Cours - Patrons de conception - #1

Principes SOLID, Patron Singleton.

Guillaume Santini

17 janvier 2024

Plan

Programmation orientée objet (POO)

Les concepts de base de la POO

Classes/objets Les classes sont des modèles pour créer des objets qui communiquent entre eux par messages (appels de méthodes).

Encapsulation On cache (`private`) la structure d'un objet et on ne révèle (`public`) que les fonctions (méthodes) nécessaires.

Héritage Une classe peut hériter des données et des méthodes d'autres classes.

Polymorphisme (d'héritage) Une méthode de même signature peut avoir des comportements (instructions) différents selon la classe où elle est (re)définie.

Programmation orientée objet (POO)

Intérêts des concepts objets de base

Classes/objets → Réutilisation Les notions de classe et d'héritage permettent de réutiliser de données et du code dans différents contextes.

Encapsulation → Sécurité L'encapsulation permet de protéger un objet contre des modifications inappropriées.

Héritage → Factorisation L'héritage permet de factoriser des données et instructions.

Polymorphisme → Souplesse Le polymorphisme permet d'utiliser un même code sur des objets de différentes classes.

La qualité d'un développement dépend de bonnes pratiques

Comment offrir les garanties suivantes ?

- code facilement maintenable,
- code aisément extensible,
- code robuste,
- code fiable.

Respecter les bonnes pratiques

- en produisant un jeu de tests,
- en produisant un code commenté,
- en produisant un code précisément documenté pour la vue publique,
- en respectant les principes d'encapsulation, de couplage faible.

Les principes SOLID

Single responsibility principle

Open/Closed principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

Les principes SOLID

Single responsibility principle

Principe	Une classe doit avoir une et une seule responsabilité,
Objectif(s)	Maintenir un couplage faible, et décorréler les fonctionnalités indépendantes, gagner en robustesse.
Mise en œuvre	Faire des petites classes, utiliser des abstractions.

Open/Closed principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

Les principes SOLID

Single responsibility principle

Open/Closed principle

Principe	Une classe doit être ouverte aux extensions mais fermée à la modification. *i.e.* Durant son cycle de vie, une classe validée et testée ne doit plus jamais être modifiée (**Fermée**). La modification de son comportement ne doit se faire que par extension (**Ouverture**).
Objectif(s)	Gagner en maintenabilité et éviter d'avoir à refaire tous les tests.
Mise en œuvre	Utiliser des abstractions, pratiquer le polymorphisme de redéfinition.

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

Les principes SOLID

Single responsibility principle

Open/Closed principle

Liskov substitution principle

Principe	Une instance de type A doit pouvoir être remplacée par une instance de type B, telle que B sous-classe de A, sans que cela ne modifie la cohérence du programme.
Objectif(s)	Robustesse.
Mise en œuvre	Les préconditions ne peuvent pas être renforcées dans une sous-classe. Les postconditions ne peuvent pas être affaiblies dans une sous-classe. Des exceptions d'un type nouveau ne peuvent pas être levées par des méthodes de la sous-classe, sauf si elles sont des sous-types des exceptions lancées par la superclasse.

Interface segregation principle

Dependency inversion principle

Les principes SOLID

Single responsibility principle

Open/Closed principle

Liskov substitution principle

Interface segregation principle

Principe	Préférer le développement de plusieurs interfaces spécifiques adaptées aux différents besoins des clients plutôt qu'une seule interface générale.
Objectif(s)	Maintenir un couplage faible et faciliter d'éventuelles refactorisations en évitant qu'une classe cliente ne dépende de méthodes dont elle n'a pas besoin.
Mise en œuvre	Utiliser les abstractions et les interfaces.

Dependency inversion principle

Les principes SOLID

Single responsibility principle

Open/Closed principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

Principe	Dépendre des abstractions, pas des implémentations.
Objectif(s)	Maintient d'un couplage faible
Mise en œuvre	Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions. Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

Open/Closed principle

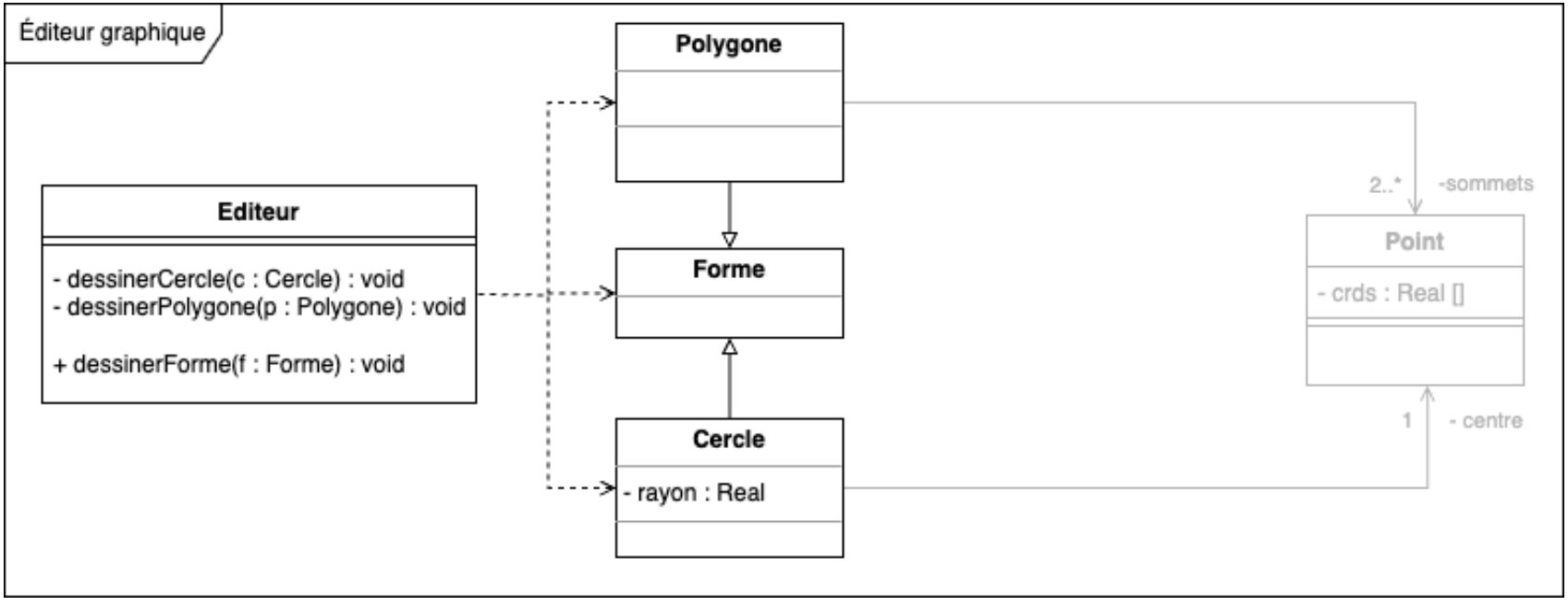
Une classe doit être ouverte aux extensions (ajouts) mais fermée aux modifications (du code existant) :

Ouverture aux extensions : le comportement d'un module doit pouvoir être étendu (*e.g.* ajout de nouvelles méthodes),

Fermeture aux modifications : L'extension du comportement doit pouvoir se faire sans modification du code existant (*e.g.* mise en place d'interfaces, utilisation du polymorphisme, ...).

$\Rightarrow$ L'ajout de fonctionnalités doit se faire en ajoutant le code et non en éditant du code existant.

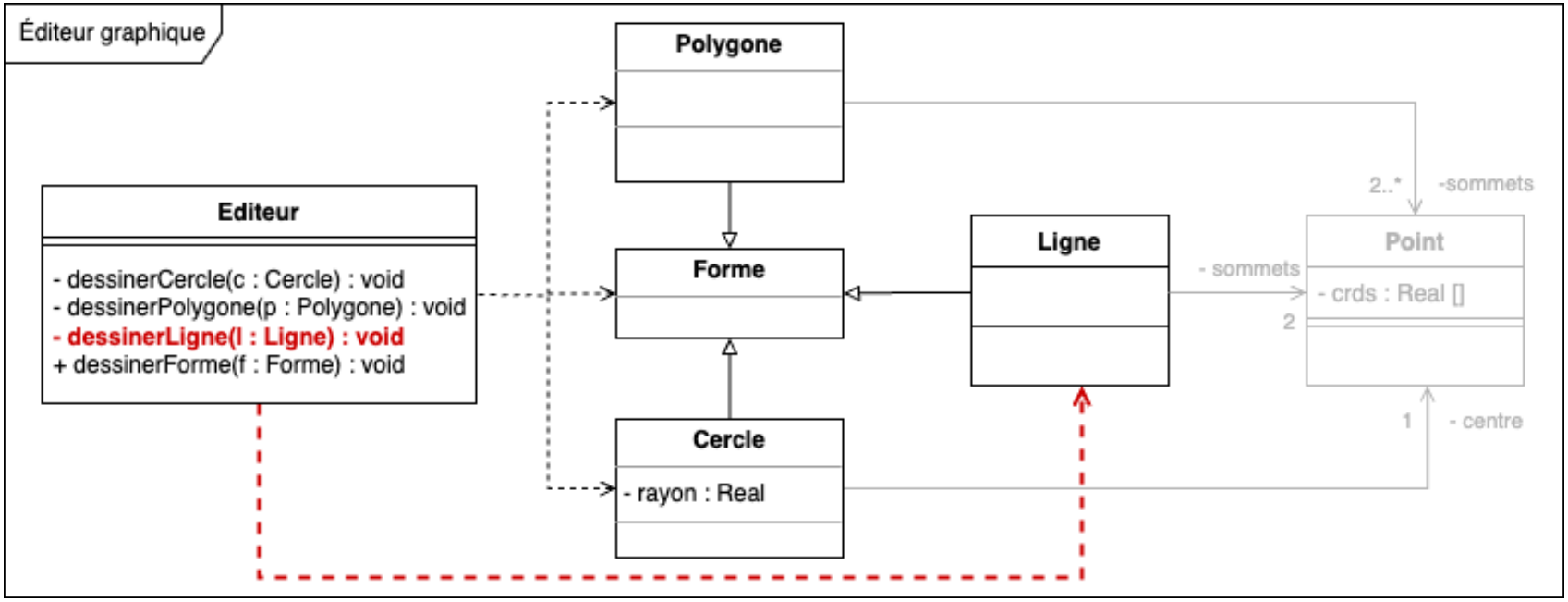
Open/Closed principle



Couplage fort des classes

La classe `Editeur` dépend des classes `Forme`, `Polygone` et `Cercle`.

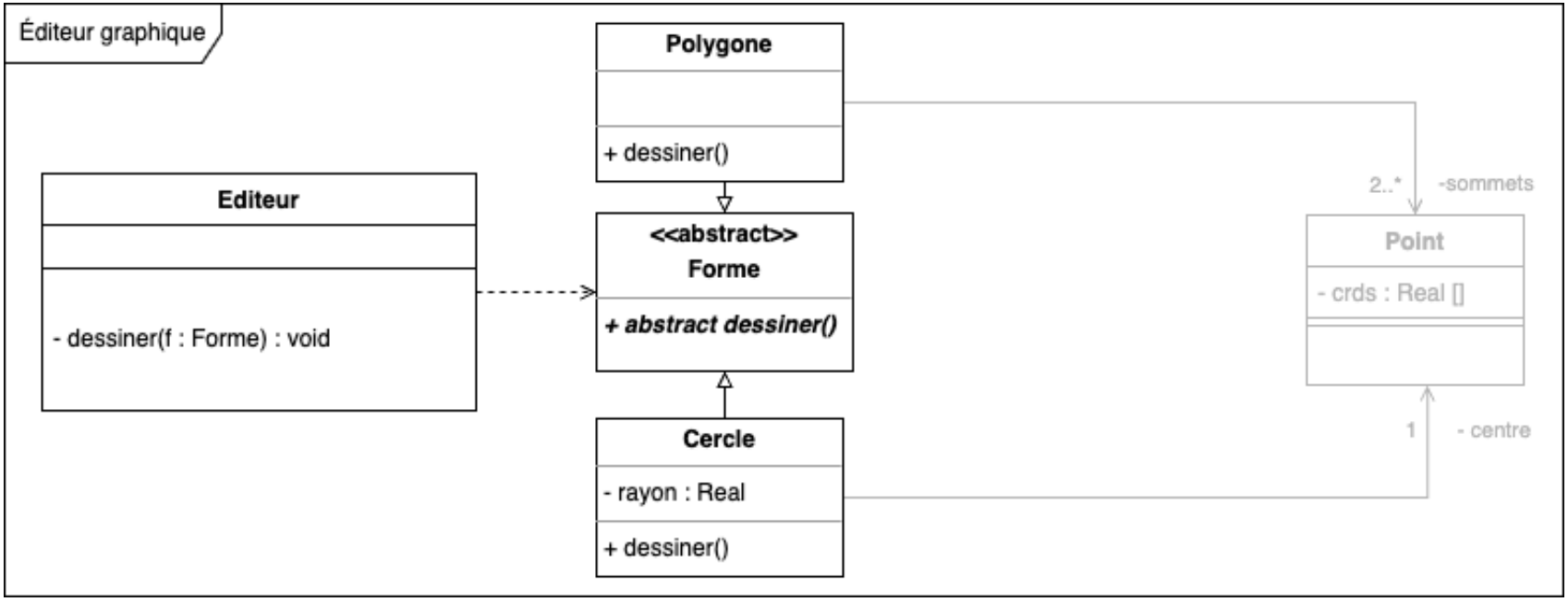
Open/Closed principle



Principe Open/Close non respecté

Ajout d'un nouveau type de forme (*e.g.* classe `Ligne`) contraint l'ouverture et la modification de la classe `Editeur` (*e.g.* ajout d'une méthode `dessinerLigne( l:Ligne) : void`).

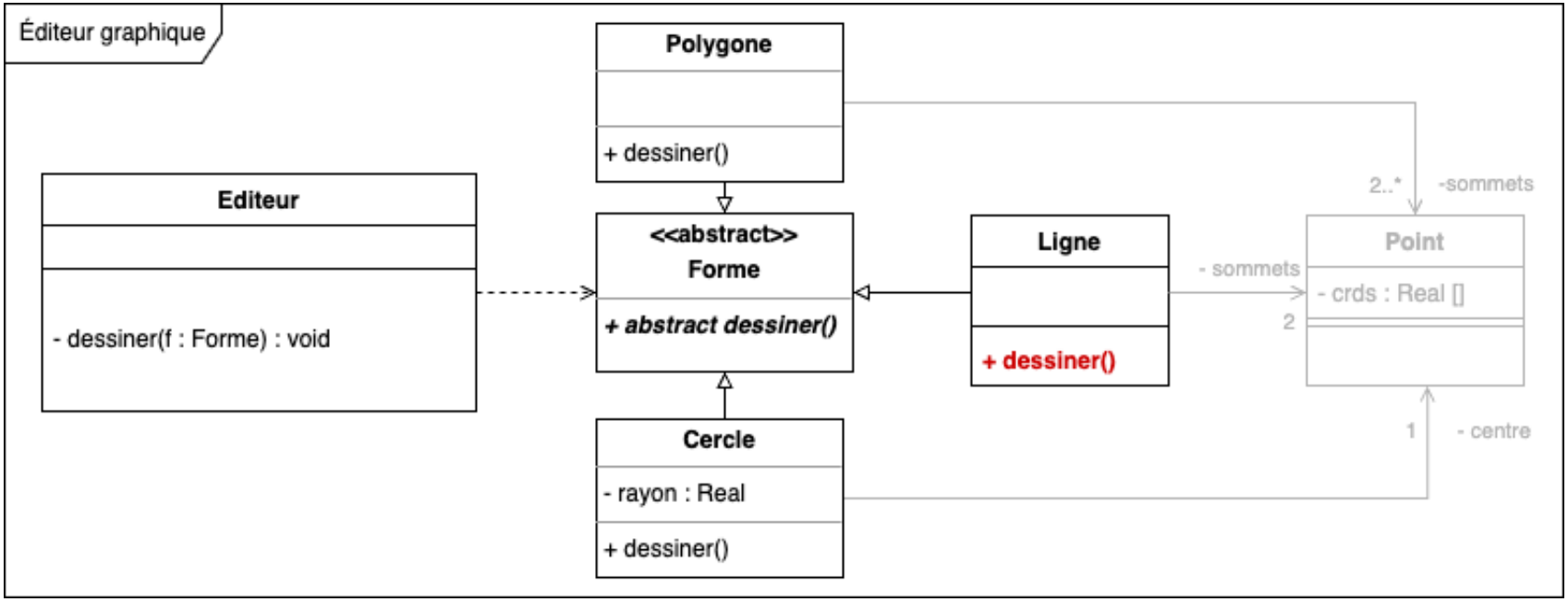
Open/Closed principle



Couplage faible

La classe `Editeur` ne dépend que de la classe abstraite `Forme` (sans implémentation).

Open/Closed principle



Principe Open/Close respecté

Ajout d'un nouveau type de forme (*e.g.* classe `Ligne`) ne contraint pas à la modification de la classe `Editeur` (*e.g.* redéfinition de la méthode `dessiner()` dans la classe `Ligne`).

Liskov substitution principle

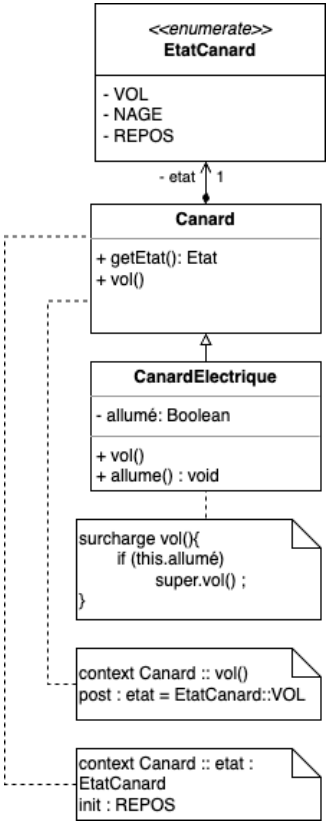
Une instance de type A doit pouvoir être remplacée par une instance de type B, telle que B sous-classe de A, sans que cela ne modifie la cohérence du programme.

⇒ Une instance d'une classe doit pouvoir être substituée sans modification par une instance d'une sous-classe (et sans que le programme ne soit altéré dans son comportement).

Ne pas introduire de modificatione dans le fonctionnement de B qui rendent inutilisable tout objet de type B utilisé comme un objet de type A.

⇒ B ne doit pas introduire de lever d'exception qui ne seraient pas des sous-classes des exceptions levées par la classe A.

Liskov substitution principle

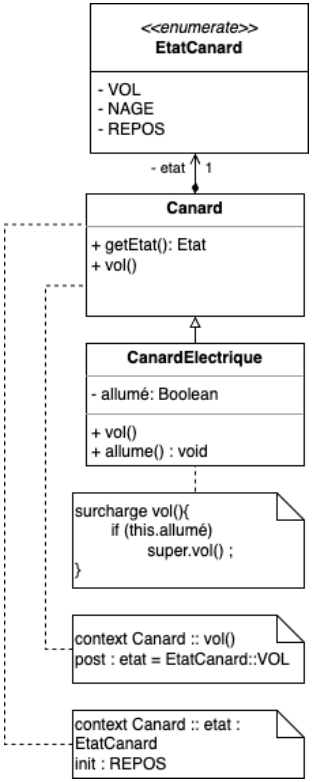



```
public class Test {                                                    1
  public void test( List<Canard> maListe ){                           2
    int nbCanardsVolants = 0 ;                                         3
    for ( Canard c : maListe ){ // on fait voler les canards           4
      c.vol();                                                         5
      if (c.getEtat() == EtatCanard.Vol)                               6
        nbCanardsVolants ++ ;                                          7
    }                                                                  8
    // on verifie qu'ils volent tous                                   9
    if ( nbCanardsVolants != this.maList.size() )                      10
      throw new Exception("Des canards ne volent pas");                11
  }                                                                    12
  public static void main( String[] args ){                            13
    List<Canard> maListe = new ArrayList<Canard>();                    14
    maListe.add( new Canard());                                        15
    maListe.add( new Canard());                                        16
    this.test( maListe );          // ---> OK                          17
    maListe.add( new CanardElectrique() ) ;                            18
    this.test( maListe) ;          // Exception : le canard            19
                                   // electrique ne vole pas           20
                                   // car il n'est pas                 21
allume} }
```

Principe de Liskov non respecté

La méthode `test()` qui utilise des instances de la classe `Canard` doit pouvoir utiliser des instances de la classe dérivée `CanardElectrique` sans que le programme ne soit altéré dans son comportement.

Liskov substitution principle




```
public class Test {                                                    1
  public static void test( List<Canard> maListe ){                    2
    int nbCanardsVolants = 0 ;                                         3
    for ( Canard c : maListe ){ // on fait voler les canards           4
      if ( c instanceof CanardElectrique )                             5
        ((CanardElectrique) c).allume();                               6
      c.vol();                                                         7
      if (c.getEtat() == EtatCanard.Vol)                               8
        nbCanardsVolants ++ ;                                          9
    }                                                                  10
    // on verifie qu'ils volent tous                                   11
    if ( nbCanardsVolants != this.maList.size() )                      12
      throw new Exception("Des canards ne volent pas");                13
  }                                                                    14
  public static void main( String[] args ){                            15
    List<Canard> maListe = new ArrayList<Canard>();                    16
    maListe.add( new Canard()); maListe.add( new Canard());            17
    Test.test( maListe );       // ---> OK                             18
    maListe.add( new CanardElectrique() ) ;                            19
    Test.test( maListe) ;       // Plus d'Exception                    20
} }                                                                    21
```

Principe de Liskov respecté

Principe Open/Closed non respecté : module non-fermé aux modifications

La méthode `test()` a besoin de connaître le type réel des objets pour pouvoir les traiter correctement.

Les patrons de conception

Fonction des patrons de conception

- Les design patterns (patrons de conception) décrivent des procédés généraux et réutilisables pour concevoir des logiciels,
- Les design patterns mettent en œuvre les principes SOLID,
- Un design pattern décrit une bonne pratique et une solution standard en réponse à un problème de conception d'un logiciel,
- Les design patterns doivent être adaptés selon les besoins.

Les patrons de conception

Classification des patrons de conception

Création (créer des objets)

- Fabrique, Fabrique abstraite, Prototype, **Singleton**, Monteur,

Structure (organiser les classes)

- **Décorateur,** Adapteur, Façade, Pont, **Composite**, Poids-mouche, Proxy

Comportement (organiser la collaboration entre objets)

- **Observateur**, **Stratégie**, Commande, Médiateur, Chaîne de responsabilité, Iterateur, Interpréteur, Etat, Patron de méthode, Visiteur, Fonction de rappel

***En gras** les patrons de conception abordés dans ce cours.

Le patron de conception **Singleton**

Motivations

- Assurer l'unicité d'un objet ou d'une ressource (cache, boite de dialogue, paramètres de préférences, objets de journalisation ...) pour éviter des conflits, des pertes de cohérence, ...
- Offrir un point d'accès global à ces ressources pour tout le système.
- Ne créer ces objets (qui peuvent consommer des ressources mémoire ou de calcul) que lorsque l'on en a besoin.

Le patron de conception **Singleton**

Motivations

- Assurer l'unicité d'un objet ou d'une ressource (cache, boite de dialogue, paramètres de préférences, objets de journalisation ...) pour éviter des conflits, des pertes de cohérence, ...
- Offrir un point d'accès global à ces ressources pour tout le système.
- Ne créer ces objets (qui peuvent consommer des ressources mémoire ou de calcul) que lorsque l'on en a besoin.

Utiliser une variable globale

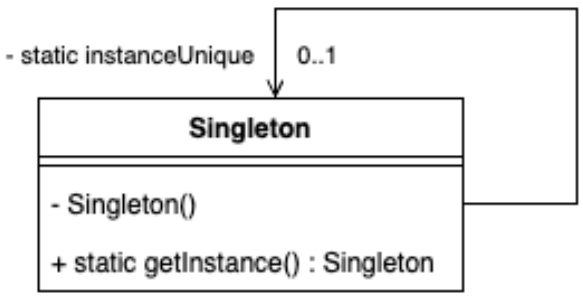
Moyennant une convention et une rigueur d'usage exclusif de cette variable dans le code, cela peut

- garantir l'unicité d'un objet,
- offrir un point d'accès global.

Par contre cela implique

- de créer et d'initialiser cette ressource dès le lancement de l'application,
- d'immobiliser des ressources même si vous n'avez pas besoin de celles-ci ...

Le patron de conception **Singleton**

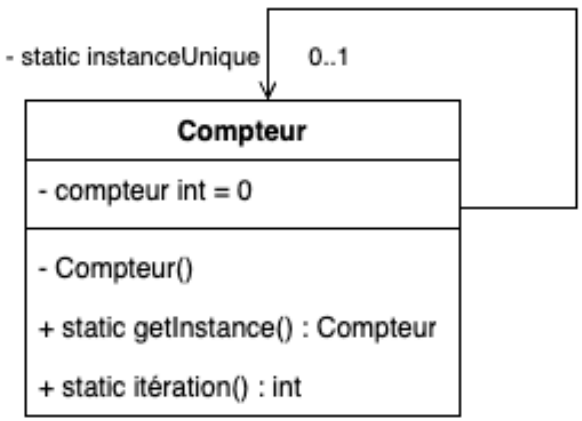



```
public class Singelton {
  private static Singleton instanceUnique ;

  private Singleton() {
    // initialisation
  }

  public static Singleton getInstance() {
    if ( Singleton.instanceUnique == null )
      Singleton.instanceUnique = new Singelton();
    return Singleton.instanceUnique;
  }
}
```

Analyse du patron

- Le constructeur est privé !
- `getInstance()` est une méthode publique de classe
  $\Rightarrow$ elle peut être invoquée alors qu'il n'y a pas d'instance,
- `getInstance()` instancie l'objet `: Singleton` et retourne la référence de l'instance unique dans tous les cas.

Adaptation de Singleton : un compteur unique




```
public class Compteur {
  private static Compteur instanceUnique ;
  private int cpt ;

  private Compteur() { this.cpt = 0 ; }

  public static Compteur getInstance() {
    if ( Compteur.instanceUnique == null )
      Compteur.instanceUnique = new Compteur();
    return Compteur.instanceUnique;
  }

  public int iteration() { return this.cpt ++ ;}
}

public class TestCompteur {
  public static void main( String [] args) {
    Compteur it1 = Compteur.getInstance();
    System.out.println(it1.iteration()) ;
    System.out.println(it1.iteration()) ;
    Compteur it2 = Compteur.getInstance() ;
    System.out.println(it2.iteration()) ;
    System.out.println(it1.iteration()) ;
  }
}
```

L'exécution de la classe de `TestCompteur` produit l'affichage des entiers de 0 à 4 car une seule instance de compteur est manipulée.

Le compteur est global et unique.

Credits

- Support soumis à copyleft : 
- Le cours de D. Bouthinon,
- **Design patterns — Tête la première**, E. & E. Freeman, ed. O'Reilly.