

06

# Patrons de conception

---



# Les patrons de conception

Les patrons de conception en programmation, également connus sous le nom de "design patterns" en anglais, sont des solutions éprouvées et réutilisables pour résoudre des problèmes courants dans le développement logiciel. Ils fournissent des modèles structurés et des approches standardisées pour la conception et l'organisation du code, ce qui facilite la création de logiciels fiables, maintenables et évolutifs.

Les patrons de conception ont été introduits pour la première fois par le livre "Design Patterns: Elements of Reusable Object-Oriented Software" écrit par Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, également appelés le "Gang of Four" (GoF). Ce livre décrit 23 patrons de conception fondamentaux, regroupés en trois catégories : les patrons de création, les patrons de structure et les patrons de comportement.

Les patrons ne se limitent pas à un langage de programmation. Il s'agit de structures que vous pouvez transférer si vous apprenez d'autres langages comme Java ou C#.

Dans le cours d'aujourd'hui, nous verrons le singleton, la fabrique et l'observateur. Cependant, au cours de la session, nous verrons d'autres que vous pourrez ajouter à votre coffre à outils de programmation.

À noter qu'il existe plusieurs façons d'arriver au même résultat en JavaScript, il est possible que les exemples diffèrent dans les différentes références.

# Singleton

Le patron de conception Singleton est un modèle de conception qui garantit qu'une classe ne possède qu'une seule instance et fournit un point d'accès global à cette instance. Ce patron est utilisé pour des classes qui gèrent d'autres classes et dont il serait illogique d'avoir plusieurs instances.

En d'autres termes, il s'agit d'assurer qu'une classe n'a qu'une seule instance unique et de fournir un moyen de récupérer cette instance partout dans l'application.

En javascript, nous assignons une copie de l'instance au prototype de la classe. Il s'agit d'une propriété qu'on appelle statique.

Elle peut être appelée n'importe où sans avoir à utiliser new devant le nom de la classe.

```
class Gestionnaire {  
    constructor() {  
        if (Gestionnaire.instance == null) {  
            Gestionnaire.instance = this;  
        } else{  
            throw new Error("Impossible de créer plusieurs instances.");  
        }  
        // Suite du constructeur  
    }  
  
    // Autres méthodes de la classe  
}  
  
const instance1 = new Gestionnaire();  
const instance2 = new Gestionnaire();  
  
console.log(instance1 === instance2); // true  
console.log(Gestionnaire.instance);
```

# Fabrique

Le patron de conception Factory (ou Fabrique) est un modèle de conception créatif qui fournit une interface pour créer des objets dans une classe de base, tout en permettant aux sous-classes de décider quels types d'objets concrets créer. Il encapsule le processus de création d'objets et permet une meilleure flexibilité en matière de création d'instances de classes sans exposer directement les détails de leur création.

Le patron Factory est utile lorsque :

- Vous voulez déléguer la création d'objets à des sous-classes ou à des méthodes spécifiques.
- Vous voulez encapsuler la logique de création d'objets afin que les parties de votre code qui utilisent les objets n'aient pas à connaître les détails de leur création.
- Vous voulez centraliser la création d'objets dans une classe ou une fonction, ce qui peut faciliter la maintenance et l'extension du code.

```
class ProduitA {  
    constructor(nom) {  
        this.nom = nom;  
    }  
}  
  
class ProduitB {  
    constructor(nom) {  
        this.nom = nom;  
    }  
}  
  
class FabriqueDeProduits {  
    creer(type, nom) {  
        if(type == "A"){  
            return new ProduitA(nom);  
        }else{  
            return new ProduitB(nom);  
        }  
    }  
}  
  
const fabrique = new FabriqueDeProduits();  
const produitA = fabrique.creer("A", "Produit A");  
const produitB = fabrique.creer("B", "Produit B");  
  
console.log(produit1.nom); // Produit A  
console.log(produit2.nom); // Produit B
```

# Observateur et événements personnalisés

Le patron Observer (ou Observateur) en JavaScript est un modèle de conception comportemental qui permet à un objet, appelé sujet, de maintenir une liste de ses dépendants, appelés observateurs, et de les informer automatiquement de tout changement d'état survenant dans le sujet.

Cela permet aux observateurs de réagir en conséquence, créant ainsi un lien lâche entre le sujet et les observateurs. C'est un peu le principe du AddEventListener.

Il est possible en JavaScript d'ajouter des événements personnalisés sur n'importe quel élément HTML. Nous créons un événement CustomEvent avec le nom de notre choix. Il est possible d'ajouter des données à passer avec le paramètre detail.

Ensuite, on ajoute un écouteur d'événement du type de notre événement personnalisé.

Ensuite on déclenche l'événement avec DispatchEvent.

```
const elementObservateur = document.querySelector("body");

// Fonction pour gérer l'événement personnalisé
function handleCustomEvent(event) {
    console.log('Évenement déclenché:', event.detail);
}

// Ajout d'un écouteur d'événement personnalisé
elementObservateur.addEventListener('monEvenement', handleCustomEvent);

// Séquence de déclenchement de l'événement
// Cr éation et d éclenchement d'un événement personnalisé
const nouvelEvenement = new CustomEvent('monEvenement', {
    detail: 'Données personnalisées'
});

// On d éclenche l'événement sur le m ême élément qui comporte
// l'écouteur d'événement
elementObservateur.dispatchEvent(nouvelEvenement);
```

# Références

Patterns - Dev (Exemples en JavaScript)  
<https://www.patterns.dev/posts>

Refactoring Guru - Tous types de langages  
<https://refactoring.guru/design-patterns>

Comprehensive Guide to JavaScript Design Patterns  
<https://www.toptal.com/javascript/comprehensive-guide-javascript-design-patterns>