**Design Patterns**

La POO a introduit la notion de réutilisation du code en créant des acteurs autonomes que sont les objets, les design patterns inventent la réutilisation des idées. Après avoir résolu un problème original on peut généraliser cette solution en créant un design pattern.

# PRINCIPE SOLID

SOLID est l'acronyme de cinq principes de base (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation et Dependency Inversion) que l'on peut appliquer au développement objet.

Ces principes, lorsqu'ils sont compris et suivis, permettent d'améliorer la cohésion, de diminuer le couplage, et de favoriser l'encapsulation d'un programme orienté objet.

En suivant ces principes, cela vous permettra de développer une application maintenable, plus fiable et plus robuste.

## SINGLE RESPONSABILITY

Que ce soit une classe ou une méthode, il est important que celle-ci ne puisse faire qu’une et une seule chose : la responsabilité unique.

Prenons l’exemple d’une machine à café : son unique utilité est de préparer du café. Si vous souhaitez de la bière, ce n’est pas la bonne machine.

« Moi j’ai une machine qui fait les deux »

Eh bien… ce n’est pas une machine à café, c’est une machine tout court (ou autre), mais pas à café. Je fais cette précision car la précision du terme est important. Ce principe va vous obliger à minimiser vos classes ou vos services et donc éviter ce que l’on appelle une « God Class ».

Pour pousser ce principe au maximum, il faut que vous réduisiez le plus possible les fonctionnalités de vos classes.

Typiquement, pour notre machine à café, nous ne pouvons pas avoir une classe qui chauffe l’eau et qui, en même temps, fasse couler le café. Ce n’est pas la même fonctionnalité !

Et si vous n’êtes pas sûr de ce que vous faites ou si vous pensez ne pas avoir suffisamment réduit la responsabilité de votre classe, n’hésitez pas à faire relire votre code !

## OPEN / CLOSED

Une classe doit être ouverte à l’extension et fermée à la modification. Dans les faits, c’est très simple : pourquoi voudriez-vous modifier un code source qui fonctionne ? Mis à part pour la correction d’un bug… Mais, nous sommes tous d’accord pour dire que notre code ne bug jamais ? Non ?

Si vous respectez le premier principe, vous allez considérablement réduire la densité de vos classes ainsi que la taille de vos méthodes. Vous allez multiplier les fichiers mais par expérience, je vous garantis qu’il est plus agréable d’avoir beaucoup de fichiers avec peu de lignes de code qu’un fichier de 1 000 lignes à maintenir (oui 1 000 lignes c’est 10 fois trop. J’insiste : 10 fois trop, allez… 6 ou 7 fois)

L’idée est donc de ne pas toucher à ce qui fonctionne. Grâce à l’objet vous avez différentes méthodes pour améliorer / étendre votre classe ou votre méthode :

L’héritage

Les design patterns dis « de structure »

La surcharge …

Ne serait-ce qu’avec toutes ces solutions, vous pouvez ouvrir une classe à l’extension sans avoir à la modifier !

Rien qu’avec toutes ces solutions, vous pouvez ouvrir une classe à l’extension sans avoir à la modifier !

En même temps, vous vous voyez modifier votre machine à café pour en faire une machine à bière ? Non, ça n’a pas de sens ! Et, en plus, on appellerait ça du bricolage.

## LISKOV SUBSTITUTION

On aurait pu trouver plus simple comme nom… Il nous vient de Barbara Liskov qui nous affirme que :

Si q(x) est une propriété démontrable pour tout objet x de type T, alors q(y) est vraie pour tout objet y de type S tel que S est un sous-type de T.

Pour illustrer cette phrase, vous entendrez régulièrement l’exemple suivant : “Un carré est un rectangle” (qui est bien vrai mathématique). Si l’on suit rigoureusement le principe d’héritage dans la programmation objet qui se base sur la formule : “est un”, vous allez avoir envie de faire une classe Carré qui hérite de la classe Rectangle.

Erreur ! il ne faut surtout pas car vous allez vous heurter à des problèmes de conception. Un carré ne se comporte pas comme un rectangle d’un point de vue objet pur. Quand vous allez modifier le côté d’un carré, la largeur et la hauteur vont se confondre, ce qui n’est pas vrai avec un rectangle. De fait, lorsque le carré va hériter des getters et des setters du rectangle, vous allez devoir changer fondamentalement le fonctionnement de base (la largeur, et la hauteur).

# Creational Patterns

## Abstract factory

Ce DP permet de créer des instances pour des groupes, des familles de classes différents sans que le code qui les utilise n’ait à se soucier de ces groupes ou familles.

## Builder

Ce DP décrit comment séparer la construction d’un objet complexe de sa représentation.

## Factory method

Cet DP ressemble à l’abstract factory en permettant à une méthode de créer des instances dérivées d’une classe mère. Par exemple on peut supposer une gestion de document de différents types, des CV, des articles, des news, etc, chacun ayant sa structure interne propre. La factory method de la classe « document » permettra à l’application de gérer tout type de document et sera capable de créer les instances des parties différentes du document en fonction de son type (un CV comportera les parties identité, diplômes et expérience, un article comportera les parties titre, introduction, texte et conclusion, ainsi de suite…).

## Prototype

Le prototype est une instance correctement initialisée servant de modèle à la création d’un clone. Cet DP permet par exemple d’implémenter une collection d’objets décrivant des formes géométriques qui, au lieu d’être utilisées directement pour créer des dessins seront d’abord dupliqués comme on pioche dans un stock d’outils.

## Singleton

Ce DP est la star, celui qui vient en premier si on demande de citer un DP. Il permet de s’assurer qu’une seule instance d’une classe ne peut exister à un moment donné et fournit un point d’accès vers cette instance. Par exemple, une application gérant une machine outil voudra s’assurer qu’il ne peut pas y avoir au même moment deux instances de la classe permettant de commander la machine afin d’éviter que les ordres envoyés par plusieurs instances ne se mélangent. De façon générale dès qu’un objet interagit avec une ressource unique qu’il se réserve il est classique de le voir implémenter cette DP.

# Structural patterns

## Adapter

L’adapter (adaptateur) est un procédé permettant d’habiller en quelque sorte une classe en la faisant ressembler, du point de vue de son interface, à une autre. Par exemple on peut supposer une application qui travaille sur des objets de classe A exposant les méthodes M1 et M2. L’évolution du produit fait que la classe A est abandonnée au profit d’une classe B qui fait ce que faisait A mais différemment. Pour ne pas perturber l’ancienne application et dans l’attente de la reprogrammer on concevra un adaptateur se présentera comme étant de classe A avec les méthodes M1 et M2 et qui en réalité instanciera un objet de classe B avec d’autres méthodes qui feront le travail “sous le manteau”. L’appel de la méthode M1 par l’application sera traduite par l’adaptateur en l’appel des méthodes X1, X2 et X3 du nouvel objet. De fait, l’application aura toujours l’impression de travailler sur la classe A alors que la librairie sous-jacente sera totalement différente. Les nouvelles applications utiliseront elles directement la nouvelle classe.

## Bridge

Ce DP créé un pont entre une classe et l’implémentation de tout ou partie de son code. On l’utilise par exemple lorsqu’on sait que l’implémentation d’une classe risque de changer dans le futur alors qu’on ne désire pas modifier l’application utilisatrice. On fige l’interface, utilisée par le client, et on délègue l’implémentation à d’autres classes. La classe d’interface étant liée à l’exécution à une classe d’implémentation contenant le code utile.

## Composite

Le composite est un DP permettant de créer des structures arborescentes d’objets ayant une relation tout/partie tout en autorisant les clients de traiter les objets individuels et les compositions de façon uniforme.

## Decorator

Avec ce pattern on dispose d’un moyen efficace pour accrocher de nouvelles fonctions à un objet dynamiquement lors de l’exécution. Cela ressemble à la fois un peu aux interfaces et à l’héritage mais c’est une troisième voie offrant des avantages différents. Les nouvelles fonctionnalités sont contenues dans des objets, l’objet parent étant dynamiquement mis en relation avec ces derniers pour en utiliser les capacités.

## Façade

Il s’agit ici de proposer une façade masquant la complexité de tout un système. Par exemple une classe GestionDesStocks permettra de manipuler depuis d’autres endroits dans une chaîne de gestion commerciale l’ensemble des modules afférents à la gestion des stocks. Bien implémentée cette DP permet une incroyable modularité et la réutilisation simplifiée du code dans des applications complexes. L’implémentation “moderne” doit pour être fortement découplée souvent passer par une conteneur IoC par exemple mais ce n’est pas une obligation du pattern.

## Flyweight

Cette DP autorise le partage de très nombreux objets de petite granularité de façon efficace.

## Proxy

Cela n’a rien à voir avec les « proxy » du Web même si dans l’esprit il y a un fond commun. En réalité ce DP définit comment un objet permet d’en contrôler un autre. On l’utilise très fréquemment pour manipuler localement la représentation d’un objet distant (Remoting/WCF, services web, etc.).

# Behavioral patterns

## Chain of responsibility

Ce DP définit une structure permettant de découpler un objet émettant des requêtes des objets chargés de les traiter. Dans une chaîne de responsabilité la requête est transmise d’objet en objet jusqu’à ce que l’un d’entre eux la traite.

## Command

https://scottlilly.com/c-design-patterns-the-command-pattern/

Lorsqu’un système doit gérer de nombreuses commandes qui peuvent être mises en attente et être éventuellement annulée il est intéressant de transformer chaque commande en un objet placé dans une liste. Un exemple très simple peut être celui d’un programme de type calculatrice permettant un niveau quelconque de undo. Chaque élément du calcul est un objet placé dans une liste. Le module de calcule traite la liste, la touche undo efface le dernier objet de la liste.



Command (Command) :

* déclarer une interface pour exécuter un opération

ConcreteCommand (CalculatorCommand) :

* defines a binding between a Receiver object and an action
* implements Execute by invoking the corresponding operation(s) on Receiver

Client (CommandApp)

* creates a ConcreteCommand object and sets its receiver

Invoker (Invoker)

* asks the command to carry out the request

Receiver (Calculator)

* knows how to perform the operations associated with carrying out the request.

## Interpreter

Ce DP propose une structure de classes permettant, à partir d’un langage et d’une représentation de sa grammaire, de pouvoir interpréter des phrases écrites dans ce langage.

## Iterator

Ce DP propose un moyen d’accéder à tous les objets d’une liste séquentiellement sans avoir besoin d’exposer la structure interne qui maintient cette liste.

## Mediator

Cet DP définit le fonctionnement d’un objet qui gère les interactions entre d’autres objets. C’est un médiateur. Les objets considérés ont ainsi un couplage très faible entre eux ce qui coïncide avec les guidelines les plus récentes. Par exemple un objet ChatRoom permet la communication entre des objets clients, chaque client peut communiquer avec d’autres clients mais aucun n’a accès directement aux autres clients. Tous communiquent en utilisant les services du médiateur.

## Memento

Ce DP met en place un mécanisme permettant de capturer l’état interne d’un objet pour le stocker et restaurer ultérieurement l’objet dans le même état. Le DP memento peut être étendue en réalité à tout système de sérialisation. Les langages .NET comme C# exploitent les mécanismes de sérialisation offert par le framework ou des librairies tierces (Json.Net par exemple), ces mécanismes n’existaient pas “out of the box” à la création du catalogue du Gang Of Four. Avec les langages plate-formes modernes comme C# et .NET le DP memento perd de son intérêt mais il existe tout de même des cas où il peut encore jouer un rôle.

## Observer

Lorsque plusieurs objets doivent être averti d’une notification il faut trouver un moyen pour distribuer le message. C’est ce que propose la DP observer (l’observateur). Ici aussi on peut reconnaitre des éléments des langages modernes. C# sait par exemple gérer les évènements à diffusion multiple (multicast events) qui ne sont autres qu’une façon d’implémenter le DP observer…

## State

Ce DP décrit comment un objet peut modifier son comportement quand son état change. Par exemple un objet compte bancaire peut gérer différemment les retraits, le paiement d’intérêts et les frais de fonctionnement selon le solde que présente le compte. L’état interne peut être représenté par un objet qui applique les nouvelles règles. L’objet compte s’occupant de s’accrocher un objet état approprié selon sa balance.

## Strategy

Pourvoir modifier un algorithme de façon transparente peut être très utile. On peut supposer une application permettant de trier des éléments selon des méthodes différentes. Ce DP explique comment mettre en œuvre le mécanisme rendant le choix de l’algorithme transparent pour l’objet l’utilisant.

## Template method

Ce DP définit comment mettre en place un algorithme dont les étapes sont figées mais dont la réalisation de ces dernières peut être déléguée à d’autres objets offrant des variations de traitement. La DP template method créé en quelque sorte la trame immuable d’un traitement tout en laissant la liberté de réalisation de chaque étape à d’autres intervenants du système. Par exemple on peut définir dans un objet conteneur de données une méthode ExportData permettant d’exporter des données sous un format quelconque mais segmentant l’opération en étapes précises : OuvrirDestination, EcrireEntête, EcrireCorps, EcrirePied, FermerDestination. Toute exportation sera toujours faite de ces cinq étapes. Ensuite on peut implémenter des objets d’exportation implémentant ces cinq méthodes (exportation vers XML, CSV, etc.). La technique de la template method est extrêmement souple et s’adapte à de nombreux cas de figure.

## Visitor

Le visiteur peut être vu comme l’opération complémentaire à l’itérateur. Le visiteur est une classe définissant une opération sur une autre classe. L’objet visité autorise l’appel d’un visiteur en publiant généralement une méthode prenant en paramètre une instance de visiteur. Comme il est ensuite possible de créer des dérivées de la classe visiteuse, on peut créer autant d’opérations différentes pouvant être appliquée à la classe visitée sans avoir à en modifier le code. Bien utilisée ce DP est très efficace pour laisser ouverte la définition de nouveaux traitements pouvant s’appliquer sur des collections d’objets (liste de clients, de salariés, articles…). La liste expose une méthode de type TraiterAvec(TypeVisiteur leVisitieur) dont le rôle est de balayer chaque élément en lui appliquant le visiteur. On peut ainsi, pour une liste de clients, créer un visiteur « promotion » qui créditera de points cadeaux les clients répondant à certains critères, un autre jour on pourra créer un autre visiteur envoyant un mail aux clients répondant à d’autres critères, etc. C’est un véritable jeu qui vitalise les idées !

# Les DP dans le Framework .NET

Dans un article de juillet 2008 (oui ça fait loin !) je vous parlais d’un excellent petit bouquin écrit par des développeurs de l’équipe qui a écrit le Framework .NET. “Framework Design Guidelines” est vraiment toujours un super livre de référence surtout pour ceux qui doivent écrire des librairies un peu grosses pour les besoin de plusieurs applications. Vous pouvez jeter un œil à l’article de l’époque et voir si on trouve encore ce livre en vente, si c’est le cas, n’hésitez pas à l’acheter !

Ce livre évoque certains DP utilisés dans le Framework lui-même et c’est une première source qui m’a donné envie de faire une petite liste. Je n’ai jamais eu le temps de terminer ce jeu de piste (il faut dire que le code du Framework est un peu gros !) mais on peut toujours s’amuser à compléter la liste ci-dessous (et il y a matière à faire !) :

Creational patterns

Abstract Factory

On la retrouve dans DbProviderFactory de System.Data.Common, tous les membres de la classe sont des factory methods.

Builder

Toute l’infrastructure de construction des canaux de WCF. Mais plus simplement les builders de chaînes de connexion (comme SqlConnectionStringBuilder et équivalents). On trouve aussi UriBuilder qui implémente le même DP.

Factory method

IDbConnection.BeginTransaction() dans System.Data. Le type de la transaction créée dépend de l’implémentation de IDbConnection qui est instanciée. WebRequest.Create() dans un autre domaine du Framework retourne un type concret qui dépend de l’URL (son type). L’implémentation de la classe Convert peut aussi être vue comme une application de ce DP.

Prototype

Utilisé pour gérer la sérialisation / désérialisation.

Singleton

On ne trouve partout, la difficulté étant de savoir s’ils sont implémentés réellement comme des singleton ou s’il ne s’agit que de propriétés initialisées une fois. Les Settings d’une application WPF par exemple peuvent être considérés comme un singleton du point de vue de l’utilisation qui en est faite. Ce n’est pas forcément comme tel que cela est implémenté dans le code du Framework.

Structural patterns

Adapter

Les providers de ADO.NET comme par exemple SqlConnection (System.Data.SqlClient), OleDbConnection (System.Data.OleDb). Chaque provider est un adaptateur pour la base de données cible considérée. Les wrappers pour appeler du code COM (tlbimp.exe) depuis du code managé .NET est un autre exemple de mise en pratique de l’adaptateur.

Composite

Il y en a beaucoup, le plus évident étant System.Windows.Forms.Control et ses classes dérivées. Le même principe se retrouve dans System.Web.UI.Control et ses classes dérivées ou dans System.Xml.XmlNode et les classes qui en descendent.

Decorator

Sous WPF on trouve directement System.Windows.Controls.Decorator. Certaines implémentations de la classe Stream sont des décorateurs autour d’un flux interne (par exemple GZipSteam, CryptoStream, MemoryStream, BufferedStream, FileStream, IsolatedStorageStream, PipeStream,…).

Façade

Le XmlSerializer dans System.Xml.Serialization a pour but de cacher un processus plus complexe (qui inclut la génération à la volée d’assemblages) derrière une classe très simple à utiliser.

Proxy

Les proxies de Web services générés par svcutil.exe dérivant de System.ServiceModel.Clientbase<TChannel>.

Behavioral patterns

Chain of Responsability

OnBubbleEvent() dans System.Web.UI.Control et RaiseBubbleEvent().

Command

Les commandes de WPF et le ICommand très utilisé aujourd’hui pour respecter MVVM !

Interpreter

System.Linq.Expressions.Expression et toutes les classes qui tournent autour de ces expressions (représentation mémoire, interprétation, exécution).

Iterator

L’exemple peut-être le plus évident étant IEnumerable dans System.Collections et sa version générique IEnumérable<T>. Le IDataReader fonctionne aussi sur le principe de ce DP. Foreeach est bien entendu un élément du langage entièrement dédié à ce DP, le mot clé yield aussi.

Memento

La sérialisation .NET est une application du principe de ce DP.

Observer

Toute la logique des events de .NET.

Strategy

La méthode Sort dans ArrayList est un bon exemple.

Template Method

La méthode Render des Custom Controls.

Visitor

ExpressionVisitor dans System.Linq.Expressions qui est utilisé par Linq en interne.