

Les patrons de conception (design patterns)

Table des matières

I. Contexte	3
II. Les patrons de conception/Design Patterns	3
III. Exercice introduction aux patrons de conception	7
IV. Le patron de conception Singleton/Singleton Design Pattern	8
V. Exercice : Appliquer la notion	13
VI. Les patrons de conception Fabrique/Factory Design Pattern	14
VII. Exercice : Appliquer la notion	17
VIII. Le patron de conception Monteur/Builder Design Pattern	19
IX. Exercice : Appliquer la notion	23
X. Le patron de conception Prototype/Prototype Design Pattern	25
XI. Exercice : Appliquer la notion	28
XII. Essentiel	29
XIII. Auto-évaluation	29
A. Exercice final	29
B. Exercice : Défi	32
Solutions des exercices	33

I. Contexte

Durée : 1 h

Environnement de travail : E.D.I. IntelliJ Idea

Prérequis : les méthodes, les common objects, les types primitives, les collections, les fonctions anonymes

Contexte

De plus en plus des développeurs utilisent plus ou moins les mêmes **méthodologies/fonctionnalités/techniques de codage**.

À mesure que de plus en plus de programmes sont développés en utilisant les paradigmes orientés objets, l'idée de réutiliser les **méthodologies/fonctionnalités/techniques de codage** pour résoudre des problèmes courants a donné lieu à l'identification d'un nombre de modèles connus sous le nom de **patrons de conception**.

Chaque patron de conception :

- Décrit un problème récurrent dans nos programmes
- Puis propose une solution efficace à ce problème
- Afin que nous puissions utiliser et ré-utiliser cette solution autant de fois que nous le voulons.

Ces patrons de conception sont définis pour pouvoir être utilisés avec un maximum de langages orientés objets.

Le nombre de ces modèles de conception augmente constamment. Le but de ce cours n'est pas de les lister tous, mais de présenter les plus utilisés et de fournir un ou plusieurs exemples de leur implémentation en Java.

II. Les patrons de conception/Design Patterns

Objectifs

- Comprendre les patrons de conception et leurs utilités/utilisations dans le langage Java.
- Différencier les types de patrons de conception les plus utilisés dans l'univers Java.
- Savoir comment et quand il faut les utiliser.

Mise en situation

Au cours de ce chapitre, nous allons introduire les patrons de conception, parler des bénéfices d'utilisation de ces patrons de conception et aborder les classifications de ces différents patrons de conception.

Définition | Patrons de conception | Design Patterns

Un patron de conception **nomme, décrit, explique** et permet d'**évaluer** une conception d'un système extensible et réutilisable, digne d'intérêt pour un problème récurrent.

En d'autres termes, les patrons de conception sont des solutions classiques aux problèmes courants de conception de logiciels. Ce sont des sortes de plans ou de schémas que l'on peut personnaliser afin de résoudre un problème récurrent dans notre code.

Un patron de conception est un *concept général* permettant de résoudre un problème spécifique. Nous pouvons nous inspirer des principes d'un design pattern et mettre en place une solution adaptée à un programme.

Cependant, on ne peut pas faire de *copier coller* comme avec des fonctions ou des bibliothèques prêtes à l'emploi. Nous avons affaire ici à des macro-concepts/solutions et non à des morceaux de code spécifiques.

Ils ne peuvent pas non plus être *simplement implémentés* comme avec des algorithmes où les instructions/étapes sont claires et précises. Nous avons affaire ici à un plan directeur, nous pouvons nous inspirer des fonctionnalités et de leurs résultats, mais la façon de les mettre en œuvre dépend de nous.

Fondamental **Éléments d'un patron de conception**

La plupart des patrons de conception sont représentés de manière très large afin de pouvoir être reproduits n'importe où et dans n'importe quel contexte.

En général, un patron de conception possède 4 éléments essentiels :

- *Le nom* : Le nom du modèle est utilisé pour fournir un nom unique et significatif au modèle qui définit un problème de conception et une solution pour cela.
Nommer un modèle de conception permet de se référer facilement aux autres.
Il devient également facile de fournir de la documentation pour l'expliquer et le mot de vocabulaire correct facilite la réflexion sur la conception.
- *Le problème récurrent décrivant où appliquer la solution* : Ceci explique le problème et son contexte.
 - Cela peut inclure la description d'une conception spécifique de problèmes tels que la représentation d'algorithmes en tant qu'objets.
 - Parfois, le problème comprendra une liste de conditions qui doivent être remplies avant qu'on applique une fonctionnalité.
- *La solution intéressante* : Décrit les éléments qui composent la conception, leurs relations, leurs responsabilités et leurs collaborations.
- *Les analyses de la solution et ses variantes* :
 - Les avantages et inconvénients de l'utilisation du patron
 - Les considérations sur l'implémentation
 - Exemples de code
 - Utilisations connues
 - Schémas de connexes...

Classification des patrons de conception

Les patrons de conception diffèrent par leur complexité, leur documentation et la mesure dans laquelle ils s'appliquent à un système en cours de conception.

Les patrons de conception peuvent être classés en fonction de leur intention ou de leur objectif.

On distingue 3 principaux groupes de patrons de conception:

- *Les patrons de création* : ils décrivent des mécanismes pour créer des objets.
- *Les patrons de structure* : ils décrivent comment assembler des objets et des classes dans des structures plus grandes.
- *Les patrons de comportement* : ils décrivent comment mettre en place des formes de collaboration entre objets.

Syntaxe

Nous avons dans le tableau ci-dessous une liste de patrons de conception selon leurs groupes respectifs.

Les patrons de création/ Creational Patterns	Les patrons de structure/ Structural Patterns	Les patrons de comportement/ Behavioral Patterns
Singleton	Adaptateur/Adapter	État/State
Fabrique/Factory	Decorateur/Decorator	Visiteur/Visitor
Fabrique abstraite/Abstract Factory	Composite	Itérateur/Iterator
Monteur/Builder	Procuration/Proxy	Observateur/Observer
Prototype	Pont/Bridge	Commande/Command
	Façade/facade	Médiateur/Mediator
	Poids mouche/Flyweight	Stratégie/Strategy
		Interpréteur/Interpreter
		Memento/Memento
		Patron de Méthode/Template Method
		Chaîne de responsabilité/Chain of Responsibility

Les patrons de création

Dans cette catégorie, il existe 5 patrons principaux :

Patron de Conception	Rôle
I. Singleton	Garantir qu'une seule instance d'un objet existe dans l'application, et pas plus.
II. Fabrique/Factory	Créer un objet dont le type dépend du contexte.
III. Fabrique abstraite/Abstract Factory	Fournir une interface unique pour instancier des objets d'une même famille.
IV. Monteur/builder	Construire des objets complexes en utilisant une approche étape par étape.
V. Prototype	Créer de nouveaux objets en clonant les objets que vous définissez comme prototypes.

Les patrons de structure

Dans cette catégorie, il existe 7 patrons principaux :

Patron de Structure/Structural Patterns	Rôle
I. Adaptateur/Adapter	Faire correspondre les interfaces de différentes classes.
II. Décorateur/Decorator	Ajouter des fonctionnalités aux objets de manière dynamique.
III. Composite	Avoir une structure arborescente d'objets simples et composites.

IV. Procuration/Proxy	Utiliser un substitut pour un objet.
V. Pont/Bridge	Séparer une grosse classe ou un ensemble de classes connexes en deux hiérarchies.
VI. Façade/Facade	Avoir une classe unique qui représente un sous-système entier.
VII. Poids mouche/Flyweight	Utiliser pour réduire l'encombrement de la mémoire.

Les patrons de comportement

Dans cette catégorie, il existe 11 patrons principaux :

Patron de Conception	Rôle
I. Visiteur/Visitor	Créer un objet dont le type dépend du contexte.
II. Observateur/Observer	Un moyen de notifier un changement à un certain nombre de classes.
III. Itérateur/Iterator	Accéder séquentiellement aux éléments d'une collection.
IV. Commande/Command	Encapsuler une demande de commande en tant qu'objet.
V. Médiateur/Mediator	Définir la communication simplifiée entre les classes.
VI. Stratégie/Strategy	Encapsuler un algorithme dans une classe.
VII. Interpréteur/Interpreter	Un moyen d'inclure des éléments linguistiques dans un programme.
VIII. Memento/Memento	Capturer et restaurer l'état interne d'un objet.
IX. Patron de Méthode/Template Method	Reporter les étapes exactes d'un algorithme à une sous-classe.
X. Chaîne de Responsabilité/Chain of Responsibility	Une manière de transmettre une demande entre une chaîne d'objets.
XI. État/State	Modifier le comportement d'un objet lorsque son état change.

Nous illustrerons les **patrons de création** qui sont les plus utilisés dans l'univers de programmation en Java.

Conseil Bénéfices des patrons de conception

- **Flexibilité** : En utilisant des modèles de conception, votre code devient flexible. Cela permet de fournir le niveau d'abstraction correct grâce auquel les objets sont faiblement couplés les uns aux autres, ce qui facilite la modification de votre code.

- **Ré-utilisabilité** : Des objets et des classes faiblement couplés peuvent rendre votre code plus réutilisable. Ce genre de code devient facile à tester par rapport à un code hautement couplé.
- **Vocabulaire commun** : Un vocabulaire partagé facilite le partage de votre code et de vos pensées avec d'autres membres de l'équipe. Cela crée plus de compréhension entre les membres de l'équipe concernant le code.
- **Utilisation des bonnes pratiques** : Les patrons de conception utilisent les solutions qui ont été appliquées avec succès aux problèmes. En apprenant ces modèles et le problème connexe, un développeur inexpérimenté en apprend beaucoup sur la conception de logiciels.

Syntaxe **À retenir**

- Utiliser les patrons de conception facilite la réutilisation des techniques de programmation qui ont fait leurs preuves.
- Le gain de temps énorme que procure l'utilisation des patrons de conception n'est pas négligeable sur un projet.
- On distingue 3 catégories de patron de conception qui sont les **patrons de création**, les **patrons de structure** et les **patrons de comportement**.

Il existe en outre quelques autres patrons de conceptions (**DAO, IOC et MCV, J2EE...**) qui n'ont pas été retenus dans la liste initiale des patrons de conception fournie dans le livre intitulé ***Design Patterns - Elements of Reusable Object-Oriented Software*** écrit par le "Gang of Four".

Complément **Pour aller plus loin**

Le livre des principaux contributeurs à l'identification des patrons de conception (Erich Gamma, Richard Helm, Ralph Johnson et John Vissides) : <http://wiki.c2.com/?DesignPatternsBook>¹

III. Exercice introduction aux patrons de conception

Exercice

Exercice

Quels sont les trois principaux groupes de patrons de conception ?

- ☐ Les patrons de création
- ☐ Les patrons de sécurisation
- ☐ Les patrons d'accès
- ☐ Les patrons de structure
- ☐ Les patrons de conception
- ☐ Les patrons de comportement

Exercice

Quels sont les bénéfices de l'utilisation des patrons de conception ?

¹ Design Patterns par Erich Gamma, Richard Helm, Ralph Johnson et John Vissides

- ☐ La flexibilité
- ☐ La sécurité
- ☐ La ré-utilisabilité
- ☐ Le vocabulaire commun
- ☐ Utilisation de bonnes pratiques

Exercice

Quel est le but d'un patron de conception ?

- ☐ Il fournit un morceau de code pour chaque problème rencontré dans le développement informatique
- ☐ Il permet d'apporter une solution à des problèmes récurrents
- ☐ Il fournit des bonnes pratiques de développement
- ☐ Il permet de découper un projet informatique en bloc de code

Exercice

Les patrons de conception ont été inventés pour quel langage ?

- ☐ Java
- ☐ C
- ☐ PHP
- ☐ Python
- ☐ Ils ne dépendent pas d'un seul langage

IV. Le patron de conception Singleton/Singleton Design Pattern

Objectifs

- Comprendre la problématique et l'intention derrière le patron de conception **Singleton**.
- Comprendre la solution proposée au problème récurrent de la programmation Java que ce patron propose.
- Savoir les différentes possibilités d'utilisation de ce patron et éviter les pièges d'utilisation abusive.

Mise en situation

La plupart des programmes ont souvent besoin d'une seule instance d'une classe qui puisse être accessible partout dans le programme.

Ces instances peuvent être par exemple des objets nécessaires pour la journalisation (log), la communication avec la BDD...

Comme ils sont fréquemment utilisés dans un programme, les recréer à chaque fois qu'on en a besoin rajoute de la complexité inutile et de la redondance de code.

Le patron de création Singleton que nous allons voir dans ce chapitre nous propose une solution efficace pour pallier ces problèmes.

Introduction

Si un programme n'a besoin que d'une seule instance d'une classe et que cette instance doit être accessible dans de nombreuses parties différentes d'un programme, nous contrôlons à la fois l'instanciation et l'accès en faisant de cette classe un **Singleton**.

Il existe plusieurs approches différentes pour mettre en œuvre un patron Singleton en Java et qui ont toutes des concepts communs tels que :

- Un constructeur `private` : pour restreindre l'instanciation de classe par d'autres classes en dehors de classe de Singleton.
- Une variable `private static` : de la classe Singleton qui est la seule instance de la classe.
- Une méthode `public static` : qui retourne l'instance de la classe, c'est le point d'accès global pour le monde extérieur pour obtenir l'instance de la classe Singleton.

On va voir ces différentes approches avec des exemples d'implémentation.

Exemple	Approches d'implémentation
	<ul style="list-style-type: none"> • Initiation hâtive/impatiente : Dans une initialisation hâtive, une instance d'une classe est créée bien avant qu'elle ne soit réellement requise. Cela se fait principalement au démarrage du système et c'est la méthode la plus simple pour créer une classe Singleton. <pre> 1 public class HativeSingleton { 2 private static final HativeSingleton instance = 3 new HativeSingleton(); 4 5 private HativeSingleton() { } 6 7 private static HativeSingleton getInstance() { 8 return instance; 9 } 10 }</pre> <p>Cet extrait de code fonctionne bien et c'est l'approche à utiliser si votre classe Singleton n'utilise pas beaucoup de ressources.</p> <p>Sinon, cette approche n'est pas très appropriée pour des programmes utilisant des grosses ressources telles que les systèmes de fichiers, les connections aux bases de données...</p> <ul style="list-style-type: none"> • Initiation paresseuse/tardive/patiente : En programmation informatique, l'initialisation paresseuse est la tactique consistant à retarder la création d'un objet, le calcul d'une valeur ou tout autre processus coûteux jusqu'à la première fois que cela est nécessaire. <p>Dans un patron Singleton, l'initialisation paresseuse limite la création de l'instance jusqu'à ce qu'elle soit demandée pour la première fois</p> <pre> 1 public class TardiveSingleton { 2 /** 3 * L'utilisation du mot clé volatile (Java 5+) permet d'éviter le cas où le 4 * tardiveSingleton.instance est non null 5 */ 6 private static volatile TardiveSingleton instance = null; 7 8 private TardiveSingleton() { } 9 10 public static TardiveSingleton getInstance() { 11 if (instance == null) { 12 synchronized (TardiveSingleton.class) { 13 instance = new TardiveSingleton(); 14 } 15 } 16 }</pre>

```

14     }
15     return instance;
16 }
17 }
    
```

Cet extrait de code fonctionne bien dans un programme avec un seul thread, mais présente des inconvénients par rapport à un programme ayant un système multithread et que plusieurs threads se retrouvent à l'intérieur de la condition `if () { ... }` en même temps.

- **Thread Safe Singleton** : C'est le moyen le plus simple de créer une classe Singleton. Il s'agit de synchroniser la méthode d'accès globale afin qu'un seul thread à la fois puisse exécuter cette méthode.

```

1 public class ThreadSafeSingleton {
2     private static ThreadSafeSingleton instance;
3
4     private ThreadSafeSingleton(){}
5
6     public static synchronized ThreadSafeSingleton getInstance(){
7         if(instance == null){
8             instance = new ThreadSafeSingleton();
9         }
10        return instance;
11    }
12 }
    
```

Cet extrait de code fonctionne bien et assure la sécurité des threads. Néanmoins il réduit les performances en raison du coût associé à la méthode synchronisée.

Pour éviter cette surcharge à chaque fois, vous avez aussi le principe de verrouillage réverifié.

- **Verrouillage réverifié** : Ce principe consiste à réverifier la variable d'instance dans un bloc synchronisé de la précédente approche pour résoudre le problème des systèmes multithreads et garantir qu'une seule instance d'une classe Singleton est créée.

```

1 public class TardiveSingleton {
2     /**
3      * L'utilisation du mot clé volatile (Java 5+) permet d'éviter le cas où le
4      * tardiveSingleton.instance est non null
5      */
6     private static volatile TardiveSingleton instance = null;
7
8     private TardiveSingleton() { }
9
10    public static TardiveSingleton getInstance() {
11        if (instance == null) {
12            synchronized (TardiveSingleton.class) {
13                if (instance == null) {
14                    instance = new TardiveSingleton();
15                }
16            }
17            return instance;
18        }
19    }
20 }
    
```

- **Enum Java** : Java garantit que toute valeur `enum` n'est instanciée qu'une seule fois dans un programme.

```

1 public enum EnumSingleton {
2
3     INSTANCE;
4 }
    
```

```

5     public static void doSomething(){ ... }
6 }

```

L'inconvénient de cet extrait de code est que le type `enum` n'autorise pas l'initialisation tardive.

- **readResolve()** : Supposons qu'un programme soit distribué et qu'il sérialise les objets dans un système de fichiers pour les lire ultérieurement si nécessaire.

Le problème avec une classe Singleton sérialisée est que chaque fois que nous la désérialisons, elle crée une nouvelle instance de la classe.

Prenons une classe Singleton qui implémente l'interface `Serializable` (pour sérialiser en Java) :

```

1 import java.io.*;
2
3 public class SerialisedSingleton implements Serializable {
4     private volatile static SerialisedSingleton instance = null;
5
6     public static SerialisedSingleton getInstance() {
7         if (instance == null) {
8             instance = new SerialisedSingleton();
9         }
10        return instance;
11    }
12    private int year = 2022;
13
14
15    public int getYear() {
16        return year;
17    }
18
19    public void setYear(int year) {
20        this.year = year;
21    }
22
23    static SerialisedSingleton premiereInstance = SerialisedSingleton.getInstance();
24
25    public static void main(String[] args) throws IOException {
26        /* Serialisation dans un fichier */
27        try (ObjectOutput output = new ObjectOutputStream(
28            new FileOutputStream("fichierPerso.ser"));
29            ObjectInput input = new ObjectInputStream(
30                new FileInputStream("fichierPerso.ser")))
31        ) {
32
33            output.writeObject(instance);
34            premiereInstance.setYear(2020);
35
36            SerialisedSingleton deuxiemeInstance = (SerialisedSingleton) input.readObject();
37
38            System.out.println("La première année ==> " + premiereInstance.getYear());
39            System.out.println("la deuxième année ==> " + deuxiemeInstance.getYear());
40        } catch (IOException | ClassNotFoundException e) {
41            e.printStackTrace();
42        }
43    }
44 }

```

```

1 C:\>javac SerialisedSingleton.java
2 La première année ==> 2021

```

```
3 la deuxième année ==> 2022
```

On voit que les deux variables ont des valeurs différentes de la variable `year` (année) ce qui implique qu'on a deux instances de notre classe `SerialisedSingleton` ().

Cependant, on ne veut pas avoir plusieurs instances de notre classe `SerialisedSingleton`.

Pour résoudre ce problème, nous devons inclure une méthode `readResolve()` dans notre classe `SerialisedSingleton`.

```
1 @Serial
2 protected Object readResolve() {
3     return getInstance();
4 }
```

Cette méthode sera appelée pour désérialiser l'objet et à l'intérieur de cette méthode renvoyer l'instance existante pour garantir qu'il n'y a qu'une seule instance à l'échelle de l'application.

```
1 import java.io.*;
2
3 public class SerialisedSingleton implements Serializable {
4     private volatile static SerialisedSingleton instance = null;
5
6     public static SerialisedSingleton getInstance() {
7         if (instance == null) {
8             instance = new SerialisedSingleton();
9         }
10        return instance;
11    }
12
13    @Serial
14    protected Object readResolve() {
15        return instance;
16    }
17
18    private int year = 2022;
19
20    public int getYear() {
21        return year;
22    }
23
24    public void setYear(int year) {
25        this.year = year;
26    }
27
28    static SerialisedSingleton premiereInstance = SerialisedSingleton.getInstance();
29
30    public static void main(String[] args) throws IOException {
31        /* Serialisation dans un fichier */
32        try (ObjectOutput output = new ObjectOutputStream(
33            new FileOutputStream("fichierPerso.ser"));
34            ObjectInput input = new ObjectInputStream(
35                new FileInputStream("fichierPerso.ser")))
36        {
37
38            output.writeObject(instance);
39            premiereInstance.setYear(2021);
40
41            SerialisedSingleton deuxiemeInstance = (SerialisedSingleton) input.readObject();
42        }
```

```

43         System.out.println("La première année ==> " + premiereInstance.getYear());
44         System.out.println("la deuxième année ==> " + deuxiemeInstance.getYear());
45     } catch (IOException | ClassNotFoundException e) {
46         e.printStackTrace();
47     }
48 }
49 }
50

```

```

1 C:\>javac SerialisedSingleton.java
2 La première année ==> 2021
3 la deuxième année ==> 2021

```

Cet extrait de code donne un résultat correct qui garantit que nous n'aurons qu'une seule instance de notre classe Singleton à chaque fois que nous en aurons besoin.

Remarque	Avantages et Inconvénients
-----------------	-----------------------------------

- **Avantages :**
 - Garantir l'unicité de l'instance d'une classe et garantir le point d'accès global à cette instance.
 - L'objet du Singleton est uniquement initialisé la première fois qu'il est appelé.
- **Inconvénients :**
 - Cachez une mauvaise conception, les composants peuvent avoir trop de visibilité les uns par rapport aux autres.
 - Étant donné que le constructeur d'une classe Singleton est `private` et qu'il n'est pas possible de remplacer la méthode `static`, tester un programme l'implémentant peut être assez compliqué car de nombreux frameworks Java reposent sur l'héritage lors de la création d'objets fictifs (Mocks en Java).

Syntaxe	À retenir
----------------	------------------

Utilisez le Singleton lorsque :

- L'une de vos classes doit fournir une seule instance à tous ses clients. Par exemple, une base de données partagée entre toutes les parties d'un programme.
- Vous voulez un contrôle absolu sur vos variables globales.
- Vous avez des classes donnant accès aux paramètres de configuration de l'application.
- Vous avez des classes contenant des ressources accessibles en mode partagé.

Complément	Aller plus loin : Parallélisme avec les autres patrons de conception de création
-------------------	---

Les patrons de conception **Fabrique Abstraite**, **Monteur** et **Prototype** peuvent tous être implémentés comme des patrons de conception **Singleton**.

V. Exercice : Appliquer la notion

Question

Nous voulons créer une classe commune pour la connectivité de base de données.

Celle-ci sera appelée très fréquemment pour établir la connexion du programme à la base de données.

Quelle approche allez-vous proposer de mettre en œuvre ? Gardez à l'esprit que vous ne devez pas créer une nouvelle connexion à chaque fois que vous demandez à vous connecter à la base de données (éviter la redondance).

Indice :

Avec une simple approche, chaque fois que la connexion DB est appelée, un nouvel objet sera créé. Cela conduira certainement à la fuite de mémoire et à l'exception de type `OutOfMemoryException` (À éviter !)

Indice :

L'approche correcte consiste donc à générer une classe Singleton pour la connexion à la base de données afin qu'une seule instance soit créée quel que soit le nombre de fois qu'elle est appelée pour la connectivité à la base de données à partir de différentes classes.

VI. Les patrons de conception Fabrique/Factory Design Pattern

Objectifs

- Comprendre la problématique et l'intention derrière le patron de conception **Fabrique**, **Méthode de Fabrique** et **Fabrique Abstraite**.
- Comprendre la solution proposée au problème récurrent de la programmation Java que ce patron propose.
- Connaître les différentes possibilités d'utilisation de ces patrons et éviter les pièges d'utilisation abusive.

Mise en situation

Une **Fabrique** est un patron de conception de création qui définit une interface pour créer des objets dans une classe-mère, mais délègue le choix des types d'objets à créer aux sous-classes.

Ce patron fonctionne sur le concept d'encapsulation en cachant la logique d'instanciation des objets aux applications clientes.

Méthode de Fabrique

La **Méthode de Fabrique** indique qu'il suffit de définir une interface ou une classe abstraite pour créer un objet, mais que les sous-classes décident de la classe à instancier.

En d'autres termes, les sous-classes sont responsables de la création de l'instance de la classe.

Syntaxe

```
1 public class SomeImplementation implements SomeInterface { ... }
2
3 interface SomeInterface { ... }
4
5 class SomeInterfaceFactory {
6     public SomeInterface newInstance() {
7         return new SomeImplementation();
8     }
9 }
```

Dans cet extrait de code, le code client n'a jamais besoin de savoir `SomeImplementation` et à la place il fonctionne en termes de `SomeInterface`.

Exemple Dans la JVM

Les exemples les plus connus de ce patron pour la JVM sont les méthodes de création de collection (`Singleton()`, `SingletonList()`, `SingletonMap()`) de la classe `Collections`.

Ces méthodes renvoient toutes des instances de la collection appropriée (`Set`, `List` ou `Map`) mais le type exact n'est pas pertinent.

La méthode `Stream.of()` et les nouvelles méthodes `Set.of()`, `List.of()` et `Map.ofEntries()` permettent de faire (fabriquer) de même avec des collections plus larges.

De plus, les méthodes `getInstance()` des classes `java.util.Calendar`, `ResourceBundle` et `NumberFormat` utilisent le patron Fabrique ainsi que la méthode `valueOf()` dans les classes Wrappers comme `Boolean`, `Integer`...

Exemple Implémentation personnalisée

Dans cet exemple, nous allons créer une interface `Polygone` qui sera implémentée par plusieurs classes concrètes. Une Fabrique De Polygone sera utilisée pour récupérer les objets de cette famille :

Tout d'abord, nous créons l'interface **Polygone** :

```
1 public interface Polygone {
2     String getType();
3 }
```

Ensuite, nous allons créer des implémentations de polygones comme le Carré, le Triangle, etc. qui implémentent cette interface et retournent un objet de type `Polygone`.

```
1 public static class Triangle implements Polygone {
2
3     @Override
4     public String getType() {
5         return "Triangle";
6     }
7 }
8
9 public static class Carre implements Polygone {
10
11     @Override
12     public String getType() {
13         return "Carré";
14     }
15 }
16
17 ...
```

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

```
1 public static class FabriqueDePolygone {
2     public Polygone getPolygone(int nombreDeCotes) {
3         return switch (nombreDeCotes) {
4             case 3 -> new Triangle();
5             case 4 -> new Carre();
6             case 5 -> new Pentagone();
7             case 6 -> new Hexagone();
8             ...
9             default -> throw new IllegalArgumentException("Ce polygone n'est pas utilisé
dans ce programme, Désolé!");
10        };
11    }
12 }
```

Remarquez comment le client peut s'appuyer sur cette Fabrique pour nous donner un polygone approprié, sans avoir à initialiser l'objet directement.

Fabrique Abstraite

Une **Fabrique Abstraite** est un patron de conception qui fonctionne comme une fabrique qui génère d'autres fabriques en fournissant une couche d'abstraction supplémentaire.

Exemple Dans la JVM

Les exemples les plus connus de ce patron pour la JVM se portent sur les méthodes `newInstance()` des classes du package XML (`javax.xml.parsers.DocumentBuilderFactory`, `javax.xml.transform.TransformerFactory` et `javax.xml.xpath.XPathFactory`).

Exemple Implémentation personnalisée

La **Fabrique Abstraite** définit une interface pour la création de chaque objet mais délègue la véritable création des produits aux classes concrètes de la **Fabrique**. Chaque type de Fabrique correspond à une certaine variété d'objets.

Le code client appelle les méthodes de création d'un objet **Fabrique** plutôt que de créer directement les objets à l'aide d'un constructeur (`new`). Tous les objets seront donc compatibles car chaque **Fabrique** a sa propre variante de l'objet.

Le code client manipule les **Fabriques** et les objets uniquement via leurs interfaces **Abstraites**, ce qui lui permet de fonctionner avec n'importe quelle variante de l'objet créée par un objet **Fabrique**. Créez simplement une nouvelle classe **Fabrique** concrète et transmettez-la au code client.

Prenons par exemple une interface et ces implémentations concrètes :

```
1 interface FileSystem { ... }
2
3 class LocalFileSystem implements FileSystem { ... }
4
5 class NetworkFileSystem implements FileSystem { ... }
```

Ensuite, on a une interface et quelques implémentations concrètes pour implémenter une **Fabrique** :

```
1 interface FileSystemFactory { FileSystem newInstance(); }
2
3 class LocalFileSystemFactory implements FileSystemFactory { ... }
4
5 class NetworkFileSystemFactory implements FileSystemFactory { ... }
```

On a alors une **Méthode de Fabrique** pour obtenir la **Fabrique Abstraite** à travers laquelle on obtient une instance réelle.

```
1 class Example {
2     static FileSystemFactory getFactory(String fs) {
3         FileSystemFactory factory;
4         if ("local".equals(fs)) {
5             factory = new LocalFileSystemFactory();
6         } else if ("network".equals(fs)) {
7             factory = new NetworkFileSystemFactory();
8         }
9         return factory;
10    }
11 }
12 }
```

Dans cet extrait de code, on a une interface `FileSystemFactory` qui a deux implémentations concrètes.

Nous avons sélectionné l'implémentation exacte au moment de l'exécution, mais le code qui l'utilise n'a pas besoin de se soucier de l'instance réellement utilisée.

Ils renvoient chacun une instance concrète différente de l'interface `FileSystem`, mais encore une fois, notre code n'a pas besoin de se soucier exactement de l'instance que nous avons.

Souvent, nous obtenons la **Fabrique** elle-même en utilisant une autre **Méthode de Fabrique** comme décrit ci-dessus.

Dans notre exemple ici, la méthode `getFactory()` est elle-même une **Méthode de Fabrique** qui renvoie une `FileSystemFactory` **Abstraite** qui est ensuite utilisée pour construire un `FileSystem`.

Remarque Avantages et Inconvénients

- **Avantages :**

- Vous pouvez déplacer tout le code de création d'objets vers un seul endroit, ce qui permet une meilleure maintenabilité (**principe de responsabilité unique**).
- Vous pouvez ajouter de nouveaux types d'objets dans le programme sans endommager l'existant.
- Vous êtes assurés que les produits d'une fabrique sont compatibles entre eux.

- **Inconvénients :**

- Le code peut devenir plus complexe puisque vous devez introduire de nombreuses sous-classes pour la mise en place du patron. La condition optimale d'intégration du patron dans du code existant se présente lorsque vous avez déjà une hiérarchie existante de classes de création.

Syntaxe À retenir

Quand utiliser :

- Une **Fabrique** : lorsque vous ne souhaitez pas exposer la logique d'instanciation d'objet au client/appelant.
- Une **Méthode de Fabrique** : pour définir une interface pour créer un objet, mais laisser les sous-classes décider quelle classe instancier.
- Une **Fabrique Abstraite** : lorsque vous souhaitez fournir une interface à des familles d'objets liés ou dépendants sans spécifier leurs classes concrètes.

Complément Parallélisme avec les autres patrons de conception de création

- La conception commence à l'aide de la **Méthode de Fabrique** (moins compliquée, plus personnalisable, les sous-classes prolifèrent) et évolue vers la **Fabrique Abstraite** à mesure que le concepteur découvre où plus de flexibilité est nécessaire
- Les classes de **Fabrique Abstraite** sont souvent implémentées avec des méthodes de **Fabrique**, mais elles peuvent également être implémentées à l'aide d'un **Prototype**.

VIII. Exercice : Appliquer la notion

Question

Nous allons pouvoir mettre en pratique une Méthode de Fabrique à l'aide d'un exemple.

Prenons un exemple du cas d'utilisation de la vie quotidienne le plus courant : des achats en ligne.

Supposons que l'un des magasins de commerce électronique vende une offre combinée de vêtements de sport pour hommes, à savoir Nike, Adidas et Jordan.

On veut que les clients du magasin de basket puissent fournir différentes baskets selon les différents choix et sélections des utilisateurs.

Indice :

Le patron Fabrique résout le problème en fournissant les objets selon les baskets sélectionnées par le client.

Voici un exemple pratique d'implémentation du programme en Java à l'aide du patron des Méthodes de Fabrique.

```

1 public abstract class Vetement {
2     public abstract void getShirt();
3
4     public abstract void getSurvet();
5
6     public abstract void getBasket();
7 }
```

La classe `Vetement` ci-dessus est une classe abstraite et a des méthodes abstraites (principalement les produits qui seront ajoutés aux offres du magasin).

Indice :

Pour créer les classes de vêtements Nike, Adidas et Jordan, on va créer des classes les représentant étendant toute la classe abstraite `Vetement`.

```

1     static class Nike extends Vetement {
2         @Override
3         public void shirt() { System.out.println("Ajoutez un T-Shirt dans le panier de
vêtement Nike!"); }
4         @Override
5         public void survet() { System.out.println("Ajoutez un Survet dans le panier de
vêtement Nike!"); }
6         @Override
7         public void basket() { System.out.println("Ajoutez un Basket dans le panier de
vêtement Nike!"); }
8     }
9
10    static class Adidas extends Vetement {
11        @Override
12        public void shirt() { System.out.println("Ajoutez un T-Shirt dans le panier de
vêtement Adidas!"); }
13        @Override
14        public void survet() { System.out.println("Ajoutez un Survet dans le panier de
vêtement Adidas!"); }
15        @Override
16        public void basket() { System.out.println("Ajoutez un Basket dans le panier de
vêtement Adidas!"); }
17    }
18
19    static class Jordan extends Vetement {
20        @Override
21        public void shirt() { System.out.println("Ajoutez un T-Shirt dans le panier de
vêtement Jordan!"); }
22        @Override
23        public void survet() { System.out.println("Ajoutez un Survet dans le panier de
vêtement Jordan!"); }
24        @Override
25        public void basket() { System.out.println("Ajoutez un Basket dans le panier de
vêtement Jordan!"); }
26    }
```

Indice :

Ensuite on crée une classe abstraite dans laquelle il y a une méthode abstraite `creerPanierVetement ()` qui a un argument de type `String`, ce sera le nom du vêtement que les clients passeront de la classe concrète.

```

1     static class FabriqueVetement {
2         public static Vetement creerPanierVetement(String vetement) {
3             Vetement panier = switch (vetement) {
4                 case "nike" -> new Nike();
```

```

5         case "adidas" -> new Adidas();
6         case "jordan" -> new Jordan();
7         default -> throw new IllegalArgumentException("Pas de vetement de ce
modèle");
8     };
9     panier.getShirt();
10    panier.getSurvet();
11    panier.getBasket();
12
13    return panier;
14 }
15 }

```

IX. Le patron de conception Monteur/Builder Design Pattern

Objectifs

- Comprendre la problématique et l'intention derrière le patron de conception **Monteur**.
- Comprendre la solution proposée au problème récurrent de la programmation Java que ce patron propose.
- Savoir les différentes possibilités d'utilisation de ce patron et éviter les pièges d'utilisation abusive.

Mise en situation

La plupart des programmes ont souvent une classe avec des constructeurs complexes ou plusieurs nombres de paramètres et présentent des difficultés pour créer des objets de la même manière que les paramètres sont définis.

Le modèle **Monteur** résout ces types de problèmes en fournissant un moyen de créer l'objet étape par étape et une méthode qui renvoie l'objet final.

Introduction

Il existe plusieurs approches différentes pour mettre en œuvre un patron Monteur en Java, commençons par la plus basique :

```

1 class CarBuilder {
2     private String marque = "Ford";
3     private String modele = "Fiesta";
4     private int portes = 5;
5     private String couleur = "Black";
6
7     public Car build() {
8         return new Car(marque, modele, portes, couleur);
9     }
10 }

```

Cet extrait de code permet de fournir individuellement des valeurs pour la marque , le modèle , les portes et la couleur puis lorsque nous construisons la voiture , tous les arguments du constructeur sont résolus en valeurs stockées.

Exemple Dans la JVM

Les exemples les plus connus de ce patron pour la JVM sont les classes `StringBuilder` et `StringBuffer` qui sont des générateurs qui permettent de construire des chaînes de caractère en fournissant plusieurs autres paramètres.

Par exemple, la classe `Stream.Builder` :

```

1 Stream.Builder<Integer> builder = Stream.builder<Integer>();

```

```

2 builder.add(1);
3 builder.add(2);
4 if (condition) {
5     builder.add(3);
6     builder.add(4);
7 }
8 builder.add(5);
9 Stream<Integer> stream = builder.build();
    
```

De plus, les méthode `put()` des classes `java.nio.ByteBuffer`, `CharBuffer`, `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer` et `DoubleBuffer` et `addComponent()` de la classe `javax.swing.GroupLayout.Group`.

Exemple Implémentation personnalisée

Prenons par exemple une situation où un site de commerce électronique demande des informations personnelles à ses clients sous la forme d'un formulaire où au moins le nom doit être saisi.

Le site web souhaite stocker les informations des clients dans un objet immuable. On peut le faire en appelant directement le constructeur de la classe du formulaire. Cependant, le formulaire comporte plusieurs champs et on ne veut pas obliger le client à les remplir tous.

On va donc implémenter un patron Monteur pour ne pas créer le constructeur avec les différentes combinaisons.

```

1 public class FormulaireClient {
2     private String nom; // Obligatoire
3     private String age;
4     private String sex;
5     private String type;
6
7     private FormulaireClient(FormulaireClientBuilder builder) {
8         this.nom = builder.nom;
9         this.age = builder.age;
10        this.sex = builder.sex;
11        this.type = builder.type;
12    }
13
14    @Override
15    public String toString() {
16        return "FormulaireClient{" +
17            "nom='" + nom + '\'' +
18            ", age='" + age + '\'' +
19            ", sex='" + sex + '\'' +
20            ", type='" + type + '\'' +
21            '}';
22    }
23 }
    
```

Dans cet extrait de code, la classe `FormulaireClient` a un constructeur qui accepte l'objet générateur et initialise les paramètres de classe.

```

1 public class FormulaireClient {
2     ...
3     private static class FormulaireClientBuilder {
4         private String nom;
5         private String age;
6         private String sex;
7         private String type;
8
9         public FormulaireClientBuilder(String nom) {
    
```

```

10         this.nom = nom;
11     }
12
13     public FormulaireClientBuilder addAge(String age) {
14         this.age = age;
15         return this;
16     }
17
18     public FormulaireClientBuilder addSex(String sex) {
19         this.sex = sex;
20         return this;
21     }
22
23     public FormulaireClientBuilder addType(String type) {
24         this.type = type;
25         return this;
26     }
27
28     public FormulaireClient build() {
29         return new FormulaireClient(this);
30     }
31 }
32 }

```

Dans cet extrait de code, la classe interne `FormulaireClientBuilder` qui agit comme un générateur et a un constructeur public qui n'a que les paramètres requis comme arguments.

Cette classe a une méthode distincte d'ajout (`add ... ()`) pour chacun des paramètres facultatifs des champs non obligatoires du formulaire.

On a aussi l'implémentation de la classe **Builder** (Monteur) qui appelle le constructeur de la classe `FormulaireClientBuilder` et passe les arguments via le constructeur.

```

1 public class FormulaireClient {
2     private String nom; // Obligatoire
3     private String age;
4     private String sex;
5     private String type;
6
7     private FormulaireClient(FormulaireClientBuilder builder) {
8         this.nom = builder.nom;
9         this.age = builder.age;
10        this.sex = builder.sex;
11        this.type = builder.type;
12    }
13
14    @Override
15    public String toString() {
16        return "FormulaireClient{" +
17            "nom='" + nom + '\'' +
18            ", age='" + age + '\'' +
19            ", sex='" + sex + '\'' +
20            ", type='" + type + '\'' +
21            '}';
22    }
23
24    private static class FormulaireClientBuilder {
25        private String nom;
26        private String age;

```

```

27     private String sex;
28     private String type;
29
30     public FormulaireClientBuilder(String nom) {
31         this.nom = nom;
32     }
33
34     public FormulaireClientBuilder addAge(String age) {
35         this.age = age;
36         return this;
37     }
38
39     public FormulaireClientBuilder addSex(String sex) {
40         this.sex = sex;
41         return this;
42     }
43
44     public FormulaireClientBuilder addType(String type) {
45         this.type = type;
46         return this;
47     }
48
49     public FormulaireClient build() {
50         return new FormulaireClient(this);
51     }
52 }
53
54 public static void main(String[] args) {
55     FormulaireClient informationClient =
56         new FormulaireClient.FormulaireClientBuilder("Jane Doe")
57             .addAge("25")
58             .addSex("F")
59             .addType("Premium").build();
60
61     System.out.println(informationClient);
62 }
63
64 }
```

```
1 C:\>javac FormulaireClient.java
```

```
2 FormulaireClient{nom='Jane Doe', age='25', sex='F', type='Premium'}
```

On remarque que la classe `FormulaireClient` n'a pas de constructeur public. On a donc forcément utilisé `FormulaireClientBuilder` pour obtenir le formulaire.

Remarque	Avantages et Inconvénients
-----------------	-----------------------------------

- **Avantages :**

- Vous pouvez construire les objets étape par étape et les déléguer ou les exécuter récursivement.
- Vous pouvez réutiliser le même code de construction lorsque vous construisez différentes représentations des produits.

- **Inconvénients :**

- Le Monteur nécessite de créer beaucoup nouvelles classes, ce qui accroît la complexité générale du code.

Syntaxe **À retenir**

Utiliser le patron de conception Monteur lorsque :

- Le processus de création d'un objet est extrêmement complexe, avec de nombreux paramètres obligatoires et facultatifs
- Il faut construire une arborescence composite ou d'autres objets complexes.
- Le client attend des représentations différentes pour l'objet qui est construit.

Complément **Parallélisme avec les autres patrons de conception de création**

- Le **Monteur** se concentre sur la construction d'objets complexes étape par étape. La **Fabrique Abstraite** se spécialise dans la création de familles d'objets associés. La **Fabrique Abstraite** retourne le produit immédiatement, alors que le **Monteur** vous permet de lancer des étapes supplémentaires avant de récupérer le produit.
- Vous pouvez utiliser le **Monteur** lorsque vous créez des arbres **Composites** complexes, car vous pouvez programmer les étapes de la construction récursivement.

X. Exercice : Appliquer la notion

Question

Nous avons un logiciel écrit en Java pour une banque dont nous voulons représenter les comptes bancaires.

Nous voulons garder une trace du taux d'intérêt mensuel applicable à chaque compte bancaire, de la date à laquelle le compte bancaire a été ouvert et éventuellement la succursale dans laquelle le compte bancaire en question a été ouvert en plus des autres informations nécessaires telles que le numéro de compte.

Ci-dessous notre code d'implémentation naïve des comptes bancaires.

```

1 public class CompteBancaire {
2     private long numeroDeCompte;
3     private String proprietaireDuCompte;
4     private String succursale;
5     private double soldeDuCompte;
6     private double tauxDInteret;
7
8     public CompteBancaire(long numeroDeCompte, String proprietaireDuCompte, String succursale,
9 double soldeDuCompte, double tauxDInteret) {
10         this.numeroDeCompte = numeroDeCompte;
11         this.proprietaireDuCompte = proprietaireDuCompte;
12         this.succursale = succursale;
13         this.soldeDuCompte = soldeDuCompte;
14         this.tauxDInteret = tauxDInteret;
15     }
16
17     public long getNumeroDeCompte() {
18         return numeroDeCompte;
19     }
20
21     public void setNumeroDeCompte(long numeroDeCompte) {
22         this.numeroDeCompte = numeroDeCompte;
23     }
24
25     public String getProprietaireDuCompte() {
26         return proprietaireDuCompte;
27     }
28 }

```

```

27
28     public void setProprietaireDuCompte(String proprietaireDuCompte) {
29         this.proprietaireDuCompte = proprietaireDuCompte;
30     }
31
32     public String getSuccursale() {
33         return succursale;
34     }
35
36     public void setSuccursale(String succursale) {
37         this.succursale = succursale;
38     }
39
40     public double getSoldeDuCompte() {
41         return soldeDuCompte;
42     }
43
44     public void setSoldeDuCompte(double soldeDuCompte) {
45         this.soldeDuCompte = soldeDuCompte;
46     }
47
48     public double getTauxDInteret() {
49         return tauxDInteret;
50     }
51
52     public void setTauxDInteret(double tauxDInteret) {
53         this.tauxDInteret = tauxDInteret;
54     }
55
56     @Override
57     public String toString() {
58         return "CompteBancaire{" +
59             "numeroDeCompte=" + numeroDeCompte +
60             ", proprietaireDuCompte='" + proprietaireDuCompte + '\'' +
61             ", succursale='" + succursale + '\'' +
62             ", soldeDuCompte=" + soldeDuCompte + " euros " +
63             ", tauxDInteret=" + tauxDInteret + " %" +
64             '}';
65     }
66
67     public static void main(String[] args) {
68         CompteBancaire compte = new CompteBancaire(34L, "Joel", "Montpellier", 5000, 2.5);
69         CompteBancaire autreCompte = new CompteBancaire(75L, "Jean ", "Paris", 2.2, 1250);
70         System.out.println(compte.toString());
71         System.out.println(autreCompte.toString());
72     }
73 }

```

```
1 C:\>javac CompteBancaire.java
```

```

2 CompteBancaire[numeroDeCompte=34, proprietaireDuCompte='Joel', succursale='Montpellier',
  soldeDuCompte=5000.0 euros , tauxDInteret=2.5 %]
3 CompteBancaire[numeroDeCompte=75, proprietaireDuCompte='Jean ', succursale='Paris',
  soldeDuCompte=2.2 euros , tauxDInteret=1250.0 %] // OUPS, taux d'intérêt de Jean 🤖

```

On constate ici que l'ordre des paramètres peut être ambigu pour des classes comme celles-ci ou ayant encore bien plus de paramètres avec des types consécutifs ou non identiques.

Si on a plusieurs arguments consécutifs du même type, il est facile de les échanger accidentellement. Étant donné que le compilateur ne le considère pas comme une erreur, cela peut se manifester comme un problème quelque part au moment de l'exécution et cela peut se transformer en un exercice de débogage délicat.

Quelle approche allez-vous proposer de mettre en œuvre pour implémenter ce programme correctement ?

Indice :

Approche simple consistant à atténuer le problème en appelant le constructeur sans argument, puis configurer le compte via des méthodes Setter à la place d'un constructeur avec argument susceptible de provoquer des erreurs accidentelles.

Cette approche n'est pas appropriée car elle conduit à un autre problème :

- Que se passera-t-il si on oublie d'appeler une méthode de Setter particulière ?

Cela conduit à avoir des objets qui ne sont que partiellement initialisés. Le compilateur ne verrait aucun problème et on n'aura donc pas résolu la question.

Cette approche révèle d'un autre problème et ainsi on a deux problèmes spécifiques que nous devons résoudre.

Indice :

La bonne approche consiste donc à implémenter un **Monteur** car on aura la possibilité d'écrire du code lisible et compréhensible pour configurer des objets complexes comme celui qu'on a ici.

Le **Monteur** va contenir tous les champs de la classe `CompteBancaire`. On va donc déléguer la configuration de tous les champs que nous voulons dans le Monteur.

On va remplacer le constructeur `public` de la classe `CompteBancaire` par un constructeur `private` et ainsi déléguer aussi la création des comptes bancaire au Monteur.

XI. Le patron de conception Prototype/Prototype Design Pattern

Objectifs

- Comprendre la problématique et l'intention derrière le patron de conception **Prototype**.
- Comprendre la solution proposée au problème récurrent de la programmation Java que ce patron propose.
- Savoir les différentes possibilités d'utilisation de ces patrons et éviter les pièges d'utilisation abusive.

Mise en situation

Le patron **Prototype** fonctionne sur le concept de clonage d'objet, ce qui aide à la création de nouveaux objets à partir d'objets existants sans rendre le code dépendant de leur classe.

Dans les patrons de création, le **Prototype** est largement utilisé pour éviter le coût des créations d'objets. **Prototype** résout le problème de la création d'objets coûteux et chronophages.

Le Prototype fournit un mécanisme pour copier l'objet original dans un nouvel objet, puis y apporter des modifications en fonction des besoins exprimés.

En Java, on implémente un Prototype en utilisant la méthode `Object.clone()` de l'interface `Cloneable`. Cela produit un *clone superficiel* de l'objet créant ainsi une nouvelle instance.

Nous pouvons également copier les objets existants en personnalisant/redéfinissant la méthode `Object.clone()` afin de faire des *copies profondes*.

Syntaxe

```
1 public class Prototype implements Cloneable {
2     private Map<String, String> contents = new HashMap<>();
3
4     public void setValue(String key, String value) {
5         // ...
6     }
7     public String getValue(String key) {
```

```

8      // ...
9  }
10
11  @Override
12  public Prototype clone() {
13      Prototype result = new Prototype();
14      this.contents.entrySet().forEach(entry -> result.setValue(entry.getKey(),
15  entry.getValue()));
16      return result;
17  }

```

Dans cet extrait de code, on voit bien qu'on redéfinit la méthode `clone()` de la classe racine Java **Object** pour avoir une copie profonde de notre objet.

Exemple Dans la JVM

Les exemples les plus connus de ce patron pour la JVM sont les classes qui implémentent l'interface `Cloneable` (`PKIXCertPathBuilderResult`, `PKIXBuilderParameters`, `PKIXParameters`, `PKIXCertPathBuilderResult` et `PKIXCertPathValidatorResult`).

Un autre exemple est la classe `java.util.Date` qui, elle, redéfinit complètement la méthode `clone()`.

Exemple Implémentation personnalisée par copie profonde

Le patron de conception de Prototype exige que l'objet que vous copiez fournisse exclusivement une fonction de copie (Rappel : en java la méthode `clone()` de l'interface **Cloneable** pour faire des copies).

Cependant, l'utilisation ou non de la copie des propriétés de l'objet dépend des exigences et de sa décision de conception.

```

1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class Employees implements Cloneable {
5      private final List<String> listeDesEmployees;
6
7      public Employees() { this.listeDesEmployees = new ArrayList<>(); }
8
9      public Employees(List<String> listeDesEmployees) { this.listeDesEmployees =
10 listeDesEmployees; }
11
12  public void loadData() {
13      this.listeDesEmployees.add("Joel");
14      this.listeDesEmployees.add("Jean");
15      this.listeDesEmployees.add("Jeremy");
16  }
17
18  public List<String> getListeDesEmployees() { return listeDesEmployees; }
19
20  @Override
21  public Object clone() throws CloneNotSupportedException {
22      Object clone = super.clone();
23      List<String> liste = new ArrayList<>(this.getListeDesEmployees());
24      return new Employees(liste);
25  }
26
27  public static void main(String[] args) throws CloneNotSupportedException {
28      Employees listeDesEmployees = new Employees();

```

```

28     listeDesEmployees.loadData();
29
30     /* Utilisation de la méthode de clonage pour avoir la liste des employés */
31     Employees premiereListe = (Employees) listeDesEmployees.clone();
32     Employees deuxiemeListe = (Employees) listeDesEmployees.clone();
33
34     List<String> liste1 = premiereListe.getListeDesEmployees();
35     liste1.add("Rambo");
36     List<String> liste2 = deuxiemeListe.getListeDesEmployees();
37     liste2.remove("Jeremy");
38
39     System.out.println("La liste des employés : " +
40 listeDesEmployees.getListeDesEmployees());
41     System.out.println("La première liste ====>: " + liste1);
42     System.out.println("La deuxième liste ====>: " + liste2);
43 }

```

```

1 C:\>javac Employees.java
2 La liste des employés : [Joel, Jean, Jeremy]
3 La première liste ====>: [Joel, Jean, Jeremy, Rambo]
4 La deuxième liste ====>: [Joel, Jean]

```

Nous avons ici redéfini la méthode `clone()` pour fournir une copie complète de la liste des employés.

Cet extrait de code est la preuve que nous gagnons du temps car si le clonage n'avait pas été fourni, nous aurions dû appeler la base de données à chaque fois pour récupérer la liste des employés.

Remarque Avantages et Inconvénients

• Avantages :

- Vous pouvez cloner les objets sans les coupler avec leurs classes concrètes.
- Vous clonez des prototypes pré-construits et vous pouvez vous débarrasser du code d'initialisation redondant.
- Vous pouvez créer des objets complexes plus facilement.
- Vous obtenez une alternative à l'héritage pour gérer les modèles de configuration d'objets complexes.

• Inconvénients :

- Cloner des objets complexes dotés de références circulaires peut se révéler très difficile.
- La gestion des objets clonés est difficile pour des applications à grande échelle.
- Difficultés de choix entre la copie superficielle et la copie profonde.

Syntaxe À retenir

Quand utiliser un **Prototype** :

- Lorsque les classes à instancier sont spécifiées au moment de l'exécution.
- Si certains objets sont requis, ils doivent être exactement les mêmes que l'objet existant.
- Lors de la création d'un objet plus grand et complexe à partir de zéro. Il serait plus simple et plus économique de copier l'objet déjà existant.
- Si vous avez besoin d'une application indépendante de la façon dont les ressources sont créées et composées. En d'autres termes si vous ne voulez pas que votre code dépende des classes concrètes des objets que vous clonez.

- Si vous voulez réduire le nombre de sous-classes quand elles ne diffèrent que dans leur manière d'initialiser leurs objets respectifs. Ces sous-classes pourraient avoir été prévues pour créer des objets similaires dans des configurations spécifiques.

Complément Parallélisme avec les autres patrons de conception de création

- Le **Prototype** se révèle utile lorsque vous voulez sauvegarder des copies de Commandes dans l'historique.
- Le **Prototype** n'est pas basé sur l'héritage, il n'a donc pas ses désavantages. Mais le **Prototype** requiert une initialisation compliquée pour l'objet cloné. La **Fabrique** est basée sur l'héritage, mais n'a pas besoin d'une étape d'initialisation.

XII. Exercice : Appliquer la notion

Question

Nous avons une entreprise de smartphones qui produit des milliers de téléphones avec le même matériel et le même logiciel, mais avec un modèle différent pour chaque téléphone (couleur ou prix qui varie selon les modèles, etc.).

```

1 public abstract class Smartphone {
2     private String modele;
3     private int prix;
4     private int prixAdditionnel = 0;
5
6     public String getModele() { return modele; }
7
8     public void setModele(String modele) {
9         this.modele = modele;
10    }
11
12    public int getPrix() {
13        return prix + prixAdditionnel;
14    }
15
16    public void setPrix(int prix) {
17        this.prix = prix;
18    }
19
20    public void setPrixAdditionnel(int prixAdditionnel) {
21        this.prixAdditionnel = prixAdditionnel;
22    }
23
24    @Override
25    public String toString() {
26        return "Smartphone{" +
27            "modele='" + modele + '\'' +
28            ", prix=" + prix +
29            ", prixAdditionnel=" + prixAdditionnel +
30            '}';
31    }
32 }
33
    
```

Quelle approche allez-vous proposer de mettre en œuvre pour implémenter ce programme correctement ?

Indice :

L'approche simple constitue à créer un nouvel objet à chaque fois qu'un nouveau client va passer une commande.

Du fait que nous devrions modifier ces données plusieurs fois, cette approche simple n'est donc pas appropriée car on va à chaque fois créer des nouveaux objets à partir de rien.

Indice :

La bonne approche consiste donc à implémenter un **Prototype** car on va cloner l'objet exigé existant dans un nouvel objet permettant alors de faire la manipulation de données sur l'objet cloné.

XIII. Essentiel**XIV. Auto-évaluation****A. Exercice final****Exercice**

Exercice

Quelle assertion est vraie à propos des patrons de conceptions/design patterns?

- ☐ Un modèle de conception est une solution élégante et réutilisable à un problème courant (code répétitif, codage de tout à zéro, ajout de propriétés supplémentaires ou de fonctions risquant de perturber l'existant...) dans la conception des logiciels.
- ☐ Un modèle de conception est un algorithme élégant et réutilisable à implémenter pour développer des applications comme on le souhaite.
- ☐ Un modèle de conception est un programme/une fonction ou librairie Java destiné à fournir des solutions pour bien développer nos logiciels en Java.
- ☐ Un modèle de conception est le framework Java le plus utilisé juste après Spring.
- ☐ Aucune bonne réponse.

Exercice

Quels sont les différents types de patrons de conception ?

- ☐ Le Patron Singleton
 - Le Patron Fabrique/Fabrique Abstraite
 - Le Patron Monteur
 - Le Patron Prototype
- ☐ Le Patron Adaptateur
 - Le Patron Décorateur
 - Le Patron Composite
 - Le Patron Procuration
 - Le Patron Façade
 - Le Patron Pont
 - Le Patron Poids mouche

- ☐ Le Patron Chaîne de responsabilité
 - La Commande
 - Le Patron Itérateur
 - Le Patron Médiateur
 - Le Patron Memento
 - Le Patron Observateur
 - Le Patron État
 - Le Patron Stratégie
 - Le Patron de Méthode
 - Le Patron Visiteur

- ☐ Les Patrons de Création
 - Les Patrons de Structure
 - Les Patrons de Comportement

- ☐ Aucune bonne réponse.

Exercice

Quelle assertion est vraie à propos du patron Singleton ?

- ☐ Ce patron crée un objet sans exposer la logique de création derrière et fait référence à un objet nouvellement créé à l'aide d'une interface commune.
- ☐ Ce patron implique qu'une interface est responsable de la création d'une Fabrique d'objets associée sans spécifier explicitement leurs classes.
- ☐ Ce patron garantit qu'à chaque instance donnée, un et un seul objet de la classe existe dans la JVM. Fournit aussi un moyen d'accès global à cet objet.
- ☐ Aucune bonne réponse

Exercice

Quelles sont les différentes manières de créer un patron Singleton en Java ?

- ☐ Initiation paresseuse/tardive/patiente
- ☐ Verrouillage réverifié
- ☐ Initiation hâtive/impatiente
- ☐ Énumération Java
- ☐ Aucune bonne réponse

Exercice

Est-il possible de créer un clone d'un objet Singleton ?

- ☐ Oui, si la classe de Singleton implémente une interface `Cloneable`.
- ☐ Non, les classes de Singleton ne peuvent être clonées.

Exercice

Quelles assertions est sont vraies à propos des Méthode de Fabrique ?

- ☐ Ce patron implique une seule classe qui est responsable de créer un objet tout en s'assurant qu'un seul objet est créé.
- ☐ Patron de création qui fournit une spécification pour l'installation d'une classe à l'aide d'une interface ou d'une classe abstraite.
- ☐ Dans ce patron, la méthode de modèle elle-même doit être définitive, de sorte que la sous-classe ne peut pas la remplacer et modifier les étapes, mais si nécessaire, elles peuvent être rendues abstraites afin que la sous-classe puisse les implémenter en fonction du besoin.
- ☐ Avec les Méthodes de Fabrique, la logique réelle de l'instanciation des objets est reportée aux sous classes en fonction du type nécessaire. Le client peut alors créer un objet qu'il souhaite sans connaître la logique réelle.
- ☐ Aucune bonne réponse.

Exercice

Qu'est-ce que le patron de conception Fabrique Abstraite?

- ☐ Il s'agit d'un sous-type de modèle de création qui est utilisé lorsque nous avons besoin d'une couche supplémentaire d'abstraction au-dessus d'une famille d'objets du patron Fabrique.
- ☐ Il s'agit d'un sous-type de modèle de création qui fournit une spécification pour l'instanciation d'une classe à l'aide d'une interface ou d'une classe abstraite.
- ☐ Il s'agit d'un patron utilisé pour ajouter des fonctionnalités supplémentaires à un objet particulier lors de l'exécution.

Exercice

Quelle est la différence entre les Méthodes de Fabrique et la Fabrique Abstraite ?

- ☐ Les Méthodes de Fabrique sont une Fabrique et la Fabrique Abstraite n'en est pas une.
- ☐ Les Méthodes de Fabrique utilisent l'héritage tandis que la Fabrique Abstraite utilise la composition comme mécanisme de création d'objet.
- ☐ Les Méthodes de Fabrique créent un objet unique tandis que la Fabrique Abstraite crée un groupe d'objets.
- ☐ Aucune bonne réponse

Exercice

Quel patron de conception est préféré pour créer un objet complexe ?

- ☐ Singleton, car ce patron offre un accès global à l'objet.
- ☐ Méthode de Fabrique, car le client peut créer l'objet souhaité sans connaître la logique réelle.
- ☐ Fabrique Abstraite, car ce patron crée des Fabriques qui à leur tour créent des objets en reportant davantage la logique de création d'objet à leurs sous-classes.
- ☐ Monteur, car il s'agit d'une extension du patron Fabrique qui est créée pour résoudre des problèmes associés aux Méthodes de Fabrique et Fabrique Abstraite.
- ☐ Aucune bonne réponse

Exercice

Laquelle de ces assertions est vraie à propos du patron Prototype ?

- ☐ Ce patron crée un objet complexe en utilisant des objets simples et en utilisant une approche de création étape par étape.
- ☐ Ce patron fonctionne principalement en créant un clone de l'objet existant au lieu de créer une nouvelle instance à chaque fois afin d'optimiser les performances.
- ☐ Ce patron fonctionne comme un pont entre deux interfaces incompatibles.
- ☐ Ce patron est utilisé lorsque nous devons découpler une abstraction de son implémentation afin que les deux puissent varier indépendamment.

B. Exercice : Défi

Si vous êtes familier avec les Méthodes de Fabriques en Java, vous remarquerez que nous avons une seule classe de Fabrique.

Cette classe de Fabrique renvoie différentes sous-classes en fonction de l'entrée fournie et la classe de Fabrique utilise l'instruction `if/else` ou `switch` comme on a vu avec la notion qu'on a appliquée précédemment.

Avec le patron Fabrique Abstraite, on se débarrasse du bloc `if/else` et on a une classe Fabrique pour chaque sous-classe.

En suite une classe de Fabrique Abstraite renvoie les sous-classes en fonction de la classe de Fabrique d'entrée.

Cela peut sembler déroutant, mais l'implémentation sera facilement compréhensible car il n'y a pas beaucoup de différences entre les patrons Méthodes de Fabrique et Fabrique Abstraite.

Question

Nous allons procéder par l'implémentation du programme des magasins de vêtements Nike, Adidas et Jordan à l'aide du patron Fabrique Abstraite cette fois.

En effet, on a vu comment l'implémenter en Méthodes de Fabrique dans l'exercice sur les Fabriques.

Le défi ici est donc de le ré-implémenter/transformer en Fabrique Abstraite.

Indice :

Tout d'abord on va créer une classe abstraite/interface de Fabrique Abstraite de vetement.

```
1 public abstract static class VetementFabriqueAbstraite {
2     abstract Vetement creerPanierVetement();
3 }
```

Notez que la méthode `creerPanierVetement()` va renvoie une instance de la super classe `Vetement`.

Indice :

Ensuite on va créer des classes de Fabrique qui vont toutes étendre la classe abstraite `VetementFabriqueAbstraite` et renvoyer leurs sous-classes respectives (sous-classes de type `Vetement`).

```
1 static class NikeFabriqueAbstraite extends VetementFabriqueAbstraite {
2     @Override
3     Vetement creerPanierVetement() {
4         Vetement panier = new Nike();
5         panier.getShirt();
6         panier.getSurvet();
7         panier.getBasket();
8         return panier;
9     }
10 }
11
12 static class AdidasFabriqueAbstraite extends VetementFabriqueAbstraite {
13     @Override
14     Vetement creerPanierVetement() {
15         Vetement panier = new Adidas();
16         panier.getShirt();
```



```

17         panier.getSurvet();
18         panier.getBasket();
19         return panier;
20     }
21 }
22
23 static class JordanFabriqueAbstraite extends VetementFabriqueAbstraite {
24     @Override
25     Vetement creerPanierVetement() {
26         Vetement panier = new Jordan();
27         panier.getShirt();
28         panier.getSurvet();
29         panier.getBasket();
30         return panier;
31     }
32 }

```

Indice :

Nous allons finalement créer une classe consommateur qui fournira le point d'entrée aux classes clientes pour créer des sous-classes.

```

1     static class FabriqueVetement {
2         public static Vetement getVetement(VetementFabriqueAbstraite fabrique) {
3             return fabrique.creerPanierVetement();
4         }
5     }

```

On note que la méthode `getVetement()` est une méthode simple qui prend `VetementFabriqueAbstraite` comme argument et renvoie un objet de type `Vetement`.

Solutions des exercices

Exercice p. 7 Solution n°1

Exercice

Quels sont les trois principaux groupes de patrons de conception ?

- ☒ Les patrons de création
- ☐ Les patrons de sécurisation
- ☐ Les patrons d'accès
- ☒ Les patrons de structure
- ☐ Les patrons de conception
- ☒ Les patrons de comportement

Exercice

Quels sont les bénéfices de l'utilisation des patrons de conception ?

- ☒ La flexibilité
- ☐ La sécurité
- ☒ La ré-utilisabilité
- ☒ Le vocabulaire commun
- ☒ Utilisation de bonnes pratiques

Exercice

Quel est le but d'un patron de conception ?

- ☐ Il fournit un morceau de code pour chaque problème rencontré dans le développement informatique
- ☒ Il permet d'apporter une solution à des problèmes récurrents
- ☐ Il fournit des bonnes pratiques de développement
- ☐ Il permet de découper un projet informatique en bloc de code

Exercice

Les patrons de conception ont été inventés pour quel langage ?

- ☐ Java
- ☐ C
- ☐ PHP
- ☐ Python
- ☒ Ils ne dépendent pas d'un seul langage

Exercice p. Solution n°2

```

1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.SQLException;
4
5 /**
6  * la classe est final car un singleton n'est pas censé avoir d'héritier
7  */
8 final class ClasseDeConnexionSingleton {
9
10     /**
11      * L'utilisation du mot clé volatile (Java 5+) permet d'éviter le cas où le
12      * notreSingleton.instance est non null
13      */
14     private static volatile ClasseDeConnexionSingleton instance;
15     private Connection connection;
16
17     /**
18      * La présence d'un constructeur privé supprime le constructeur public par défaut.
19      * De plus, seul le singleton peut s'instancier lui-même.
20      */
21     private ClasseDeConnexionSingleton() throws SQLException {
22         try {
23             Class.forName("org.mysql.Driver");
24             String username = "root";
25             String password = "root";
26             String url = "jdbc:mysql://localhost:3306/jdbc";
27             this.connection = DriverManager.getConnection(url, username, password);
28         } catch (ClassNotFoundException ex) {
29             System.out.println("Something is wrong with the DB connection String : " +
30                 ex.getMessage());
31         }
32     }
33
34     public Connection getConnection() {
35         return connection;
36     }
37
38     /**
39      * Méthode permettant de renvoyer une instance de notre classe Singleton
40      */
41     public static ClasseDeConnexionSingleton getInstance() throws SQLException {
42         if (instance == null) {
43             instance = new ClasseDeConnexionSingleton();
44         } else if (instance.getConnection().isClosed()) {
45             instance = new ClasseDeConnexionSingleton();
46         }
47         return instance;
48     }
49 }

```

Exercice p. Solution n°3

```

1 public abstract class Vetement {
2     public static void main(String[] args) {
3         FabriqueVetement panier = new FabriqueVetement();
4         Vetement nike = panier.creerPanierVetement("Nike");
5         Vetement adidas = panier.creerPanierVetement("Adidas");

```

```
6      Vetement jordan = panier.creerPanierVetement("Jordan");
7  }
8
9  public abstract void getShirt();
10
11 public abstract void getSurvet();
12
13 public abstract void getBasket();
14
15 static class Nike extends Vetement {
16     @Override
17     public void getShirt() {
18         System.out.println("Ajoutez un T-Shirt dans le panier de vêtement Nike!");
19     }
20
21     @Override
22     public void getSurvet() {
23         System.out.println("Ajoutez un Survet dans le panier de vêtement Nike!");
24     }
25
26     @Override
27     public void getBasket() {
28         System.out.println("Ajoutez un Basket dans le panier de vêtement Nike!");
29     }
30 }
31
32 static class Adidas extends Vetement {
33     @Override
34     public void getShirt() {
35         System.out.println("Ajoutez un T-Shirt dans le panier de vêtement Adidas!");
36     }
37
38     @Override
39     public void getSurvet() {
40         System.out.println("Ajoutez un Survet dans le panier de vêtement Adidas!");
41     }
42
43     @Override
44     public void getBasket() {
45         System.out.println("Ajoutez un Basket dans le panier de vêtement Adidas!");
46     }
47 }
48
49 static class Jordan extends Vetement {
50     @Override
51     public void getShirt() {
52         System.out.println("Ajoutez un T-Shirt dans le panier de vêtement Jordan!");
53     }
54
55     @Override
56     public void getSurvet() {
57         System.out.println("Ajoutez un Survet dans le panier de vêtement Jordan!");
58     }
59
60     @Override
61     public void getBasket() {
62         System.out.println("Ajoutez un Basket dans le panier de vêtement Jordan!");
63     }
64 }
```

```

64     }
65
66     static class FabriqueVetement {
67         public Vetement creerPanierVetement(String vetement) {
68             Vetement panier = switch (vetement) {
69                 case "Nike" -> new Nike();
70                 case "Adidas" -> new Adidas();
71                 case "Jordan" -> new Jordan();
72                 default -> throw new IllegalArgumentException("Pas de vetement de ce
modèle");
73             };
74             panier.getShirt();
75             panier.getSurvet();
76             panier.getBasket();
77
78             return panier;
79         }
80     }
81 }

```

```

1 C:\>javac Vetement.java
2 Ajout d'un T-Shirt dans le panier de vêtement Nike
3 Ajout d'un Survet dans le panier de vêtement Nike
4 Ajout d'un Basket dans le panier de vêtement Nike
5 Ajout d'un T-Shirt dans le panier de vêtement Adidas
6 Ajout d'un Survet dans le panier de vêtement Adidas
7 Ajout d'un Basket dans le panier de vêtement Adidas
8 Ajout d'un T-Shirt dans le panier de vêtement Jordan
9 Ajout d'un Survet dans le panier de vêtement Jordan
10 Ajout d'un Basket dans le panier de vêtement Jordan

```

Exercice p. Solution n°4

```

1 public class CompteBancaire {
2     private long numeroDeCompte; // Obligatoire
3     private String proprietaireDuCompte;
4     private String succursale;
5     private double soldeDuCompte;
6     private double tauxDInteret;
7
8     private CompteBancaire() {
9     }
10
11     public static void main(String[] args) {
12         CompteBancaire compte = new CompteBancaire.Monteur(34L)
13             .duProprietaireDuCompte("Joel")
14             .deLaSuccursale("Montpellier")
15             .duSoldeDuCompe(5000)
16             .duTauxDInteret(2.5)
17             .monteur();
18         CompteBancaire autreCompte = new CompteBancaire.Monteur(75L)
19             .duProprietaireDuCompte("Jean")
20             .deLaSuccursale("Paris")
21             .duSoldeDuCompe(1000)
22             .duTauxDInteret(2.2)
23             .monteur();
24         System.out.println(compte.toString());
25         System.out.println(autreCompte.toString());

```

```

26     }
27
28     @Override
29     public String toString() {
30         return "Monteur{" +
31             "numeroDeCompte=" + numeroDeCompte +
32             ", proprietaireDuCompte='" + proprietaireDuCompte + '\'' +
33             ", succursale='" + succursale + '\'' +
34             ", soldeDuCompte=" + soldeDuCompte + " euros" +
35             ", tauxDInteret=" + tauxDInteret + " %" +
36             '}';
37     }
38
39     private static class Monteur {
40         private final long numeroDeCompte; // Obligatoire
41         private String proprietaireDuCompte;
42         private String succursale;
43         private double soldeDuCompte;
44         private double tauxDInteret;
45
46         public Monteur(long numeroDeCompte) {
47             this.numeroDeCompte = numeroDeCompte;
48         }
49
50         public Monteur duProprietaireDuCompte(String proprietaireDuCompte) {
51             this.proprietaireDuCompte = proprietaireDuCompte;
52             return this; // En renvoyant le constructeur à chaque fois, nous pouvons créer une
53             interface fluide.
54         }
55
56         public Monteur deLaSuccursale(String succursale) {
57             this.succursale = succursale;
58             return this;
59         }
60
61         public Monteur duSoldeDuCompe(double soldeDuCompte) {
62             this.soldeDuCompte = soldeDuCompte;
63             return this;
64         }
65
66         public Monteur duTauxDInteret(double tauxDInteret) {
67             this.tauxDInteret = tauxDInteret;
68             return this;
69         }
70
71         public CompteBancaire monteur() {
72             CompteBancaire compte = new CompteBancaire();
73             compte.numeroDeCompte = this.numeroDeCompte;
74             compte.proprietaireDuCompte = this.proprietaireDuCompte;
75             compte.succursale = this.succursale;
76             compte.soldeDuCompte = this.soldeDuCompte;
77             compte.tauxDInteret = this.tauxDInteret;
78             return compte;
79         }
80     }

```

```
1 C:\>javac CompteBancaire.java
```

```

2 Monteur{numeroDeCompte=34, proprietaireDuCompte='Joel', succursale='Montpellier',
soldeDuCompte=5000.0 euros, tauxDInteret=2.5 %}
3 Monteur{numeroDeCompte=75, proprietaireDuCompte='Jean', succursale='Paris',
soldeDuCompte=1000.0 euros, tauxDInteret=2.2 %}

```

Cet extrait de code résout bien notre problème car il est plus détaillé, plus clair et donc il n'entraînera pas d'erreur accidentelle lors du passage de paramètre à la création d'objets.

Exercice p. Solution n°5

```

1 public class Main {
2     public static void main(String[] args) throws CloneNotSupportedException {
3
4         SmartPhone note20 = new Samsung("Note20");
5         SmartPhone iphone12 = new Apple("iPhone12");
6         SmartPhone huaweiP40 = new Huawei("HuaweiP40");
7         SmartPhone nokia8 = new Nokia("Nokia8");
8
9         System.out.println(note20);
10        System.out.println(iphone12);
11        System.out.println(huaweiP40);
12        System.out.println(nokia8);
13
14        System.out.println("====> Version Pro <====");
15
16        SmartPhone iphone12Pro = iphone12.clone();
17        iphone12Pro.setPrixAdditionnel(250);
18        System.out.println(iphone12Pro);
19
20        SmartPhone huaweiP40Pro = huaweiP40.clone();
21        huaweiP40Pro.setPrixAdditionnel(100);
22        System.out.println(huaweiP40Pro);
23    }
24
25    public abstract static class SmartPhone implements Cloneable {
26        private String modele;
27        private int prix;
28        private int prixAdditionnel = 0;
29
30        public String getModele() {
31            return modele;
32        }
33
34        public void setModele(String modele) {
35            this.modele = modele;
36        }
37
38        public int getPrix() {
39            return prix + this.prixAdditionnel;
40        }
41
42        public void setPrix(int prix) {
43            this.prix = prix;
44        }
45
46        public void setPrixAdditionnel(int prixAdditionnel) {
47            this.prixAdditionnel = prixAdditionnel;
48        }

```



```

49
50     public SmartPhone clone() throws CloneNotSupportedException {
51         return (SmartPhone) super.clone();
52     }
53
54     @Override
55     public String toString() {
56         return "Smartphone{" +
57             "modele='" + getModele() + '\'' +
58             ", prix=" + getPrix() + " euros " +
59             ", prixAdditionnel=" + prixAdditionnel + " euros " +
60             '}';
61     }
62 }
63
64 public static class Samsung extends SmartPhone {
65     public Samsung(String model) {
66         this.setPrix(850);
67         this.setModele(model);
68     }
69
70     @Override
71     public SmartPhone clone() throws CloneNotSupportedException {
72         return super.clone();
73     }
74 }
75
76 public static class Apple extends SmartPhone {
77     public Apple(String model) {
78         this.setPrix(900);
79         this.setModele(model);
80     }
81
82     @Override
83     public SmartPhone clone() throws CloneNotSupportedException {
84         return super.clone();
85     }
86 }
87
88 static class Huawei extends SmartPhone {
89     public Huawei(String model) {
90         this.setPrix(700);
91         this.setModele(model);
92     }
93
94     @Override
95     public SmartPhone clone() throws CloneNotSupportedException {
96         return super.clone();
97     }
98 }
99
100 static class Nokia extends SmartPhone {
101     public Nokia(String model) {
102         this.setPrix(500);
103         this.setModele(model);
104     }
105
106     @Override

```

```

107         public SmartPhone clone() throws CloneNotSupportedException {
108             return super.clone();
109         }
110     }
111 }

1 C:\>javac Main.java
2 Smartphone{modele='Note20', prix=850 euros , prixAdditionnel=0 euros }
3 Smartphone{modele='IPhone12', prix=900 euros , prixAdditionnel=0 euros }
4 Smartphone{modele='HuaweiP40', prix=700 euros , prixAdditionnel=0 euros }
5 Smartphone{modele='Nokia8', prix=500 euros , prixAdditionnel=0 euros }
6 ==> Version Pro <==
7 Smartphone{modele='IPhone12', prix=1150 euros , prixAdditionnel=250 euros }
8 Smartphone{modele='HuaweiP40', prix=800 euros , prixAdditionnel=100 euros }

```

Exercice p. 29 Solution n°6

Exercice

Quelle assertion est vraie à propos des patrons de conceptions/design patterns?

- ☒ Un modèle de conception est une solution élégante et réutilisable à un problème courant (code répétitif, codage de tout à zéro, ajout de propriétés supplémentaires ou de fonctions risquant de perturber l'existant...) dans la conception des logiciels.
- ☐ Un modèle de conception est un algorithme élégant et réutilisable à implémenter pour développer des applications comme on le souhaite.
- ☐ Un modèle de conception est un programme/une fonction ou librairie Java destiné à fournir des solutions pour bien développer nos logiciels en Java.
- ☐ Un modèle de conception est le framework Java le plus utilisé juste après Spring.
- ☐ Aucune bonne réponse.



L'assertion 2 est fausse car un algorithme définit toujours clairement un ensemble d'actions qui va vous mener vers un objectif, alors qu'un patron, c'est la description d'une solution à un plus haut niveau.

L'assertion 3 est fausse car vous ne pouvez pas vous contenter de trouver un patron et de le recopier dans votre programme comme vous le feriez avec des fonctions ou des librairies prêtes à l'emploi. Un modèle de conception, ce n'est pas un bout de code spécifique, mais plutôt un concept général pour résoudre un problème précis.

Les assertions 4 et 5 sont fausses.

Exercice

Quels sont les différents types de patrons de conception ?

- ☐ Le Patron Singleton
 - Le Patron Fabrique/Fabrique Abstraite
 - Le Patron Monteur
 - Le Patron Prototype
- ☐ Le Patron Adaptateur
 - Le Patron Décorateur
 - Le Patron Composite
 - Le Patron Procuration
 - Le Patron Façade

Le Patron Pont

Le Patron Poids mouche

☐ Le Patron Chaîne de responsabilité

La Commande

Le Patron Itérateur

Le Patron Médiateur

Le Patron Memento

Le Patron Observateur

Le Patron État

Le Patron Stratégie

Le Patron de Méthode

Le Patron Visiteur

☒ Les Patrons de Création

Les Patrons de Structure

Les Patrons de Comportement

☐ Aucune bonne réponse.



L'assertion 4 est vraie car on distingue 3 types de patrons de conception :

- *Les patrons de création* décrivent des mécanismes pour créer des objets.
- *Les patrons de structure* décrivent comment assembler des objets et des classes dans des structures plus grandes.
- *Les patrons de comportement* décrivent comment mettre en place des formes de collaboration entre objets.

Les assertions 1, 2 et 3 désignent respectivement des patrons de conception, des patrons de création, de structure et de comportement.

Exercice

Quelle assertion est vraie à propos du patron Singleton ?

- ☐ Ce patron crée un objet sans exposer la logique de création derrière et fait référence à un objet nouvellement créé à l'aide d'une interface commune.
- ☐ Ce patron implique qu'une interface est responsable de la création d'une Fabrique d'objets associée sans spécifier explicitement leurs classes.
- ☒ Ce patron garantit qu'à chaque instance donnée, un et un seul objet de la classe existe dans la JVM. Fournit aussi un moyen d'accès global à cet objet.
- ☐ Aucune bonne réponse



Les assertions 1 et 2 font référence aux autres patrons de création et pas du tout au patron Singleton.

Exercice

Quelles sont les différentes manières de créer un patron Singleton en Java ?

- ☒ Initiation paresseuse/tardive/patiente
- ☐ Verrouillage réverifié
- ☒ Initiation hâtive/impatiente

☐ Énumération Java

☐ Aucune bonne réponse



Il existe deux moyens populaires pour implémenter le patron Singleton :

- Initiation hâtive/impatiente : une instance d'une classe est créée bien avant qu'elle ne soit réellement requise.
- Initiation paresseuse/tardive/patiente : initialisation qui limite la création de l'instance jusqu'à ce qu'elle soit demandée pour la première fois.

Le Verrouillage réverifié et l'Enum Java sont des moyens de création d'une classe Singleton Thread-Safe.

Exercice

Est-il possible de créer un clone d'un objet Singleton ?

☒ Oui, si la classe de Singleton implémente une interface `Cloneable`.

☐ Non, les classes de Singleton ne peuvent être clonées.

Exercice

Quelles assertions est sont vraies à propos des Méthode de Fabrique ?

☐ Ce patron implique une seule classe qui est responsable de créer un objet tout en s'assurant qu'un seul objet est créé.

☒ Patron de création qui fournit une spécification pour l'installation d'une classe à l'aide d'une interface ou d'une classe abstraite.

☐ Dans ce patron, la méthode de modèle elle-même doit être définitive, de sorte que la sous-classe ne peut pas la remplacer et modifier les étapes, mais si nécessaire, elles peuvent être rendues abstraites afin que la sous-classe puisse les implémenter en fonction du besoin.

☒ Avec les Méthodes de Fabrique, la logique réelle de l'instanciation des objets est reportée aux sous classes en fonction du type nécessaire. Le client peut alors créer un objet qu'il souhaite sans connaître la logique réelle.

☐ Aucune bonne réponse.



L'assertion 1 parle du patron Singleton et l'assertion 3 du patron Méthode de modèle/Template Method du type de patron de comportement.

Exercice

Qu'est-ce que le patron de conception Fabrique Abstraite?

☒ Il s'agit d'un sous-type de modèle de création qui est utilisé lorsque nous avons besoin d'une couche supplémentaire d'abstraction au-dessus d'une famille d'objets du patron Fabrique.

☐ Il s'agit d'un sous-type de modèle de création qui fournit une spécification pour l'instanciation d'une classe à l'aide d'une interface ou d'une classe abstraite.


☐ Il s'agit d'un patron utilisé pour ajouter des fonctionnalités supplémentaires à un objet particulier lors de l'exécution.



L'assertion 2 parle de méthode Fabrique, l'assertion 3 du patron Décorateur de type de patron de structure.

Exercice


Quelle est la différence entre les Méthodes de Fabrique et la Fabrique Abstraite ?

- ☐ Les Méthodes de Fabrique sont une Fabrique et la Fabrique Abstraite n'en est pas une.
 - ☒ Les Méthodes de Fabrique utilisent l'héritage tandis que la Fabrique Abstraite utilise la composition comme mécanisme de création d'objet.
 - ☒ Les Méthodes de Fabrique créent un objet unique tandis que la Fabrique Abstraite crée un groupe d'objets.
 - ☐ Aucune bonne réponse
-  L'assertion 1 est fausse car la Fabrique Abstraite est une Fabrique de Fabriques, donc une sorte d'ensemble de Fabriques.

Exercice

Quel patron de conception est préféré pour créer un objet complexe ?

- ☐ Singleton, car ce patron offre un accès global à l'objet.
- ☐ Méthode de Fabrique, car le client peut créer l'objet souhaité sans connaître la logique réelle.
- ☐ Fabrique Abstraite, car ce patron crée des Fabriques qui à leur tour créent des objets en reportant davantage la logique de création d'objet à leurs sous-classes.
- ☒ Monteur, car il s'agit d'une extension du patron Fabrique qui est créée pour résoudre des problèmes associés aux Méthodes de Fabrique et Fabrique Abstraite.
- ☐ Aucune bonne réponse

 Le **Monteur** se concentre sur la construction d'objets complexes étape par étape. Le **Fabrique Abstraite** se spécialise dans la création de familles d'objets associés. Le **Fabrique Abstraite** retourne le produit immédiatement, alors que le **Monteur** vous permet de lancer des étapes supplémentaires avant de récupérer le produit.

Le singleton quant à lui n'est pas du tout approprié pour la création des objets complexes.

Exercice

Laquelle de ces assertions est vraie à propos du patron Prototype ?

- ☐ Ce patron crée un objet complexe en utilisant des objets simples et en utilisant une approche de création étape par étape.
- ☒ Ce patron fonctionne principalement en créant un clone de l'objet existant au lieu de créer une nouvelle instance à chaque fois afin d'optimiser les performances.
- ☐ Ce patron fonctionne comme un pont entre deux interfaces incompatibles.
- ☐ Ce patron est utilisé lorsque nous devons découpler une abstraction de son implémentation afin que les deux puissent varier indépendamment.

Exercice p. Solution n°7

```

1 public class Demo {
2
3     public static void main(String[] args) {
4         Vetement nike = FabriqueVetement.getVetement(new NikeFabriqueAbstraite());
5         Vetement adidas = FabriqueVetement.getVetement(new AdidasFabriqueAbstraite());
6         Vetement Jordan = FabriqueVetement.getVetement(new JordanFabriqueAbstraite());
7     }
8
9     public abstract static class VetementFabriqueAbstraite {
10         abstract Vetement creerPanierVetement();

```

```

11     }
12
13     public abstract static class Vetement {
14
15         public abstract void getShirt();
16
17         public abstract void getSurvet();
18
19         public abstract void getBasket();
20     }
21
22     static class FabriqueVetement {
23         public static Vetement getVetement(VetementFabriqueAbstraite fabrique) {
24             return fabrique.creerPanierVetement();
25         }
26     }
27
28     static class NikeFabriqueAbstraite extends VetementFabriqueAbstraite {
29         @Override
30         Vetement creerPanierVetement() {
31             Vetement panier = new Nike();
32             panier.getShirt();
33             panier.getSurvet();
34             panier.getBasket();
35             return panier;
36         }
37     }
38
39     static class AdidasFabriqueAbstraite extends VetementFabriqueAbstraite {
40         @Override
41         Vetement creerPanierVetement() {
42             Vetement panier = new Adidas();
43             panier.getShirt();
44             panier.getSurvet();
45             panier.getBasket();
46             return panier;
47         }
48     }
49
50     static class JordanFabriqueAbstraite extends VetementFabriqueAbstraite {
51         @Override
52         Vetement creerPanierVetement() {
53             Vetement panier = new Jordan();
54             panier.getShirt();
55             panier.getSurvet();
56             panier.getBasket();
57             return panier;
58         }
59     }
60
61     static class Nike extends Vetement {
62         @Override
63         public void getShirt() {
64             System.out.println("Ajoutez un T-Shirt dans le panier de vêtement Nike!");
65         }
66
67         @Override
68         public void getSurvet() {

```

```

69         System.out.println("Ajoutez un Survet dans le panier de vêtement Nike!");
70     }
71
72     @Override
73     public void getBasket() {
74         System.out.println("Ajoutez un Basket dans le panier de vêtement Nike!");
75     }
76 }
77
78 static class Adidas extends Vetement {
79     @Override
80     public void getShirt() {
81         System.out.println("Ajoutez un T-Shirt dans le panier de vêtement Adidas!");
82     }
83
84     @Override
85     public void getSurvet() {
86         System.out.println("Ajoutez un Survet dans le panier de vêtement Adidas!");
87     }
88
89     @Override
90     public void getBasket() {
91         System.out.println("Ajoutez un Basket dans le panier de vêtement Adidas!");
92     }
93 }
94
95 static class Jordan extends Vetement {
96     @Override
97     public void getShirt() {
98         System.out.println("Ajoutez un T-Shirt dans le panier de vêtement Jordan!");
99     }
100
101     @Override
102     public void getSurvet() {
103         System.out.println("Ajoutez un Survet dans le panier de vêtement Jordan!");
104     }
105
106     @Override
107     public void getBasket() {
108         System.out.println("Ajoutez un Basket dans le panier de vêtement Jordan!");
109     }
110 }
111 }
112

```

```

1 C:\>javac Demo.java
2 Ajoutez un T-Shirt dans le panier de vêtement Nike!
3 Ajoutez un Survet dans le panier de vêtement Nike!
4 Ajoutez un Basket dans le panier de vêtement Nike!
5 Ajoutez un T-Shirt dans le panier de vêtement Adidas!
6 Ajoutez un Survet dans le panier de vêtement Adidas!
7 Ajoutez un Basket dans le panier de vêtement Adidas!
8 Ajoutez un T-Shirt dans le panier de vêtement Jordan!
9 Ajoutez un Survet dans le panier de vêtement Jordan!
10 Ajoutez un Basket dans le panier de vêtement Jordan!

```