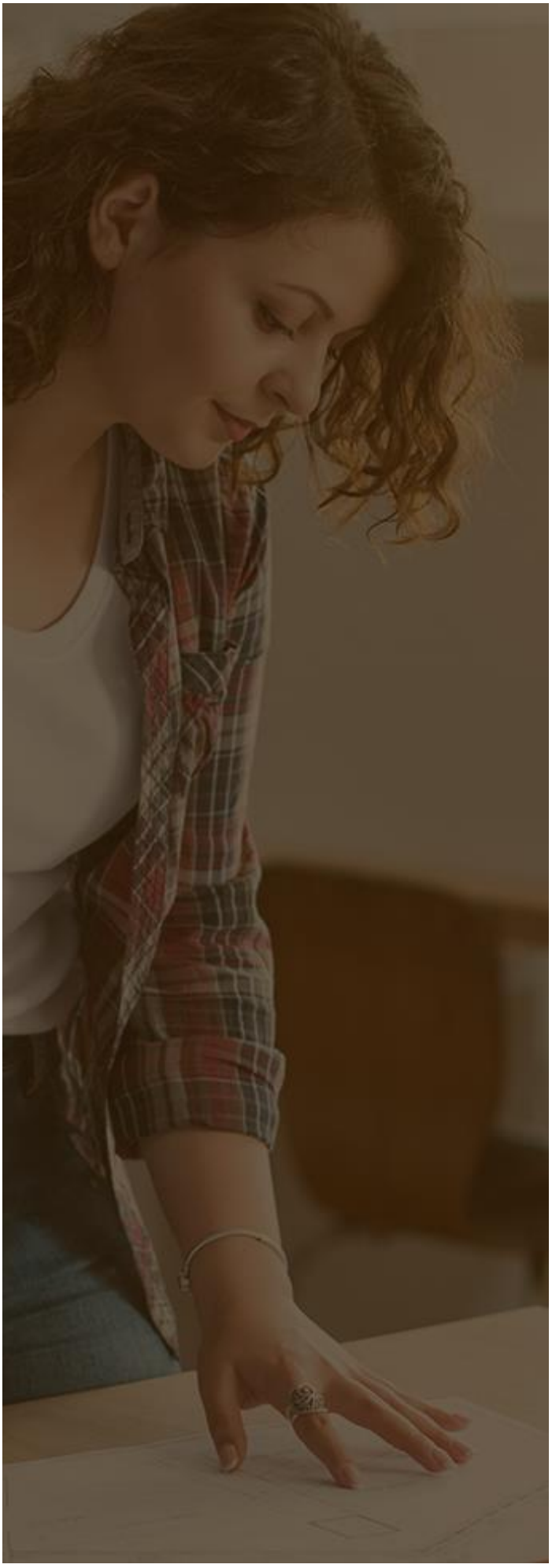
# Projet Programmation système et programmation objet

# Rapport de specification technique

*Projet Academique*



# TABLE DES MATIÈRES

Introduction.....3

1. Design pattern observateur (observer) 4
2. design pattern strategie (strategy).....5
3. design pattern singleton.....6
4. design pattern fabrique(factory).....7
5. design pattern decorateur (decorator) 8

# INTRODUCTION

Dans ce projet de système, nous serons confrontés à la résolution de problèmes en situation d'entreprise. Pour atteindre nos objectifs, nous utiliserons différentes compétences et ressources disponibles. Parmi les concepts clés que nous appliquerons pour résoudre la situation du projet, nous avons identifié cinq patrons de conception qui nous seront utiles :

1. DP Observer
2. DP Strategy
3. DP Singleton
4. DP Factory
5. DP Decorator

# 1. DESIGN PATTERN OBSERVATEUR (OBSERVER)

Le design pattern Observateur est utilisé pour établir une relation de notification entre un sujet (observable) et plusieurs observateurs. Dans votre cas, vous pouvez utiliser ce design pattern pour permettre aux différents clients d'observer le restaurant et de réagir en conséquence. Voici comment vous pouvez le mettre en œuvre :

Définissez une interface IObserver qui déclare une méthode Update() pour mettre à jour les observateurs lorsqu'un événement important se produit.

Implémentez une classe Restaurant qui agira en tant que sujet observé. Elle aura des méthodes pour s'abonner, se désabonner et notifier les observateurs.

Définissez des classes pour les observateurs, par exemple ClientObserver, ChefDeRangObserver, etc. Ces classes implémenteront l'interface IObserver et réagiront aux mises à jour du restaurant.

Le design pattern Observateur vous permettra de gérer les interactions entre les clients et le restaurant de manière flexible et extensible.

# 2. DESIGN PATTERN STRATEGIE (STRATEGY)

Le design pattern Stratégie est utilisé pour encapsuler des algorithmes ou des comportements interchangeables. Dans votre cas, vous pouvez l'utiliser pour définir différentes stratégies de comportement pour les clients dans le restaurant. Par exemple, vous pouvez avoir des stratégies pour les clients calmes, bruyants, pressés, etc. Voici comment vous pouvez l'implémenter :

Créez une interface IClientStrategy qui déclare une méthode Execute() pour définir le comportement des clients.

Implémentez des classes concrètes qui implémentent l'interface IClientStrategy pour chaque type de comportement de client.

Dans votre modèle (Model), ajoutez une propriété ClientStrategy qui représente la stratégie actuelle du client.

Lorsque vous souhaitez exécuter le comportement du client, appelez la méthode Execute() de la stratégie actuelle du client.

Le design pattern Stratégie vous permet de modifier dynamiquement le comportement des clients sans avoir à changer directement leur implémentation.

# 3. DESIGN PATTERN SINGLETON

Le design pattern Singleton est utilisé pour garantir qu'une classe n'a qu'une seule instance et fournir un point d'accès global à cette instance. Dans votre cas, vous pouvez l'utiliser pour des classes qui doivent être uniques, telles que le chef de rang ou le maître d'hôtel. Voici comment vous pouvez l'implémenter :

Ajoutez une méthode statique GetInstance() dans chaque classe qui doit être un singleton, comme ChefDeRang et MaitreDHotel.

À l'intérieur de la méthode GetInstance(), vérifiez si l'instance de la classe existe déjà. Si oui, renvoyez cette instance. Sinon, créez une nouvelle instance et renvoyez-la.

Assurez-vous que le constructeur de la classe est privé pour empêcher la création d'instances supplémentaires en dehors de la méthode GetInstance().

Le design pattern Singleton vous permet d'accéder facilement à une seule instance d'une classe spécifique dans toute votre application.

# 4. DESIGN PATTERN FABRIQUE(FACTORY)

Le design pattern Fabrique est utilisé pour encapsuler la création d'objets et fournir une interface commune pour créer différentes instances de classes dérivées. Dans votre cas, vous pouvez l'utiliser pour créer des clients de différents types de manière flexible. Voici comment vous pouvez l'implémenter :

Créez une interface IClient qui déclare les méthodes communes que tous les types de clients doivent implémenter.

Implémentez des classes concrètes qui représentent différents types de clients, toutes implémentant l'interface IClient.

Créez une classe fabrique ClientFactory qui expose une méthode de création CreateClient() prenant en compte les paramètres spécifiques (nom, groupe, comportement, etc.) et retournant une instance appropriée de IClient en fonction de ces paramètres.

Le design pattern Fabrique vous permet de créer des instances de clients de manière cohérente et centralisée, tout en vous permettant d'ajouter facilement de nouveaux types de clients à l'avenir.

# 5. DESIGN PATTERN DECORATEUR (DECORATOR)

Le modèle Décorateur est un modèle de conception structurel qui permet d'ajouter de nouvelles fonctionnalités à un objet existant de manière dynamique, sans modifier sa structure de base. Il utilise la composition plutôt que l'héritage pour étendre les fonctionnalités d'un objet. Le décorateur enveloppe l'objet d'origine dans un objet décorateur qui fournit des fonctionnalités supplémentaires tout en conservant l'interface de l'objet d'origine. Cela permet d'ajouter ou de supprimer des fonctionnalités à un objet à tout moment, de manière flexible et modulaire. Voici comment vous pouvez l'implémenter :

Lorsqu’on veut utiliser le décorateur, on crée une instance de la classe Plongeur ou Commis, puis vous créez une instance de la classe AideSupplementaire en lui passant l'objet Plongeur ou Commis en tant que paramètre. Ensuite, on utilise cette instance de AideSupplementaire pour appeler la méthode "aider". Cela ajoutera le comportement supplémentaire spécifié dans le décorateur tout en conservant les fonctionnalités de base de la classe Plongeur ou Commis.