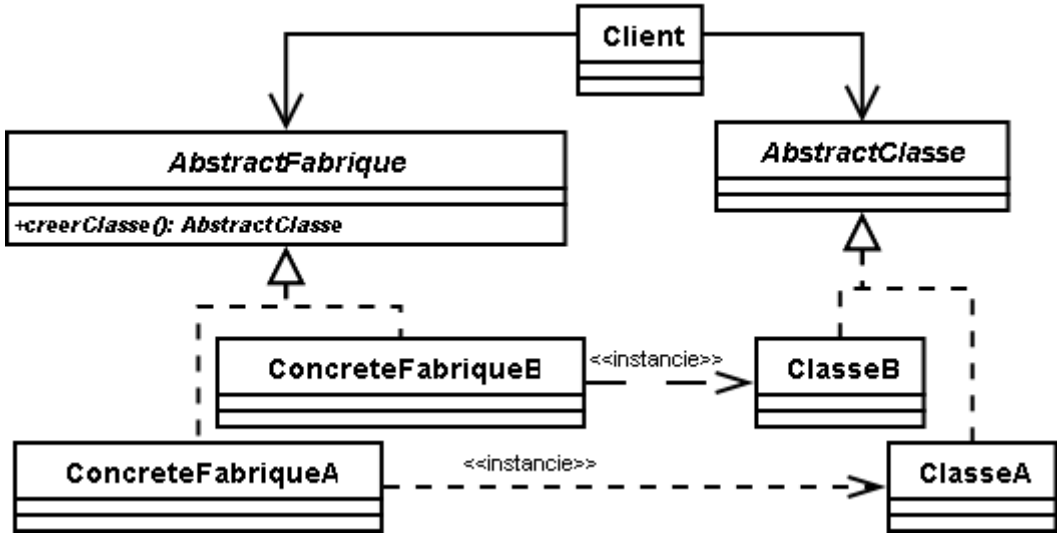
Design Patterns du Gang of Four appliqués à Java

Table des matières

IV. CREATIONNELS (CREATIONAL PATTERNS) ▲

IV-A. Fabrique abstraite (Abstract Factory ou Kit) ▲



LIEN VERS LE DICTIONNAIRE DES DEVELOPPEURS :

Abstract Factory

AUTRE RESSOURCE SUR DEVELOPPEZ.COM :

Le kit par Sébastien MERIC

OBJECTIFS :

Fournir une interface pour créer des objets d'une même famille sans préciser leurs classes concrètes.

RAISONS DE L'UTILISER :

Le système utilise des objets qui sont regroupés en famille. Selon certains critères, le système utilise les objets d'une famille ou d'une autre. Le système doit utiliser ensemble les objets d'une famille.

Cela peut être le cas des éléments graphiques d'un look and feel : pour un look and feel donné, tous les graphiques créés doivent être de la même famille.

La partie cliente manipulera les interfaces des objets ; ainsi il y aura une indépendance par rapport aux classes concrètes. Chaque fabrique concrète permet d'instancier une famille d'objets (éléments graphiques du même look and feel) ; ainsi la notion de famille d'objets est renforcée.

RESULTAT :

Le Design Pattern permet d'isoler l'appartenance à une famille de classes.

RESPONSABILITES :

- **AbstractFabrique** : définit l'interface des méthodes de création. Dans le diagramme, il n'y a qu'une méthode de création pour un objet d'une classe. Mais, le diagramme sous-entend un nombre indéfini de méthodes pour un nombre indéfini de classes.

- **ConcreteFabriqueA** et **ConcreteFabriqueB** : implèmentent l'interface et instancient la classe concrète appropriée.
- **AbstractClasse** : définit l'interface d'un type d'objet instancié.
- **ClasseA** et **ClasseB** : sont des sous-classes concrètes d'**AbstractClasse**. Elles sont instanciées par les ConcreteFabrique.
- La partie cliente fait appel à une Fabrique pour obtenir une nouvelle instance d'**AbstractClasse**. L'instanciation est transparente pour la partie cliente. Elle manipule une **AbstractClasse**.

IMPLEMENTATION JAVA :

AbstractClasse.java
Sélectionnez

```
/**
 * Définit l'interface d'un type d'objet instancié.
 */
public interface AbstractClasse {

    /**
     * Méthode d'affichage du nom de la classe.
     */
    public void afficherClasse();
}
```

ClasseA.java
Sélectionnez

```
/**
 * Sous classe de AbstractClasse
 * Cette classe est instanciée par ConcreteFabriqueA
 */
public class ClasseA implements AbstractClasse {

    /**
     * Implémentation de la méthode d'affichage
     */
    public void afficherClasse() {
        System.out.println("Objet de classe 'ClasseA'");
    }
}
```

ClasseB.java
Sélectionnez

```
/**
 * Sous classe de AbstractClasse
 * Cette classe est instanciée par ConcreteFabriqueB
 */
public class ClasseB implements AbstractClasse {

    /**
     * Implémentation de la méthode d'affichage
     */
```

```
        public void afficherClasse() {
            System.out.println("Objet de classe 'ClasseB'");
        }
    }
```

AbstractFabrique.java
Sélectionnez

```
/**
 * Définit l'interface de la méthode de création.
 */
public interface AbstractFabrique {

    /**
     * Méthode de création d'un objet de classe AbstractClasse.
     * @return L'objet créé.
     */
    public AbstractClasse creerClasse();
}
```

ConcreteFabriqueA.java
Sélectionnez

```
/**
 * Implémente l'interface "AbstractFabrique".
 */
public class ConcreteFabriqueA implements AbstractFabrique {

    /**
     * La méthode de création instancie un objet "ClasseA".
     * @return Un objet "ClasseA" qui vient d'être créé.
     */
    public AbstractClasse creerClasse() {
        return new ClasseA();
    }
}
```

ConcreteFabriqueB.java
Sélectionnez

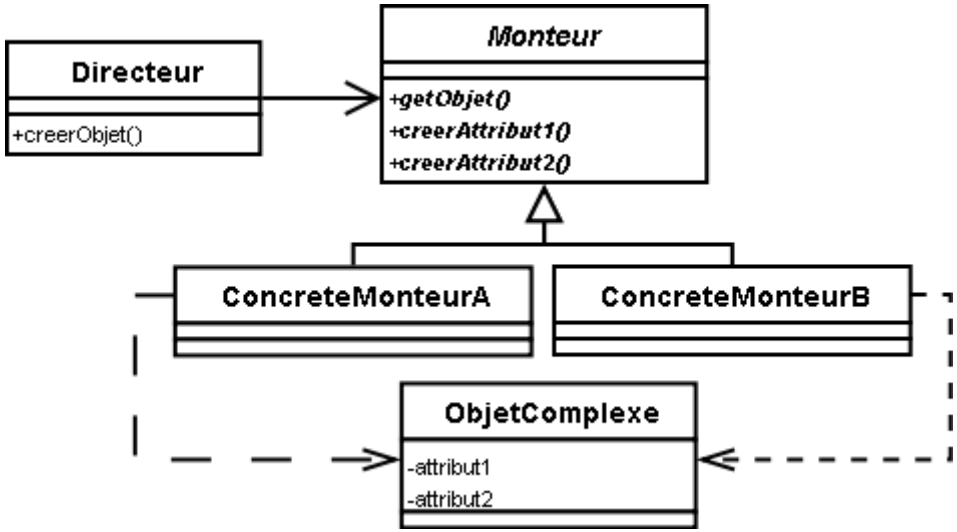
```
/**
 * Implémente l'interface "AbstractFabrique".
 */
public class ConcreteFabriqueB implements AbstractFabrique {

    /**
     * La méthode de création instancie un objet "ClasseB".
     * @return Un objet "ClasseB" qui vient d'être créé.
     */
    public AbstractClasse creerClasse() {
        return new ClasseB();
```

```
        }
    }
```

AbstractFactoryPatternMain.java
Sélectionnez

```
public class AbstractFactoryPatternMain {

    public static void main(String[] args) {
        // Création des fabriques
        AbstractFabrique lFactory1 = new ConcreteFabriqueA();
        AbstractFabrique lFactory2 = new ConcreteFabriqueB();

        // Création de deux "AbstractClasse" à partir de chaque fabrique
        AbstractClasse lClasse1 = lFactory1.creerClasse();
        AbstractClasse lClasse2 = lFactory2.creerClasse();

        // Appel d'une méthode d'"AbstractClasse" qui affiche un message
        // Ce message permet de vérifier que chaque "AbstractClasse"
        // est en fait une classe différente
        lClasse1.afficherClasse();
        lClasse2.afficherClasse();

        // -------------------------
        // Affichage :
        // Objet de classe 'ClasseA'
        // Objet de classe 'ClasseB'
    }
}
```

IV-B. Monteur (Builder) ▲



LIEN VERS LE DICTIONNAIRE DES DEVELOPPEURS :

Builder

AUTRE RESSOURCE SUR DEVELOPPEZ.COM :

Le monteur par Sébastien MERIC

OBJECTIFS :

- Séparer la construction d'un objet complexe de sa représentation.
- Permettre d'obtenir des représentations différentes avec le même procédé de construction.

RAISONS DE L'UTILISER :

Le système doit instancier des objets complexes. Ces objets complexes peuvent avoir des représentations différentes.

Cela peut être le cas des différentes fenêtres d'une IHM. Elles comportent des éléments similaires (titre, boutons), mais chacune avec des particularités (libellés, comportements).

Afin d'obtenir des représentations différentes (fenêtres), la partie cliente passe des monteurs différents au directeur. Le directeur appelera des méthodes du monteur retournant les éléments (titre, bouton). Chaque implémentation des méthodes des monteurs retourne des éléments avec des différences (libellés, comportements).

RESULTAT :

Le Design Pattern permet d'isoler des variations de représentations d'un objet.

RESPONSABILITES :

- **ObjetComplexe** : est la classe d'objet complexe à instancier.

- **Monteur** : définit l'interface des méthodes permettant de créer les différentes parties de l'objet complexe.
- **ConcreteMonteurA** et **ConcreteMonteurB** : implémentent les méthodes permettant de créer les différentes parties. Les classes conservent l'objet durant sa construction et fournissent un moyen de récupérer le résultat de la construction.
- **Directeur** : construit un objet en appelant les méthodes d'un **Monteur**.
- La partie cliente instancie un **Monteur**. Elle le fournit au **Directeur**. Elle appelle la méthode de contruction du **Directeur**.

IMPLEMENTATION JAVA :

ObjetComplexe.java
Sélectionnez

```
/**
 * L'objet complexe
 * Dans l'exemple, on pourrait considérer
 * l'attribut1 comme un libellé
 * et l'attribut2 comme une dimension
 * La classe de l'attribut2 peut varier selon le "Monteur"
 */
public class ObjetComplexe {

    // Les attributs de l'objet complexe
    private String attribut1;
    private Number attribut2;

    // Les méthodes permettant de fixes les attributs
    public void setAttribut1(String pAttribut1) {
        attribut1 = pAttribut1;
    }

    public void setAttribut2(Number pAttribut2) {
        attribut2 = pAttribut2;
    }

    /**
     * Méthode permettant d'afficher l'état de l'objet
     * afin de permettre de mettre en évidence
     * les différences de "montage".
     */
    public void afficher() {
        System.out.println("Objet Complexe : ");
        System.out.println("\t- attribut1 : " + attribut1);
        System.out.println("\t- attribut2 : " + attribut2);
        System.out.println("\t- classe de l'attribut2 : " + attribut2.getClass());
    }
}
```

Monteur.java
Sélectionnez

```
/**
 * Définit l'interface des méthodes permettant
 * de créer les différentes partie
 * de l'objet complexe.
 */
public abstract class Monteur {

    protected ObjetComplexe produit;

    /**
     * Crée un nouveau produit
     * Aucune des parties n'est créée
     * à ce moment là.
     */
    public void creerObjet() {
        produit = new ObjetComplexe();
    }

    /**
     * Retourne l'objet une fois fini.
     */
    public ObjetComplexe getObjet() {
        return produit;
    }

    // Les méthodes de création des parties

    public abstract void creerAttribut1(String pAttribut1);
    public abstract void creerAttribut2(double pAttribut2);
}
```

ConcreteMonteurA.java
Sélectionnez

```
/**
 * Implémente les méthodes permettant
 * de créer les parties de l'objet complexe.
 */
public class ConcreteMonteurA extends Monteur {

    /**
     * Méthode de création de l'attribut attribut1
     * Précise que l'attibut2 représente une dimention en centimètres
     */
    public void creerAttribut1(String pAttribut1) {
        produit.setAttribut1(pAttribut1 + " (avec dimension en centimètre)");
    }

    /**
     * Méthode de création de l'attribut attribut2
     * Stocke la valeur dans un Float sans modification
```

```
         */
        public void creerAttribut2(double pAttribut2) {
            produit.setAttribut2(new Float(pAttribut2));
        }
    }
```

ConcreteMonteurB.java
Sélectionnez

```
/**
 * Implémente les méthodes permettant
 * de créer les parties de l'objet complexe.
 */
public class ConcreteMonteurB extends Monteur {

    /**
     * Méthode de création de l'attribut attribut1
     * Précise que l'attibut2 représente une dimention en pouces
     */
    public void creerAttribut1(String pAttribut1) {
        produit.setAttribut1(pAttribut1 + " (avec dimension en pouces)");
    }

    /**
     * Méthode de création de l'attribut attribut2
     * Stocke la valeur dans un Doucle en le convertissant en pouces
     */
    public void creerAttribut2(double pAttribut2) {
        produit.setAttribut2(new Double(pAttribut2 * 2.54));
    }
}
```

Directeur.java
Sélectionnez

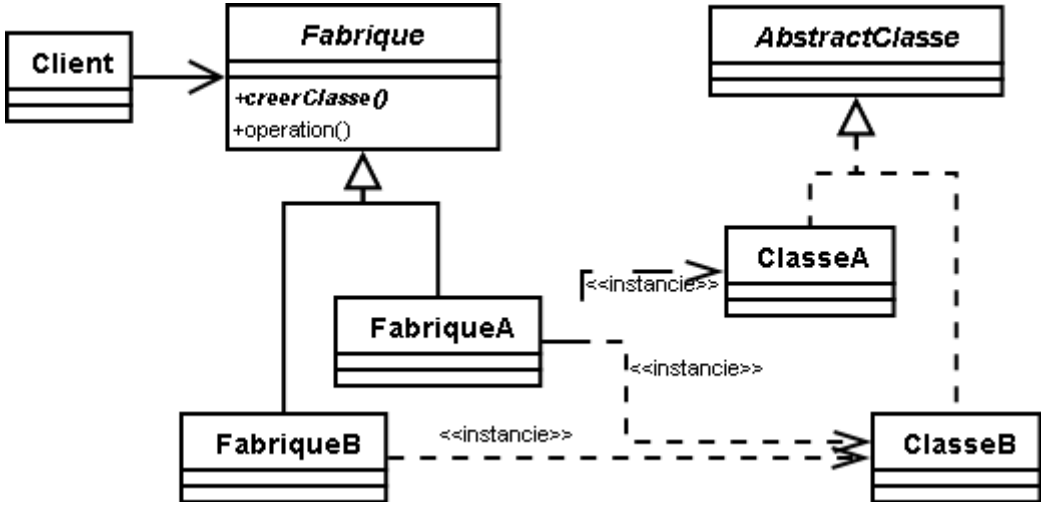
```
/**
 * Contruit un objet en appelant les méthodes d'un "Monteur".
 */
public class Directeur {
    private Monteur monteur;

    Directeur(Monteur pMonteur) {
        monteur = pMonteur;
    }

    /**
     * Crée un objet.
     * Appelle les méthodes de création
     * des parties du "Monteur".
     */
    public ObjetComplexe creerObjet() {
```

```
            monteur.creerObjet();

            monteur.creerAttribut1("libelle de l'objet");
            monteur.creerAttribut2(12);

            return monteur.getObjet();
        }
    }
```

BuilderPatternMain.java
Sélectionnez

```
public class BuilderPatternMain {

    public static void main(String[] args) {
        // Instancie les objets directeur et monteur
        Monteur lMonteurA = new ConcreteMonteurA();
        Directeur lDirecteurA = new Directeur(lMonteurA);
        Monteur lMonteurB = new ConcreteMonteurB();
        Directeur lDirecteurB = new Directeur(lMonteurB);

        // Appel des différentes méthodes de création
        ObjetComplexe lProduitA = lDirecteurA.creerObjet();
        ObjetComplexe lProduitB = lDirecteurB.creerObjet();

        // Demande l'affichage des valeurs des objets
        // pour visualiser les différences de composition
        lProduitA.afficher();
        lProduitB.afficher();

        // Affichage :
        // Objet Complexe :
        //         - attribut1 : libelle de l'objet (avec dimension en centimètre)
        //         - attribut2 : 12.0
        //         - classe de l'attribut2 : class java.lang.Float
        // Objet Complexe :
        //         - attribut1 : libelle de l'objet (avec dimension en pouces)
        //         - attribut2 : 30.48
        //         - classe de l'attribut2 : class java.lang.Double
    }
}
```

IV-C. Fabrique (Factory Method ou Virtual Constructor) ▲



OBJECTIFS :

- Définir une interface pour la création d'un objet, mais laisser les sous-classes décider quelle classe instancier.
- Déléguer l'instanciation aux sous-classes.

RAISONS DE L'UTILISER :

Dans le fonctionnement d'une classe, il est nécessaire de créer une instance. Mais, au niveau de cette classe, on ne connaît pas la classe exacte à instancier.

Cela peut être le cas d'une classe réalisant une sauvegarde dans un flux sortant, mais ne sachant pas s'il s'agit d'un fichier ou d'une sortie sur le réseau.

La classe possède une méthode qui retourne une instance (interface commune au fichier ou à la sortie sur le réseau). Les autres méthodes de la classe peuvent effectuer les opérations souhaitées sur l'instance (écriture, fermeture). Les sous-classes déterminent la classe de l'instance créée (fichier, sortie sur le réseau). Une variante du Pattern existe : la méthode de création choisit la classe de l'instance à créer en fonction de paramètres en entrée de la méthode ou selon des variables de contexte.

RESULTAT :

Le Design Pattern permet d'isoler l'instanciation d'une classe concrète.

RESPONSABILITES :

- **AbstractClasse** : définit l'interface de l'objet instancié.
- **ClasseA** et **ClasseB** : sont des sous-classes concrètes d'**AbstractClasse**. Elles sont instanciées par les classes **Fabrique**.
- **Fabrique** : déclare une méthode de création (**creerClasse**). C'est cette méthode qui a la responsabilité de l'instanciation d'un objet **AbstractClasse**. Si d'autres méthodes de la classe ont besoin d'une instance de **AbstractClasse**, elles font appel à la méthode de création. Dans l'exemple, la méthode **operation()** utilise une instance de **AbstractClasse** et fait donc appel à la méthode **creerClasse**. La méthode de création peut être paramétrée ou non. Dans l'exemple, elle est paramétrée, mais le paramètre n'est significative que pour la **FabriqueA**.
- **FabriqueA** et **FabriqueB** : substituent la méthode de création. Elles implémentent une version différente de la méthode de création.
- La partie cliente utilise une sous-classe de **Fabrique**.

IMPLEMENTATION JAVA :

AbstractClasse.java
Sélectionnez

```
/**
 * Définit l'interface de l'objet instancié.
 */
public interface AbstractClasse {

    /**
     * Méthode permettant d'afficher le nom de la classe.
     * Cela permet de mettre en evidence la classe créée.
     */
    public void afficherClasse();
}
```

ClasseA.java
Sélectionnez

```
/**
 * Sous-class de AbstractClasse.
 */
public class ClasseA implements AbstractClasse {

    /**
     * Implémentation de la méthode d'affichage.
     * Indique qu'il s'agit d'un objet de classe ClasseA
     */
    public void afficherClasse() {
        System.out.println("Objet de classe 'ClasseA'");
    }
}
```

ClasseB.java
Sélectionnez

```
/**
 * Sous-class de AbstractClasse.
 */
public class ClasseB implements AbstractClasse {

    /**
     * Implémentation de la méthode d'affichage.
     * Indique qu'il s'agit d'un objet de classe ClasseB
     */
    public void afficherClasse() {
        System.out.println("Objet de classe 'ClasseB'");
    }
}
```

Fabrique.java
Sélectionnez

```
/**
 * Déclare la méthode de création.
 */
public abstract class Fabrique {

    private boolean pIsClasseA = false;

    /**
     * Méthode de création
     */
    public abstract AbstractClasse creerClasse(boolean pIsClasseA);

    /**
     * Méthode appelant la méthode de création.
     * Puis, effectuant une opération.
     */
    public void operation() {
        // Change la valeur afin de varier le paramètre
        // de la méthode de création
        pIsClasseA = !pIsClasseA;

        // Récupère une instance de classe "AbstractClasse"
        AbstractClasse lClasse = creerClasse(pIsClasseA);

        // Appel la méthode d'affichage de la classe
        // afin de savoir la classe concrète
        lClasse.afficherClasse();
    }
}
```

FabriqueA.java
Sélectionnez

```
/**
 * Substitue la méthode "creerClasse".
 * Instancie un objet "ClasseA".
 */
public class FabriqueA extends Fabrique {

    /**
     * Méthode de création
     * La méthode retourne un objet ClasseA, si le paramètre est true.
     * La méthode retourne un objet ClasseB, sinon.
     * @return Un objet de classe ClasseA ou ClasseB.
     */
    public AbstractClasse creerClasse(boolean pIsClasseA) {
        if(pIsClasseA) {
            return new ClasseA();
```

```
            }
            else {
                return new ClasseB();
            }
        }
    }
```

FabriqueB.java
Sélectionnez

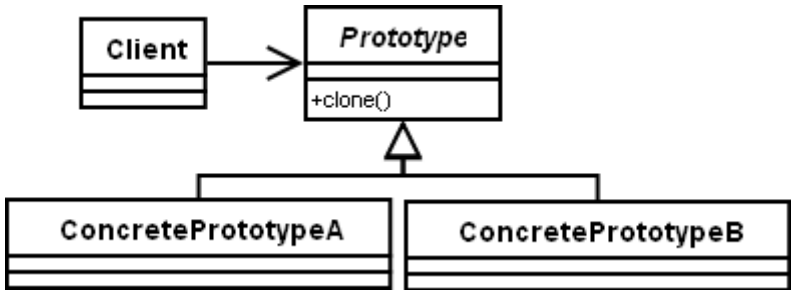
```
/**
 * Substitue la méthode "creerClasse".
 * Instancie un objet "ClasseB".
 */
public class FabriqueB extends Fabrique {

    /**
     * Méthode de création
     * La méthode ne tient pas compte du paramètre
     * et instancie toujours un objet "ClasseB"
     * @return Un objet de classe ClasseB.
     */
    public AbstractClasse creerClasse(boolean pIsClasseA) {
        return new ClasseB();
    }
}
```

FactoryMethodPatternMain.java
Sélectionnez

```
public class FactoryMethodPatternMain {

    public static void main(String[] args) {
        // Création des fabriques
        Fabrique lFactoryA = new FabriqueA();
        Fabrique lFactoryB = new FabriqueB();

        // L'appel de cette méthode avec FabriqueA provoquera
        // l'instanciation de deux classes différentes
        System.out.println("Avec la FabriqueA : ");
        lFactoryA.operation();
        lFactoryA.operation();
        lFactoryA.operation();
        // L'appel de cette méthode avec FabriqueB provoquera
        // toujours l'instanciation de la même classe
        System.out.println("Avec la FabriqueB : ");
        lFactoryB.operation();
        lFactoryB.operation();
        lFactoryB.operation();

        // Affichage :
```

```
            // Avec la FabriqueA :
            // Objet de classe 'ClasseA'
            // Objet de classe 'ClasseB'
            // Objet de classe 'ClasseA'
            // Avec la FabriqueB :
            // Objet de classe 'ClasseB'
            // Objet de classe 'ClasseB'
            // Objet de classe 'ClasseB'
        }
    }
```

IV-D. Prototype (Prototype) ▲



OBJECTIFS :

- Spécifier les genres d'objet à créer en utilisant une instance comme prototype.
- Créer un nouvel objet en copiant ce prototype.

RAISONS DE L'UTILISER :

Le système doit créer de nouvelles instances, mais il ignore de quelle classe. Il dispose cependant d'instances de la classe désirée.

Cela peut être le cas d'un logiciel de DAO comportant un copier-coller. L'utilisateur sélectionne un élément graphique (cercle, rectangle, ...), mais la classe traitant la demande de copier-coller ne connaît pas la classe exacte de l'élément à copier.

La solution est de disposer d'une duplication des instances (élément à copier : cercle, rectangle). La duplication peut être également intéressante pour les performances (la duplication est plus rapide que l'instanciation).

RESULTAT :

Le Design Pattern permet d'isoler l'appartenance à une classe.

RESPONSABILITES :

- **Prototype** : définit l'interface de duplication de soi-même.
- **ConcretePrototypeA** et **ConcretePrototypeB** : sont des sous-classes concrètes de **Prototype**. Elles implémentent l'inteface de duplication.
- La partie cliente appelle la méthode **clone()** de la classe **Prototype**. Cette méthode retourne un double de l'instance.

IMPLEMENTATION JAVA :

Prototype.java
Sélectionnez

```
/**
 * Définit l'interface de l'objet à dupliquer.
 */
public abstract class Prototype implements Cloneable {

    protected String texte;

    /**
     * Constructeur de la classe.
     * @param pTexte
     */
    public Prototype(String pTexte) {
        texte = pTexte;
    }

    /**
     * La méthode clone est protected dans Object.
     * On doit la substituer pour la rendre visible.
     * De plus, il faut que la classe implémente l'interface Cloneable.
     * Depuis Java5, on peut retourner un sous-type de Object.
     */
    public Prototype clone() throws CloneNotSupportedException {
        return (Prototype)super.clone();
    }

    public void setTexte(String pTexte) {
        texte = pTexte;
    }

    /**
     * Méthode d'affichage des informations de l'objet.
     */
    public abstract void affiche();
}
```

ConcretePrototypeA.java
Sélectionnez

```
/**
 * Sous-class de Prototype.
 */
public class ConcretePrototypeA extends Prototype {

    public ConcretePrototypeA(String pTexte) {
        super(pTexte);
    }
```

```
    /**
     * Méthode d'affichage.
     * Indique que c'est un objet de classe ConcretePrototypeA
     * et la valeur de l'attribut texte.
     */
    public void affiche() {
        System.out.println("ConcretePrototypeA avec texte : " + texte);
    }
}
```

ConcretePrototypeB.java
Sélectionnez

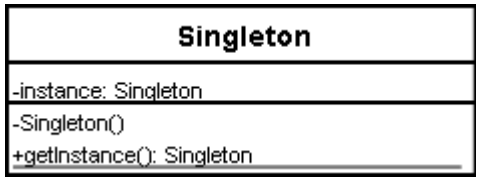
```
/**
 * Sous-class de Prototype.
 */
public class ConcretePrototypeB extends Prototype {

    public ConcretePrototypeB(String pTexte) {
        super(pTexte);
    }

    /**
     * Méthode d'affichage.
     * Indique que c'est un objet de classe ConcretePrototypeA
     * et la valeur de l'attribut texte.
     */
    public void affiche() {
        System.out.println("ConcretePrototypeB avec texte : " + texte);
    }
}
```

PrototypePatternMain.java
Sélectionnez

```
public class PrototypePatternMain {

    public static void main(String[] args) throws CloneNotSupportedException {
        // Instancie un objet de classe ConcretePrototypeA
        // et un autre de classe ConcretePrototypeB
        // de "manière traditionnelle".
        Prototype lPrototypeA = new ConcretePrototypeA("Original");
        Prototype lPrototypeB = new ConcretePrototypeB("Original");

        // Duplique les objets précédemment créés/
        Prototype lPrototypeAClone = lPrototypeA.clone();
        Prototype lPrototypeBClone = lPrototypeB.clone();

        // Affiche les objets :
        // les clones sont identiques aux originaux
```

```
            lPrototypeA.affiche();
            lPrototypeAClone.affiche();
            lPrototypeB.affiche();
            lPrototypeBClone.affiche();

            // Modifie les clones
            lPrototypeAClone.setTexte("Clone (enfait)");
            lPrototypeBClone.setTexte("Clone (enfait)");

            // Met en évidence que les clones
            // sont bien des instances à part.
            lPrototypeA.affiche();
            lPrototypeAClone.affiche();
            lPrototypeB.affiche();
            lPrototypeBClone.affiche();

            // Affichage :
            // ConcretePrototypeA avec texte : Original
            // ConcretePrototypeA avec texte : Original
            // ConcretePrototypeB avec texte : Original
            // ConcretePrototypeB avec texte : Original
            // ConcretePrototypeA avec texte : Original
            // ConcretePrototypeA avec texte : Clone (enfait)
            // ConcretePrototypeB avec texte : Original
            // ConcretePrototypeB avec texte : Clone (enfait)
        }
    }
```

IV-E. Singleton (Singleton) ▲



LIEN VERS LE DICTIONNAIRE DES DEVELOPPEURS :

Singleton

AUTRES RESSOURCES SUR DEVELOPPEZ.COM :

Le singleton par Sébastien MERIC

Le Singleton en environnement Multithread par Christophe Jollivet

OBJECTIFS :

- Restreindre le nombre d'instances d'une classe à une et une seule.
- Fournir une méthode pour accéder à cette instance unique.

RAISONS DE L'UTILISER :

La classe ne doit avoir qu'une seule instance.

Cela peut être le cas d'une ressource système par exemple.

La classe empêche d'autres classes de l'instancier. Elle possède la seule instance d'elle-même et fournit la seule méthode permettant d'accéder à cette instance.

RESULTAT :

Le Design Pattern permet d'isoler l'unicité d'une instance.

RESPONSABILITES :

Singleton doit restreindre le nombre de ses propres instances à une et une seule. Son constructeur est privé : cela empêche les autres classes de l'instancier. La classe fournit la méthode statique **getInstance()** qui permet d'obtenir l'instance unique.

IMPLEMENTATION JAVA :

Singleton.java
Sélectionnez

```
public class Singleton {

    /**
     * La présence d'un constructeur privé supprime
     * le constructeur public par défaut.
     */
    private Singleton() {
    }

    /**
     * SingletonHolder est chargé à la première exécution de
     * Singleton.getInstance() ou au premier accès à SingletonHolder.instance ,
     * pas avant.
     */
    private static class SingletonHolder {
        private final static Singleton instance = new Singleton();
    }

    /**
     * Méthode permettant d'obtenir l'instance unique.
     * @return L'instance du singleton.
     */
    public static Singleton getInstance() {
        return SingletonHolder.instance;
```

```
    }
}
```

Vous avez aimé ce tutoriel ? Alors partagez-le en cliquant sur les boutons suivants : Partager

---