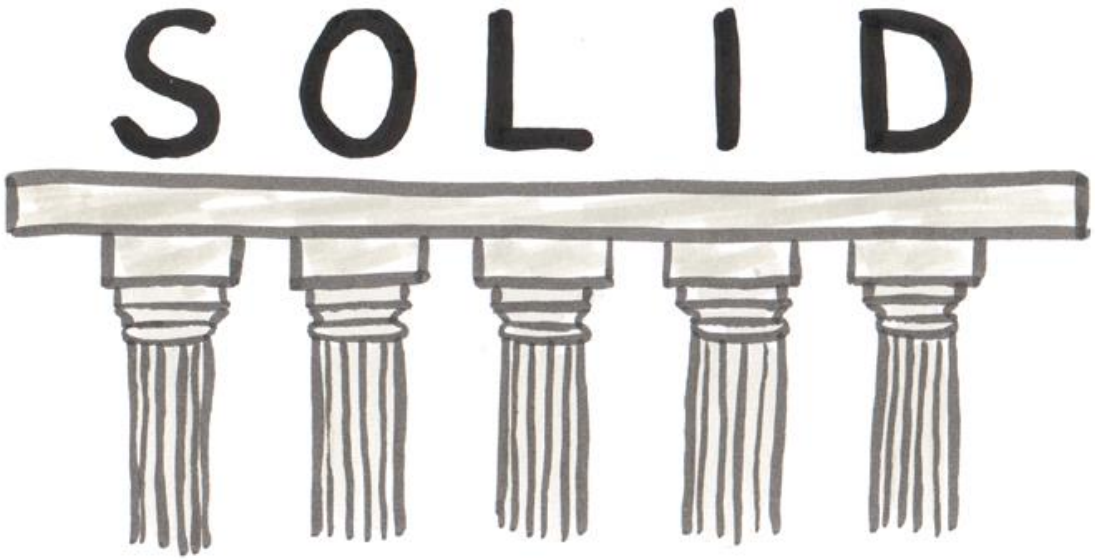
TP1 : Choix 1 (SOLID)

Sommaire



https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/

Introduction

Nous avons tous déjà coder une application en utilisant la programmation orienté objet. Il y a toujours plusieurs manières de coder, mais laquelle est la meilleure ? Quelle méthode devons-nous utiliser. Devons nous multiplier les fonctions, tout mettre dans un même fichier ou pire encore écrire tout dans le main. Nous avons appris au fur et à mesure plusieurs bonnes pratiques. SOLID à l'aide de ses 5 principes aide à mieux coder : rendre le code plus compréhensible, plus maintenable, plus robuste… Nous allons donc voir dans un premier temps une explication de chacun des 5 principes. Puis nous verrons leurs avantages. Enfin, nous étudierons deux codes ne respectant pas les principes de substitution de Liskov et le principe ouvert/fermé, nous verrons pour chacun d'entre eux comment l'implémenter en utilisant les principes SOLID.

Explication des cinq principes SOLID

Les cinq principes SOLID sont :

- Single Responsibility Principle : c'est le principe de responsabilité unique. Il a été énoncé par Martin en 2003 comme « une classe ne doit avoir qu'une seule et unique raison de changer ». C'est-à-dire que la classe doit comporter un minimum de responsabilité. Lorsqu'une classe a trop de responsabilité, cela veut dire que nous pouvons la scinder en plusieurs classes. En effet, résoudre un problème nécessite souvent de diviser le problème en tâche à faire. Cela permet de rendre un problème complexe plus simple. Le code devient plus facile à comprendre, plus facile à corriger et à tester. De plus, plus une classe possède des responsabilités, moins elle sera compatible avec les autres classes. Enfin, si l'on n'applique pas le principe de responsabilité unique et qu'on souhaite changer une méthode de la classe ne respectant pas, cela peut engendrer des modifications des autres méthodes. Par exemple, une class Livre ne doivent pas contenir des méthodes de calcul de facture, de calcul de prix et d'affichage de la facture et de sauvegarde de la facture. Les méthodes de calcul de la facture, d'affichage et de sauvegarde devrait tout être séparée en plusieurs classes. En effet, l'affichage de la facture peut changer si le calcul de la facture change ou si nous voulons changer la façon d'afficher. De même la méthode de sauvegarde de la facture change si le calcul change ou si nous changeons la sauvegarde. Peut être qu'actuellement on enregistre en pdf mais que plus tard nous voudrions l'enregistrer dans un format JSON.
- Open Close Principle : c'est le principe ouvert/fermé. Les objets évoluent dans le temps. Ils doivent être ouverts à l'extension et fermés à la modification. Si on ne modifie pas le code existant, on ne risque pas de le casser. On ne doit jamais supprimer le code existant on rajoute des couches de code par héritage ou polymorphisme et de nouvelles classes. Cela permet d'éviter de fragiliser le code, une fois que les tests unitaires ont été effectués et validés sur la classe, celle-ci ne doit plus être modifiée. Cela le rend plus réutilisable. Par exemple, si nous avons une classe voiture avec une énumération des types possibles, nous ne pouvons pas avoir d'attributs spécifique à un type et rajouter un type modifie le code source. Nous pouvons facilement créer des bugs à la suite de cet ajout. Avec le principe d'open/close, nous allons créer une classe véhicule abstraite. Nous allons ensuite créer des classes qui héritent des propriétés générales de la classe Voiture. L'exemple a été codé dans le dossier TP1_OpenClose1 (ne respecte pas le principe) et TP1_OpenClose2 (respecte le principe).
- Liskov Substitution Principle : c'est le principe de substitution de Liskov. Il est très important d'utiliser l'abstraction en programmation objet, mais en ajoutant toujours plus de niveaux d'abstraction, on finit par avoir beaucoup de dépendances. Les classes filles doivent donc être substituables à leurs classes mère. C'est-à-dire qu'une classe mère doit pouvoir utiliser les fonctions des classes filles même sans savoir qu'elles existent. Le résultat doit être le même si la fonction est utilisée par la classe mère ou par la classe fille. Cependant, ce n'est pas utilisé que pour les classes. En effet toute fonction qui utilise une instance d'un type indiqué ou d'un de ses sous-types doit

obtenir le même résultat. Un exemple concret est donné dans les dossiers TP1_Liskov1 (violation du principe) et TP1_Liskov2 (application du principe)

- Interface Segregation Principle : c'est le principe de ségrégation des interfaces. C'est une méthode permettant de séparer les responsabilités d'un module en plusieurs interfaces. Il faut conserver de petites interfaces simples pour réduire le degré de connexion entre deux éléments. Plus une interface a d'éléments plus la classe qui l'implémente en aura. Le principe de responsabilité unique nous disait d'avoir le moins de responsabilité possible dans une classe. Il faut donc également qu'une interface possède un minimum d'élément. Le principal but de ce principe est de limiter au maximum le nombre d'APIs d'un module. On va donc par exemple répartir les APIs entre les différentes interfaces qui vont ensuite être utilisées par les modules de niveau inférieur. Les modules inferieurs vont pouvoir choisir les API à utiliser et ne pas prendre en compte celles dont ils n'ont pas besoin. Par exemple, nous avons des outils de bricolage (ils représentent les APIs) qui se trouvent dans une caisse à outil. Les méthodes repeindre la porte et assembler un meuble n'auront pas besoin des mêmes outils. Nous allons utiliser un pinceau pour repeindre et une perceuse, un tournevis et des vis pour assembler le meuble. Nous n'avons pas besoin que la méthode repeindre utilise le tournevis. Nous avons donc une interface tournevis qui va être utilisée et connue seulement par les méthodes en ayant besoin.
- Dependency Inversion Principle : c'est le principe d'inversion des dépendances. Les entités doivent dépendre d'abstractions. En effet, un module de haut niveau ne doit pas dépendre du module de bas niveau, mais il doit dépendre d'abstractions. Lorsqu'on fait une modification sur un module de bas niveau, cela ne modifie pas les modules de hauts niveaux. Les modules de haut niveau sont ceux s'occupant des opérations de notre application qui ont une nature abstraite et qui contiennent une logique complexe. Ces modules contrôlent les modules de bas. Les modules de bas niveau sont des composants spécifiques se concentrant sur les détails et les petites parties des applications. De plus, les abstractions ne doivent pas dépendre des détails mais les détails doivent dépendre des abstractions. En effet, pour appliquer cela, nous allons ajouter un module (classe abstraite, interfaces…) entre les différents modules de haut-niveaux et de bas niveaux.

Avantages de chaque principe

- Avantages du Single Responsibility Principle :
    - Ce principe rend plus compréhensible les classes. En effet, le code est divisé en différentes classes ayant chacune le minimum de responsabilité. Le code est donc plus clair et moins lourd car chaque classe ne s'occupe que d'une fonctionnalité.
    - La maintenance est facilitée. En effet, les fonctionnalités étant séparées il est plus facile de remonter les bugs. De plus, il devient plus facile de créer des tests unitaires. Ils nécessitent seulement la validation de la fonctionnalité de la classe.
- Open Close Principle :
    - Il y a davantage de flexibilité par rapport à l'évolution. En effet, tous les modules sont extensibles.
    - Le code est plus robuste. En effet, on ne fait une extension que pour un module seulement donc les dépendances ne sont pas changées. De plus, les modules ne sont jamais modifiés.
- Liskov Substitution Principle :
    - Le code est plus facilement réutilisable. En effet, il y a moins de dépendance entre les différentes classes.
    - La maintenance est facilitée. En effet, le couplage étant réduit entre les classes, cela permet de tester plus aisément une classe avec des tests unitaires.
- Interface Segregation Principle :
    - Le code est plus compréhensible. En effet, la séparation des responsabilités d'un module en plusieurs interfaces permet de découper le code en plus petite partie et donc rendre la compréhension de chaque interface et même des modules plus faciles.
    - La maintenance est facilitée. En effet, les fonctionnalités étant séparées il est plus facile de remonter les bugs. De plus, il devient plus facile de créer des tests unitaires. Ils nécessitent seulement la validation de la fonctionnalité de la classe.
- Dependency Inversion Principle :
    - La maintenance est facilitée. En effet, il permet également de réduire le nombre de dépendances entre les différents modules. En effet, les abstractions ne dépendent pas des détails et les modules de hauts niveaux dépendent d'abstraction et non des modules de bas niveaux.
    - Le code est plus robuste. En effet, il permet de protéger d'un effet de propagation des bugs au sein des modules de bas niveaux. Il permet de dissocier les parties importantes des détails.

Codes ne respectant pas deux principes parmi les 5 de SOLID

Note : Pour les deux premiers codes, une modification a été apporté après les captures effectuées. La méthode description a été enlevée car elle ne respectait pas le principe de responsabilité unique. La classe pouvait être modifié si le calcul changeait ou si nous voulions changer le format du texte.

Les codes sont tous présents et fonctionnels dans le dossier à leur nom.

J'ai utilisé Clion pour coder les différents algorithmes, c'est donc un Cmake qui se trouve dans chaque dossier de projet.

Code 1 ne respectant pas : TP1_Liskov1

Le premier code ne respecte pas le troisième principe de SOLID, c'est-à-dire le principe de substitution de Liskov.

Fichier Rectangle.h :

```h
#ifndef TP1_RECTANGLE_H
#define TP1_RECTANGLE_H

#include <iostream>
using namespace std;

class Rectangle{
private:
    float largeur; //largeur du rectangle en m
    float longueur; //longueur du rectangle en m
public:
    Rectangle(); //Constructeur de la classe rectangle
    Rectangle(float largeurEdit, float longueurEdit); //Constructeur de la classe rectangle et initialisation à partir des valeurs fournie
    virtual void setLargeur(float largeurEdit); //modification de la largeur du rectangle
    virtual void setLongueur(float longueurEdit);   //modification de la longueur du rectangle
    float getLargeur(); //renvoie la valeur de la largeur
    float getLongueur();//renvoie la valeur de la longueur
    float perimetre(); //calcul du perimetre du rectangle
    float aire(); //calcul de l'aire du rectangle
    virtual void description();   //affichage du perimetre et de l'aire du rectangle
};

#endif //TP1_RECTANGLE_H
```

Fichier Carre.h :

```
#ifndef TP1_CARRE_H
#define TP1_CARRE_H

#include "Rectangle.h"
#include <iostream>
using namespace std;

class Carre : public Rectangle {
public:
    Carre();    //Constructeur de la classe carré
    Carre(float size);  //Constructeur de la classe carré et initialisation à partir des valeurs fournies
    void setLongueur(float longueurEdit) override;  //modification de la taille des côtés du carré en m
    void setLargeur(float largeurEdit) override;   //modification de la taille des côtés du carré en m
    void description() override; //affichage du perimetre et de l'aire du carré
};


#endif //TP1_CARRE_H
```

On peut voir que la classe carré est une classe fille de la classe rectangle. La classe carré dépend de la classe rectangle pour les attributs et les calculs d'aire et périmètre. De plus, un carré possède une longueur et une largeur égale, il y a du coup beaucoup de redite dans les différentes méthodes.

Par exemple les méthodes Carre ::setLongueur et Carre ::setLargeur sont exactement les mêmes.

```
void Carre::setLongueur(float longueurEdit) {   //Changement de la longueur des côtés en m
    Rectangle::setLargeur( largeurEdit: longueurEdit); //carré -> largeur = longueur
    Rectangle::setLongueur(longueurEdit);  //carré -> largeur = longueur
}

void Carre::setLargeur(float largeurEdit) {   //Changement de la longueur des côtés en m
    Rectangle::setLargeur(largeurEdit);   //carré -> largeur = longueur
    Rectangle::setLongueur( longueurEdit: largeurEdit);  //carré -> largeur = longueur
}
```

Nous sommes tout le temps obligé de redéfinir les méthodes de longueur et largeur et modifier les deux lorsque nous changeons la taille du carré.

Test des fonctions dans le main :

```cpp
int main() {
    /*
    Rectangle rectangle = Rectangle();  //création d'un rectangle
    rectangle.setLongueur(3);   //modification de la longueur du rectangle
    rectangle.setLargeur(2);    //modification de la largeur du rectangle
    testAire(rectangle);    //test de l'aire du rectangle
*/
    Rectangle carre = Carre( size: 5); //création d'un carré de côtés 5m
    testAire( rectangle: carre); //test de l'aire du carré

    Carre carre2 = Carre( size: 5); //création d'un carré de côtés 5m
    carre2.setLongueur( longueurEdit: 10);
    cout<<"Aire du carré2 : "<<carre2.aire()<<endl; //test de l'aire du carré 2

    return 0;
}

void testAire(Rectangle rectangle){
    rectangle.setLongueur( longueurEdit: 10);  //modifie la longueur
    cout << "Aire calculée " << rectangle.aire() <<endl;    //Affiche l'aire du rectangle
}
```

```
main
TP1
Variables  Debugger  Console
C:\Users\lempe\Desktop\Debian\UQAC\POO\TP1_Liskov1\cmake-build-debug\TP1
Aire calculée 50
Aire du carré2 : 100

Process finished with exit code 0
```

Nous testons d'abord un carré en appelant les fonctions de la classe mère. Le carré initial fait 5m de côté. Nous modifions la longueur a 10 avec une fonction de la classe mère puis nous calculons l'aire. L'aire d'un carré de 10m est 100m^2. Ici, nous trouvons 50. Le résultat de la fonction utilisée par la classe mère n'est pas le même que le résultat de la classe fille. Le principe de substitution de Liskov n'est pas respecté. En effet, les classes filles doivent donc être substituables à leurs classes mère. C'est-à-dire que le résultat doit être le même si la fonction est utilisée par la classe mère ou par la classe fille.

Code 2 respectant les principes SOLID : TP1_Liskov2

Pour appliquer le principe de Liskov, j'ai utilisé une interface. Ma classe forme est une classe mère des classes rectangle et carré. La classe rectangle n'a pas été changé. La classe carré a l'attribut size qui correspond à la taille du carré. Elle ne dépend donc plus de la classe rectangle. Le setter et le getter a été fait. La classe forme possède deux méthodes virtuelles : pour calculer l'aire, pour calculer le périmètre. Les classes rectangle et carré possèdent des méthodes override permettant de remplacer ses méthodes de la classe mère.

Carre.h :

```
#ifndef TP1_CARRE_H
#define TP1_CARRE_H

#include "Forme.h"
#include <iostream>
using namespace std;

class Carre: public Forme{
private:
    float size; //taille des côtés du carré en m
public:
    Carre();//Constructeur de la classe carré
    Carre(float sizeEdit);//Constructeur de la classe carré et initialisation à partir des valeurs fournies
    void setSize(float sizeEdit); //modification de la taille des côtés du carré en m
    float getSize();    //renvoie la valeur de la taille du carré en m
    float aire() override; //calcul du perimetre du carré, remplace la méthode de calcul du perimetre de la forme
    float perimetre() override;//calcul de l'aire du carré, remplace la méthode de calcul de l'aire de la forme
    void description() override;//affichage du perimetre et de l'aire du carré, remplace la méthode de calcul du perimetre de la forme
};


#endif //TP1_CARRE_H
```

Test du principe de Liskov :

```cpp
#include ...

using namespace std;

int main() {

    Forme* s[5]; //liste des différentes formes
    s[0] = new Carre( sizeEdit: 4);     //création d'un carré de côté 4m
    s[1] = new Rectangle( largeurEdit: 4, longueurEdit: 5);     //création d'un rectangle de largeur 4m et de longueur 5m
    s[2] = new Carre( sizeEdit: 5);     //création d'un carré de côté 5m
    s[3] = new Rectangle( largeurEdit: 2, longueurEdit: 3);     //création d'un rectangle de largeur 2m et de longueur 3m
    s[4] = new Carre( sizeEdit: 8);     //création d'un carré de côté 8m

    for (int i=0; i<5; ++i) {
        s[i]->description();    //affichage des perimetres et aires des formes contenues dans la liste
    }

    for (int i=0; i<5; ++i) {
        delete s[i];    //liberation de la mémoire
```

```
TP1
Le carré a un perimètre de 16 mètre.
Le carré a une aire de 16 mètre carré.
Le rectangle a un perimètre de 18 mètre.
Le rectangle a une aire de 20 mètre carré.
Le carré a un perimètre de 20 mètre.
Le carré a une aire de 25 mètre carré.
Le rectangle a un perimètre de 10 mètre.
Le rectangle a une aire de 6 mètre carré.
Le carré a un perimètre de 32 mètre.
Le carré a une aire de 64 mètre carré.
```

On peut voir que nous avons initialisé un tableau de forme. Ensuite les formes ont été réparties entre rectangle et carré. L'appel des méthodes descriptions de forme a bien donné le même résultat que les méthodes carré ou rectangle. Le principe de Liskov est donc respecté.

Code 3 ne respectant pas les principes SOLID : TP1_OpenClose1

Le code ne respecte pas le deuxième principe de SOLID, c'est-à-dire le principe ouvert/fermé.

La classe voiture représente une voiture qui veut être vendue. Il faut donc donner les informations sur la voiture à l'acheteur.

Voiture.h :

```h
#ifndef TP1_VOITURE_H
#define TP1_VOITURE_H

#include <iostream>

class Voiture {
private:
    int annee;  //année de production de la voiture
    int prix;   //prix demandé par le vendeur
    float kilometrage;  //nombre de kilometrage de la voiture
    float liquideReservoir;  //carburant dans le reservoir (pas pour voiture electrique)
public : enum typeVoiture{Essence, Electrique, Diesel};  //Differents types possibles du véhicule
public : enum aspectVoiture{Mauvais, Moyen, Bon, Excellent}; //Differents aspects possible du véhicule
private:
    typeVoiture type;//Type du véhicule
    aspectVoiture aspect;//Note de l'aspect du véhicule
public:
    Voiture(int anneeP, int prixA, float km, typeVoiture typeV,aspectVoiture aspectV);  //Constructeur de la classe Voiture, on a besoin de toutes les informations sauf l
    int getAnnee(); //renvoie l'année de production de la voiture à vendre
    int getPrix();//renvoie le prix attendu par le vendeur
    float getKilometrage(); //renvoie le nombre de kms qu'a effectué la voiture
    typeVoiture getType();  //donne le type du véhicule
    aspectVoiture getAspect();  //donne l'aspect du vehicule
    float getLquideReservoir();     //renvoie le carburant present dans la voiture en ce moment (peut evoluer entre la mise en vente et l'achat)
    void setLiquidReservoir(float liquide);     //permet de modifier le carburant disponible, voitures electriques ne possedent pas de carburant;
};
#endif //TP1_VOITURE_H
```

On peut voir qu'en ajoutant un type de véhicule, on est obligé de modifier cette classe. Par exemple, le type hybride devrait être rajouté. Le principe Open/Close dit que la classe est ouverte à l'extension et fermée à la modification. Or nous sommes obligés de la modifier pour ajouter la classe hybride. De plus si nous voulons ajouter le type Hydrogène qui, comme le type électrique, ne consomme pas de carburant, nous sommes obligés de modifier le code source de la classe Voiture.

Code 4 respectant les principes SOLID : TP1_OpenClose2

Pour que la classe respecte le principe ouvert/fermé, j'ai utilisé la programmation objet. En effet, on peut rendre la classe Voiture, classe mère des types de voiture. Les voitures diesel et essence utilisent du fuel pour fonctionner donc on crée une classe les regroupant. Pour la voiture électrique, on ajoute seulement une classe fille à la classe Voiture. Si nous voulons rajouter le type hydrogène, nous pouvons créer une nouvelle classe qui sera la fille de la classe Voiture. Le principe de l'open/close est respecté, lorsqu'on ajoute de nouveaux types de véhicules, on crée une nouvelle classe qui dépend de la classe Voiture, mais jamais nous n'avons besoin de modifier la classe Voiture. On a même pu ajouter des attributs de la classe électrique pour préciser la valeur de l'autonomie de la batterie.

Voiture.h :

```h
#ifndef TP1_VOITURE_H
#define TP1_VOITURE_H

#include <iostream>

class Voiture {
protected:
    int annee;  //année de production de la voiture
    int prix;   //prix demandé par le vendeur
    float kilometrage;  //nombre de kilometrage de la voiture
public : enum aspectVoiture{Mauvais, Moyen, Bon, Excellent}; //Differents aspects possible du véhicule
protected:
    aspectVoiture aspect;//Note de l'aspect du véhicule
public:
    Voiture(int anneeP, int prixA, float km, aspectVoiture aspectV);  //Constructeur de la classe Voiture, on a beso
    int getAnnee(); //renvoie l'année de production de la voiture à vendre
    int getPrix();//renvoie le prix attendu par le vendeur
    float getKilometrage(); //renvoie le nombre de kms qu'a effectué la voiture
    aspectVoiture getAspect();  //donne l'aspect du vehicule
};
#endif //TP1_VOITURE_H
```

Electrique.h :

```h
#ifndef TP1_ELECTRIQUE_H
#define TP1_ELECTRIQUE_H
#include "voiture.h"

class Electrique : public Voiture {
private:
    int capaciteBatterie;   //batterie restante dans la voiture (autonomie)
public:
    Electrique(int anneeP1, int prixA1, float km1, aspectVoiture aspectV1, int capacite);   //constructeur de la classe Electrique qui utilise l
    void setCapacite(int capacite); //permet de modifier la capacité disponible (autonomie restante) (peut evoluer entre la mise en vente et l'a
    int getCapacite(); //renvoie la capacité/autonomie presente dans la voiture en ce moment (peut evoluer entre la mise en vente et l'achat)
};


#endif //TP1_ELECTRIQUE_H
```

Fuel.h :

```h
#ifndef TP1_FUEL_H
#define TP1_FUEL_H

#include "voiture.h"

class Fuel: public Voiture {
private:
    int liquideReservoir;   //carburant dans le reservoir
public:
    Fuel(int anneeP, int prixA, float km, Voiture::aspectVoiture aspectV, int liquide);   //constructeur de la classe Fuel qui
    void setLiquide(int liquide); //permet de modifier le carburant disponible (peut evoluer entre la mise en vente et l'achat)
    int getLiquide();//renvoie le carburant present dans la voiture en ce moment (peut evoluer entre la mise en vente et l'acha
};

#endif //TP1_FUEL_H
```

Comme nous l'avons vu, nous pouvons toujours ajouté de nouveaux types de véhicules en ne modifiant pas les classes existantes. Le principe ouvert/fermé est donc respecté.

Conclusion

Nous avons pu voir les différents principes SOLID :

- Responsabilité unique (Single responsibility principle)
- Ouvert/fermé (Open/closed principle)
- Substitution de Liskov (Liskov substitution principle)
- Ségrégation des interfaces (Interface segregation principle)
- Inversion des dépendances (Dependency inversion principle)

Nous avons également pu voir la façon dont on les implémente, les avantages de chacun…

Les principes SOLID permettent d'améliorer le code qui devient plus lisible, facile à maintenir, extensible, réutilisable et sans répétition. Ces principes s'appliquent en programmation objet avancé.