Génie logiciel - Patrons de conception

Nuwan Herath

2022-2023

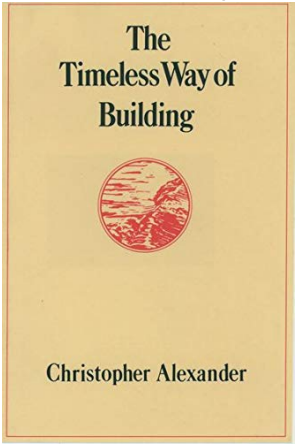
Introduction

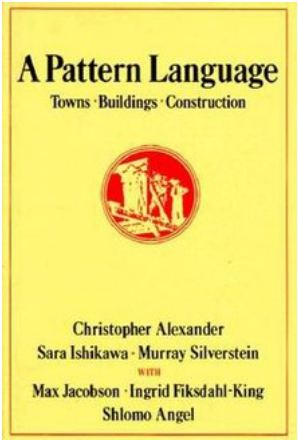
Pourquoi étudier les patrons de conception ?

Exploiter l'expérience et les leçons tirées par d'autres développeurs qui ont déjà suivi le même chemin, rencontré les mêmes problèmes de conception et survécu au voyage
Au lieu de **réutiliser du code**, les patrons permettent de **réutiliser de l'expérience**

L'origine des patrons

Idée de l'architecte Christopher Alexander





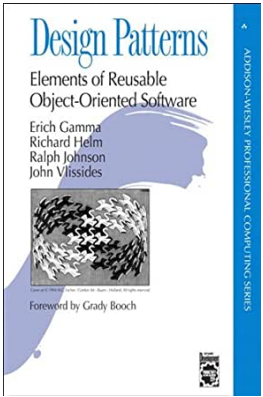
La bande des quatre

Le *Gang of Four* (GoF) :

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides

Ils sont à l'origine du concept pour le développement logiciel, grâce à leur livre en 1995

Références


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


Un guide d'apprentissage
Tête la première
Design Patterns

Les bases de la programmation orientée objet

Abstraction	*à partir d'un problème, extraction des variables pertinentes pour construire un modèle informatique*
Encapsulation	*regroupement de données et de méthodes en des structures (objet, classe)*
Polymorphisme	*multiple utilisation d'un opérateur*
Héritage	*création de classes à partir de classes existantes*

Les principes de la programmation orientée objet

- Encapsuler ce qui varie
- Préférer la composition/l'encapsulation à l'héritage
- Programmer des interfaces, non des implémentations
- Coupler faiblement les objets qui interagissent
- Les classes doivent être ouvertes à l'extension, mais fermées à la modification
- Dépendre des abstractions, mais ne pas dépendre des classes concrètes
- Ne parler qu'a ses amis
- Une classe ne doit avoir qu'une seule raison de changer

L'essentiel des principes

S'ils ne fallait retenir que trois choses...

- Encapsuler ce qui varie
- Préférer la composition à l'héritage
- Utiliser les interfaces

Les patrons de la classification GoF

		Objectif		
		De construction	Structuraux	Comportementaux
Portée	Classe	Fabrique	Adaptateur	Interpréteur Patron de méthode
	Objet	Fabrique abstraite Monteur Prototype Singleton	Adaptateur Pont Composite Décorateur Façade Poids-mouche Procuration	Chaîne de responsabilité Commande Itérateur Médiateur Mémento Observateur Etat Stratégie Visiteur

Qu'est-ce que les patrons de conception ?

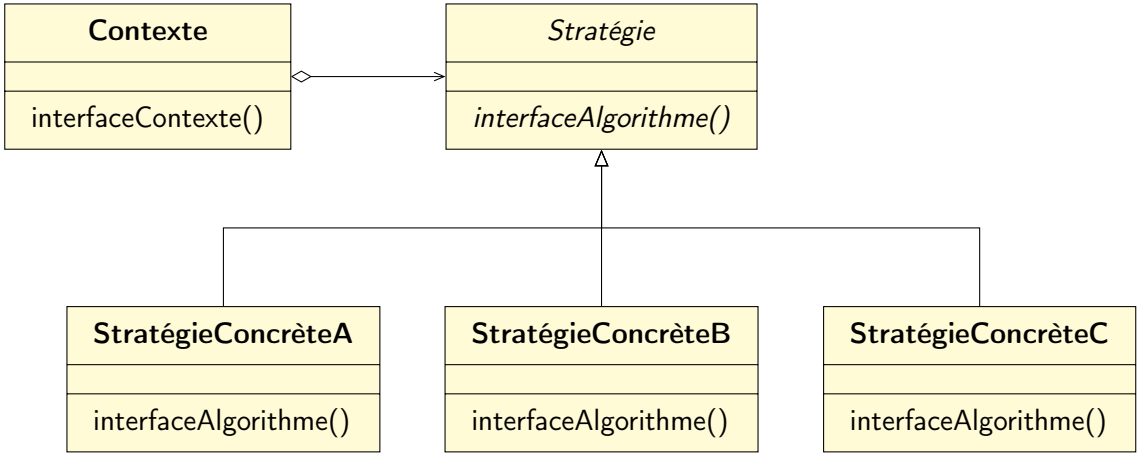
Soyons patient...
Ils sont nombreux, donc étudions les par l'exemple

Etude des patrons

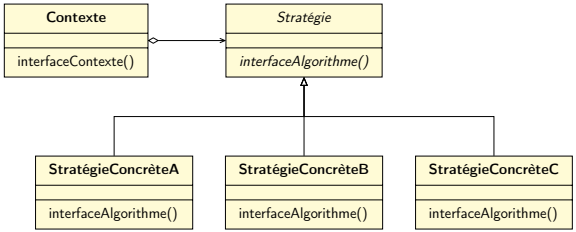
Etude des patrons

Stratégie

Diagramme de classe



Principes mis en œuvre



- Encapsuler ce qui varie
- Programmer des interfaces, non des implémentations
- Préférer la composition à l'héritage

Interface ?

"interface" = "supertype"
"interface" ≠ `interface` (en Java)

L'idée est d'exploiter le polymorphisme en programmant un supertype pour que l'objet réel à l'exécution ne soit pas enfermé dans le code
Autrement dit, le type déclaré des variables doit être un supertype, généralement une interface ou une classe abstraite
Ainsi, les objets affectés à ces variables peuvent être n'importe quelle implémentation concrète du supertype, ce qui signifie que la classe qui les déclare n'a pas besoin de savoir quels sont les types des objets réels !

Zoom sur les principes

Encapsuler ce qui varie

Extraire les parties variables et les encapsuler permettra plus tard de les modifier ou de les augmenter sans affecter celles qui ne varient pas

Programmer des interfaces, non des implémentations

Utiliser une interface cache l'implémentation réelle

Préférer la composition à l'héritage

La composition permettra de changer d'implémentation au moment de l'exécution

Application réelle

Exemple

Au football, lorsqu'on arrive vers la fin de la rencontre, si l'équipe A mène l'équipe B au score avec 1-0, au lieu d'attaquer, l'équipe A se met à défendre.

"Deschamps, c'est pas le Joga Bonito mais tant que ça gagne ça va pour moi
Le football c'est aussi de la tactique, allez dire ça à Hazard et Courtois"

— Flynt, *Champions du monde*

Application informatique

Exemple

Lorsqu'une première mémoire est remplie, on se met à stocker dans la prochaine mémoire accessible.
Donc un test est nécessaire à l'exécution avant le stockage de données et on s'adapte en fonction.

Etude des patrons

Décorateur

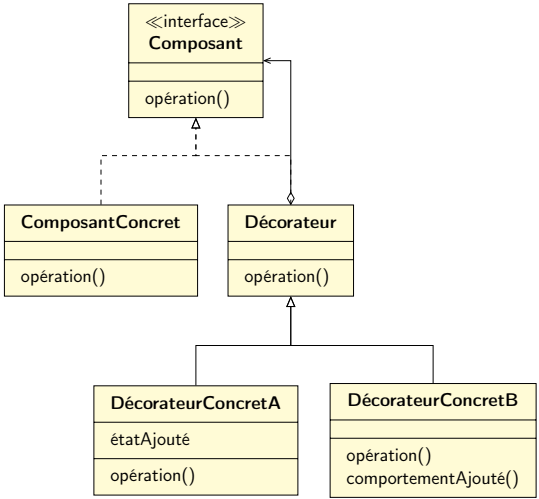
Définition

Definition

Le patron Décorateur attache dynamiquement des responsabilités supplémentaires à un objet
Il fournit une alternative souple à la dérivation, pour étendre les fonctionnalités

Il permet d'étendre facilement les classes pour incorporer de nouveaux comportements sans modifier le code existant
Il produit des conceptions résistantes au changement et suffisamment souples pour accepter de nouvelles fonctionnalités répondant à l'évolution des besoins
$\rightarrow$ le rêve de tout développeur soumis aux besoins changeant des utilisateurs

Diagramme de classe



Le principe satisfait par le patron Décorateur

Les classes doivent être ouvertes à l'extension, mais fermées à la modification

En composant dynamiquement des objets, on peut ajouter de nouvelles fonctionnalités en écrivant du code au lieu de modifier le code existant; les risques d'introduire des bogues ou de provoquer des effets de bord inattendus sont significativement réduits

Etude des patrons

Les fabriques

Eviter new

Programmer des interfaces, non des implémentations

```
Canard canard = new
   Colvert();
```

```
Canard canard;
if (dansLaMare) {
  canard = new Colvert();
} else if (aLaChasse) {
  canard = new Leurre();
} else if (dansLaBaignoire) {
  canard = new
     CanardEnPlastique();
}
```

- duplication des conditions dans le code
- maintenance et mises à jour difficiles

Pizzeria

```
Pizza commanderPizza() {
  Pizza pizza = new Pizza();

  pizza.preparer();
  pizza.cuire();
  pizza.couper();
  pizza.emballer();

  return pizza;
}
```

```
Pizza commanderPizza(String type) {
  Pizza pizza;

  if (type.equals("fromage")) {
    pizza = new PizzaFromage();
  } else if (type.equals("grecque") {
    pizza = new PizzaGrecque();
  } else if (type.equals("poivrons") {
    pizza = new PizzaPoivrons();
  }

  pizza.preparer();
  pizza.cuire();
  pizza.couper();
  pizza.emballer();

  return pizza;
}
```

La pression du marché

L'ennemi du développeur : le changement

- La pizza grècque ne se vend pas bien
- On souhaite ajouter une pizza aux fruits de mer et une végétarienne

```
Pizza commanderPizza(String type) {
  Pizza pizza;

  if (type.equals("fromage")) {
    pizza = new PizzaFromage();
  // } else if (type.equals("grecque") {
    // pizza = new PizzaGrecque();
  } else if (type.equals("poivrons") {
    pizza = new PizzaPoivrons();
  } else if (type.equals("fruitsDeMer")) {
    pizza = new PizzaFruitsDeMer();
  } else if (type.equals("vegetarienne")) {
    pizza = new PizzaVegetarienne();
  }

  pizza.preparer();
  pizza.cuire();
  pizza.couper();
  pizza.emballer();
}
```

Rappel de deux principes de la POO

- Les classes doivent être ouvertes à l'extension, mais fermées à la modification
- Encapsuler ce qui varie

Extraire ce qui varie et l'encapsuler

```
public class SimpleFabriqueDePizzas {
  public Pizza creerPizza(String type) {
    Pizza pizza;

    if (type.equals("fromage")) {
    pizza = new PizzaFromage();
    } else if (type.equals("poivrons")) {
    pizza = new PizzaPoivrons();
    } else if (type.equals("fruitsDeMer")) {
    pizza = new PizzaFruitsDeMer();
    } else if (type.equals("vegetarienne")) {
    pizza = new PizzaVegetarienne();
    }
    return pizza;
  }
}
```

Questions **pas** bêtes

N'a-t-on pas juste transférer le problème ? D'autres classes peuvent avoir besoin d'une fabrique, tout étant maintenant à un endroit, les modifications sont plus simples

Ne peut-on pas définir une méthode statique pour la fabrique ? Oui, c'est possible, mais alors on ne pourra pas sous-classer

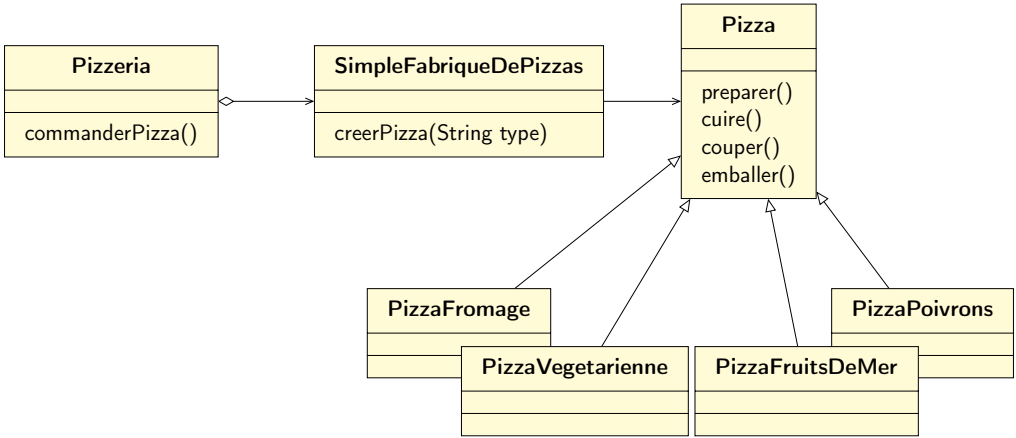
Retour à la pizzeria

Grâce à la composition, l'opérateur `new` a été remplacé par une méthode de création

```
public class Pizzeria {
  SimpleFabriqueDePizzas fabrique;

  public Pizzeria(SimpleFabriqueDePizzas fabrique) {
    this.fabrique = fabrique;
  }

  public Pizza commanderPizza(String type) {
    Pizza pizza;

    pizza = fabrique.creerPizza(type);

    pizza.preparer();
    pizza.cuire();
    pizza.couper();
    pizza.emballer();
    return pizza;
  }
  // ...
}
```

Vue d'ensemble

La brique de base des fabriques



Des franchises

La pizerria est une telle réussite qu'apparaissent des filiales : une à Brest avec une pâte fine et une à Strasbourg avec une pâte épaisse et beaucoup de sauce

Des franchises

La pizerria est une telle réussite qu'apparaissent des filiales : une à Brest avec une pâte fine et une à Strasbourg avec une pâte épaisse et beaucoup de sauce
Une approche

```
FabriqueDePizzasBrest fabriqueBrest = new FabriqueDePizzasBrest();
Pizzeria boutiqueBrest = new Pizzeria(fabriqueBrest);
boutiqueBrest.commanderPizza("vegetarienne");

FabriqueDePizzasStrasbourg fabriqueStrasbourg = new FabriqueDePizzasStrasbourg();
Pizzeria boutiqueStrasbourg = new Pizzeria(fabriqueStrasbourg);
boutiqueStrasbourg.commanderPizza("vegetarienne");
```

Trop de liberté

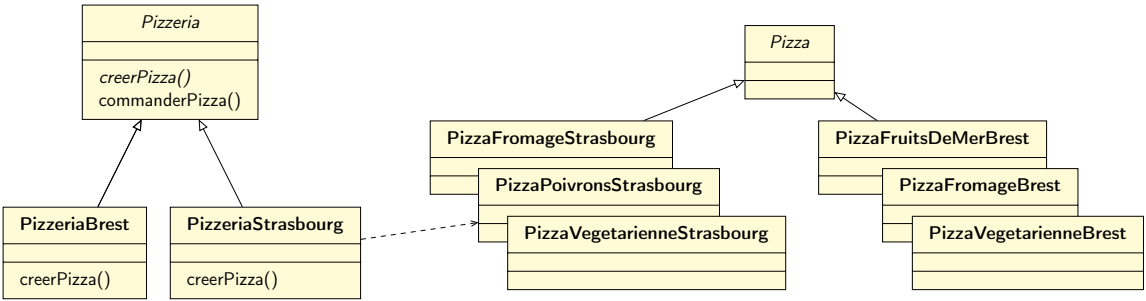
Les franchises commencent à utiliser leurs propres procédures : modes de cuisson différents, oubli du découpage, achat de boîtes différentes

Equilibre contrôle / liberté

```
public abstract class Pizzeria {
  public Pizza commanderPizza(String type) {
    Pizza pizza;

    pizza = creerPizza(type);
    pizza.preparer();
    pizza.cuire();
    pizza.couper();
    pizza.emballer();

    return pizza;
  }

  protected abstract Pizza creerPizza(String type);
}
```

- La pizzeria devient abstraite
- `creerPizza()` redevient un appel à une méthode de `Pizzeria`
- Cette méthode de fabrication est maintenant abstraite dans `Pizzeria`

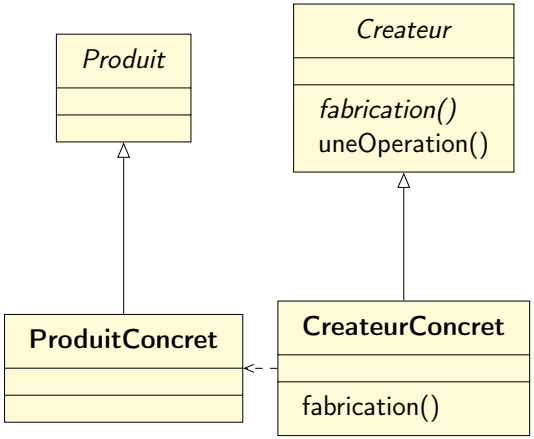
Le patron Fabrication appliqué à la pizzeria



Définition

Definition

Le patron Fabrication définit une interface pour la création d'un objet, mais en laissant aux sous-classes le choix des classes à instancier
Fabrication permet à une classe de déléguer l'instanciation à des sous-classes



Un nouveau principe

Dépendre d'abstractions, ne pas dépendre de classes concrètes

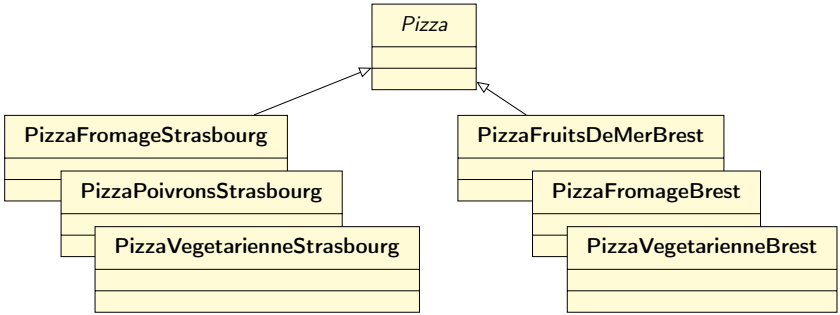
De la Fabrication à la Fabrique abstraite

Les franchises appliquent bien les procédures, mais elles ont des spécificités régionales

- pâte épaisse vs pâte fine
- sauce tomate cerise vs sauce marinara
- mozzarella vs parmesan
- moules surgelées vs moules fraîches

Des pizzas de différents styles

Plutôt que d'avoir deux pizzas par types



gérons les spécificités régionales avec une fabrique

Des fabriques d'ingrédients

```java
public interface
    FabriqueIngredientsPizza {
  public Pate creerPate();
  public Sauce creerSauce();
  public Fromage creerFromage();
  public Legumes[]
      creerLegumes();
  public Poivrons
      creerPoivrons();
  public Moules creerMoules();
}
```

```java
public class FabriqueIngredientsPizzaBrest
      implements FabriqueIngredientsPizza {
  public Pate creerPate() {
    return new PateFine();
  }
  // ...
}

public class FabriqueIngredientsPizzaStrasbourg
      implements FabriqueIngredientsPizza {
  public Pate creerPate() {
    return new PateEpaisse();
  }
  // ...
}
```

Contrôle des ingrédients

La pizza recupère ses ingrédients de la fabrique appropriée

```java
public abstract class Pizza {
  // ...
  abstract void preparer();
}
```

```java
public class PizzaFromage extends Pizza {
  FabriqueIngredientsPizza fabriqueIngredients;
  public PizzaFromage(FabriqueIngredientsPizza
      fabriqueIngredients) {
    this.fabriqueIngredients = fabriqueIngredients;
  }

  void preparer() {
    System.out.println("Preparation de " + nom);
    pate = fabriqueIngredients.creerPate();
    sauce = fabriqueIngredients.creerSauce();
    fromage = fabriqueIngredients.creerFromage();
  }
}
```

Les franchisés utilisent les bonnes pizzas

```
public class PizzeriaBrest extends Pizzeria {
  protected Pizza creerPizza(String choix) {
    Pizza pizza;
    FabriqueIngredientsPizza fabriqueIngredients = new FabriqueIngredientsPizzaBrest();

    if (choix.equals("fromage")) {
      pizza = new PizzaFromage(fabriqueIngredients);
      pizza.setNom("Pizza au fromage style Brest")
    }
    // ...
    return pizza;
  }
}
```

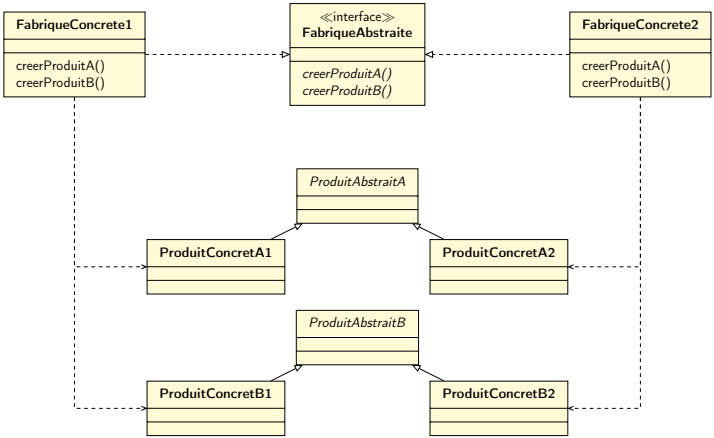
Le patron Fabrique abstraite

Definition

Le patron Fabrique Abstraite fournit une interface pour créer des familles d'objets apparentés ou dépendants sans avoir à spécifier leurs classes concrètes

En écrivant du code qui utilise une interface, on découple ce code de la fabrique réelle qui crée les produits
Cela nous permet d'implémenter toute une gamme de fabriques qui créent des produits destinés à différents contextes

Diagramme


FabriqueConcrete1
creerProduitA()
creerProduitB()
≪interface≫
FabriqueAbstraite
creerProduitA()
creerProduitB()
FabriqueConcrete2
creerProduitA()
creerProduitB()
ProduitAbstraitA
ProduitConcretA1
ProduitConcretA2
ProduitAbstraitB
ProduitConcretB1
ProduitConcretB2

Etude des patrons

Singleton

Créer un objet unique

Est-ce utile ?

Exemple

Pools de threads, caches, boîtes de dialogue, objets qui gèrent des préférences et des paramètres de registre, objets utilisés pour la journalisation et objets qui servent de pilotes à des périphériques comme les imprimantes et les cartes graphiques...

Comment créer un objet unique ?

Créer un objet

Comment créer un objet unique ?

Créer un objet

```
new MonObjet();
```

Comment créer un objet unique ?

Créer un objet `new MonObjet();`

Peut-on l'instancier plusieurs fois ?

Comment créer un objet unique ?

Créer un objet	`new MonObjet();`
Peut-on l'instancier plusieurs fois ?	Seulement si c'est une classe publique

Comment créer un objet unique ?

Créer un objet

```
new MonObjet();
```

Peut-on l'instancier plusieurs fois ?

Seulement si c'est une classe publique

Mais rien n'interdit d'écrire ceci

```
public MaClasse {
  private MaClasse() {}
}
```

Comment créer un objet unique ?

Créer un objet

```
new MonObjet();
```

Peut-on l'instancier plusieurs fois ?

Seulement si c'est une classe publique

Mais rien n'interdit d'écrire ceci

```
public MaClasse {
  private MaClasse() {}
}
```

Une classe qui ne peut pas être instanciée...

Comment créer un objet unique ?

Créer un objet

```
new MonObjet();
```

Peut-on l'instancier plusieurs fois ?

Seulement si c'est une classe publique

Mais rien n'interdit d'écrire ceci

```
public MaClasse {
  private MaClasse() {}
}
```

Une classe qui ne peut pas être instanciée...

Sauf par...

Comment créer un objet unique ?

Créer un objet

`new MonObjet();`

Peut-on l'instancier plusieurs fois ?

Seulement si c'est une classe publique

Mais rien n'interdit d'écrire ceci

```
public MaClasse {
  private MaClasse() {}
}
```

Une classe qui ne peut pas être instanciée...

Sauf par...

Une méthode de `MaClasse`

Comment créer un objet unique ?

Créer un objet
`new MonObjet();`

Peut-on l'instancier plusieurs fois ?
Seulement si c'est une classe publique

Mais rien n'interdit d'écrire ceci

```
public MaClasse {
  private MaClasse() {}
}
```

Une classe qui ne peut pas être instanciée...

Sauf par...
Une méthode de `MaClasse`

On ne peut pas instancier `MaClasse`

Comment créer un objet unique ?

Créer un objet
Peut-on l'instancier plusieurs fois ?
Mais rien n'interdit d'écrire ceci

```
new MonObjet();
```

Seulement si c'est une classe publique

```
public MaClasse {
  private MaClasse() {}
}
```

Une classe qui ne peut pas être instanciée...

Sauf par...
On ne peut pas instancier `MaClasse`

Une méthode de `MaClasse`
Mais on pourrait avoir une méthode de classe

Implémentation du patron Singleton

```
public class Singleton {
  private static Singleton uniqueInstance;
  // autres variables d'instance
  private Singleton() {}
  public static Singleton getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }
  // autres methodes
}
```

Vue d'ensemble

Les patrons de la classification GoF étudiés

		Objectif		
		De construction	Structuraux	Comportementaux
Portée	Classe	**Fabrique**	Adaptateur	Interpréteur Patron de méthode
	Objet	**Fabrique abstraite** Monteur Prototype **Singleton**	Adaptateur Pont Composite **Décorateur** Façade Poids-mouche Procuration	Chaîne de responsabilité Commande Itérateur Médiateur Mémento Observateur Etat **Stratégie** Visiteur