Génie logiciel : introduction aux patrons de conception (*design patterns*)

Karën Fort

karen.fort@univ-lorraine.fr / https://members.loria.fr/KFort

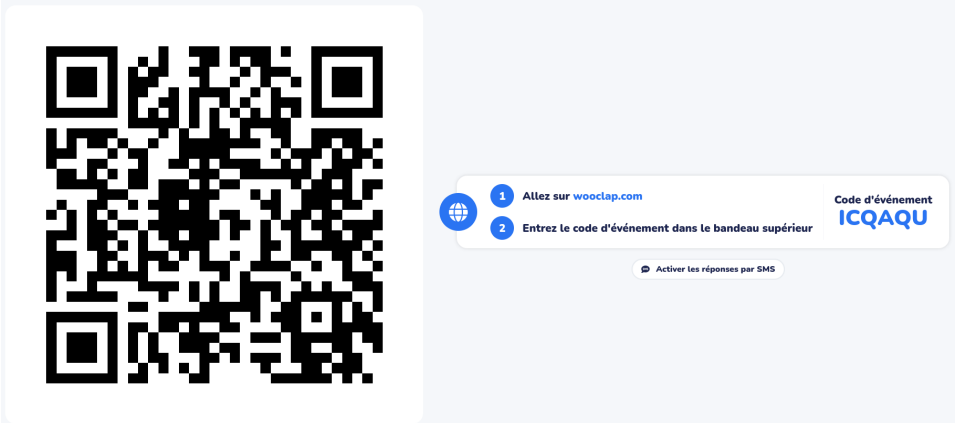
Institut des sciences du Digital Management & Cognition
COMPOSANTE DE L'UNIVERSITÉ DE LORRAINE

UNIVERSITÉ DE LORRAINE

Quelques sources d'inspiration

- Indispensable : https://refactoring.guru/design-patterns
- Présentation de Forrest Knight : https://www.youtube.com/watch?v=BJatgOiiht4
- Des exemples : https://www.javaguides.net/2025/02/top-10-design-patterns-every-java-developer-should-know.html
- D'autres exemples : https://www.digitalocean.com/community/tutorials/java-design-patterns-example-tutorial
- https://fr.wikipedia.org/wiki/Patron_de_conception

Wooclap

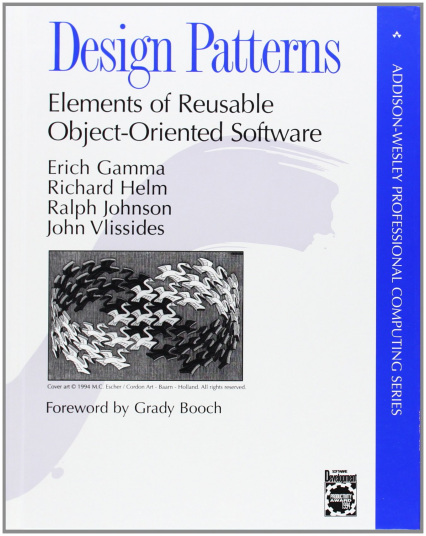

1 Allez sur wooclap.com
2 Entrez le code d'événement dans le bandeau supérieur
Code d'événement
ICQAQU
Activer les réponses par SMS

Wooclap

Exercice

Qu'est-ce qu'un patron de conception (design patterns) ?

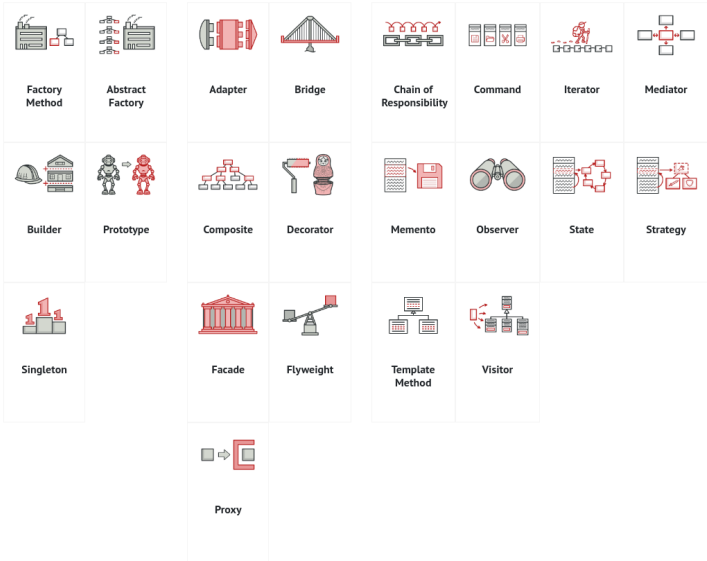
Un livre publié en 1994 par le “Gang of Four”



Patrons de conception : principes

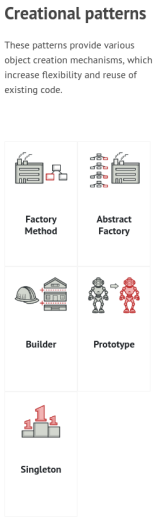
- solutions à des problèmes de programmation classiques (23 dans le livre)
- pour les language objets
- permettent de créer un code simple et élégant, plus facile à maintenir
- très utilisés en Java, notamment (vous devez être capables de les reconnaître)

Patrons de conception : un catalogue structuré



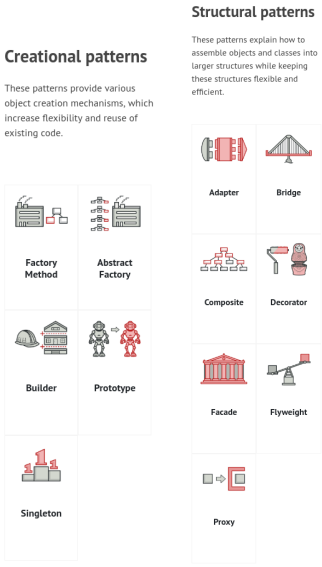
https://refactoring.guru/design-patterns/catalog

Patrons de conception : un catalogue structuré



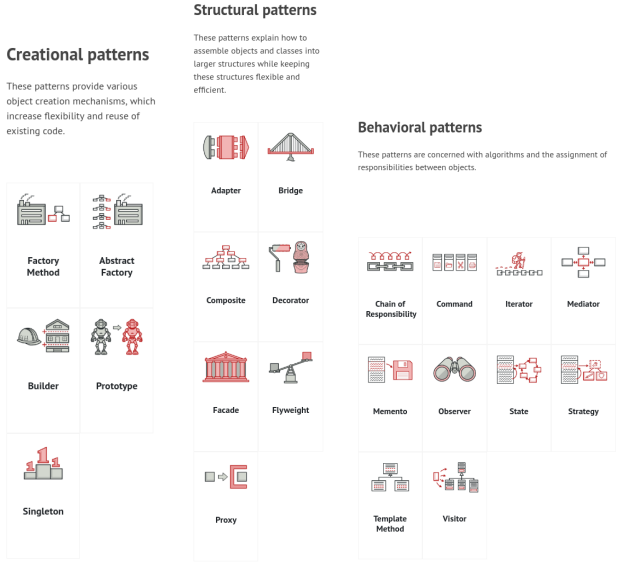
https://refactoring.guru/design-patterns/catalog

Patrons de conception : un catalogue structuré



https://refactoring.guru/design-patterns/catalog

Patrons de conception : un catalogue structuré



https://refactoring.guru/design-patterns/catalog

Singleton

Permet d'assurer qu'il n'y a toujours qu'une seule instance d'une classe :

- une méthode pour obtenir cette unique instance
- un mécanisme pour empêcher la création d'autres instances

Concrètement :

- le constructeur doit être privé (pour empêcher les autres objets d'utiliser l'opérateur *new* de cette classe)
- créer une méthode de création statique qui se comporte comme un constructeur et qui va appeler le constructeur privé pour créer un objet et le sauvegarder dans un champ statique (les appels suivants vont retourner cet objet en cache)

Exemple de Singleton

```java
public class Singleton {

    // Volatile ensures visibility across threads and prevents instruction reordering
    private static volatile Singleton instance;

    // Private constructor prevents direct instantiation
    private Singleton() {
        if (instance != null) {
            throw new RuntimeException("Use getInstance() to create an object");
        }
    }

    // Double-checked locking for thread safety
    public static Singleton getInstance() {
        if (instance == null) {  // First check (No lock)
            synchronized (Singleton.class) {
                if (instance == null) {  // Second check (With lock)
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

https://www.javaguides.net/2025/02/top-10-design-patterns-every-java-developer-should-know.html

Builder (monteur)

Sépare le processus de construction d'un objet du résultat obtenu :

- utile quand il y a de nombreux paramètres de création, en particulier optionnels

Concrètement :

- au lieu d'une méthode pour créer un objet, à laquelle est passée un ensemble de paramètres, la classe comporte une méthode pour créer un objet, le builder
- le builder comporte des propriétés qui peuvent être modifiées et une méthode pour créer l'objet final en tenant compte de toutes les propriétés

Exemple de Builder

```java
public class User {
    private final String name;
    private final int age;
    private final String email;

    // Private constructor to enforce Builder usage
    private User(UserBuilder builder) {
        this.name = builder.name;
        this.age = builder.age;
        this.email = builder.email;
    }

    // Static inner Builder class
    public static class UserBuilder {
        private final String name;  // Mandatory field
        private int age;            // Optional
        private String email;       // Optional

        public UserBuilder(String name) { this.name = name; }

        public UserBuilder age(int age) {
            this.age = age;
            return this;
        }

        public UserBuilder email(String email) {
            this.email = email;
            return this;
        }

        public User build() {
            return new User(this);
        }
    }
}
```

https://www.javaguides.net/2025/02/top-10-design-patterns-every-java-developer-should-know.html

Factory (fabrique)

Permet de créer des familles d'objets (interface pour créer des objects sans spécifier leur classe exacte) :

- réduit les dépendances
- améliore la flexibilité

Concrètement :

- une fabrique retourne une instance d'une classe parmi plusieurs possibles, en fonction des paramètres qui ont été fournis.
- toutes les classes ont un lien de parenté, et des méthodes communes, et chacune est optimisée en fonction d'une certaine donnée

Exemple de Factory

```java
interface Vehicle {
    void manufacture();
}

// Concrete Implementations
class Car implements Vehicle {
    public void manufacture() { System.out.println("Manufacturing a Car!"); }
}

class Bike implements Vehicle {
    public void manufacture() { System.out.println("Manufacturing a Bike!"); }
}

// Factory Class
class VehicleFactory {
    public static Vehicle getVehicle(String type) {
        return switch (type.toLowerCase()) {
            case "car" -> new Car();
            case "bike" -> new Bike();
            default -> throw new IllegalArgumentException("Invalid vehicle type!");
        };
    }
}
```

https://www.javaguides.net/2025/02/top-10-design-patterns-every-java-developer-should-know.html

Wooclap

Exercice

Est-ce que vous avez déjà vu ces patrons de création dans du code (par ex. en entreprise) ?

Façade (facade)

Fournit une interface simplifiée vers un sous-système complexe (interactions entre sous-systèmes) :

- permet de contrôler l'ordre des opérations
- permet de cacher les détails techniques des sous-systèmes

Concrètement :

- juste de la programmation propre
- on limite les fonctionnalités disponibles pour ne donner accès qu'à l'indispensable

Exemple de Façade

```
package com.journaldev.design.facade;

import java.sql.Connection;

public class HelperFacade {

	public static void generateReport(DBTypes dbType, ReportTypes reportType, Strin
		Connection con = null;
		switch (dbType){
		case MYSQL:
			con = MySqlHelper.getMySqlDBConnection();
			MySqlHelper mySqlHelper = new MySqlHelper();
			switch(reportType){
			case HTML:
				mySqlHelper.generateMySqlHTMLReport(tableName, con);
				break;
			case PDF:
				mySqlHelper.generateMySqlPDFReport(tableName, con);
				break;
			}
			break;
		case ORACLE:
			con = OracleHelper.getOracleDBConnection();
			OracleHelper oracleHelper = new OracleHelper();
			switch(reportType){
			case HTML:
				oracleHelper.generateOracleHTMLReport(tableName, con);
				break;
			case PDF:
				oracleHelper.generateOraclePDFReport(tableName, con);
				break;
			}
			break;
		}

	}

	public static enum DBTypes{
		MYSQL,ORACLE;
	}

	public static enum ReportTypes{
		HTML,PDF;
	}
}
```

https://www.digitalocean.com/community/tutorials/facade-design-pattern-in-java

Exemple de Façade

```
package com.journaldev.design.test;

import java.sql.Connection;

import com.journaldev.design.facade.HelperFacade;
import com.journaldev.design.facade.MySqlHelper;
import com.journaldev.design.facade.OracleHelper;

public class FacadePatternTest {

        public static void main(String[] args) {
                String tableName="Employee";

                //generating MySql HTML report and Oracle PDF report without using Faca
                Connection con = MySqlHelper.getMySqlDBConnection();
                MySqlHelper mySqlHelper = new MySqlHelper();
                mySqlHelper.generateMySqlHTMLReport(tableName, con);

                Connection con1 = OracleHelper.getOracleDBConnection();
                OracleHelper oracleHelper = new OracleHelper();
                oracleHelper.generateOraclePDFReport(tableName, con1);

                //generating MySql HTML report and Oracle PDF report using Facade
                HelperFacade.generateReport(HelperFacade.DBTypes.MYSQL, HelperFacade.Re
                HelperFacade.generateReport(HelperFacade.DBTypes.ORACLE, HelperFacade.R
        }

}
```

https://www.digitalocean.com/community/tutorials/facade-design-pattern-in-java

Adapter (adaptateur)

Permet de faire collaborer deux interfaces incompatibles/plus compatibles

Concrètement :

- cache la complexité de la conversion

Exemple d'Adapter

```
interface MediaPlayer {
    void play(String file);
}

class MP3Player implements MediaPlayer {
    public void play(String file) {
        System.out.println("Playing MP3 file: " + file);
    }
}

// Adapter to convert AdvancedMediaPlayer into MediaPlayer
class MediaAdapter implements MediaPlayer {
    private final AdvancedMediaPlayer advancedPlayer = new AdvancedMediaPlayer();

    public void play(String file) {
        advancedPlayer.playAdvancedFormat(file);
    }
}
```

https://www.javaguides.net/2025/02/top-10-design-patterns-every-java-developer-should-know.html

Wooclap

Exercice

Est-ce que vous avez déjà vu ces patrons de structure dans du code (par ex. en entreprise) ?

Strategy (stratégie)

S'applique à une classe qui réalise une action spécifique de différentes manières (app de navigation : en voiture, à pied, en train, etc.) :

- extraire tous les algos dans des classes séparées appelées strategies

Concrètement :

- trois rôles : le contexte, la stratégie et les implémentations :
  - La stratégie est l'interface commune aux différentes implémentations — typiquement une classe abstraite.
  - Le contexte est l'objet qui va associer un algorithme avec un processus

Exemple de Strategy

```java
// Strategy Interface
interface PaymentStrategy {
    void pay(int amount);
}

// Concrete Strategies
class CreditCardPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Credit Card.");
    }
}

class PayPalPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using PayPal.");
    }
}
```

```java
// Context Class
class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void checkout(int amount) {
        paymentStrategy.pay(amount);
    }
}

// Usage
public class StrategyPatternExample {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        cart.setPaymentStrategy(new CreditCardPayment());
        cart.checkout(500);

        cart.setPaymentStrategy(new PayPalPayment());
        cart.checkout(300);
    }
}
```

https://www.javaguides.net/2025/02/top-10-design-patterns-every-java-developer-should-know.html

Observer (observateur)

Établit une relation un à plusieurs entre des objets, où lorsqu'un objet change, plusieurs autres objets sont avisés du changement (par ex, notifications, messages, etc)

Concrètement :

- les changements sur un objet (le sujet) sont notifiés aux multiples objets dépendants (observateurs)
- c'est l'objet qui change qui prévient les autres

Exemple d'Observer

```java
import java.util.ArrayList;
import java.util.List;

// Observer Interface
interface Observer {
    void update(String message);
}

// Concrete Observer
class User implements Observer {
    private String name;

    public User(String name) { this.name = name; }

    public void update(String message) {
        System.out.println(name + " received notification: " + message);
    }
}

// Subject Interface
interface Observable {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(String message);
}
```

```java
// Concrete Subject
class NotificationService implements Observable {
    private List<Observer> observers = new ArrayList<>();

    public void addObserver(Observer observer) { observers.add(observer); }
    public void removeObserver(Observer observer) { observers.remove(observer); }
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}

// Usage
public class ObserverPatternExample {
    public static void main(String[] args) {
        NotificationService service = new NotificationService();

        Observer user1 = new User("Alice");
        Observer user2 = new User("Bob");

        service.addObserver(user1);
        service.addObserver(user2);

        service.notifyObservers("New video uploaded!");
    }
}
```

https://www.javaguides.net/2025/02/top-10-design-patterns-every-java-developer-should-know.html

Wooclap

Exercice

Est-ce que vous avez déjà vu ces patrons de comportement dans du code (par ex. en entreprise) ?

Wooclap

Exercice

Avez-vous déjà vu d'autres patrons de conception non présentés ici dans du code (par ex. en entreprise) ?

Historique et motivations

Quelques patrons de conception

Pour finir

CQFR : Ce Qu'il Faut Retenir

- être conscient que vous les utilisez
- pratiquer leur utilisation
- respecter le nommage