**1. Singleton**

* **Description** : Garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global.
* **Exemple** :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

 **Avantages** :

* Contrôle global de l'accès à une instance unique.
* Réduction des ressources inutiles.

 **Inconvénients** :

* Peut introduire des dépendances cachées.
* Difficulté dans les tests unitaires (couplage fort).

**Indicateurs à chercher :**

* **Une méthode statique souvent nommée getInstance() ou équivalente.**
* **Un constructeur privé empêchant l’instanciation depuis l'extérieur.**
* **Une variable statique contenant l'instance unique.**

**Comment l’identifier :**

* **Motifs statiques** : Recherchez les classes qui ont des variables/méthodes statiques gérant une instance unique.
* **Logs ou commentaires** mentionnant des restrictions d'instanciation.

**2. Factory Method**

* **Description** : Définit une interface pour créer des objets, mais permet aux sous-classes de décider quel objet instancier.
* **Exemple** :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

 **Avantages** :

* Simplifie la création d'objets complexes.
* Permet d'étendre sans modifier le code existant.

 **Inconvénients** :

* Nécessite des sous-classes pour chaque type d'objet.

**Indicateurs à chercher :**

* Une méthode qui retourne des instances d’une classe sans exposer les détails de leur construction.
* Des classes héritant d'une superclasse ou implémentant une interface, qui surchargent cette méthode pour fournir des implémentations spécifiques.

**Comment l’identifier :**

* **Méthodes avec des noms comme create, make, ou build**, suivies de l’instanciation d’objets.
* **Abstraction** : Les classes utilisent des interfaces ou des superclasses pour fournir des implémentations.

**3. Observer**

* **Description** : Définit une dépendance "un-à-plusieurs" entre des objets pour qu'un changement dans un objet notifie automatiquement tous ses dépendants.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

 **Avantages** :

* Découple le sujet de ses observateurs.
* Permet une flexibilité dans la gestion des événements.

 **Inconvénients** :

* Peut devenir complexe avec beaucoup d'observateurs.

**Indicateurs à chercher :**

* Une classe centrale (*Sujet*) qui maintient une liste d'observateurs ou d'écouteurs.
* Des méthodes comme addObserver(), removeObserver(), ou notifyObservers().

**Comment l’identifier :**

* Recherchez des listes ou des collections d’objets observateurs.
* Cherchez des boucles invoquant une méthode commune sur plusieurs objets (ex. : update()).
* Vérifiez la notification d'événements ou de changements d’état.

**4. Decorator**

* **Description** : Ajoute dynamiquement des responsabilités à un objet sans affecter les autres objets de la même classe.
* **Exemple** :

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

 **Avantages** :

* Flexible pour ajouter des comportements dynamiques.
* Respecte le principe *Open/Closed*.

 **Inconvénients** :

* Peut introduire des couches inutiles de complexité.

**Indicateurs à chercher :**

* Une classe ayant un champ ou un constructeur recevant une instance d’une même interface ou superclasse.
* Des appels relayés vers cette instance, avec un comportement étendu.

**Comment l’identifier :**

* Recherche d'un **enchaînement d’appels d’une méthode** à travers plusieurs classes.
* Les **constructeurs prennent une instance de la même interface** pour étendre le comportement.

**5. Strategy**

* **Description** : Définit une famille d'algorithmes interchangeables qui peuvent être appliqués dynamiquement.
* **Exemple** :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

 **Avantages** :

* Réduit le couplage entre l'algorithme et son contexte.
* Favorise la réutilisabilité et l'extensibilité.

 **Inconvénients** :

* Complexité accrue en raison de la multiplication des classes.

**Indicateurs à chercher :**

* Une interface définissant une ou plusieurs méthodes, implémentée par plusieurs classes.
* Une classe *Context* qui contient une instance d'une stratégie et l'utilise pour exécuter des opérations.

**Comment l’identifier :**

* Cherchez des classes où un comportement est délégué dynamiquement à une autre classe.
* Les noms tels que setStrategy ou des relations avec des interfaces d’exécution sont souvent des indices.
* Présence de polymorphisme pour exécuter différentes stratégies.

**Comment l’identifier :**

* Cherchez des classes où un comportement est délégué dynamiquement à une autre classe.
* Les noms tels que setStrategy ou des relations avec des interfaces d’exécution sont souvent des indices.
* Présence de polymorphisme pour exécuter différentes stratégies.

**Exemple de Code :**

**Classe parent avec méthode Template :**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Sous-classe 1 :**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Sous-classe 2 :**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Utilisation :**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

Comment identifier le Template Method Pattern dans un projet ?

1. Motifs dans le code :

* Recherchez une méthode finale (final en Java) ou principale appelant d'autres méthodes abstraites ou protégées.
* Repérez des séquences de méthodes qui semblent répétitives mais varient dans certains détails.
* Ex. : prepareIngredients(), cookDish().

2. Hiérarchies de classes :

* Examinez les classes abstraites dans une hiérarchie. Vérifiez si elles contiennent une méthode principale qui suit un modèle rigide.

3. Diagrammes UML :

* Cherchez une classe abstraite avec une méthode principale et des méthodes redéfinies dans les sous-classes.

4. Noms des méthodes :

* Les méthodes des classes parent peuvent inclure des noms explicites comme templateMethod() ou des termes évoquant une séquence logique (ex. : execute, process, build).

5. Utilisation dans le code client :

* Le code client appelle uniquement la méthode principale (par exemple, cook() dans l’exemple). Les sous-classes ne sont pas directement appelées.

**Composite Pattern**

**Description :**

Le pattern Composite est utilisé pour représenter des structures d’objets hiérarchiques comme des arbres. Il permet de manipuler des objets simples (*feuilles*) et des objets composés (*nœuds*) de manière uniforme, en utilisant la même interface.

**Indicateurs à chercher pour identifier le Composite Pattern :**

1. **Interface commune :**
   * Une interface ou classe abstraite est utilisée par tous les composants (simples ou composés).
   * Exemple : Component, Shape, ou Node.
2. **Relations hiérarchiques :**
   * Une classe composite contient une collection d’objets de la même interface ou classe abstraite.
   * Exemple : List<Component> dans une classe comme Composite.
3. **Récursivité :**
   * Les objets composites contiennent eux-mêmes des composants (simples ou composites).
   * Les opérations sur un composite impliquent souvent des appels récursifs (ex. : un draw() qui parcourt tous les enfants).
4. **Méthodes uniformes :**
   * Les composants simples et composites partagent une méthode commune, souvent définie dans l'interface.
   * Exemple : draw(), move(), execute().

**Exemple de Code :**

**Interface Commune :**

Une image contenant texte, Police, capture d’écran, Graphique

Description générée automatiquement

Feuilles :

Une image contenant texte, capture d’écran

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Composite :

Une image contenant texte, capture d’écran

Description générée automatiquement

Utilisation :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**Avantages du Composite Pattern :**

1. **Uniformité :**
   * Les objets simples et composites sont manipulés de manière identique grâce à une interface commune.
2. **Simplicité :**
   * Réduit la complexité de gestion des structures hiérarchiques (ex. : arbres, graphes).
3. **Flexibilité :**
   * Il est facile d’ajouter de nouveaux types de composants ou de combiner des composites sans modifier le code existant.

**Inconvénients :**

1. **Complexité initiale :**
   * La mise en place d'une hiérarchie avec des composants simples et composites peut être complexe si la structure est petite.
2. **Faible contrôle sur les composants :**
   * L’utilisation uniforme peut masquer des différences essentielles entre les composants simples et composites, ce qui peut entraîner des erreurs.

**Comment identifier le Composite Pattern dans un projet ?**

**1. Structure hiérarchique :**

* Recherchez des classes qui forment des arbres ou d'autres relations imbriquées. Exemple typique : gestion d'une interface utilisateur ou de fichiers (ex. : *fichier* et *dossier*).

**2. Interface commune :**

* Vérifiez si plusieurs classes implémentent une même interface ou héritent d’une même classe abstraite.

**3. Méthodes récursives :**

* Les méthodes comme draw() ou execute() sont souvent appelées récursivement sur les composants enfants.

**4. Collections internes :**

* Cherchez des classes contenant une collection d'objets de leur propre type ou de leur interface parent (List<Component>).

**5. Diagrammes UML :**

* Recherchez une classe contenant des relations récursives (composition d'objets de même type).