



Comprendre les différents design patterns de construction


Table des matières

- 1. Introduction
- 2. Problématique générale
- 3. Les modèles de construction
 - 3.1. Fabrique abstraite (Abstract Factory)
 - 3.1.1. Définition
 - 3.1.2. Diagramme de classe
 - 3.1.3. Implémentation en C#
 - 3.2. Monteur (Builder)
 - 3.2.1. Définition
 - 3.2.2. Diagramme de classe
 - 3.2.3. Implémentation en C#
 - 3.3. Fabrique (Factory Method)
 - 3.3.1. Définition
 - 3.3.2. Diagramme de classe
 - 3.3.3. Implémentation en C#
 - 3.4. Prototype (Prototype)
 - 3.4.1. Définition
 - 3.4.2. Diagramme de classe
 - 3.4.3. Implémentation en C#
 - 3.5. Singleton (Singleton)
 - 3.5.1. Définition
 - 3.5.2. Diagramme de classe
 - 3.5.3. Implémentation en C#
- 4. Récapitulatif
- 5. Liens
- 6. Remerciements

Comprendre les différents design patterns de construction fait partie d'une suite d'articles que j'ai écrits pour expliquer comment implémenter les 23 modèles de conception les plus connus. Dans cet article, nous allons nous concentrer sur le fonctionnement des design patterns liés à la construction d'objets, les deux autres familles feront le sujet d'un autre article. N'hésitez pas à laisser votre avis sur le contenu de l'article directement via le forum : 11 commentaires ★★★★★

Article lu -1 fois.

L'auteur

Jean-Michel Ormes 

L'article

Publié le 4 avril 2011 - Mis à jour le 4 avril 2011

Version PDF Version hors-ligne

ePub, Azw et Mobi

Liens sociaux



1. Introduction ▲

Qu'est-ce qu'un design pattern (ou patron de conception) ? Il s'agit tout simplement d'un schéma qui forme une solution à un problème connu ou récurrent. Ce sont des solutions connues et dont la conception est due à l'expérience de programmeurs. Le concept de design patterns est né des travaux de quatre personnes (Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides plus communément appelés « Gang of Four ») dans leur ouvrage « Design Patterns : Elements of Reusable Object-Oriented Software ». De façon générale, on utilise un design pattern pour diminuer le temps nécessaire au développement d'une application et pour augmenter la qualité du résultat attendu à un traitement donné.

2. Problématique générale ▲

Afin de mieux illustrer la mise en œuvre des modèles de conception, nous allons prendre un exemple d'étude de cas. Vous êtes à la tête d'un géant constructeur d'ordinateurs et de pièces détachées. Il vous faut imaginer tout le processus, allant de la création des ordinateurs ou des pièces jusqu'à leur livraison.

3. Les modèles de construction ▲

Les patterns de construction déterminent comment faire l'instanciation et la configuration des classes et des objets. Ces patterns permettent également d'encapsuler les classes concrètes et de rendre indépendant la façon dont les objets sont créés.

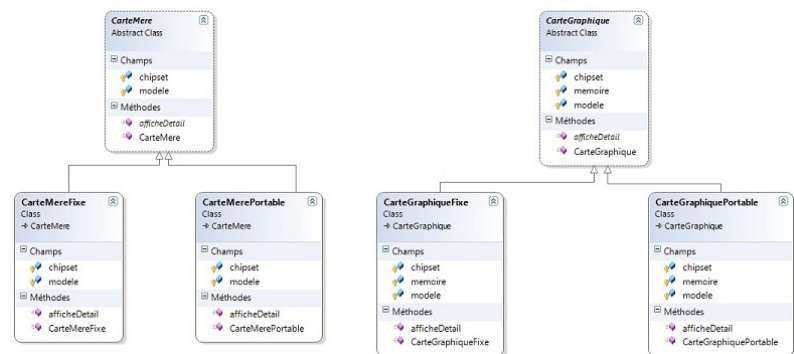
3.1. Fabrique abstraite (Abstract Factory) ▲

3.1.1. Définition ▲

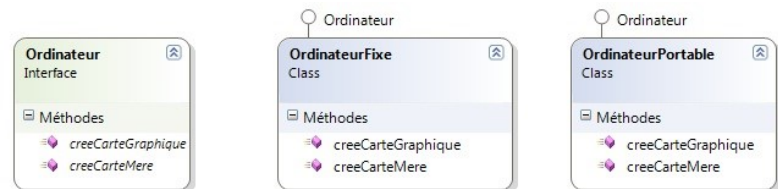
Le pattern Abstract Factory permet, à partir d'une interface, de créer une famille d'objets sans pour autant spécifier de classes concrètes. C'est de ce pattern dont nous aurons besoin pour créer nos différentes pièces pour ordinateurs fixes ou ordinateurs portables. Pour chaque type d'ordinateur ou de pièce, nous disposons d'une classe abstraite et de n sous-classes, qui définiront chacune un modèle spécifique. Un client qui passe une commande personnalisée en choisissant lui-même ses pièces est un exemple concret d'utilisation de ce pattern. Ça veut dire que chaque composant est une classe abstraite, et que chaque version du composant est une sous-classe.

3.1.2. Diagramme de classe ▲

Les diagrammes de classe suivants permettent de mieux comprendre comment fonctionne le pattern Abstract Factory :



Nous avons créé deux classes abstraites CarteMere et CarteGraphique qui contiennent chacune deux sous-classes concrètes (CarteMereFixe, CarteMerePortable, CarteGraphiqueFixe et CarteGraphiquePortable). Par souci de simplicité, je n'ai pas créé les autres classes d'objets appartenant à un ordinateur. Nous avons également créé une interface Ordinateur qui contient la signature des méthodes permettant de créer les différentes pièces pour un ordinateur fixe ou portable. Deux classes OrdinateurFixe et OrdinateurPortable implémentent cette interface.



3.1.3. Implémentation en C# ▲

Voici un exemple de code simplifié d'utilisation du pattern Abstract Factory. Tout d'abord les classes abstraites ainsi que leurs sous-classes concrètes :

CarteMere.cs
Sélectionnez

```
public abstract class CarteMere
{
    protected String modele;
    protected String chipset;

    public CarteMere(String modele, String chipset)
    {
        this.modele = modele;
        this.chipset = chipset;
    }

    public abstract void afficheDetail();
}
```

CarteMereFixe.cs
Sélectionnez

```
public class CarteMereFixe : CarteMere
{
    protected String modele;
    protected String chipset;
```

```
public CarteMereFixe(String modele, String chipset)
    : base(modele, chipset)
{
    this.modele = modele;
    this.chipset = chipset;
}

public override void afficheDetail()
{
    Console.WriteLine("Carte mere pour fixe cree : " + modele +
        " , chipset integre :" + chipset);
}
}
```

CarteMerePortable.cs

Sélectionnez

```
public class CarteMerePortable : CarteMere
{
    protected String modele;
    protected String chipset;

    public CarteMerePortable(String modele, String chipset)
        : base(modele, chipset)
    {
        this.modele = modele;
        this.chipset = chipset;
    }

    public override void afficheDetail()
    {
        Console.WriteLine("Carte mere pour portable cree : " + modele +
            " , chipset integre :" + chipset);
    }
}
```

CarteGraphique.cs

Sélectionnez

```
public abstract class CarteGraphique
{
    protected String modele;
    protected String memoire;
    protected String chipset;

    public CarteGraphique(String modele, String memoire,
        String chipset)
    {
        this.modele = modele;
        this.memoire = memoire;
        this.chipset = chipset;
    }

    public abstract void afficheDetail();
}
```

CarteGraphiquePortable.cs

Sélectionnez

```
public class CarteGraphiquePortable : CarteGraphique
{
    protected String modele;
    protected String memoire;
    protected String chipset;

    public CarteGraphiquePortable(String modele, String memoire,
        String chipset)
        : base(modele, memoire, chipset)
    {
        this.modele = modele;
        this.memoire = memoire;
        this.chipset = chipset;
    }

    public override void afficheDetail()
    {
        Console.WriteLine("Carte graphique pour portable cree : " + modele +
            " , " + memoire + " Mo de RAM, chipset integre :" + chipset);
    }
}
```

CarteGraphiqueFixe.cs

Sélectionnez

```
public class CarteGraphiqueFixe : CarteGraphique
{
    protected String modele;
    protected String memoire;
    protected String chipset;

    public CarteGraphiqueFixe(String modele, String memoire,
        String chipset)
        : base(modele, memoire, chipset)
    {

```

```

        this.modele = modele;
        this.memoire = memoire;
        this.chipset = chipset;
    }

    public override void afficheDetail()
    {
        Console.WriteLine("Carte graphique pour fixe cree : " + modele +
            " , " + memoire + " Mo de RAM, chipset integre : " + chipset);
    }
}

```

Voici le code source de l'interface **Ordinateur** ainsi que ses deux classes d'implémentation **OrdinateurFixe** et **OrdinateurPortable** :

Ordinateur.cs

Sélectionnez

```

public interface Ordinateur
{
    CarteMere creeCarteMere(String modele, String chipset);

    CarteGraphique creeCarteGraphique(String modele, String memoire, String chipset);
}

```

OrdinateurFixe.cs

Sélectionnez

```

public class OrdinateurFixe : Ordinateur
{
    public CarteMere creeCarteMere(string modele, string chipset)
    {
        return new CarteMereFixe(modele , chipset);
    }

    public CarteGraphique creeCarteGraphique(string modele, string memoire, string chipset)
    {
        return new CarteGraphiqueFixe(modele, memoire, chipset);
    }
}

```

OrdinateurPortable.cs

Sélectionnez

```

public class OrdinateurPortable : Ordinateur
{
    public CarteMere creeCarteMere(string modele, string chipset)
    {
        return new CarteMerePortable(modele, chipset);
    }

    public CarteGraphique creeCarteGraphique(string modele, string memoire, string chipset)
    {
        return new CarteGraphiquePortable(modele, memoire, chipset);
    }
}

```

Voici un petit programme en mode console permettant de tester le pattern Abstract Factory :

Program.cs

Sélectionnez

```

static void Main(string[] args)
{
    Ordinateur ordinateur;

    // Stock de cartes meres et cartes graphiques
    CarteGraphique[] carteGraphique = new CarteGraphique[10];
    CarteMere[] carteMere = new CarteMere[10];

    // Un client arrive et passe commande d'un Ordinateur
    Console.WriteLine("Voulez-vous commander un " +
        "ordinateur fixe (1) ou portable (2)?");
    String choix = Console.ReadLine();

    if (choix.Equals("1"))
    {
        ordinateur = new OrdinateurFixe();
    }
    else
    {
        ordinateur = new OrdinateurPortable();
    }

    carteGraphique[0] = ordinateur.creeCarteGraphique("modele1", "512", "chipset32");
    carteMere[0] = ordinateur.creeCarteMere("modele1", "chipset32");

    Console.WriteLine("Votre demande a été traitée.");
    carteGraphique[0].afficheDetail();
    carteMere[0].afficheDetail();
    Console.ReadKey();
}

```

Et le résultat que l'on obtient :

```

file:///C:/Users/Jean-Michel/Documents/Visual Studio 2008/Projects/Design patterns dvp/Abstract...
Voulez-vous commander un ordinateur fixe <1> ou portable <2>?
2
Votre demande a été traitée.
Carte graphique pour portable créé : modele1 , 512 Mo de RAM. chipset intégré :
chipset32
Carte mère pour portable créé : modele1 , chipset intégré : chipset32
  
```

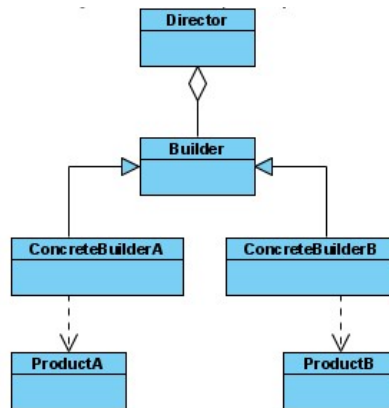
3.2. Monteur (Builder)▲

3.2.1. Définition▲

Le pattern Monteur permet de créer des objets complexes à partir d'autres objets. Concrètement, c'est comme si on assemblait plusieurs objets pour les "monter" et n'en faire qu'un. Exemple : un menu. L'objet menu se compose d'autres objets (sandwich, frites, boisson, etc.). Dans notre cas, nous allons nous servir de ce pattern pour monter différentes tours de PC.

3.2.2. Diagramme de classe▲

Le diagramme de classe suivant permet de mieux comprendre comment fonctionne le pattern Builder :



Le Bildeur est la classe abstraite qui contient le produit. Le Director est celui qui construit un objet utilisant la méthode de conception Bildeur. ConcreteBuilderA et ConcreteBuilderB fournissent une implémentation du Bildeur. Ce sont ces classes qui construisent et assemblent les différentes parties de nos tours. ProductA et ProductB correspondent à nos tours en cours de construction.

3.2.3. Implémentation en C#▲

Voici un exemple de code d'utilisation du pattern Builder :

Tour.cs

Sélectionnez

```

public class Tour
{
    private String alimentation = "";
    private String carteMere = "";
    private String boitier = "";
    private String disqueDur = "";
    private String memoireRAM = "";
    private String carteGraphique = "";
    private String processeur = "";
    private String carteSon = "";

    public void setAlimentation(String alimentation) { this.alimentation = alime
    public void setCarteMere(String carteMere) { this.carteMere = carteMere; }
    public void setBoitier(String boitier) { this.boitier = boitier; }
    public void setDisqueDur(String disqueDur) { this.disqueDur = disqueDur; }
    public void setMemoireRAM(String memoireRAM) { this.memoireRAM = memoireRAM;
    public void setCarteGraphique(String carteGraphique) { this.carteGraphique =
    public void setProcesseur(String processeur) { this.processeur = processeur;
  
```

```

        public void setCarteSon(String carteSon) { this.carteSon = carteSon; }

        public void Informations()
        {
            Console.WriteLine("Tour creee : " + alimentation + ", " +
                carteMere + ", " + boitier + ", " + disqueDur + ", " +
                memoireRAM + ", " + carteGraphique + ", " + processeur + ", " +
                carteSon);
        }
    }
}

```

MonteurTour.cs Sélectionnez

```

public abstract class MonteurTour
{
    protected Tour tour;

    public Tour getTour() { return tour; }
    public void creerNouvelleTour() { tour = new Tour(); }

    public abstract void monterAlimentation();
    public abstract void monterCarteMere();
    public abstract void monterBoitier();
    public abstract void monterDisqueDur();
    public abstract void monterMemoireRAM();
    public abstract void monterCarteGraphique();
    public abstract void monterProcesseur();
    public abstract void monterCarteSon();
}

```

MonteurTour1.cs Sélectionnez

```

class MonteurTour1 : MonteurTour
{
    public override void monterAlimentation() {tour.setAlimentation("Alimentatio
    public override void monterCarteMere() {tour.setCarteMere("Carte mere ATX");
    public override void monterBoitier() {tour.setBoitier("Boitier moyen");}
    public override void monterDisqueDur() {tour.setDisqueDur("DD 7200 tours");}
    public override void monterMemoireRAM() {tour.setMemoireRAM("1024Mo");}
    public override void monterCarteGraphique() {tour.setCarteGraphique("ATI Rad
    public override void monterProcesseur() { tour.setProcesseur("Intel Dual Cor
    public override void monterCarteSon() { tour.setCarteSon("Creative Sound Bla
}

```

MonteurTour2.cs Sélectionnez

```

class MonteurTour2 : MonteurTour
{
    public override void monterAlimentation() { tour.setAlimentation("Alimentati
    public override void monterCarteMere() { tour.setCarteMere("Carte mere DTX")
    public override void monterBoitier() { tour.setBoitier("Boitier moyen"); }
    public override void monterDisqueDur() { tour.setDisqueDur("DD 5400 tours");
    public override void monterMemoireRAM() { tour.setMemoireRAM("2048Mo"); }
    public override void monterCarteGraphique() { tour.setCarteGraphique("Nvidia
    public override void monterProcesseur() { tour.setProcesseur("Intel Dual Cor
    public override void monterCarteSon() { tour.setCarteSon("Creative Sound Bla
}

```

Technicien.cs Sélectionnez

```

class Technicien
{
    private MonteurTour monteurTour;

    public void setMonteurTour(MonteurTour m) { monteurTour = m; }
    public Tour getTour() { return monteurTour.getTour(); }

    public void construireTour()
    {
        monteurTour.creerNouvelleTour();
        monteurTour.monterAlimentation();
        monteurTour.monterCarteMere();
        monteurTour.monterBoitier();
        monteurTour.monterDisqueDur();
        monteurTour.monterMemoireRAM();
        monteurTour.monterCarteGraphique();
        monteurTour.monterProcesseur();
    }
}

```


disque et de finaliser le disque. Dans notre exemple, Le Produit est la classe DisqueDur. Le Produit Concret est une classe qui correspond à un disque dur spécifique. Cette classe hérite de la classe Produit. Dans notre exemple, les produits concrets sont les classes DisqueDurATA et DisqueDurSCSI.

3.3.3. Implémentation en C#▲

Voici un exemple de code d'utilisation du pattern Factory :

DisqueDur.cs
Sélectionnez

```
public abstract class DisqueDur
{
    public abstract void setNbTours(String nombre);

    public abstract void Tester();

    public abstract void Finaliser();
}
```

FabriqueDisqueDur.cs
Sélectionnez

```
public abstract class FabriqueDisqueDur
{
    public DisqueDur creerDisqueDur(String controleur)
    {
        DisqueDur disqueDur;

        disqueDur = setControleur(controleur);
        disqueDur.setNbTours("7200");
        disqueDur.Tester();
        disqueDur.Finaliser();

        return disqueDur;
    }

    protected abstract DisqueDur setControleur(String typeControleur);
}
```

FabriqueDisqueDurATA.cs
Sélectionnez

```
public class FabriqueDisqueDurATA : FabriqueDisqueDur
{
    override protected DisqueDur setControleur(String controleur)
    {
        DisqueDur disqueDurATA = null;
        if (controleur == "ATA")
            disqueDurATA = new DisqueDurATA();

        return disqueDurATA;
    }
}
```

FabriqueDisqueDurSCSI.cs
Sélectionnez

```
public class FabriqueDisqueDurSCSI : FabriqueDisqueDur
{
    override protected DisqueDur setControleur(String controleur)
    {
        DisqueDur disqueDurSCSI = null;
        if (controleur == "SCSI")
            disqueDurSCSI = new DisqueDurSCSI();

        return disqueDurSCSI;
    }
}
```

DisqueDurSCSI.cs
Sélectionnez

```
public class DisqueDurSCSI: DisqueDur
{
    public DisqueDurSCSI()
    {
        Console.WriteLine("Disque Dur SCSI cree");
    }

    public override void setNbTours(String nombre)
    {
        Console.WriteLine("Nb tours : " + nombre + " tours");
    }

    public override void Tester()
    {
        Console.WriteLine("Tests en cours..");
    }

    public override void Finaliser()
```



```

    {
        Console.WriteLine("Disque dur SCSI operationnel");
    }
}

```

DisqueDurATA.cs
Sélectionnez

```

public class DisqueDurATA : DisqueDur
{
    public DisqueDurATA()
    {
        Console.WriteLine("Disque Dur ATA cree");
    }

    public override void setNbTours(String nombre)
    {
        Console.WriteLine("Nb tours : " + nombre + " tours");
    }

    public override void Tester()
    {
        Console.WriteLine("Tests en cours..");
    }

    public override void Finaliser()
    {
        Console.WriteLine("Disque dur ATA operationnel");
    }
}

```

Program.cs
Sélectionnez

```

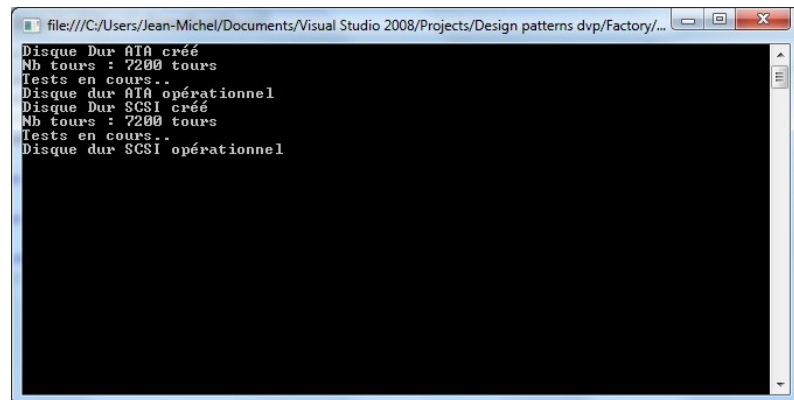
class Program
{
    static void Main(string[] args)
    {
        FabriqueDisqueDur disqueDurATA = new FabriqueDisqueDurATA();
        disqueDurATA.creerDisqueDur("ATA");

        FabriqueDisqueDur disqueDurSCSI = new FabriqueDisqueDurSCSI();
        disqueDurSCSI.creerDisqueDur("SCSI");

        Console.ReadKey();
    }
}

```

Et le résultat que l'on obtient :



```

file:///C:/Users/Jean-Michel/Documents/Visual Studio 2008/Projects/Design patterns dvp/Factory/...
Disque Dur ATA créé
Nb tours : 7200 tours
Tests en cours..
Disque dur ATA opérationnel
Disque Dur SCSI créé
Nb tours : 7200 tours
Tests en cours..
Disque dur SCSI opérationnel

```

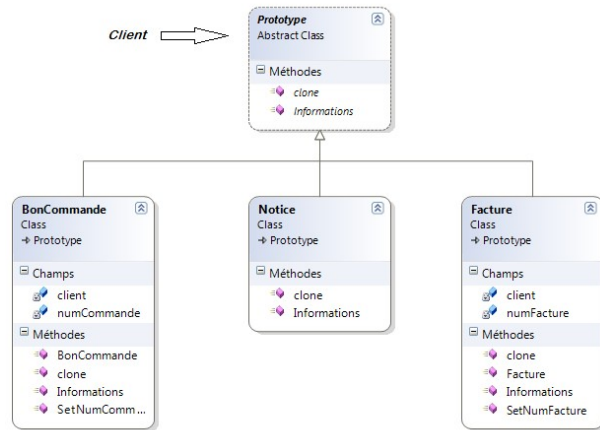
3.4. Prototype (Prototype) ▲

3.4.1. Définition ▲

Le pattern Prototype permet de créer de nouveaux objets à partir d'objets existants. Ces objets sont appelés prototypes et possèdent une capacité de duplication. Dans le cadre de notre super méga géant constructeur d'ordinateurs et de pièces détachées, ce design pattern permettrait de créer toute la panoplie de papiers (bon de commande, facture, notice d'utilisation, etc.).

3.4.2. Diagramme de classe ▲

Le diagramme de classe suivant permet de mieux comprendre comment fonctionne le pattern Prototype :



La classe Prototype est le squelette principal qui permet de créer les nouvelles copies. Les classes BonCommande, Facture et Notice sont les sous-classes spécifiques ayant leurs propres attributs. La méthode Clone() retourne une copie de l'objet.

3.4.3. Implémentation en C#▲

Voici un exemple de code d'utilisation du pattern Prototype :

Prototype.cs
Sélectionnez

```
public abstract class Prototype
{
    public abstract Prototype clone();

    public abstract void Informations();
}
```

Notice.cs
Sélectionnez

```
public class Notice : Prototype
{
    public override Prototype clone()
    {
        return (Prototype)this.MemberwiseClone();
    }

    public override void Informations()
    {
        Console.WriteLine("Notice d'utilisation generee");
    }
}
```

Facture.cs
Sélectionnez

```
public class Facture : Prototype
{
    String numFacture = "";
    String client = "";

    public Facture(String Client)
    {
        this.client = Client;
    }

    public override Prototype clone()
    {
        return (Prototype)this.MemberwiseClone();
    }

    public override void Informations()
    {
        Console.WriteLine("Facture n°{0} generee pour {1}", numFacture, client);
    }

    public void SetNumFacture(String Facture)
    {
        this.numFacture = Facture;
    }
}
```

BonCommande.cs
Sélectionnez

```
public class BonCommande : Prototype
{

```

```

String numCommande = "";
String client = "";

public BonCommande(String Client)
{
    this.client = Client;
}

public override Prototype clone()
{
    return (Prototype)this.MemberwiseClone();
}

public override void Informations()
{
    Console.WriteLine("Bon de commande n°{0} genere pour {1}", numCommande,
    }

public void SetNumCommande(String numCommande)
{
    this.numCommande = numCommande;
}
}

```

Main.cs

Sélectionnez

```

class Program
{
    static void Main(string[] args)
    {
        String numFacture1 = "F.2011.001";
        String numFacture2 = "F.2011.002";

        String Client1 = "Jean-Michel";
        String Client2 = "Philippe";

        Facture f1 = new Facture(Client1);
        f1.SetNumFacture(numFacture1);
        Console.WriteLine("1ere sortie : ");
        f1.Informations();

        Facture f2 = (Facture)f1.clone();
        Console.WriteLine("2eme sortie : ");
        f2.Informations(); // Avant modification du numero de facture
        f2.SetNumFacture(numFacture2);
        Console.WriteLine("3eme sortie : ");
        f2.Informations(); // Apres modification

        Console.ReadKey();
    }
}

```

Et le résultat que l'on obtient :

```

file:///C:/Users/Jean-Michel/Documents/Visual Studio 2008/Projects/Design patterns dvp/Prototyp...
1ere sortie :
Facture n°F.2011.001 générée pour Jean-Michel
2eme sortie :
Facture n°F.2011.001 générée pour Jean-Michel
3eme sortie :
Facture n°F.2011.002 générée pour Jean-Michel

```

Initialement, on crée une facture ayant pour numéro "F.2011.001", pour le client Jean-Michel. On clone ensuite cette facture puis on vérifie que le résultat est bien celui attendu, à savoir que le client est toujours Jean-Michel et que le numéro de facture est inchangé. On modifie ensuite le numéro de facture.

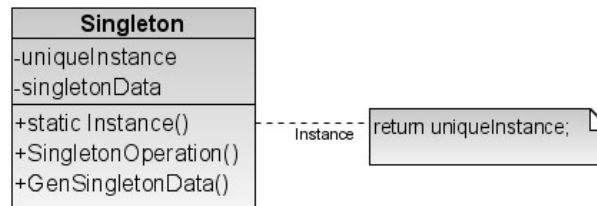
3.5. Singleton (Singleton) ▲

3.5.1. Définition ▲

Le pattern Singleton permet de s'assurer qu'une classe ne possède qu'une seule instance. Ce pattern sera utile par exemple quand on effectuera nos tests de démarrage sur un disque dur. En effet, celui-ci ne devra pas être démarré deux fois.

3.5.2. Diagramme de classe ▲

Le diagramme de classe suivant permet de mieux comprendre comment fonctionne le pattern Singleton :



Le plus important dans la classe Singleton est la méthode Instance(). Celle-ci est implémentée de façon à ne créer qu'une seule instance. Par ailleurs, pour vérifier qu'une instance est déjà créée, on utilise un attribut privé appelé **_instance**.

3.5.3. Implémentation en C# ▲

Voici un exemple de code d'utilisation du pattern Singleton :

TestDisqueDur.cs
Sélectionnez

```

public class TestDisqueDur
{
    public DateTime dt;

    private static TestDisqueDur _instance = null;

    private TestDisqueDur() { }

    public static TestDisqueDur Instance()
    {
        if (_instance == null)
        {
            _instance = new TestDisqueDur();
            Console.WriteLine("Démarrage du disque....");
        }

        return _instance;
    }

    public void afficheDetail()
    {
        Console.WriteLine("Disque dur démarre.");
    }
}
  
```

Voici un programme permettant de tester le pattern Singleton :

Program.cs
Sélectionnez

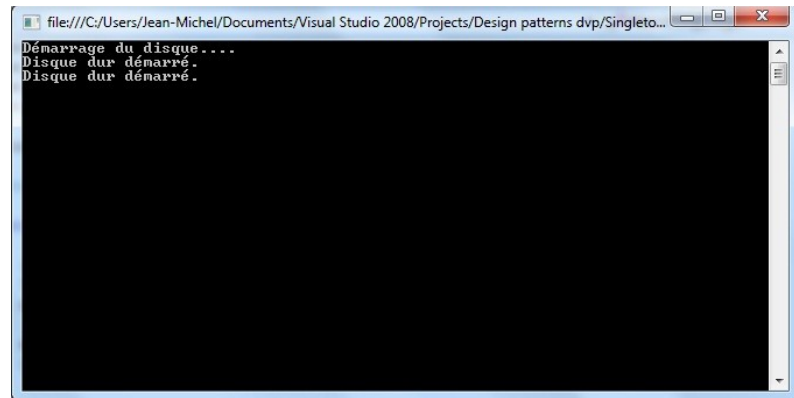
```

class Program
{
    static void Main(string[] args)
    {
        //Instanciation du disque
        TestDisqueDur disque = TestDisqueDur.Instance();
        disque.afficheDetail();

        AutreDisque();
        Console.ReadKey();
    }

    public static void AutreDisque()
    {
        //On essaie d'instancier un nouveau disque
        TestDisqueDur disque = TestDisqueDur.Instance();
        disque.afficheDetail();
    }
}
  
```

Et le résultat que l'on obtient :



Initialement, on crée un disque dur de test puis on appelle la méthode `afficheDetail()` de ce disque. Comme prévu, on a bien le démarrage qui s'effectue. On appelle ensuite la méthode `AutreDisque()`. Dans cette même méthode, on essaye d'instancier un nouveau disque dur de test et on appelle la méthode `afficheDetail()`. Vu que l'attribut `_instance` n'est pas null, la méthode `Instance()` ne créera pas de nouveau disque dur de test et renverra l'instance déjà créée.

4. Récapitulatif ▲

Nous venons de passer en revue les différents patterns de construction. Voici un récapitulatif des cas où il faut implémenter ces patterns :

- On utilise le pattern Fabrique Abstraite :
 - Quand on doit créer une famille d'objets qui se ressemblent.
 - Quand les objets d'une famille peuvent évoluer.
- On utilise le pattern Monteur :
 - Quand on doit créer un objet complexe qui nécessite plusieurs étapes.
- On utilise le pattern Fabrique :
 - Quand on doit créer un objet concret en fonction d'un paramètre donné.
 - Quand on a besoin de faire évoluer ou ajouter des classes concretes sans modifier la classe abstraite.
- On utilise le pattern Prototype :
 - Quand on doit créer de nouveaux objets à partir d'objets existants soit par copie, soit par clonage.
 - Quand la création d'un objet est plus coûteuse que de le copier.
- On utilise le pattern Singleton :
 - Quand un objet ne doit posséder qu'une seule et unique instance sous peine de générer une erreur.
 - Quand un objet créé deux fois est susceptible de générer une erreur.

5. Liens ▲

Je vous invite à lire également ces liens :

Le pattern adaptateur par Florian CASABLANCA
 Le pattern Singleton par Ronald VASSEUR
 Les critiques de livres

6. Remerciements▲

Je tiens à remercier ClaudeLELOUP pour sa relecture orthographique ainsi que Philippe Vialatte et 3DArchi pour leurs remarques techniques.

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

