Design Pattern

TP

Arnaud Palin Sainte Agathe 24-25

TP 1 : Implémentation d'un Singleton (30 minutes)

Contexte :

Vous travaillez sur une application Node.js où vous devez gérer une configuration unique accessible dans toute l'application.

Objectif :

Implémenter un **Singleton** pour gérer une configuration qui ne doit être instanciée qu'une seule fois dans l'application.

Étapes :

1. **Initialisation :**
   - Créez un projet Node.js en exécutant `npm init -y`.
   - Installez TypeScript si nécessaire : `npm install typescript --save-dev`.
2. **Créer le Singleton :**
   - Dans un fichier `Config.ts`, implémentez une classe `Config` qui stocke des paramètres tels que :
     - `databaseUrl`
     - `port`
   - Assurez-vous que cette classe ne peut être instanciée qu'une seule fois.
3. **Exigences Fonctionnelles :**
   - La classe doit avoir une méthode `getConfig()` pour récupérer la configuration.
   - Elle doit permettre de mettre à jour les paramètres via une méthode `updateConfig()`.
   - Tout changement de configuration doit être accessible depuis toutes les parties de l'application.
4. **Test :**
   - Créez deux fichiers `main1.ts` et `main2.ts`.
   - Importez `Config` dans chaque fichier et vérifiez qu'ils partagent la même instance.

TP 2 : Implémentation d'une Factory (30 minutes)

Contexte :

Vous travaillez sur un système de gestion de contenu pour une plateforme multimédia.

Vous devez créer une **Factory** pour produire différents types de contenu (articles, vidéos, images) sans exposer la logique de création.

Objectif :

Implémenter une **Factory** pour gérer la création de contenu en fonction de son type.

Étapes :

1. **Initialisation :**
   - Utilisez le même projet Node.js que précédemment ou créez-en un nouveau.
   - Créez un fichier `ContentFactory.ts` pour implémenter la factory.
2. **Créer la Factory :**
   - Définissez une interface `Content` avec une méthode `getContentDetails()`.
   - Implémentez des classes pour chaque type de contenu (`Article`, `Video`, `Image`).
   - Créez une classe `ContentFactory` qui génère le bon type de contenu en fonction d'un paramètre.
3. **Exigences Fonctionnelles :**
   - La Factory doit supporter au moins trois types de contenu : `Article`, `Video`, et `Image`.
   - Chaque classe doit avoir des propriétés spécifiques, comme :
     - Article : `title`, `body`
     - Video : `title`, `duration`
     - Image : `url`, `resolution`
4. **Test :**
   - Créez un fichier `main.ts`.
   - Utilisez la Factory pour produire plusieurs contenus et afficher leurs détails.

TP 3: Mise en Pratique des Principes SOLID (1h)

Contexte :

Vous travaillez sur une application de gestion de commandes pour un restaurant. L'objectif est d'implémenter plusieurs fonctionnalités tout en respectant les principes **SOLID** pour garantir un code propre, extensible et maintenable.

Objectifs du TP :

1. Appliquer les cinq principes SOLID :
   - **S**ingle Responsibility Principle (SRP)
   - **O**pen/Closed Principle (OCP)
   - **L**iskov Substitution Principle (LSP)
   - **I**nterface Segregation Principle (ISP)
   - **D**ependency Inversion Principle (DIP)
2. Organiser le code en suivant une approche modulaire et orientée objet.

Contexte Technique :

Vous devez implémenter les fonctionnalités suivantes :

1. Gérer différents types de commandes (repas, boissons, desserts).
2. Calculer le total des commandes.
3. Implémenter des remises pour certains types de clients (VIP, régulier).
4. Ajouter facilement de nouvelles fonctionnalités sans modifier les classes existantes.

Instruction

Instructions :

1. Étape 1 : Respecter le SRP (Single Responsibility Principle)

- Créez une classe `Order` qui stocke les éléments d'une commande.
- Créez une classe `OrderCalculator` qui calcule le total d'une commande.

2. Étape 2 : Respecter le OCP (Open/Closed Principle)

- Implémentez des classes pour différents types de commandes (`Meal`, `Drink`, `Dessert`).
- Ajoutez une méthode `getPrice()` pour calculer le prix de chaque type.
- Permettez d'ajouter de nouveaux types de commandes sans modifier les classes existantes.

3. Étape 3 : Respecter le LSP (Liskov Substitution Principle)

- Assurez-vous que toutes les classes dérivées (ex. : `Meal`, `Drink`, `Dessert`) respectent le comportement attendu de la classe de base `MenuItem`.
- Testez en remplaçant une instance d'une sous-classe par une autre sans casser le programme.

. Étape 4 : Respecter le ISP (Interface Segregation Principle)

- Divisez une interface volumineuse en interfaces spécifiques.
- Exemple : Une interface `Taxable` pour les articles soumis à une taxe spécifique.

Étape 5 : Respecter le DIP (Dependency Inversion Principle)

- Implémentez une abstraction pour les stratégies de remise (`DiscountStrategy`).
- Créez des classes spécifiques pour chaque type de remise.

Tests :

1. Ajoutez des éléments à une commande.
2. Calculez le total sans remise.
3. Appliquez une remise pour un client VIP.
4. Ajoutez un nouveau type de remise (par exemple, une remise de 5 € fixe) et testez sans modifier les classes existantes.

TP 4 : Implémentation du Pattern Observer (40 minutes)

Contexte :

Vous travaillez sur une application Node.js de gestion de notifications. Vous devez implémenter le pattern **Observer** pour permettre à plusieurs modules de réagir à des événements.

Objectif :

- Implémenter un système de notification basé sur le pattern **Observer**.
- Permettre à plusieurs modules (observers) de s'abonner aux notifications.

Instructions :

1. **Étape 1 : Créer la classe `Subject`**
   - Cette classe gère une liste d'observers.
   - Implémentez des méthodes pour :
     - Ajouter un observer.
     - Supprimer un observer.
     - Notifier tous les observers.
2. **Étape 2 : Implémenter des Observers**
   - Implémentez au moins deux observers qui réagissent différemment aux notifications :
     - Un logger qui écrit un message dans la console.
     - Un module d'alerte qui envoie une alerte (simulée).
3. **Étape 3 : Tester le système**
   - Créez une instance de `Subject`.
   - Abonnez vos deux observers.
   - Envoyez une notification pour vérifier leur réaction.

TP 5 : Implémentation du Pattern Strategy (40 minutes)

Contexte :

Vous travaillez sur un calculateur de prix pour un site de commerce en ligne. Différentes stratégies de réduction s'appliquent selon le type de client (standard, VIP, ou promotion).

Objectif :

- Implémenter le pattern **Strategy** pour appliquer différentes stratégies de réduction.
- Permettre au système de choisir dynamiquement la stratégie en fonction du type de client.

Instructions :

1. **Étape 1 : Créer une interface `DiscountStrategy`**
   - Définissez une méthode `calculate(price: number): number`.
2. **Étape 2 : Implémenter des stratégies**
   - Standard (aucune réduction).
   - VIP (10% de réduction).
   - Promotion (5 € de réduction fixe).
3. **Étape 3 : Implémenter un contexte**
   - Créez une classe `PriceCalculator` qui utilise une stratégie pour calculer le prix final.
4. **Étape 4 : Tester le système**
   - Créez une instance de `PriceCalculator`.
   - Changez dynamiquement de stratégie et testez avec différents prix

TP 6 : Implémentation du Pattern Command (40 minutes)

Contexte :

Vous travaillez sur une application qui gère des actions utilisateur dans une interface graphique (annuler, rétablir, exécuter).

Objectif :

- Implémenter le pattern **Command** pour encapsuler des actions dans des objets.
- Permettre d'exécuter, d'annuler et de rétablir des actions.

Instructions :

1. **Étape 1 : Créer une interface `Command`**
   - Méthodes : `execute()` et `undo()`.
2. **Étape 2 : Implémenter des commandes**
   - Ajouter un élément.
   - Supprimer un élément.
3. **Étape 3 : Implémenter un invocateur**
   - Créez une classe `Invoker` pour gérer les commandes et permettre d'annuler ou rétablir les actions.
4. **Étape 4 : Tester le système**
   - Créez plusieurs commandes.
   - Exécutez-les et testez les annulations/rétablissements.

TP Final : Développement d’un Gestionnaire de Tâches Collaboratif

Contexte :

Vous devez réaliser une application front-end collaborative pour gérer des tâches dans une équipe.

L’objectif est d’appliquer les concepts étudiés en intégrant au moins **3 patterns de design** dans le projet.

Objectif :

- Concevoir un **gestionnaire de tâches** en collaborant avec votre binôme.
- Implémenter au moins 3 patterns parmi :
  - **Observer** : Notifications ou synchronisation des tâches.
  - **Strategy** : Différentes stratégies de tri des tâches.
  - **Factory** : Création d'objets tâche avec différentes propriétés.
  - **Command** : Gestion des actions comme ajout, suppression, ou mise à jour d’une tâche.

Fonctionnalités demandées :

1. **Gestion des tâches** :
   - Ajouter une tâche avec un titre et une description.
   - Supprimer une tâche.
   - Marquer une tâche comme terminée.
2. **Tri des tâches** :
   - Implémentez un tri dynamique selon :
     - Tâches terminées.
     - Tâches par ordre alphabétique.
3. **Notifications** :
   - Lorsqu'une tâche est ajoutée, terminée ou supprimée, affichez une notification.

Instructions :

Étape 1 : Créez la structure du projet

1. Utilisez **Node.js** et un framework front-end comme **React** ou **Vue.js**.
2. Créez les fichiers nécessaires pour organiser le projet :
   - `TaskFactory.js` : Implémentation du pattern Factory.
   - `NotificationSystem.js` : Implémentation du pattern Observer.
   - `SortStrategy.js` : Implémentation du pattern Strategy.
   - `TaskManager.js` : Gestion globale des tâches.

Étape 2 : Implémentation des Patterns

1. Pattern Factory (Création des Tâches)

- Créez une classe Factory pour générer des objets tâche avec différentes propriétés.

TP Final : Développement d’un Gestionnaire de Tâches Collaboratif

Livrables :

1. **Code source complet** avec une structure organisée.
2. **Vidéo ou capture d’écran** montrant les fonctionnalités :
   - Création de tâches avec le Factory.
   - Notifications fonctionnelles.
   - Tri dynamique avec les stratégies.

Conseils pour réussir :

- Divisez les tâches entre les deux membres du binôme :
  - L’un peut implémenter les patterns (Factory et Observer).
  - L’autre peut se concentrer sur l'interface utilisateur et le tri (Strategy).
- Testez chaque partie indépendamment avant d’intégrer.

Extensions possibles :

- Ajoutez le pattern **Command** pour gérer les actions d'annulation et de rétablissement.
- Implémentez une persistance des tâches en localStorage.

Rendu :

projet.arnaud@gmail.com -> IIM4-DP-*VotreNom*

Lien vers votre GitHub (avec un README.md avec vos noms, prénom et mail)

Avant Dimanche !