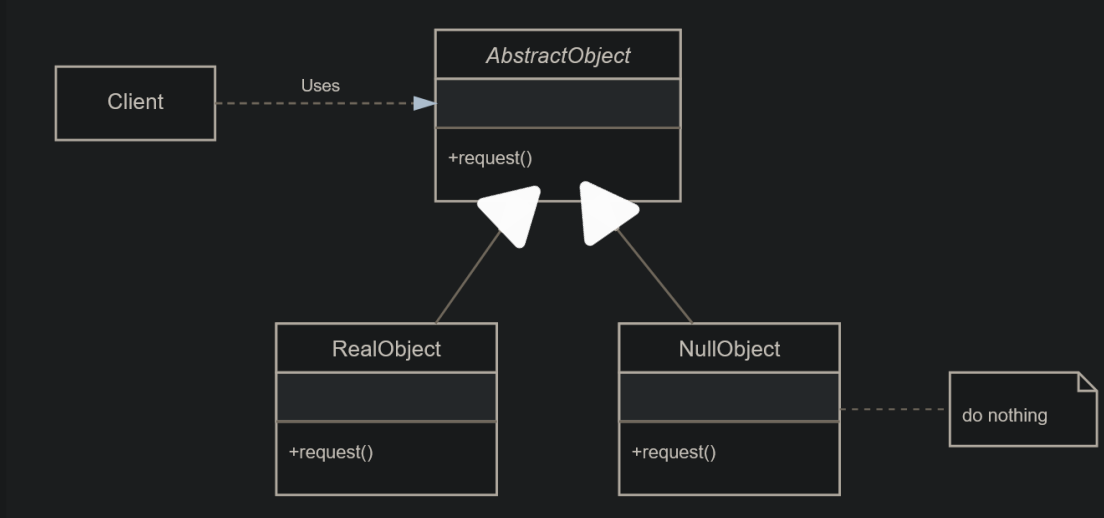
Pattern Null Object

Définition

Le **pattern Null Object** est un pattern de conception comportemental qui fournit un objet avec un comportement neutre (par défaut) à la place de `null` ou `undefined` .

🎯 **Objectif :** Éliminer les vérifications null répétitives et éviter les NullPointerException en encapsulant l'absence d'objet dans un objet qui ne fait rien.

Diagramme UML



Client
Utilise AbstractObject sans se soucier de sa nature

AbstractObject
Interface ou classe abstraite définissant request()

RealObject
Implémentation avec comportement réel

NullObject
Implémentation "ne fait rien"

Problème

Sans le pattern :

```
if (customer != null) {
  customer.sendEmail();
}

if (user != null) {
  user.getName();
} else {
  return "Guest";
}
```

- Vérifications null partout dans le code
- Code verbeux et répétitif
- Risque de NullPointerException

Solution

Avec le pattern :

```
// Plus besoin de vérification
customer.sendEmail();

// Retourne toujours un objet valide
String name = user.getName();
```

- ✓ Suppression des vérifications null
- ✓ Code plus propre et lisible
- ✓ Comportement cohérent et prévisible

Avantages

Simplification du code client
Élimine les vérifications null répétitives, rendant le code plus propre et lisible

Réduction des erreurs
Prévient les NullPointerException et autres erreurs liées aux valeurs null

Principe Open/Closed
Respecte le principe SOLID : ouvert à l'extension, fermé à la modification

Comportement polymorphique
Traite les objets null et réels de manière uniforme via l'interface commune

Maintenance facilitée
Centralise la logique de comportement par défaut en un seul endroit

Tests plus simples
Facilite les tests unitaires en fournissant un comportement prévisible

Inconvénients

Complexité accrue
Ajoute des classes supplémentaires au projet, augmentant la complexité du code

Masquage des erreurs
Peut cacher des bugs en masquant l'absence d'objet au lieu de signaler une erreur

Difficulté de débogage
Peut rendre le débogage plus difficile car les erreurs ne sont pas immédiatement visibles

Mémoire et performance
Crée des instances d'objets supplémentaires qui consomment de la mémoire

Sur-utilisation
Tendance à être sur-utilisé là où une simple vérification null serait plus appropriée

Comportement implicite
Le comportement 'ne fait rien' peut ne pas être évident pour les développeurs qui lisent le code

Exemple d'Implémentation

1. Interface/Classe Abstraite

AbstractObject

```
public abstract class Customer {
    public abstract String getName();
    public abstract boolean isNull();
}
```

2. Objet Réel

RealObject

```
public class RealCustomer extends Customer {
    private String name;

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

    @Override
    public String getName() {
        return name;
    }

    @Override
    public boolean isNull() {
        return false;
    }
}
```

3. Null Object

NullObject

```
public class NullCustomer extends Customer {
    @Override
    public String getName() {
        return "Guest";  // Comportement par défaut
    }

    @Override
    public boolean isNull() {
        return true;
    }
}
```

4. Factory

Factory Pattern

```
public class CustomerFactory {
    private static final String[] names = {"Alice", "Bob", "Charlie"};

    public static Customer getCustomer(String name) {
        for (String n : names) {
            if (n.equalsIgnoreCase(name)) {
                return new RealCustomer(name);
            }
        }
        return new NullCustomer();  // Retourne Null Object
    }
}
```

5. Utilisation

Client

```
public class Main {
    public static void main(String[] args) {
        Customer customer1 = CustomerFactory.getCustomer("Alice");
        Customer customer2 = CustomerFactory.getCustomer("Unknown");

        // Pas de vérification null nécessaire !
        System.out.println(customer1.getName());  // Alice
        System.out.println(customer2.getName());  // Guest
    }
}
```

Quand l'utiliser ?

Vérifications null fréquentes
Quand le code contient de nombreuses vérifications null répétitives

Comportement par défaut acceptable
Quand un comportement neutre ou par défaut a du sens métier

Collections d'objets
Pour éviter les vérifications dans les itérations de collections

Design Patterns
Avec Strategy, State, ou Command patterns pour simplifier le code

APIs et bibliothèques
Pour garantir que les méthodes retournent toujours un objet valide

Quand l'éviter ?

Erreurs critiques
Quand l'absence d'objet indique une erreur qui doit être signalée

Cas simples
Pour des vérifications null ponctuelles qui n'alourdissent pas le code

Comportement ambigu
Quand 'ne rien faire' pourrait créer de la confusion ou des bugs

Performance critique
Quand chaque allocation d'objet compte (systèmes embarqués, temps réel)

Debugging nécessaire
Quand il est important de tracer et déboguer les valeurs null

Relations avec d'autres Patterns

Pattern Strategy

Le Null Object peut être utilisé comme une **stratégie par défaut** dans le pattern Strategy. Au lieu de vérifier si une stratégie existe, on peut fournir un NullStrategy qui ne fait rien.

```
interface PaymentStrategy {
    void pay(double amount);
}

class CreditCardStrategy implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paying " + amount + " with credit card");
    }
}

// NullObject comme stratégie par défaut
class NullPaymentStrategy implements PaymentStrategy {
    public void pay(double amount) {
        // Ne fait rien - comportement neutre
    }
}

// Usage
PaymentStrategy strategy = getPaymentMethod(); // Peut retourner NullPaymentStrategy
strategy.pay(100); // Pas de vérification null nécessaire
```

💡 **Avantage :** Le client n'a pas besoin de vérifier si une stratégie existe. La NullStrategy fournit un comportement sûr par défaut.

Pattern State

Le Null Object peut représenter un **état initial ou invalide** dans le pattern State. Cela évite les vérifications null lors des transitions d'état.

```
interface ConnectionState {
    void connect();
    void disconnect();
}

class ConnectedState implements ConnectionState {
    public void connect() { /* Déjà connecté */ }
    public void disconnect() { /* Déconnexion */ }
}

// NullObject comme état initial
class DisconnectedState implements ConnectionState {
    public void connect() { /* Connexion */ }
    public void disconnect() { /* Déjà déconnecté - ne fait rien */ }
}

class Connection {
    private ConnectionState state = new DisconnectedState(); // État par défaut

    public void connect() {
        state.connect(); // Pas de vérification null
    }
}
```

💡 **Avantage :** Garantit qu'un objet a toujours un état valide, même s'il n'est pas encore initialisé.

Comparaison des utilisations

Pattern	Rôle du Null Object	Bénéfice principal
Strategy	Stratégie par défaut ou "do nothing"	Évite les vérifications avant exécution
State	État initial ou invalide	Garantit un état valide à tout moment
Null Object	Remplaçant pour null	Élimine les NullPointerException

Exemple combiné : Strategy + State + Null Object

```
class ShoppingCart {
    private DiscountStrategy discount = new NullDiscount(); // Null Object
    private CartState state = new EmptyCartState(); // State pattern

    public void setDiscountStrategy(DiscountStrategy discount) {
        this.discount = discount; // Peut être null-safe
    }

    public double calculateTotal() {
        double total = state.calculateTotal();
        return discount.apply(total); // Toujours safe
    }
}

// Les trois patterns travaillent ensemble :
// - State gère l'état du panier (vide, avec items, checkout)
// - Strategy gère les différentes stratégies de réduction
// - Null Object fournit des implémentations par défaut sûres
```

Cas d'Usage Courants

Logger / Console
Un NullLogger qui ne log rien peut remplacer un logger réel en production

```
logger.log("message")
```

Utilisateurs
Un NullUser ou GuestUser pour les utilisateurs non authentifiés

```
user.getName() // "Guest"
```

Collections
Collections vides ou itérateurs sans éléments

```
list.forEach(item → ...)
```

Stratégies
Stratégie par défaut qui ne fait aucun traitement spécial

```
strategy.execute()
```

Données
Repository retournant un objet vide au lieu de null

```
repo.find(id) // EmptyData
```

Services
Service stub pour tests ou mode dégradé

```
service.call() // No-op
```

Points Clés à Retenir

- Supprime les vérifications explicites de null grâce à la refactorisation du code client
- Remplace null par un objet concret via le polymorphisme
- Encapsule un comportement neutre au lieu d'une absence d'objet