Notes sur le module : *Architecture .NET & C #*

Armel Pitelet

19 décembre 2021

Table des matières

Fichier de cours correspondant : POEC_Architecture.pdf

1 Sur l'architecture

Définition : **Arhitecture logiciel**.

Décrit d'une manière symbolique et schématique les différents éléments d'un ou de plusieurs systèmes informatiques, leurs interrelations et leurs interactions. Contrairement aux spécifications produites par l'analyse fonctionnelle, le modèle d'architecture, produit lors de la phase de conception, ne décrit pas ce que doit réaliser un système informatique mais plutôt comment il doit être conçu de manière à répondre aux spécifications. L'analyse décrit le « quoi faire » alors que l'architecture décrit le "comment le faire".

2 Principes SOLID

Voir directement SOLID pour la définition et l'intérêt.

- Slide n°7 : Un code ne respectant pas le Single Responsibility Principle (SRP) se repère aux commentaires du type, dans ce cas, sauf si, sauf quand, ou... Le fractionnement des responsabilité permet une meilleur maintenabilité. Le nom de la classe doit donner la responsabilité de cette dernière afin de rendre le code plus lisible.

- Slide n°8 : Pour respecter l'Open Close Principle (OCP) il faut éviter de faire des conditions particulières. Il faut par exemple éviter de spécifier du code par type. Il est préférable d'utiliser l'héritage pour rajouter un comportement différent. Il vaut mieux remplacer les `if/switch` (en fonction d'un type) par de l'héritage.

- Slide n°9 : Pour respecter le Liskov Substitution Principle (LSP) il faut que les classes d'un sous types (hérité d'un parent) aient le même comportement que le parent. Plus précisément on ne doit pas altérer la réaction d'une classe (un ajout doit rester un ajout, et ne pas lever une exception par exemple).

- Slide n°10 : Pour respecter le Interface Segregation (ISP) il ne faut pas implémenter des fonctionnalités que l'on ne souhaite pas utiliser. Il vaut mieux séparer (ségréger).

- Slide n°11 : Pour respecter le Dependancy Inversion Principle (DSP) il vaut mieux éviter les couplage fort entre les classes. Il est préférable d'injecter une dépendance par le constructeur plutôt que d'utiliser une dépendance directement "hardcodée". Cela mène à un couplage faible.

Remarque : *Sur le DdC.*

Il va falloir sérieusement repenser le DdC pour respecter ces principes (se renseigner sur la modélisation des interfaces)

3 Sur les DP

3.1 Définitions

Définition : **Patrons de conception**.

Solutions classiques à des problèmes récurrents de la conception de logiciels. Chaque patron est

une sorte de plan ou de schéma que l'on peut personnaliser afin de résoudre un problème récurrent dans votre code. Il en existe trois types :

- **Création** : qui fournit des mécanismes de création d'objets, ce qui augmente la flexibilité et la réutilisation du code.
- **Structurel** : qui explique comment assembler des objets et des classes en de plus grandes structures, tout en les gardant flexibles et efficaces.
- **Comportemental** : qui met en place une communication efficace et répartit les responsabilités entre les objets.

Définition : **Anti-patron**.

Ou *antipattern* est une erreur courante de conception. Il apparait quand un DP est mal utilisé, entrainant lenteur exccessive, maintenance élevé, comportements anormaux et bugs. Voir : https://fr.wikipedia.org/wiki/Antipattern.

3.2 Liste des DP vues en cours

Pattern : **Singleton**, (type : *Création*)
Description : Garantie que l'instance d'une classe n'existe qu'en un seul exemplaire, tout en fournissant un point d'accès global à cette instance.
Exemple : Singleton → Singleton.sln : Program.cs
Avantages :

- Garantie l'unicité d'une instance de classe.
- Fournit un point d'accès global à cette instance.
- L'instance est uniquement initialisée la première fois qu'elle est appelée.

Inconvénients :

- Ne respecte pas le SRP car il résout deux problèmes à la fois.
- Peut masquer une mauvaise conception.
- Doit bénéficier d'un traitement spécial pour fonctionner correctement en multithread. Sans, c'est un *antipattern* puisqu'il est alors possible de créer plusieurs instances.
- Difficile à tester par conception car ne permet pas l'héritage (utilisé dans de nombreux frameworks de test.)

Liens : https://refactoring.guru/fr/design-patterns/singleton,
https://www.goprod.bouhours.net/?page=pattern&pat_id=19.

Pattern : **Facade**, (type : *Structurel*)
Description : Procure une interface unifié à un ensemble d'interfaces dans un sous système pour un accès simplifié à une librairie, un framework ou n'importe quel ensemble complexe de classe. Cela permet de masquer la complexité d'un système (comme une bibliothèque) tout en fournissant un point d'entré dans ce dernier. On peut par exemple déléguer les échanges avec une base de donnée à une Façade.
Exemple : Facade → Facade.sln : Program.cs, DataAccess.cs, Facade.cs, SubSys.cs.
Avantages :

- Permet d'isoler son code de la complexité d'un sous système.

Inconvénients :

- Peut devenir un objet omniscient (*antipattern*) qui reconnait ou fait trop de chose, ce qui viole le SRP.
- Peut réduire les fonctionnalité de la librairi masqué par rapport à une utilisation direct de cette dernière.

Liens : https://refactoring.guru/fr/design-patterns/facade,
https://www.goprod.bouhours.net/?page=pattern&pat_id=9.

Remarque : *Sur la façade.*

il faut mettre la façade en `public` et les sous système de cette façade en visibilité `internal`. De cette manière seule la façade à accès à ses sous systèmes.

Note :

Pour utiliser `internal` il faut placer la facade et ses sous systèmes dans un nouveau projet de type bibliothèque de classe (voir la différence dans les .csproj) sinon les fichiers sont considérés dans le même package ce qui tue l'intêret du `internal`.

Pattern : **Strategy**, (type : *Comportemental*)
Description : Définit une famille d'algorithmes, les encapsules et les rends interchangeables. Cela permet de faire varier le comportement exact d'un algorithm indépendamment de l'utilisateur.
Exemple : ExempleStrategy → ExempleStrategy.sln : Program.cs, IRouteStrategy.cs, Navigator.cs, RoadStrategy.cs, WalkingStrategy.cs.
Avantages :

- Permet de permuter un algorithme utilisé dans un objet à l'exécution.
- Sépare les détail d'implémentation d'un algorithme du code qui l'utilise.
- Permet de remplacer l'héritage par de la composition.
- Respecte l'OCP ce qui permet d'ajouter de nouvelles stratégies sans avoir à modifier le contexte.

Inconvénients :

- Rend le programme plus complexe en raison du nombre de classe (devant respecter une interface) accompagnant le patron (encombre le code avec des classes et interfaces supplémentaires).
- C'est à l'utilisateur de comparer les stratégies pour choisir la bonne.

Liens : https://refactoring.guru/fr/design-patterns/strategy,
https://www.goprod.bouhours.net/?page=pattern&pat_id=21.

Remarque : *Sur la Stratégie.*

La classe utilisable par le client possède une interface qui définit les algorithmes utilisables. C'est cette même classe qui utilise les algorithmes mais le client doit avoir accès aux différents algorithmes pour les passer à la classe utilisable.

Pattern : **Factory**, (type : *Création*)
Description : Définit une interface pour créer des objets dans une classe mère mais délègue le choix des types d'objets à créer aux sous classe. La création d'objets (produits) n'est cependant pas la responsabilité principale du créateur (la factory). La classe créateur a en général déjà un fonctionnement propre lié à la nature de ses produits. La fabrique aide à découpler cette logique des produits concrets.
Exemple : Factory → FactoryExemple.sln : Program.cs, IInstrument.cs, Piano.cs, PianoFactory.cs
Avantages :

- Permet de séparer la création d'un objet de cet objet.
- Respecte le SRP. On peut déplacer tout le code de création d'un objet au même endroit ce qui assure une meilleure maintenabilité.
- Respecte l'OCP puisque l'on peut ajouter de nouveaux types de produit sans modifier les produits existants.

Inconvénients :

- Introduit de nombreuses sous classes. Interessant uniquement si l'on a déjà une hiérarchie dans la création des classes.

Liens : https://refactoring.guru/fr/design-patterns/factory-method,
https://www.goprod.bouhours.net/?page=pattern&pat_id=10.

Remarque : *Sur la Fabrique.*
La classe de création utilise une interface qui définit les sous classe à créer. La classe de création parent doit être abstraite et c'est aux sous classes de création d'instancier les objets qui doivent respecter le contrat définit par l'interface portée dans la classe de création parent.

4 Sur le C#

4.1 Mots clefs et définitions

Pour des exemples de code voir :
— BaseCSharp → BaseCSharp.sln,
— Exceptions → Exceptions.sln,
— Heritage → Heritage.sln.

Définition : **Type valeur**.
Les principaux type valeurs sont les type primitifs (*built-in*) comme les `int` ou `string`. Un type valeur est soit un type **structures**, qui encapsule les données et les fonctionnalités associées, soit un type **énumération**, qui est défini par un ensemble de constantes nommées et qui représente un choix ou une combinaison de choix . Avec les types valeur, chaque variable a sa propre copie des données et les opérations sur une variable ne peuvent pas affecter l'autre car un type valeur contient directement la donnée et pas une référence vers cette donnée. Lien : https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/builtin-types/value-types

Remarque : *Sur les types valeur.*
Par convention ils commencent par une minuscule. Laz valeur par défaut d'un type valeur dépend du type en question.

Définition : **Type reference**.
Les variables de type **référence** font référence à leurs données (objets), elles pointent vers la donnée et ne la contiennent pas directement. Ce qui n'est pas de type valeur est de type référence. Les opérations sur une variable peuvent affecter le même objet référencé par une autre variable. Lien : https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/reference-types

Remarque : *Sur les types références.*
Par convention ils commencent par une majuscule. Par défaut un type référence à la valeur nulle. Les classes créées sont de type référence.

Remarque : *Sur l'opérateur de comparaison "==".*
Il compare par défaut la référence pointé pour les type référence et la valeur sur les types valeurs.

Définition : **Propriété**.
Champ (membre) d'une classe accessible et modifiable via `getter` et `setter`. C'est en fait une méthode spéciales appelées *accesseur*. Une propriété est dit automatique si elle n'est pas lié à un attribut. Une propriété est dite calculé si sa définition inclue une opération. Une telle propriété n'a pas de `setter`. Elle n'a pas de valeur en soit mais est directement calculé à partir d'une autre (propriété).
Lien : https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/classes-and-structs/properties.

Remarque : *Sur le nommage des propriétés.*
Par convention on nomme les Propriétés avec une majuscule au début (contrairement aux attributs qui commencent par "_" ou bien `m_attribut` à la C++).

Remarque : *Sur les propriétés.*
On peut faire une propriété en `readonly` en ne mettant juste pas le setter ! Il faut alors initialiser la propriété dans sa déclaration (pas certain sa).

Remarque : *Sur les propriétés.*
On peut faire des propriété qui se base sur un attribut.

Remarque : *Sur les object initializers et les propriétés.*
L'utilisation des *initializers* (via les { Prop = ... } au lieu de ()) passe par les propiétés et pas par le constructeur pour initialiser une instance de classe ! Lien : https ://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/classes-and-structs/object-and-collection-initializers

Définition : **Polymorphisme**.
Le fait de pouvoir utiliser des objets de la même manière mais le comportement de ces objets est différents (les appels de méthodes sont identiques [signature identique] mais le résultats de ces appels est différent. L'implémentation de ces méthodes est différentes malgré une signature identique.) Plusieurs formes pour un même type d'objet (en fonction du sous type). Permet de traiter des éléments d'implémentation différentes de la même manière. Lien : https://docs.microsoft.com/fr-fr/dotnet/csharp/fundamentals/object-oriented/polymorphism.

Mot clef : **interface**.
Une interface est un ensemble de prototypes (déclaration sans définition) qui définie un contrat. Toute classes qui implémente ce contrat doit fournir une implémentation des membres définie dans l'interface. Une interface est instanciable, ce qui permet le polymorphisme (pour stocker un ensemble d'objet de type différents mais respectant tous la même interface).

Remarque : *Sur les interfaces.*
Par convention une interface se note avec un I majuscule : IMonInterface.

Mot clef : **abstract**.
Modificateur qui indique que l'élément en cours de modification a une implémentation manquante ou incomplète. On ne peut définir un membre abstrait que dans une classe abstraire, qui peut posséder des membres abstrait et non-abstraits. Une classe abstraite n'est pas instanciable. Les membres définis comme abstraits doivent être implémentés par des classes non abstraites dérivées de la classe abstraite.

Remarque : *Sur l'héritage des classes abstraites.*
Une classe abstraite peut hériter d'une classe abstraite.

Remarque : *Sur l'abstraction en UML.*
Dans un diagramme de classe une classe abstraite est représentée par un nom de classe en *italique.*

Mot clef : **sealed**.
Modificateur qui permet de marquer une classe comme "final" dans une chaine d'héritage. Mutuellement exclusive avec `abstract`.

Mot clef : **virtual**.

Modificateur de méthode, propriété ou attribut. Permet la substitution du champ dans les classes dérivées (via `override`).

Mot clef : **override**.

Modificateur qui indique la surcharge par un enfant d'un champ du parent marqué comme `abstract` ou `virtual`.

Remarque : *Sur l'appellation surcharge.*

La surcharge est à à la fois le remplacement d'une méthode mère par une méthode fille via virtual et override (la signature de la fonction ne doit pas changer pour effectuer une surcharge) et le fait de redéfinir une fonction pour lui faire accepter différent type d'argument (la signature change).

Définition : **signature**.

la signature (ou prototype) d'une méthode se compose de son nom, son type de retour, les types de ses arguments et sa visibilité.

Mot clef : **base**.

Permet d'accéder aux membre de la classe parent à partir d'une classe dérivé. On peut s'en servir pour appeler le constructeur de la classe mère dans la classe fille. Si la classe mère a un constructeur qui n'est pas par défaut, il est obligatoire d'appeler explicitement le constructeur de la classe Mère dans celui de la classe fille.

Mot clef : **this**.

Permet de faire référeence à l'instance actuelle de la classe en cours (l'objet en cours lors de l'appel). Peut également être utilisé comme modificateur du premier paramètre d'une méthode d'extension.

Mot clef : **internal**.

Visibilitée liée au package (donc ni privé, public ou protected). On ne peut pas accéder à ces sous systèmes en dehors du package mais toutes les classe du package voient les classes `internal` comme si elle étaient `public`.

Remarque : *Sur la visibilité par défaut d'une classe.*

Par défaut une classe est en visibilité `internal`.

Mot clef : **static**.

Modificateur qui rend une classe ou un membre statique. Une classe `static` n'est pas instanciable. Le compilateur force alors le caractère static de tt les attributs, méthodes et propriétés. Un membre `static` est partagé entre toutes les instances d'une classe.

Remarque : *Sur les membres statiques.*

Un membre d'une classe non-statique peut être statique. On parle alors de *membre de classe.*

Définition : **Muable/Immuable**.

Un objet immuable (*immutable*) n'est pas modifiable après sa construction contrairement à un objet muable (*mutable*). C'est une propriété *runtime*. Différence avec constante : un objet imuable l'est au *runtime* et pas à la compilation (*compiltime*).

Mot clef : **const**.

Permet de déclarer un champ ou un élément local comme constant. C'est une propriété *compil-time*, il faut donc que les variables marqués `const` soit intialisées à leur création (à leur déclaration qui doit être une définition).

Mot clef : **readonly**.

Notion similaire à `const` mais *runtime*. L'assignation au champ peut uniquement se produire dans le cadre de la déclaration ou dans un constructeur de la même classe. Un champ en lecture seule peut être affecté et réaffecté plusieurs fois dans la déclaration de champ et le constructeur. Voir : https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/readonly pour le détail exact et les autres cas d'utilisation.

Mot clef : **is**.

Opérateur qui vérifie si le résultat d'une opération est compatible avec un type donné. `is` renvoie un boléen si la convertion est possible et peut également réaliser une affectation. Il est aussi utilisé pour faire correspondre une expression à un **modèle de propriété**. Voir aussi : https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/operators/type-testing-and-cast#is-operator.

Mot clef : **as**.

L'opérateur `as` convertit explicitement le résultat d'une expression en un type de valeur de référence ou nullable. Si la conversion n'est pas possible, l'`as` opérateur retourne `null`. Contrairement à une expression de **cast**, l'`as` opérateur ne lève jamais d'exception.

Remarque : *Sur `is` et `as`.*

Ces deux opérateurs sont utilisés pour la conversion de type référence(De ce que j'ai compris essentiellement pour de la conversion dans une chaine d'héritage).

Remarque : *Sur la conversion de types valeur.*

Pour convertir les types valeur on utilise les méthodes **Parse** et **TryParse**. Le `Parse` lève une erreur en cas d'impossibilité de conversion alors que la version `TryParse` prend un argument supplémentaire (par référence) dans lequel mettre le résultat de la conversion. En cas d'impossibilité, `TryParse` ne lève pas d'erreur.

Mot clef : **ref**.

Permet d'indiquer un passage par référence. `ref` force l'initialisation d'une variable avant son passage contrairement à `out`. Pour les multiples cas d'utilisation voir : https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/ref.

Mot clef : **out**.

Permet d'indiquer un passage par référence. `out` force la non initialisation d'une variable avant son passage contrairement à `ref`.

Remarque : *Sur `ref` et `out`.*

Permettent les passages par référence. Ils ne sont utilisable que sur les variables et pas les *literals*. Attention le concept de passage par référence est différent du concept des types référence et type valeurs. On peut passer des types valeurs par référence !

Remarque : *Sur l'utilisation de `ref` et `out`.*

Attention `ref` et `out` doivent être déclarés à la fois dans la déclaration des arguments (lors de la définition/déclaration) et lors de l'appel des méthodes !

Note :

in à également l'air d'être impliqué dans les passages par référence. Il sert aussi dans les `foreach`. `in` et `out` sont impliqués dans les notions de covariance et contravariance (voir https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/concepts/covariance-contravariance/ : la covariance et la contravariance permettent la conversion de références implicite pour les types tableau).

Définition : **Encapsulation**.

Le fait de masquer l'état interne et les fonctionnalités d'un objet pour autoriser l'accès uniquement via un ensemble de fonctions déclarées `public`. L'idée est d'éviter la modification intempestive du code et d'exposer une interface simple d'utilisation malgré un fonctionnement complexe d'un objet.

Mot clef : **namespace**.

Pour de l'encapsulation. Permet de déclarer une portée qui contient un ensemble d'objet connexe.

Remarque : *Sur la notion de `namespace`.*

Par convention le namespace principale `Projet` est le nom du projet puis chaque dossier a contenue aura une syntaxe du type `Projet.NomDeDossier`, et ainsi de suite.

Mot clef : **using**.

Instruction ou directive. Sous forme de directive il permet de charger dans l'espace local (le fichier courant) les namespaces voulus. Voir : https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/using-directive (directive) et https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/using-statement (instruction) pour plus de détail.

Remarque : *sur les namespace et le using.*

Dans un projet, si les namespace sont tous identiques il est inutiles de les appeler. De plus les sous namespace doivent quand même être appelée dans les dossier contenue dans le dossier parent.

Mot clef : **when**.

Mot cléf contextuel utilisé dans les `switch` permet d'imposer une condition à l'éxécution d'un case lors d'un switch (équivalent case `if{}`).

Mot clef : **partial**.

Le `partial` (au sens Mot clef : **type partial**.

) devant une classe permet de définir une classe dans deux fichiers différents ! public partial A dans A.cs et public partial A dans B.cs résultera en une classe A unique à la compilation. Au sens Mot clef : **méthodes partielles**.

: une méthode déclarée `partial` dans une classe `partial` pourra être étendue par une méthode de signature identique dans une classe de nom identique ailleurs dans le programme. Pour plus de différence classe et méthodes partiel voir : https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods.

4.2 Remarques et liens divers

Note :

Une classe par fichier est une obligation ! ! !

Remarque : *dans un* ***switch***.

Le **default** n'est pas obligatoire... Attention, le mot clef **break** est indispensable pour éviter le "cascadage" des **case** !

Remarque : *Sur les* ***exceptions***.

Toutes les Exceptions dérivent de la classe Exception. Les exceptions remontent sur le stack jusqu'à être catch dans la chaine remontante ou bien arriver au main. Listes : https://docs.microsoft.com/fr-fr/dotnet/api/system.exception?view=net-6.0#choosing-standard-exceptions

Remarque : *Sur les string*.

Un `string` est non mutable ! (imuable). Un `String` est mutable !

Remarque : *Sur les string*.

L'override de la méthode **ToString** permet de modifier le comportement d'un objet lors d'un appel à `Console.Write(objet)`

Remarque : *Sur la méthode* ***isEquals***.

L'interface `IEquatable<T>` permet de forcer l'overrride de la méthode `isEquals` pour la comparaison entre deux objet.

Remarque : *Sur les* ***tuples***.

Les tuples sont mutables en c# !. Il vaut mieux utiliser un objet pour retourner plusieurs variables au lieu d'un tuple.

Note :

Le symbol " ?" dans la déclaration d'un argument (après l'argument) permet de quand même affecter une valeur **null** à un objet non-nullable de base (comme un type valeur).

Remarque : *Sur la visibilité des constructeurs*.

Les constructeurs sont public par défaut.

Remarque : *Sur l'appel d'une méthode avec paramètre par défaut*.

On n'est pas obligé de donner les paramètres déclarés par défaut lors de l'appel de la méthode correspondante. On peut également directement nommé le/les arguments déclarés par défaut dans n'importe quel ordre sans avoir à donner les arguments précédents en utilisant la syntaxe `fun(argNonDefault, arg3: value)` (on a ommit `arg2` [sous entendu déclaré par défaut]).

Remarque : *Sur le Chainage du constructeur*.

Fait via `: base()` entre la déclaration des arguments et la définition d'un constructeur fille. Permet d'éviter la redondance.

Remarque : *Sur les Méthodes d'extension*.

Cela permet de rajouter une méthode à une classe sans avoir à la définir dans la classe. Fonctionne uniquement avec des méthodes statiques dans une classe statique. Lien : https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/classes-and-structs/extension-methods.

Note :

On peut faire prop+tab pour créer une propriété !

Note :

Voir les **Linq Extension** (Méthode d'extension) pour tt ce qui est slice de `list` et opération

plus complexes sur les tableaux.

Remarque : *Lien sur les raccourcis de Visual Studio.*
https://docs.microsoft.com/fr-fr/visualstudio/ide/visual-csharp-code-snippets?view=vs-2022.

Remarque : *Lien pour le site des exercices :.*
https://perso.esiee.fr/~perretb/I3FM/POO1/basecsharp/index.html.

Remarque : *lien intéressant sur le C#.*
https://www.tutlane.com/tutorial/csharp/csharp-pass-by-reference-ref-with-examples.

Note :
Se renseigner sur les relational pattern (c#9 et +) : to compare an expression result with a constant
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns#relational-patterns

Remarque : *Lien sur la vérification et la conversion de type (cast).*
https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/operators/type-testing-and-cast#is-operator.
Remarque : *Lien sur boxing unboxing : conversion de type valeur en type object.*
https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/types/boxing-and-unboxing

Remarque : *Pdf certification c#.*
https://www.certification-questions.com/microsoft-pdf/70-483-pdf.html

Acronymes

DP *Design Pattern* (Patron de Conception). 1–3, *Glossaire :* DP

DSP *Dependancy Inversion Principle* (Principe d'inversion de dépendance). 2, *Glossaire :* DSP

ISP *Interface Segregation Principle* (Principe de substitution de Liskov). 2, *Glossaire :* LSP

LSP *Liskov Subtitution Principle* (Principe de substitution de Liskov). 2, *Glossaire :* LSP

OCP *Open Close Principle* (Principe d'ouverture et de fermeture). 2, 4, *Glossaire :* OPC

SRP *Single Responsibility Principle* (Principe de responsabilité unique). 2–4, *Glossaire :* SRP

Glossaire

DP Un *Design Pattern* (DP) est ... Liens utile :
https://www.goprod.bouhours.net/,
https://refactoring.guru/design-patterns/catalog.

DSP Le *Dependancy Inversion Principle* (DIP) est de D de SOLID. Une classe doit dépendre de son abstraction, pas de son implémentation. Il est préférable d'injecter une dépendance par le constructeur plutôt que d'utiliser une dépendance directement "hardcodée" dans un attribut ou une méthode. Ou encore, une classe de haut niveau (qui manage des classe de bas niveaux pour des taches complexe) ne doit pas dépendre d'une classe de plus bas niveau(qui effectue les actions les plus simples). Elles doivent dépendre toutes les deux de leur abstraction. De plus, une abstraction ne doit pas dépendre de ses détails d'implémentation mais les détails d'implémentation doivent dépendre de d'abstractions. Lien utile https://www.c-sharpcorner.com/article/solid-principles-in-c-sharp-dependency-inversion-principle/.

ISP L'*Interface Segregation Principle* (ISP) est le I de SOLID. Aucun clients ne devrait avoir à implémenter des méthodes / fonctions qu'ils n'utilisent pas. Plus précisément une classe ne doit pas avoir à être implémenté avec une interface qu'elle n'utilise pas. Lien utile : https://www.c-sharpcorner.com/article/solid-principles-in-c-sharp-interface-segregation-principle/.

LSP *Liskov Substitution Principle* (LSP) est le L de SOLID. Une classe fille devrait pouvoir se substituer à une classe mère sans affecter le résultat que l'on attend de l'instance de la classe mère. On ne doit pas altérer la réaction d'une famille de classe. Lien utile : https://www.c-sharpcorner.com/article/solid-principles-in-c-sharp-liskov-substitution-principle/.

OPC L'*Open Close Principle* (OCP) est le O de SOLID. Une entité logicielle (classe, méthodes, fonctions, ...) devrait être ouverte à l'extension mais fermé à la modification. On devrait pouvoir étendre l'implémentation sans toucher à la logique de cette dernière (via héritage par exemple). Lien utile : https://www.c-sharpcorner.com/UploadFile/pranayamr/open-close-principle/.

SOLID Ensemble de principes dont le but est de rendre le code moins bugué, plus facile à lire, plus logique, plus facilement maintenable, testable et extensible. C'est un guide de bonnes pratiques. Liens utiles:
https://github.com/harrymt/SOLID-principles,
https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/,
https://www.c-sharpcorner.com/article/solid-principles-in-c-sharp-interface-segregation-principle/,
https://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/..

SRP Le *Single Responsibility Principle* (SRP) est le S de SOLID. Une classe ne devrait avoir qu'une seul tâche. Lien utile :
https://www.c-sharpcorner.com/article/solid-single-responsibility-principle-with-c-sharp/.