Université de Monastir
Institut Supérieur d'Informatique et de Mathématiques
Département Informatique

Cours
Design Patterns et conception par contrats

ING 2 - Génie Logiciel

02

Introduction aux Design Patterns

Objectifs de ce chapitre

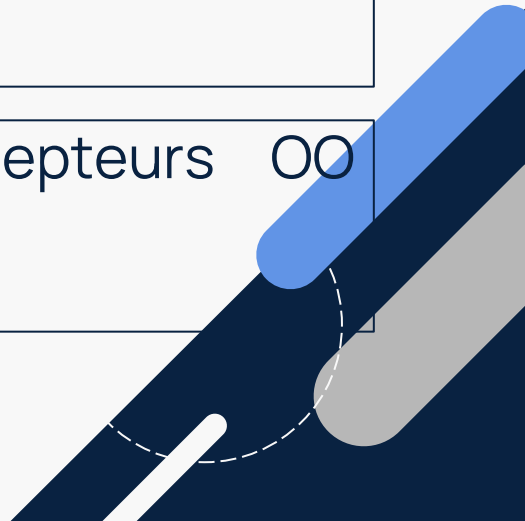
- Comprendre les défis du développement logiciel
- Identifier les problèmes liés à la maintenance, l'évolution et la réutilisabilité du code.
- Expliquer pourquoi une bonne conception est essentielle en génie logiciel.
- Définir ce qu'est un Design Pattern
- Classer les Design Patterns selon la classification GoF (Gang of Four)

Enjeu du Génie Logiciel

- Produire des conceptions **extensibles**, **maintenables** et **réutilisables**

Il ne faut pas chercher à résoudre un problème en partant de ses mécanismes de base, mais il vaut mieux réutiliser des solutions qui ont déjà fait leurs preuves.

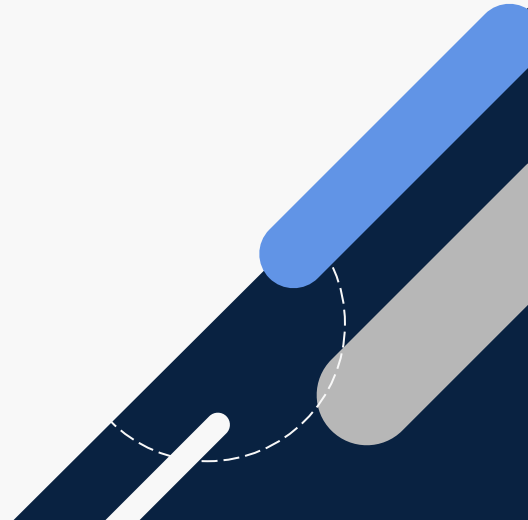
Il faut s'aider du savoir-faire des concepteurs OO expérimentés



Design Pattern-Définition

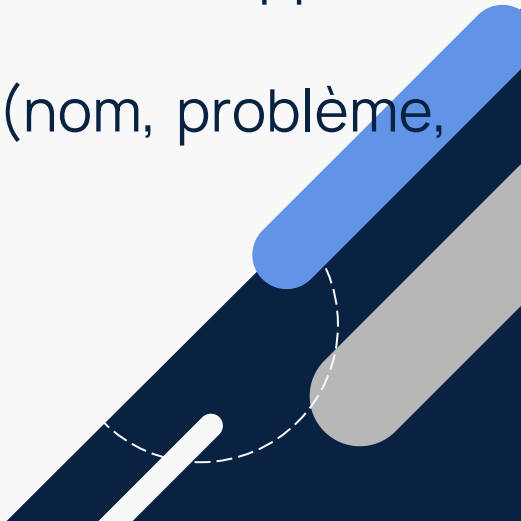
- Un Design Pattern est :

« Une solution réutilisable, éprouvée et documentée à un **problème de conception** qui se pose **fréquemment** dans le développement logiciel. »



Design Pattern-Caractéristiques

- **Réutilisable** : peut être appliqué dans différents projets.
- **Abstrait** : ce n'est pas du code prêt à l'emploi, mais un modèle de solution.
- **Commun** : fournit un vocabulaire partagé entre développeurs et architectes.
- **Documenté** : décrit de manière standardisée (nom, problème, solution, conséquences).



« Les patrons de conception vous aident à apprendre des succès des autres plutôt que de vos propres échecs. »

Mark Johnson

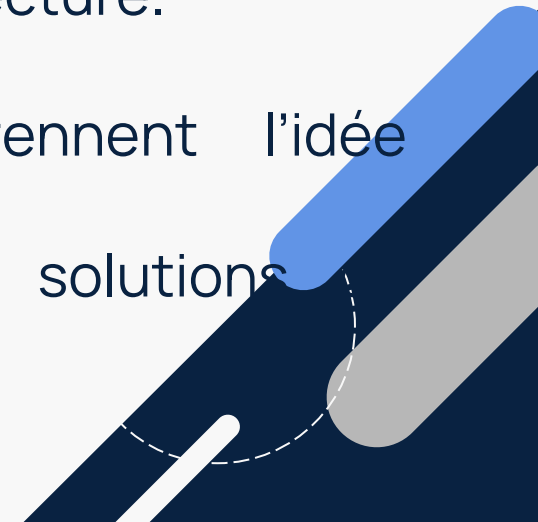
Historique des Design Patterns

- **1977 – Christopher Alexander (architecture)**

Architecte en bâtiment.
Publie “A Pattern Language”.
Introduit l’idée des patterns : solutions réutilisables à des problèmes récurrents d’urbanisme et d’architecture.

- **Années 1980 – Début en génie logiciel**

Les chercheurs en génie logiciel reprennent l’idée d’Alexander.
Utilisation de patterns pour décrire des solutions de conception logicielle réutilisables.



Historique des Design Patterns

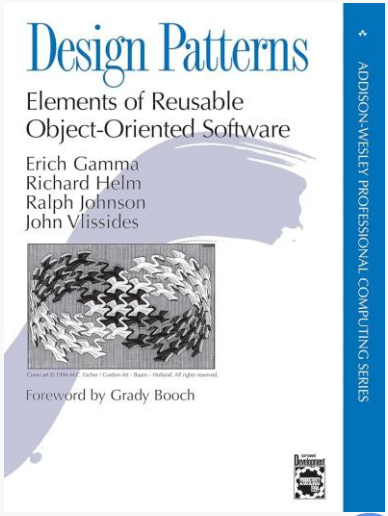
- 1994 – GoF (Gang of Four) :

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

Publient "*Design Patterns: Elements of Reusable Object-Oriented Software*".

Définissent 23 Design Patterns classés en 3 catégories: Créationnels, Structurels et Comportementaux

Ce livre devient la référence mondiale.

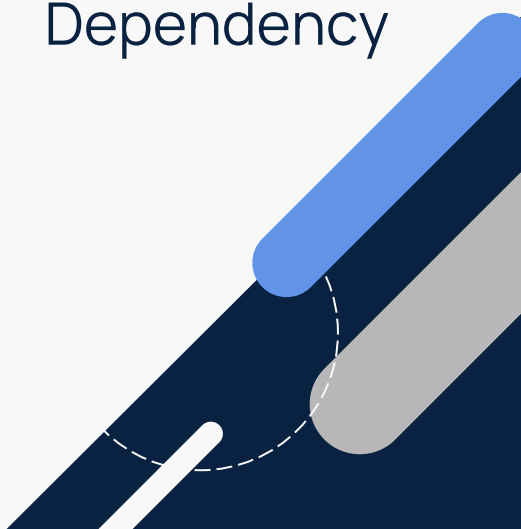


. . . . . .

Historique des Design Patterns

- **Depuis 2000 – Adoption massive des DP**

Intégrés dans les frameworks Java, C#, C++.
Essentiels pour les architectes logiciels et développeurs avancés.
De nouveaux patterns apparaissent (ex. : Dependency Injection, MVC, Microservices).



Structure d’un Design Pattern (GOF)

- **Nom du pattern:**

Unique, évocateur, facilite la communication entre développeurs.

Exemple : Observer, Strategy, Decorator

- **Problème:**

Contexte dans lequel le pattern est utile.

Le type de difficulté de conception rencontré.

Exemple (Observer) : Comment notifier plusieurs objets lorsqu’un objet change d’état, sans les coupler fortement ?

Structure d’un Design Pattern (GOF)

- **Solution:**

Description abstraite de la structure de classes/objets et de leurs collaborations.
Pas une implémentation unique → un schéma générique.

- **Conséquences (avantages / inconvénients):**

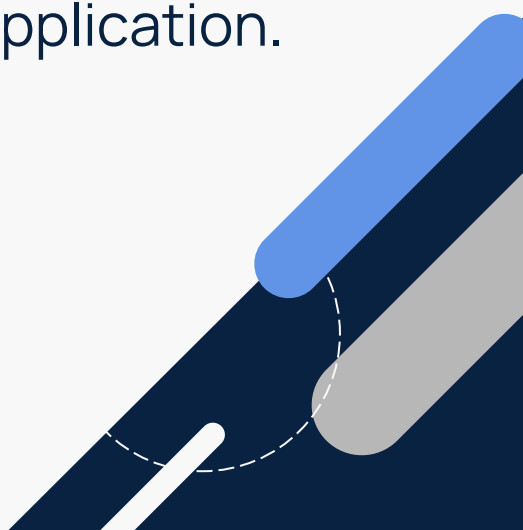
Points forts : flexibilité, découplage, extensibilité.
Points faibles : complexité accrue, surcoût potentiel.

- **Implémentation:**

Extrait de code (Java, Python, C++…).

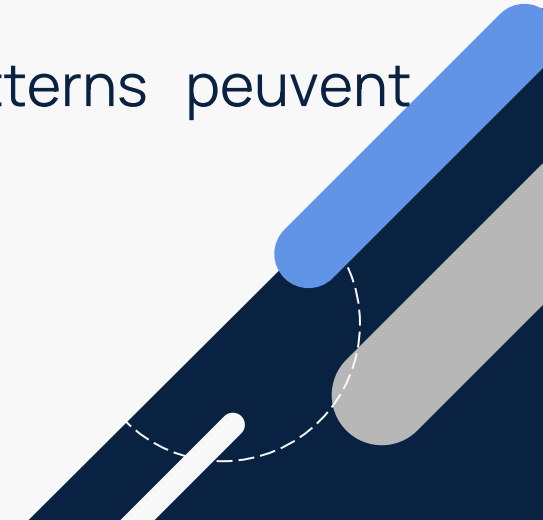
Avantages

- Faciliter la communication entre les différents développeurs.
- Réduire le couplage (interactions) entre les différentes classes(modules, composants) d'une application.
- Réduire le temps de développement d'une application.
- Faciliter la maintenance d'une application.



Inconvénients

- Difficulté à identifier quand un pattern s'applique.
- Nécessite un apprentissage et de l'expérience.
- Les patterns sont nombreux (23 patterns qui font l'unanimité, répertoriés dans un ouvrage de référence (GoF)).
- La corrélation des patterns i.e Les patterns peuvent dépendre d'autres patterns.



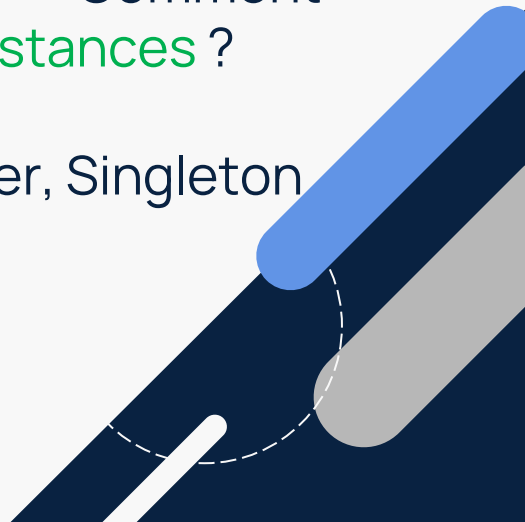
Classification des Design Patterns

- Les patrons de conception sont classés en trois catégories (GoF).

1. Patterns de Création (5)

- S'intéressent à la construction des objets : Comment organiser et déléguer la création dynamique d'instances ?

- Patterns FactoryMethod, AbstractFactory, Builder, Singleton



Classification des Design Patterns

2. Patterns de Structure (7)

- Permettent d'organiser les classes d'une application : Comment composer classes et objets pour obtenir des structures plus complexes ?

- Travaillent essentiellement sur des aspects statiques, à «l'extérieur»des classes

- Bridge, Adapter et Composite (etc.)

Classification des Design Patterns

3. Patterns de Comportement (11)

- Expliquent comment organiser les objets pour qu'ils collaborent entre eux : Comment faire collaborer classes ou objets pour construire une partie de l'application et des traitements ?

- Travaillent essentiellement sur des aspects dynamiques, à «l'intérieur»des classes et parfois au niveaux des instances

- Strategy, Decorator, Observer, Visitor (etc.)

Principes communs aux patrons

4 principes:

Principe 1. Réduire l'accessibilité des membres de classe

Principe 2. Encapsuler ce qui varie

Principe 3. Programmer pour une interface et non pour une implémentation

Principe 4. Privilégier la composition à l'héritage

Principe 1. Réduire l'accessibilité des membres de classe

- L'accès direct aux données membres d'une classe devrait être limité à la classe elle-même

- Éviter d'exposer les détails d’implémentation pour faciliter l'évolution future sans aucune conséquence sur la classe

Principe 1. Réduire l'accessibilité des membres de classe

Solution : encapsulation

- Faire des attributs privés
- Réduire l'utilisation des accesseurs (getters) et mutateurs (setters) : Leur utilisation suggère que l'objet est un fournisseur de données, alors qu'il faut considérer l'objet comme un fournisseur de services

Principe 2. Encapsuler ce qui varie

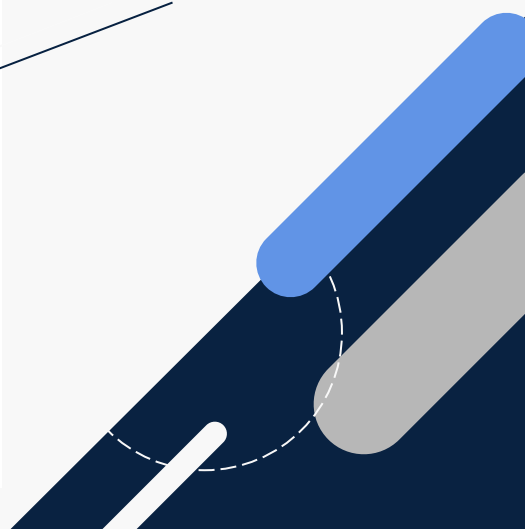
Principe:

Isoler ce qui change de ce qui reste stable.

Exemple:

```
interface PaymentStrategy { void pay(int amount); }
class CreditCardPayment implements PaymentStrategy { ... }
class PayPalPayment implements PaymentStrategy { ... }

class ShoppingCart {
   private PaymentStrategy payment;
   public void setPayment(PaymentStrategy p)
{ payment = p; }
   public void checkout() { payment.pay(total); }
}
```

l'algorithme de paiement peut changer sans modifier ShoppingCart.



Règle 3. Programmer pour une interface et non pour une implémentation

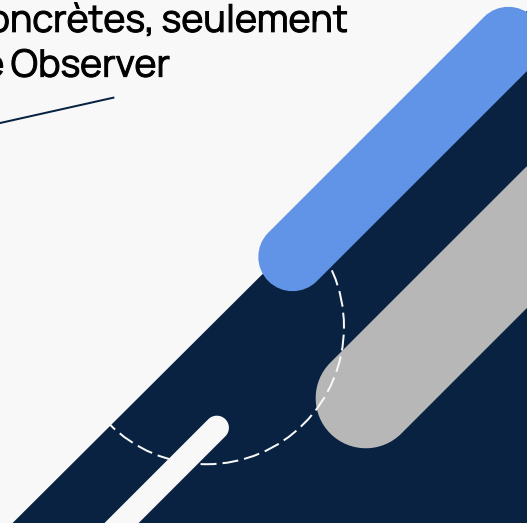
Principe:

Dépendre d'abstractions, pas de classes concrètes

Exemple (Pattern Observer) :

```
interface Observer { void update(); }
class ConcreteObserver implements Observer { ... }
class Subject {
   private List<Observer> observers;
   void notifyObservers() { for(Observer o : observers)
o.update(); }
}
```

Le Subject ne connaît pas les classes concrètes, seulement l'interface Observer



Règle 4. Privilégier la composition à l'héritage

Principe:

Les objets collaborent plutôt que créer une hiérarchie complexe.

L'héritage crée une relation “est-un” (is-a) : rigide et difficile à changer.

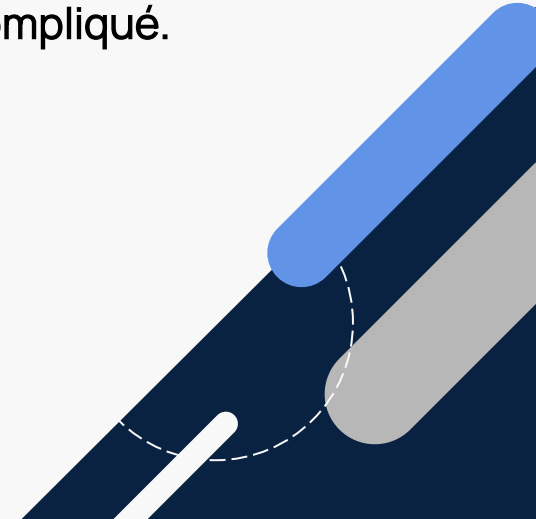
La **composition** crée une relation “a-un” (has-a) : flexible, remplaçable et extensible

Règle 4. Privilégier la composition à l'héritage

Mauvais exemple (héritage) :

```
class EmailNotifier {
  void sendEmail(String msg) {
System.out.println("Email: " + msg); }
}

class SMSNotifier extends EmailNotifier {
  void sendSMS(String msg) { System.out.println("SMS: "
+ msg); }
}
```

Problèmes :
SMSNotifier n'est pas un EmailNotifier, la hiérarchie est artificielle.
Ajouter PushNotifier ou combiner plusieurs méthodes devient compliqué.



Règle 4. Privilégier la composition à l'héritage

Bon exemple (Composition)

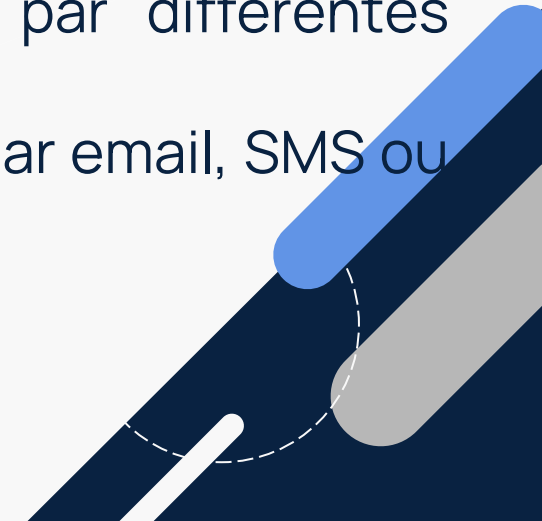
```
interface Notifier {
   void send(String msg);
}

class EmailNotifier implements Notifier {
   public void send(String msg) {
System.out.println("Email: " + msg); }
}

class SMSNotifier implements Notifier {
   public void send(String msg) {
System.out.println("SMS: " + msg); }
}
```

```
class NotificationService implements Notifier {
   private List<Notifier> notifiers = new ArrayList< >();

   public void addNotifier(Notifier n) { notifiers.add(n); }

   public void send(String msg) {
      for (Notifier n : notifiers) n.send(msg);
   }
}

// Utilisation
NotificationService service = new NotificationService();
service.addNotifier(new EmailNotifier());
service.addNotifier(new SMSNotifier());
service.send("Hello world!");
```

Exemple pratique

Système de gestion de commandes

- Contexte: L'application doit gérer différents types de commandes : Commandes en ligne, Commandes en magasin.
- Les commandes peuvent être payées par différentes méthodes : carte, PayPal, crypto.
- Le système doit pouvoir notifier le client par email, SMS ou push.



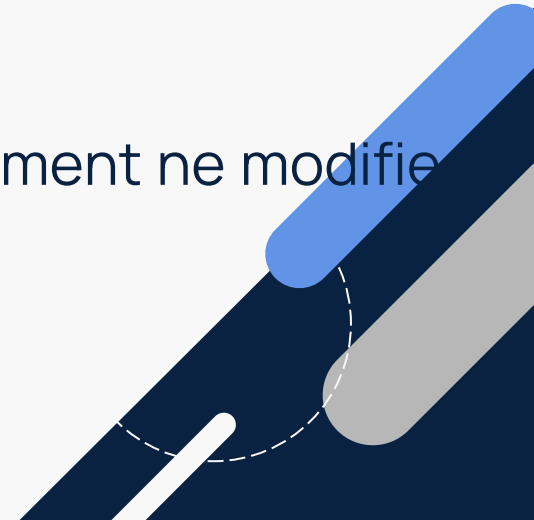
Exemple pratique

Encapsulation des variations

On isole le comportement variable (méthode de paiement)

```
interface PaymentStrategy { void pay(double amount); }
class CreditCardPayment implements PaymentStrategy { ... }
class PayPalPayment implements PaymentStrategy { ... }
```

Avantage : ajouter un nouveau mode de paiement ne modifie pas le reste du code.



Exemple pratique

Programmer pour une interface

Les notifications passent par une interface abstraite :

```
interface Notifier { void send(String msg); }
class EmailNotifier implements Notifier { ... }
class SMSNotifier implements Notifier { ... }
```

Le code qui envoie les notifications dépend de Notifier, pas des classes concrètes.

Exemple pratique

Programmer pour une interface

Les notifications passent par une interface abstraite :

```
interface Notifier { void send(String msg); }
class EmailNotifier implements Notifier { ... }
class SMSNotifier implements Notifier { ... }
```

Le code qui envoie les notifications dépend de Notifier, pas des classes concrètes.