure du Pattern Builder

Le `CarDirector` interagit avec l'interface `CarBuilder` pour déclencher les étapes de construction. Le `SportsCarBuilder` concret assemble les différentes parties de l'objet `Car`.




«interface»
CarBuilder
+ buildEngine()
+ buildWheels()
+ paint()
+ getCar()
SportsCarBuilder
- car : Car
+ buildEngine()
+ buildWheels()
+ paint()
+ getCar()
Car
- engine
- wheels
- color
+ setEngine()
+ setWheels()
+ setColor()

En Bref & Ressources

Points Clés à Retenir :

- Le Builder **découple** la construction de l'objet de sa représentation finale.
- Plusieurs builders peuvent coexister, chacun produisant une **variante différente** du produit.
- Le client peut diriger finement la construction ou **déléguer** l'ordre au directeur.
- Adapté pour des objets avec **paramètres multiples, complexes ou immuables**.

Conclusion : Le Builder offre une solution structurée pour construire des objets complexes, évitant les constructeurs surchargés et simplifiant la maintenance et l'extension.

Sources :

- Refactoring.Guru – Builder Pattern
- Wikipedia – Builder Pattern
- Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Design Patterns de Création (Partie 2)

Le Builder et le Fluent API : Construire avec Éloquence

- Le pattern **Builder** facilite la construction d’objets complexes.
- Combiné à une **implémentation fluide (Fluent API)**, il permet d’écrire du code :
  - **Clair**
  - **Lisible**
  - **Expressif**
- Grâce à des **méthodes chaînées**.

Qu’est-ce que le Fluent API dans le Builder ?

- **Principe Fondamental :**
  - Chaque méthode de configuration du builder retourne une référence à l’instance courante (`this`).
  - Cela permet de **chaîner les appels de méthodes**, rendant le processus de construction très naturel, proche du langage humain.

Avantages Clés du Fluent Builder

- **Lisibilité améliorée :** Le code ressemble à un « mini-langage » spécifique à la construction de l’objet.
- **Facilité d’utilisation :** L’objet peut être configuré en une seule expression.
- **Immutabilité souvent prise en charge :** Permet de construire des objets immuables via le builder.

Exemple en Java : La Classe Produit `Computer`

Classe `Computer` (le produit final)

```
public class Computer {
    private String CPU;
    private int RAM;
    private int storage;

    private Computer() {} // Constructeur privé pour forcer l'utilisation du builder
    // Méthode toString publique pour affichage (accessible partout).
    @Override
    public String toString() { // ⟵ méthode publique
        return "Computer [CPU=" + CPU + ", RAM=" + RAM + "GB, Storage=" + storage + "GB]";
    }
...
```

--> Suite -->

Classe `Builder` (imbriquée et fluide)

```
public static class Builder {
    private String CPU; private int RAM; private int storage;

    // Méthodes de configuration retournant "this" (le builder lui-même)
    public Builder withCPU(String CPU) { this.CPU = CPU; return this; }
    public Builder withRAM(int RAM) { this.RAM = RAM; return this; }
    public Builder withStorage(int storage) { this.storage = storage; return this; }

    // Méthode finale pour construire l'objet Computer
    public Computer build() {
        Computer computer = new Computer();
        computer.CPU = this.CPU;
        computer.RAM = this.RAM;
        computer.storage = this.storage;
        return computer;
    }
    } // ⟵ fin du Builder
} // ⟵ fin de la classe Computer
```

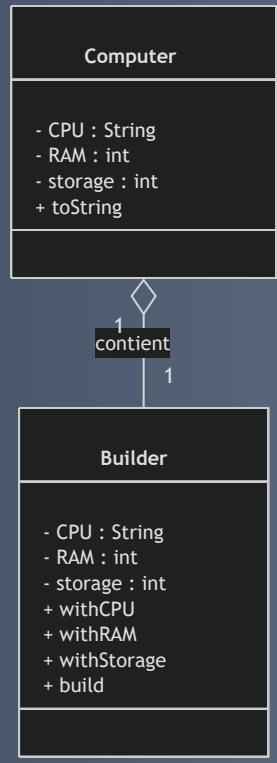
Construction du `Computer` par le Client

```java
public class Client {
    public static void main(String[] args) {
        Computer myPC = new Computer.Builder()
            .withCPU("Intel i7")     // Appel chaîné
            .withRAM(16)             // Appel chaîné
            .withStorage(512)        // Appel chaîné
            .build();                // Création de l'objet final

        System.out.println(myPC);
        // Affiche : Computer [CPU=Intel i7, RAM=16GB, Storage=512GB]
    }
}
```

- **Ce qu'on observe :**
  - Le code est **expressif** et simple à lire, mimant une phrase.
  - Il évite la confusion habituelle des multiples arguments dans un constructeur unique.
  - Le chaînage des appels rend la configuration intuitive et linéaire.

Architecture du Fluent Builder

Diagramme de Classes



Points Importants pour un Fluent Builder

- `return this` : Chaque méthode de configuration retourne l'instance du Builder pour permettre le chaînage.
- **Méthode** `build()` : L'objet final est construit et retourné par cette méthode dédiée.
- **Constructeur privé :** Le constructeur de l'objet final est privé pour forcer son instanciation via le Builder.
- **Validation :** La méthode `build()` est le lieu idéal pour vérifier la validité des paramètres avant la création de l'objet.

Synthèse & Ressources

Ce qu'il faut retenir

- Le Builder avec Fluent API est une solution élégante pour la construction d'**objets complexes**.
- Il garantit un code **plus lisible, maintenable** et réduit les erreurs.
- Il surmonte efficacement les limitations des constructeurs avec de nombreux paramètres optionnels ou ordonnés.

Sources

- Refactoring.Guru – Builder Pattern
- Effective Java, 3rd Edition, Joshua Bloch (chapitre sur le pattern Builder et Fluent API)
- Wikipedia – Builder pattern

Design Patterns de Création (Partie 2)

Builder vs Abstract Factory : Comparaison et cas d’usage

Les patterns **Builder** et **Abstract Factory** sont des design patterns de création. Bien qu’ils aident tous deux à la construction d’objets, ils répondent à des besoins distincts et s’appliquent dans des contextes différents.

Différences fondamentales

Critère	Builder	Abstract Factory
But principal	Construire un objet complexe étape par étape, avec de nombreux paramètres	Créer une famille d’objets liés ou dépendants de manière cohérente
Produit	Un seul objet complexe	Plusieurs objets qui forment une famille
Construction	Processus étape par étape, souvent avec méthodes chaînées (Fluent API)	Factory avec différentes méthodes pour chaque type d’objet
Découplage	Sépare la logique de construction et la représentation	Fournit une interface d’usine abstraite pour créer des objets liés
Utilisation typique	Configurer un objet avec de nombreux paramètres optionnels	Générer plusieurs types d’objets compatibles selon une plateforme ou variante

Builder : Construire des objets complexes étape par étape

Le **Builder** est idéal pour des objets complexes avec de nombreux paramètres optionnels. Il résout le problème des "constructeurs télescopes" (multiplication de constructeurs surchargés), améliorant la lisibilité et réduisant les erreurs.

- **Principe :** Un processus de construction détaillé, souvent via une API fluide (Fluent API) où chaque appel de méthode configure une partie de l'objet.
- **Exemple (utilisation) :**

```
// Configuration d'un objet Person avec le Builder
Person person = new Person.PersonBuilder("John", "Doe")
    .age(30)
    .phone("123456789")
    .address("123 Main St")
    .build();

System.out.println(person);
```

Cela permet de configurer l'objet de manière claire et sélective, sans avoir à gérer un grand nombre de paramètres dans un constructeur unique.

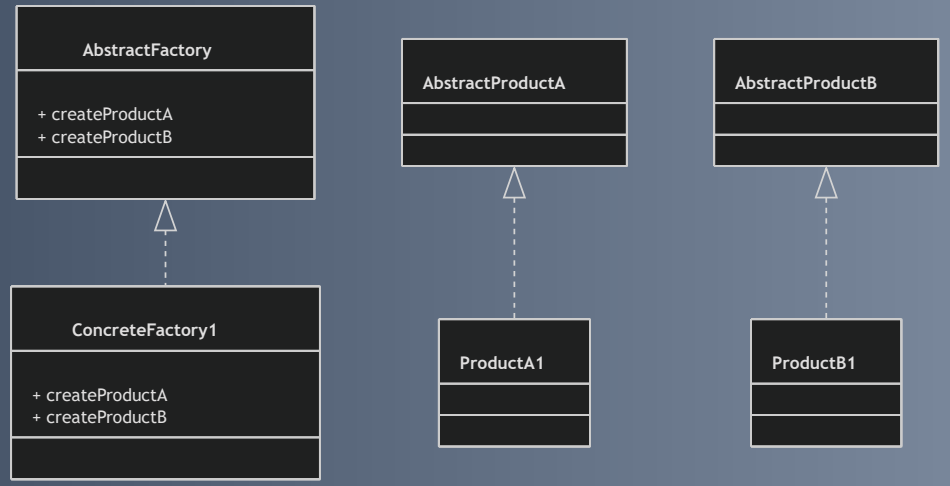
Abstract Factory : Créer des familles d’objets cohérentes

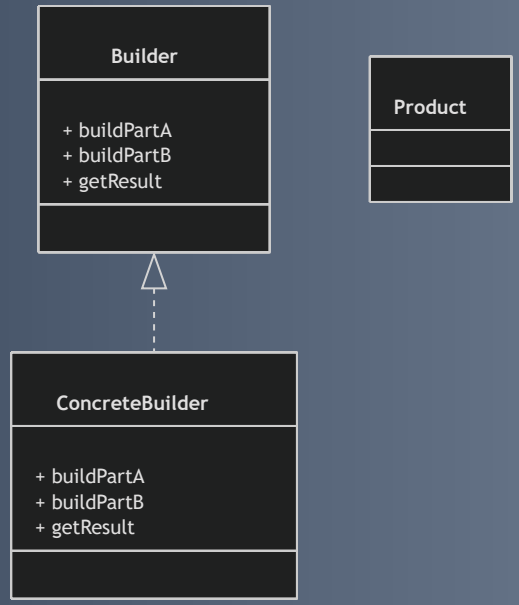
L’**Abstract Factory** se concentre sur la création d’un **ensemble complet et cohérent d’objets liés ou dépendants**.

- **But :** Fournir une interface pour créer des familles d’objets sans spécifier leurs classes concrètes.
- **Cas d’usage :** Générer des composants compatibles pour différentes plateformes (par exemple, des composants graphiques Windows vs. MacOS).
- **Garantie :** Assure la cohérence entre les objets créés, car tous les objets d’une même "famille" proviennent de la même usine concrète.

Synthèse : Quand choisir Builder ou Abstract Factory ?

Besoin / Situation	Utiliser Builder	Utiliser Abstract Factory
Construction d’un objet individuel complexe	✓	✗
Nombre élevé de paramètres optionnels	✓	✗
Créer une famille d’objets interconnectés	✗	✓
Garantir la cohérence entre les objets créés	✗	✓
Focus sur la représentation du produit final	✓	✗


AbstractFactory
+ createProductA
+ createProductB
ConcreteFactory1
+ createProductA
+ createProductB
AbstractProductA
ProductA1
AbstractProductB
ProductB1


Builder
+ buildPartA
+ buildPartB
+ getResult
Product
ConcreteBuilder
+ buildPartA
+ buildPartB
+ getResult

Ce qu’il faut retenir & Références

Ce qu’il faut retenir : Le choix entre Builder et Abstract Factory dépend du problème spécifique à résoudre.

- Le **Builder** est privilégié pour construire un **objet complexe et configurable** à la volée, avec de nombreux paramètres.
- L’**Abstract Factory** est adapté pour générer des **familles complètes d’objets compatibles** sur différents contextes ou plateformes.

Connaître ces différences permet d’appliquer le pattern approprié et d’optimiser la conception de vos logiciels.

Références :

- Refactoring.Guru – Builder vs Abstract Factory
- Wikipedia – Builder pattern
- Wikipedia – Abstract Factory pattern
- Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Design Patterns de Structure : Adapter

Le design pattern **Adapter** permet de faire collaborer des objets dont les interfaces sont incompatibles.

Il fournit une interface intermédiaire qui adapte l'usage entre eux.

Définition : L'Adapter agit comme un **pont** entre deux interfaces incompatibles. Il convertit l'interface d'une classe (l'adaptée) en une interface attendue par le client, afin que ces deux entités puissent collaborer.

Adapter : Intention et Utilité

Intention :

- Permettre à des classes ayant des interfaces incompatibles de fonctionner ensemble.
- Réutiliser des classes existantes sans modifier leur code.
- Simplifier l'intégration de composants tiers aux interfaces différentes.

Utilité : Ce pattern s'avère particulièrement utile quand un composant existant doit être réutilisé dans un contexte différent sans modifier son code.

Adapter en action : Scénario d’incompatibilité

Le Problème : Interface incompatible de lecteur audio

1. **Le système attend :** Une interface `MediaPlayer`.

```
public interface MediaPlayer {
    void play(String audioType, String fileName);
}
```

2. **Nous disposons de :** Un lecteur audio avancé avec une interface différente.

```
public interface AdvancedMediaPlayer {
    void playVlc(String fileName);
    void playMp4(String fileName);
}
public class VlcPlayer implements AdvancedMediaPlayer { /* ... */ }
public class Mp4Player implements AdvancedMediaPlayer { /* ... */ }
```

Le Défi : Comment intégrer `AdvancedMediaPlayer` (et ses implémentations) au système qui n’attend que `MediaPlayer` sans modifier le code existant ?

Adapter en action : La solution

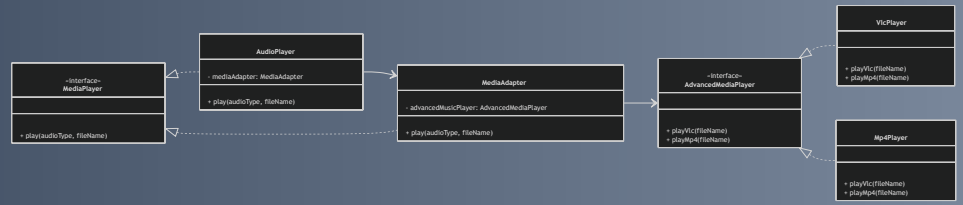
L'Adapter (`MediaAdapter` **) :** Implémente l'interface `MediaPlayer` attendue et utilise l'interface `AdvancedMediaPlayer` interne.

```
public class MediaAdapter implements MediaPlayer {
    AdvancedMediaPlayer advancedMusicPlayer;

    public MediaAdapter(String audioType){ // Initialise le bon lecteur avancé
        if(audioType.equalsIgnoreCase("vlc")){
            advancedMusicPlayer = new VlcPlayer();
        } else if(audioType.equalsIgnoreCase("mp4")){
            advancedMusicPlayer = new Mp4Player();
        }
    }

    public void play(String audioType, String fileName) {
        // Délègue l'appel au bon lecteur avancé
        if(audioType.equalsIgnoreCase("vlc")){
            advancedMusicPlayer.playVlc(fileName);
        } else if(audioType.equalsIgnoreCase("mp4")){
            advancedMusicPlayer.playMp4(fileName);
        }
    }
}
```

Le Client (`AudioPlayer`) : Utilise l'Adapter pour les formats non supportés directement.

```
public class AudioPlayer implements MediaPlayer {
    MediaAdapter mediaAdapter;

    public void play(String audioType, String fileName) {
        if(audioType.equalsIgnoreCase("mp3")){
            System.out.println("Playing mp3 file: " + fileName);
        } else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){
            mediaAdapter = new MediaAdapter(audioType); // Utilise l'Adapter
            mediaAdapter.play(audioType, fileName);
        } else {
            System.out.println("Invalid media. " + audioType + " format not supported");
        }
    }
}
```

Adapter : Vue d’ensemble structurale



Adapter : Bilan et Références

Aspect	Adapter
Objectif	Adapter une interface existante à celle attendue
Utilisation typique	Intégration de classes tierces ou légacies
Complexité	Faible à moyenne
Modèle relationnel	Composition
Type de pattern	Structurel

Ce qu'il faut retenir : L'Adapter facilite la réutilisation dans des systèmes hétérogènes en comblant le fossé entre interfaces différentes, sans modifier les classes existantes ni compromettre la modularité.

Sources :

- Refactoring.Guru – Adapter Pattern
- Wikipedia – Adapter Pattern
- Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Design Patterns de Structure : Adapter

Deux approches : Classe et Objet

Le pattern **Adapter** permet la collaboration entre des interfaces incompatibles en adaptant l'un des objets à l'interface attendue.

Il existe deux implémentations principales :

1. **L'Adapter par classe**
2. **L'Adapter par objet**

Ces approches ont leurs propres avantages et contraintes, influencées par le langage et le contexte applicatif.

L’Adapter par Classe (Class Adapter)

Principe

L’adaptateur **hérite** de l’interface cible attendue et de la classe adaptée. Il réalise l’adaptation en héritant des fonctionnalités et en les redéfinissant selon l’interface requise.

Caractéristiques

- Utilise l’**héritage multiple** (ou interfaces combinées) pour lier l’interface source et l’interface cible.
- Permet de substituer une interface de classe directement à une autre.
- **Limité** aux langages supportant l’héritage multiple (ex : C++).
- Plus **couplé** à la classe adaptée, donc moins flexible en cas de changement de la classe de base.

Exemple (Java, via interface et héritage simple)

```java
interface Target { void request(); }

class Adaptee {
    void specificRequest() { System.out.println("Appel spécifique dans Adaptee"); }
}

class ClassAdapter extends Adaptee implements Target {
    public void request() {
        specificRequest();  // adaptation via héritage
    }
}
```

L’Adapter par Objet (Object Adapter)

Principe

L’adaptateur **contient** une instance de la classe adaptée (composition) et redirige les appels vers cette instance en adaptant les interfaces.

Caractéristiques

- Repose sur la **composition** au lieu de l’héritage.
- Plus **souple** car il peut adapter des objets à runtime.
- Supporté dans **tous les langages orientés objets**.
- **Découplage** plus fort entre l’adaptateur et la classe adaptée.

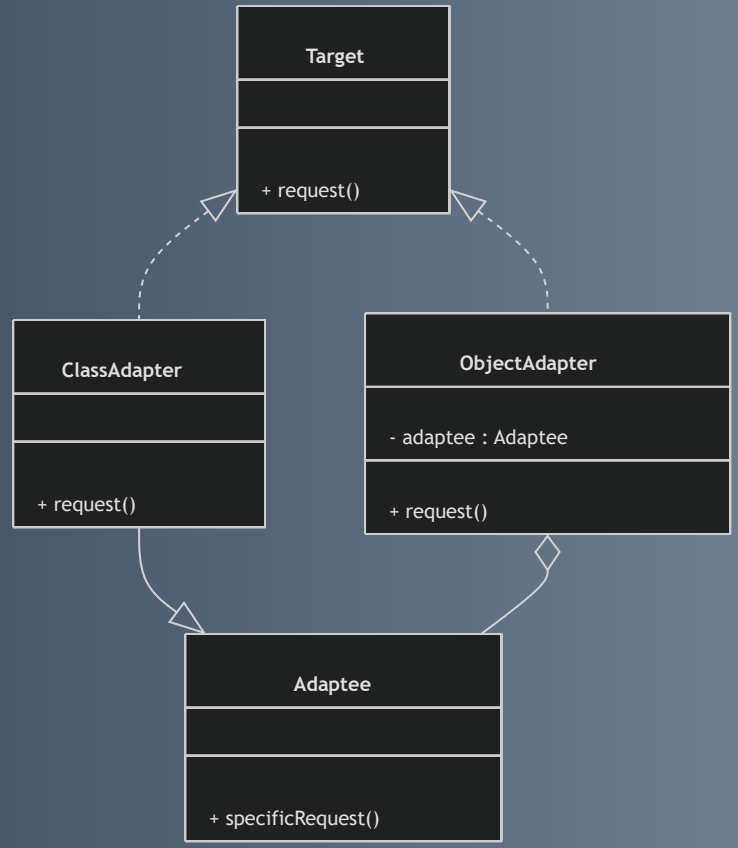
Exemple

```
interface Target { void request(); }

class Adaptee {
    void specificRequest() { System.out.println("Appel spécifique dans Adaptee"); }
}

class ObjectAdapter implements Target {
    private Adaptee adaptee;

    public ObjectAdapter(Adaptee adaptee){
        this.adaptee = adaptee;
    }

    public void request() {
        adaptee.specificRequest();  // délégation
    }
}
```

Adapter : Comparaison des Approches

Critère	Adapter par Classe	Adapter par Objet
Mécanisme	Héritage multiple	Composition
Flexibilité	Faible (fort couplage statique)	Élevée (objet adaptable à runtime)
Langage requis	Langage supportant héritage multiple (C++)	Tous langages orientés objet
Réutilisation	Réutilisation limitée sur classes non modifiables	Adaptation dynamique possible
Complexité	Simple si héritage possible	Léger surcoût d'abstraction

Adapter : Illustration Visuelle



L’Essentiel à Retenir & Ressources

Ce qu’il faut retenir

- **Adapter par classe** est simple et efficace si le langage supporte l’héritage multiple et que la classe adaptée peut être directement héritée.
- **Adapter par objet** offre plus de souplesse, favorise un découplage fort et permet l’adaptation dynamique des instances.
- La majorité des implémentations modernes privilégient l’**Adapter par objet** pour sa flexibilité et sa compatibilité universelle.

Le choix entre les deux est stratégique et dépend du langage, des besoins de flexibilité et de l’architecture globale.

Pour aller plus loin

- Refactoring.Guru – Adapter Pattern
- Wikipedia – Adapter pattern
- Gamma, Helm, Johnson, Vlissides – *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Design Patterns de Structure : L’Adapter

Faciliter l’intégration et la migration de code

Le pattern **Adapter** facilite la collaboration entre des composants dont les interfaces sont incompatibles. C’est un besoin courant lors :

- De l’intégration de bibliothèques externes.
- De la migration ou refonte de systèmes existants.

Cas d’usage 1 : Intégration de Bibliothèques Tierces

La problématique : L’interface d’une bibliothèque tierce ne correspond pas à celle attendue par l’application, et il n’est pas possible ou recommandé de la modifier.

La solution : L’Adapter joue le rôle de **pont**. Il encapsule la bibliothèque tierce et fournit une interface compatible à l’application.

Exemple : Interface attendue par l’application

```
public interface ImageConverter {
    void convert(String sourcePath, String destPath);
}
```

Exemple 1 : Adaptation d'une bibliothèque de conversion d'images

La bibliothèque tierce (incompatible) :

```
class ThirdPartyImageLib {
    public void changeFormat(String inputFile, String outputFile, String type) {
        System.out.println("Conversion en " + type + " : " + inputFile + " → " + outputFile);
    }
}
```

L'Adapter : Une interface compatible pour l'application

```
public class ImageConverterAdapter implements ImageConverter {
    private ThirdPartyImageLib thirdParty;

    public ImageConverterAdapter(ThirdPartyImageLib thirdParty) {
        this.thirdParty = thirdParty;
    }

    @Override
    public void convert(String sourcePath, String destPath) {
        // L'adaptateur "traduit" l'appel vers la bibliothèque tierce
        thirdParty.changeFormat(sourcePath, destPath, "PNG"); // Ex: toujours en PNG
    }
}
```

Utilisation dans l'application principale :

```
public class App {
    public static void main(String[] args) {
        ThirdPartyImageLib lib = new ThirdPartyImageLib();
        ImageConverter converter = new ImageConverterAdapter(lib); // Utilise l'adaptateur

        converter.convert("photo.jpg", "photo_converted.png"); // Appel via l'interface attendue
    }
}
```

Cas d’usage 2 : Migration progressive de code existant

La problématique : Lors d’une migration ou refonte, des parties d’un ancien système doivent être conservées. L’ancienne API ne correspond plus aux nouvelles spécifications.

La solution : L’adaptateur permet d’encapsuler l’implémentation ancienne, en fournissant la nouvelle interface attendue par le système moderne.

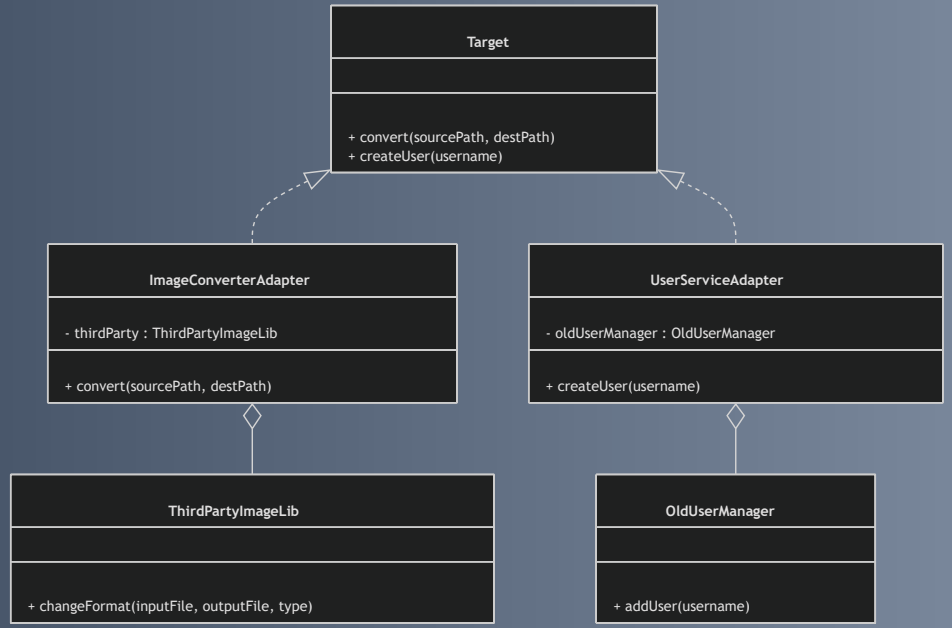
Exemple : Ancienne classe et nouvelle interface cible

```
// Ancienne classe
class OldUserManager {
    public void addUser(String username) {
        System.out.println("Utilisateur " + username + " ajouté dans l'ancien système");
    }
}

// Nouvelle interface cible
interface UserService {
    void createUser(String username);
}
```

L’Adapter pour faire le pont :

```
class UserServiceAdapter implements UserService {
    private OldUserManager oldUserManager;

    public UserServiceAdapter(OldUserManager oldUserManager) {
        this.oldUserManager = oldUserManager;
    }

    public void createUser(String username) {
        // L'adaptateur "traduit" le nouvel appel vers l'ancien système
        oldUserManager.addUser(username);
    }
}
```

L’Adapter : Structure et Bénéfices



Résumé des cas d’usage :

Cas d’usage	Rôle de l’Adapter
Intégration de bibliothèques tierces	Adaptation d’une interface tierce pour qu’elle corresponde à l’interface client
Migration/réécriture progressive	Encapsulation de l’ancien système sous la nouvelle interface pour compatibilité

L’Adapter : Un pilier pour la flexibilité logicielle

Ce qu’il faut retenir : L’adaptateur joue un rôle clé dans la modernisation et la maintenance des systèmes logiciels. Il assure une intégration fluide sans modifier les composants tiers ou hérités, minimisant ainsi les risques et coûts de réécriture.

Sources :

- Refactoring.Guru – Adapter Pattern
- Wikipedia – Adapter Pattern
- Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Design Patterns de Structure : Decorator

Ajouter des responsabilités dynamiquement

Le pattern **Decorator** permet d’ajouter dynamiquement des responsabilités ou comportements supplémentaires à un objet sans modifier sa structure.
C’est une alternative souple à la sous-classe pour étendre les fonctionnalités.

Pourquoi l’utiliser ?

- Éviter une prolifération rigide de sous-classes.
- Enrichir un objet à l’exécution.
- Maintenir la flexibilité de l’architecture.

Le Pattern Decorator : Mécanisme et Intentions

Comment ça marche ?

Le Decorator est un pattern structurel. Il **enveloppe un objet original** dans un autre objet décorateur, qui **implémente la même interface**. Grâce à cette composition, le décorateur peut :

- Intercepter les appels.
- Modifier les comportements.
- Compléter les fonctionnalités de l'objet original.

Objectifs clés :

- **Ajout dynamique :** Permettre l'ajout de fonctionnalités à un objet à l'exécution.
- **Flexibilité :** Offrir une extension flexible et modulaire des comportements.
- **Responsabilité unique :** Respecter ce principe en cloisonnant les ajouts dans des décorateurs spécifiques.

Démonstration Pratique : Gestion de Données (Java I/O Inspired)

Le Composant de Base : Interface et Implémentation

Pour illustrer, imaginons une source de données simple.

1. Interface de base : Définit les opérations fondamentales.

```
public interface DataSource {
    void writeData(String data);
    String readData();
}
```

2. Classe concrète (composant de base) : La source de données primaire.

```java
public class FileDataSource implements DataSource {
    private String filename;
    private String storage = "";

    public FileDataSource(String filename) {
        this.filename = filename;
    }

    @Override
    public void writeData(String data) {
        // Simule l'écriture dans un fichier
        storage = data;
    }

    @Override
    public String readData() {
        return storage;
    }
}
```

Démonstration Pratique : Ajouter des Comportements

Les Décorateurs : Enrichir sans modifier

Nous allons ajouter une fonctionnalité de chiffrement sans toucher à `FileDataSource`.

1. Décorateur abstrait : La base pour tous les décorateurs. Il délègue par défaut.

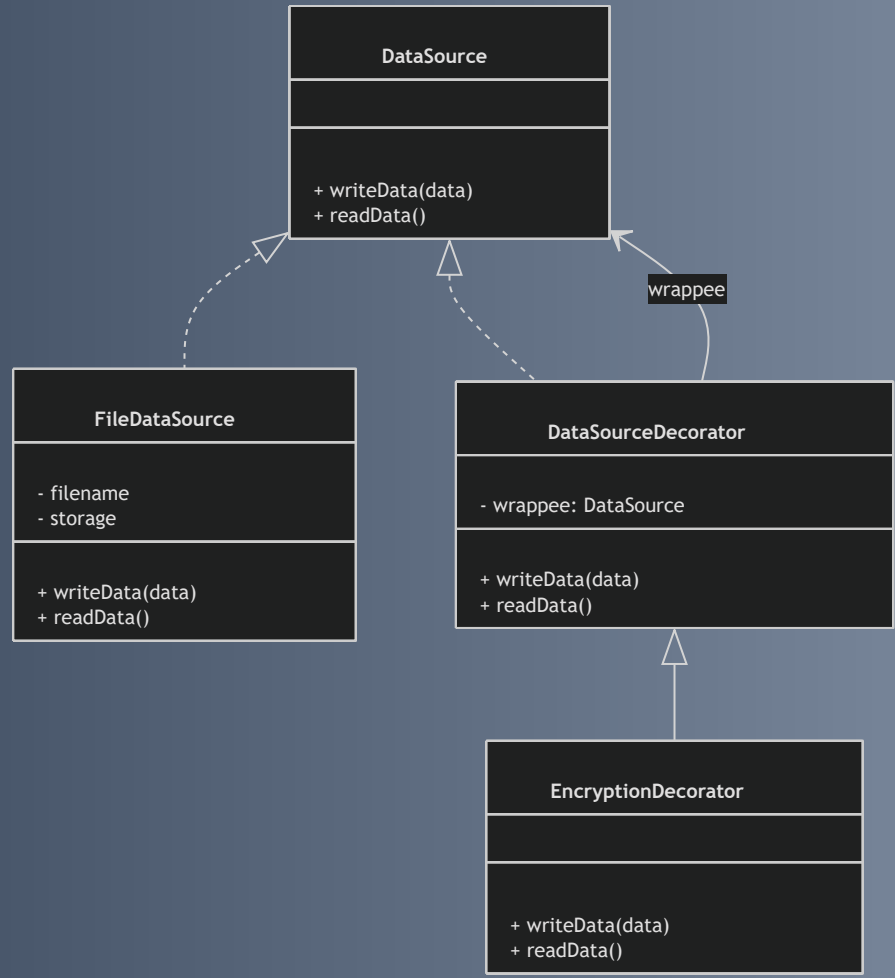
```java
public class DataSourceDecorator implements DataSource {
    protected DataSource wrappee; // L'objet enveloppé

    public DataSourceDecorator(DataSource source) {
        this.wrappee = source;
    }

    @Override
    public void writeData(String data) {
        wrappee.writeData(data); // Délégation
    }

    @Override
    public String readData() {
        return wrappee.readData(); // Délégation
    }
}
```

2. Décorateur concret : Chiffrement simple

```java
public class EncryptionDecorator extends DataSourceDecorator {

    public EncryptionDecorator(DataSource source) {
        super(source);
    }

    @Override
    public void writeData(String data) {
        String encrypted = "ENCRYPT(" + data + ")";
        super.writeData(encrypted); // Ajout avant délégation
    }

    @Override
    public String readData() {
        String data = super.readData();
        return data.replace("ENCRYPT(", "").replace(")", ""); // Ajout après délégation
    }
}
```

3. Utilisation : Empiler les fonctionnalités

```java
public class Client {
    public static void main(String[] args) {
        DataSource fileSource = new FileDataSource("file.txt");
        // On "décore" fileSource avec un chiffrement
        DataSource encryptedSource = new EncryptionDecorator(fileSource);

        encryptedSource.writeData("mySensitiveData");
        System.out.println(encryptedSource.readData());  // Affiche : mySensitiveData
    }
}
```


DataSource
+ writeData(data)
+ readData()
wrappee
FileDataSource
- filename
- storage
+ writeData(data)
+ readData()
DataSourceDecorator
- wrappee: DataSource
+ writeData(data)
+ readData()
EncryptionDecorator
+ writeData(data)
+ readData()

Synthèse et Pour Aller Plus Loin

Ce qu'il faut retenir du Decorator :

Aspect	Decorator
But	Ajouter dynamiquement des responsabilités
Structure	Composition (objets décorateurs enveloppent l'objet)
Extensibilité	Très élevée, sans multiplier les sous-classes
Couplage	Faible, permet séparation des responsabilités
Domaine d'application	Bibliothèques I/O, interfaces graphiques, logs, …

Conclusion : Le pattern Decorator offre une méthode élégante pour enrichir les fonctionnalités d'un objet à l'exécution sans le rigidifier par une hiérarchie complexe de classes. Il favorise la modularité, la réutilisabilité et un code plus maintenable.

Sources et références :

- Refactoring.Guru – Decorator Pattern
- Wikipedia – Decorator pattern
- Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Design Patterns de Structure

Decorator : Encapsulation et Composition

Le pattern **Decorator** s'appuie sur l'**encapsulation** et la **composition**.

Ces principes fondamentaux permettent d'ajouter dynamiquement et de manière transparente des fonctionnalités à un objet existant, sans en modifier la structure ni créer une hiérarchie complexe de sous-classes.

L’Encapsulation au cœur du Decorator

L’**encapsulation** masque l’objet original à l’intérieur du décorateur.

Le décorateur agit comme un **intercepteur** :

- Il reçoit les appels du client.
- Il peut les modifier, les étendre ou les filtrer.
- Il délègue ensuite ces appels à l’objet encapsulé (avant ou après son propre traitement).

L’objet décoré conserve son interface, mais son comportement est enrichi "à la volée".

La Composition : Fondation de la Flexibilité

Le Decorator privilégie la **composition** plutôt que l'héritage statique pour sa flexibilité.

- Le décorateur **implémente la même interface** que l'objet qu'il décore.
- Il **contient une référence** à une instance de cette interface (l'objet à décorer).
- Quand une méthode est appelée sur le décorateur, il peut :
  - Ajouter ou modifier un comportement.
  - Puis déléguer l'appel à l'objet interne.

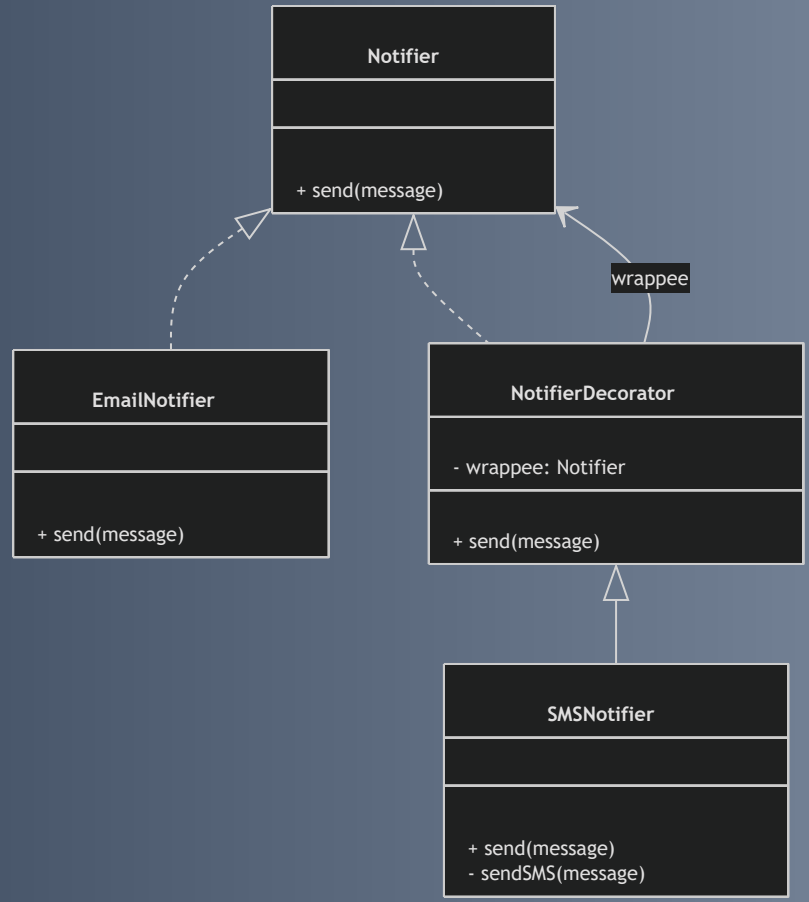
Cette approche permet d'**empiler plusieurs décorateurs**, chacun ajoutant une responsabilité spécifique à l'objet de base.

Exemple concret : Système de notifications (Java)

```java
// 1. Interface commune
public interface Notifier {
    void send(String message);
}

// 2. Classe concrète de base
public class EmailNotifier implements Notifier {
    public void send(String message) {
        System.out.println("Envoi de mail: " + message);
    }
}

// 3. Décorateur abstrait
public abstract class NotifierDecorator implements Notifier {
    protected Notifier wrappee; // Référence à l'objet décoré

    public NotifierDecorator(Notifier notifier) {
        this.wrappee = notifier;
    }
    public void send(String message) {
        wrappee.send(message); // Délégation par défaut
    }
}
```

-> Suite ->

```java
// 4. Décorateur concret : Ajout SMS
public class SMSNotifier extends NotifierDecorator {
    public SMSNotifier(Notifier notifier) { super(notifier); }

    @Override
    public void send(String message) {
        super.send(message); // Appel au comportement de base
        sendSMS(message);    // Ajout du comportement SMS
    }
    private void sendSMS(String message) {
        System.out.println("Envoi de SMS: " + message);
    }
}

// 5. Utilisation (Client)
public class Client {
    public static void main(String[] args) {
        Notifier notifier = new EmailNotifier(); // Notifier de base
        // Ajout dynamique d'un comportement SMS
        Notifier smsNotifier = new SMSNotifier(notifier);

        smsNotifier.send("Hello Decorator!");
        // Output :
        // Envoi de mail: Hello Decorator!
        // Envoi de SMS: Hello Decorator!
    }
}
```


Notifier
+ send(message)
wrappee
EmailNotifier
+ send(message)
NotifierDecorator
- wrappee: Notifier
+ send(message)
SMSNotifier
+ send(message)
- sendSMS(message)

Points clés & Sources

Concept	Explication
Encapsulation	Le décorateur cache l'objet original auquel il délègue les appels.
Composition	Le décorateur possède une référence à un objet de même interface, rendant le comportement modifiable à la volée.
Extensibilité	Plusieurs décorateurs peuvent être empilés pour enrichir un objet.
Faible couplage	Le décorateur manipule l'objet via son interface, limitant la dépendance à une implémentation concrète.

Grâce à l'encapsulation et à la composition, le pattern Decorator met en place un mécanisme puissant permettant d'enrichir les objets de manière modulaire, favorisant la réutilisabilité et la maintenance du code.

Sources :

- Refactoring.Guru – Decorator Pattern
- Wikipedia – Decorator pattern
- Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Design Patterns de Structure

Decorator : Flexibilité et dynamique, au-delà de l’héritage

Le pattern **Decorator** est une alternative puissante à l’héritage pour ajouter des fonctionnalités. Il se distingue par sa capacité à étendre le comportement d’un objet de manière **dynamique**, offrant une flexibilité et une maintenance accrues.

Héritage vs Decorator : Deux philosophies d’extension

L’Héritage : Extension Statique

- Crée une nouvelle classe dérivée, modifiant le comportement du parent.
- **Extension statique** à la compilation.
- Nécessite une hiérarchie de classes pour chaque combinaison.
- Provoque souvent une **explosion du nombre de sous-classes** (le "diamant").
- Comportement *figé* dans la classe héritée.

Le Decorator : Extension Dynamique

- Repose sur la **composition** et l’encapsulation.
- **Ajoute dynamiquement** des responsabilités à un objet à l’exécution.
- Permet d’**empiler plusieurs décorateurs** pour des combinaisons flexibles.
- **Évite la multiplication des sous-classes**.
- Ajoute ou retire facilement des fonctionnalités **sans modifier la classe originale**.

Le piège de l’héritage : L’explosion des classes

```
class Beverage {
    String getDescription() { return "Boisson"; }
    double cost() { return 1.0; }
}

class Coffee extends Beverage { /* ... */ }

class CoffeeWithMilk extends Coffee {
    @Override
    String getDescription() { return super.getDescription() + ", lait"; }
    @Override
    double cost() { return super.cost() + 0.5; }
}

class CoffeeWithMilkAndSugar extends CoffeeWithMilk {
    @Override
    String getDescription() { return super.getDescription() + ", sucre"; }
    @Override
    double cost() { return super.cost() + 0.3; }
}
// ... et ainsi de suite pour CHAQUE combinaison possible !
```

Inconvénients majeurs :

- **Multiplication exponentielle** des classes.
- Rigidité : les combinaisons sont fixes à la compilation.

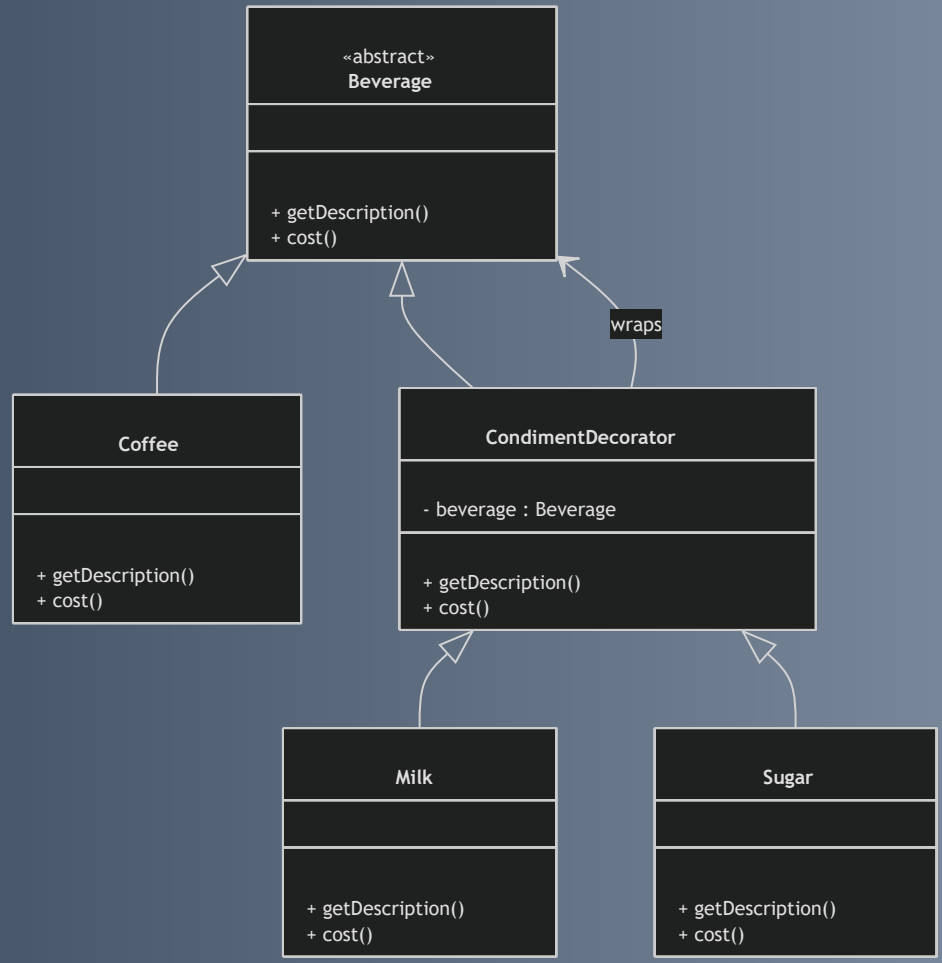
Decorator en action : Composition et agilité

```
abstract class Beverage { /* ... */ } // Interface commune
class Coffee extends Beverage { /* ... */ }

abstract class CondimentDecorator extends Beverage { // Le Decorator abstrait
    protected Beverage beverage;
    CondimentDecorator(Beverage beverage) { this.beverage = beverage; }
    // Délègue les appels au composant wrappé
}

class Milk extends CondimentDecorator { /* ... */ }
class Sugar extends CondimentDecorator { /* ... */ }
```

Utilisation dynamique : Empilez les fonctionnalités !

```
Beverage myDrink = new Coffee(); // Une simple boisson
myDrink = new Milk(myDrink);     // J'y ajoute du lait
myDrink = new Sugar(myDrink);    // J'y ajoute du sucre

System.out.println(myDrink.getDescription() + " €" + myDrink.cost());
// Sortie : "Café, lait, sucre €2.8"
```

Avantages : Combinaisons à la volée, sans créer de nouvelles classes.


«abstract»
Beverage
+ getDescription()
+ cost()
wraps
Coffee
+ getDescription()
+ cost()
CondimentDecorator
- beverage : Beverage
+ getDescription()
+ cost()
Milk
+ getDescription()
+ cost()
Sugar
+ getDescription()
+ cost()

Critère	Héritage	Decorator
Extensibilité	Statique, nombreuses sous-classes	Dynamique, empilement et combinaisons flexibles
Nombre de classes	Explosif pour multiples combinaisons	Réduit grâce aux décorateurs composables
Couplage	Fort (parent-fille)	Faible (interface et composition)
Modification à la volée	Impossible	Possible, comportement dynamique à l’exécution
Maintenance	Complexe (hiérarchies étendues)	Plus simple (modularité)

L’essentiel à retenir & Références

Ce qu’il faut retenir

La **composition** via le pattern **Decorator** offre une alternative élégante à l’héritage pour l’extension des fonctionnalités. Il apporte :

- Une **souplesse** inédite dans l’ajout de comportements.
- Une **modularité** accrue des composants.
- Une **maintenance** facilitée, évitant l’explosion des classes.

Sources

- Refactoring.Guru – Decorator Pattern
- Wikipedia – Decorator pattern
- Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.