**DESCRIPTION DES DIIFERENTS DESIGN PATTERN UTILISES**

1. **FACTORY**

Le besoin de créer des objets variés en fonction de certains paramètres est courant dans les applications. Par exemple, une usine peut produire différents types de produits selon les spécifications fournies. Pour répondre à cette nécessité, le design pattern Factory est utilisé, centralisant et simplifiant le processus de création d'objets en confiant cette tâche à des classes spécialisées appelées "usines". L’objectif principal de ce pattern est d’offrir une interface commune pour la création d’objets, tout en laissant aux sous-classes ou à des méthodes spécifiques le soin de déterminer le type exact d’objet à instancier. De cette manière, le code principal n’a pas besoin de connaître les détails des classes à créer, ce qui améliore la lisibilité, la modularité et facilite les futures modifications ou extensions. Dans notre application, ce concept a été appliqué pour gérer les différents membres de l’équipe de la salle du restaurant, représentée par la classe Staff dans notre diagramme UML. Cette équipe comprend des rôles tels que Serveur, Maître d’hôtel, et d’autres. Grâce au design pattern Factory, nous avons délégué la responsabilité de créer chaque type de membre du personnel à des classes ou méthodes spécifiques. Cela permet de gérer facilement l’ajout de nouveaux rôles ou la modification des rôles existants sans impacter le code principal. En somme, le pattern Factory simplifie et structure le processus de création d'objets complexes tout en maintenant un faible couplage entre les composants.

1. **SINGLETON**

Le pattern Singleton a pour objectif de garantir qu'une seule instance d'une classe existe et d'en offrir un accès global. Pour cela, le constructeur de la classe est rendu privé, ce qui empêche sa création depuis l'extérieur. Une méthode statique, généralement nommée GetInstance, permet d'obtenir cette instance unique. Si elle existe déjà, la méthode la retourne ; sinon, elle en crée une nouvelle. Dans notre application, nous avons utilisé le Singleton pour gérer la connexion à la base de données, assurant ainsi qu'une seule connexion soit maintenue, ce qui évite les problèmes liés à des accès multiples pouvant compromettre l'intégrité des données. Le Singleton a également été appliqué au module de journalisation, centralisant l'accès aux fichiers de logs et empêchant les écritures concurrentes d'endommager ces fichiers. Ce design pattern est particulièrement adapté aux systèmes nécessitant une gestion stricte des ressources partagées ou sensibles.

1. **OBSERVATEUR**

Certaines classes d’une application contiennent des données ou des états qui peuvent changer fréquemment. Lors de ces changements, il est souvent nécessaire de notifier automatiquement d’autres classes dépendantes afin qu’elles puissent réagir en conséquence. Le design pattern Observateur répond à ce besoin en établissant une relation entre un sujet (l’objet observé) et un ensemble d’observateurs.

Ce pattern repose sur un mécanisme d’abonnement : les observateurs s’enregistrent auprès du sujet pour recevoir des notifications chaque fois qu’il est mis à jour. Cela garantit une synchronisation automatique entre l’état du sujet et les actions des observateurs, tout en maintenant un faible couplage entre les classes concernées.

Dans notre application, ce modèle est utilisé pour synchroniser les modifications du modèle métier avec l’interface utilisateur. Par exemple, lorsqu’un état change dans le backend, la vue est immédiatement mise à jour pour refléter ces modifications. De plus, ce pattern est utilisé pour permettre aux clients de communiquer leurs besoins à l’équipe en salle en envoyant des notifications aux membres concernés.

Dans le diagramme de classes, l’implémentation du pattern Observateur repose sur des Events, qui fonctionnent comme des délégués en C# .NET. Ces délégués facilitent la communication entre le sujet et ses observateurs, offrant une méthode efficace pour gérer les notifications tout en préservant la modularité et la maintenabilité du code.

1. **BRIDGE**

Le **design pattern Bridge** est conçu pour découpler une abstraction de son implémentation, permettant ainsi leur évolution indépendante. Plutôt que d'établir une relation statique entre une interface et ses implémentations concrètes, ce modèle introduit un pont qui agit comme un connecteur flexible entre ces deux dimensions. L'abstraction joue alors le rôle d'un médiateur, assurant une interaction cohérente tout en offrant la possibilité de modifier ou d'étendre les implémentations sans perturber la structure globale de l'application.

Dans le contexte présent, le pattern Bridge est utilisé pour gérer les **modes de déplacement** des membres de l’équipe du restaurant. Chaque rôle, qu’il s’agisse d’un **serveur**, d’un **maître d’hôtel** ou d’un **commis**, peut adopter un comportement de déplacement spécifique, tel qu’un déplacement rapide pour gérer une commande urgente ou un déplacement discret pour préserver l’ambiance de la salle. Grâce à cette approche, l'interface définissant les déplacements est découplée de ses implémentations spécifiques. Cela permet une extensibilité accrue, rendant possible l’ajout ou la modification des comportements de déplacement sans impact direct sur la logique principale.

Ce modèle offre une **évolutivité** et une **maintenabilité** remarquables, en permettant de gérer indépendamment les variations d’abstractions (les différents rôles) et d’implémentations (les modes de déplacement). Il en résulte un code plus modulaire et facile à maintenir, évitant les duplications et facilitant l’intégration de nouveaux éléments, tels qu’un rôle supplémentaire ou une méthode de déplacement innovante.

1. **STRATEGY**

Le **design pattern Strategy** est utilisé pour permettre à une classe d'adapter dynamiquement son comportement ou celui d'une de ses méthodes en fonction d'un contexte spécifique. Ce modèle repose sur l'encapsulation des différentes stratégies, où chaque stratégie représente une manière distincte d'exécuter une opération, dans des classes séparées. La classe principale interagit uniquement avec une **interface commune**, ce qui simplifie la sélection, la substitution ou l'extension des stratégies à exécuter.

Dans le cadre de notre application, ce pattern est appliqué pour gérer les **modes de commande des clients**. Chaque client peut adopter une méthode de commande spécifique selon ses préférences : par exemple, certains préfèrent passer leur commande complète en une seule fois, tandis que d'autres optent pour une commande progressive, ajoutant des plats au fur et à mesure. Grâce au design pattern Strategy, ces différentes méthodes sont encapsulées dans des **stratégies indépendantes**, permettant au système de s’adapter de manière flexible au comportement du client.

Ce modèle favorise une **modularité accrue** en isolant les logiques de commande distinctes, ce qui simplifie l’ajout ou la modification de nouvelles stratégies sans impacter la classe principale. De plus, il renforce la **maintenabilité** en réduisant les dépendances, en limitant le couplage entre les composants et en favorisant la réutilisation des stratégies dans d’autres parties de l’application si nécessaire.