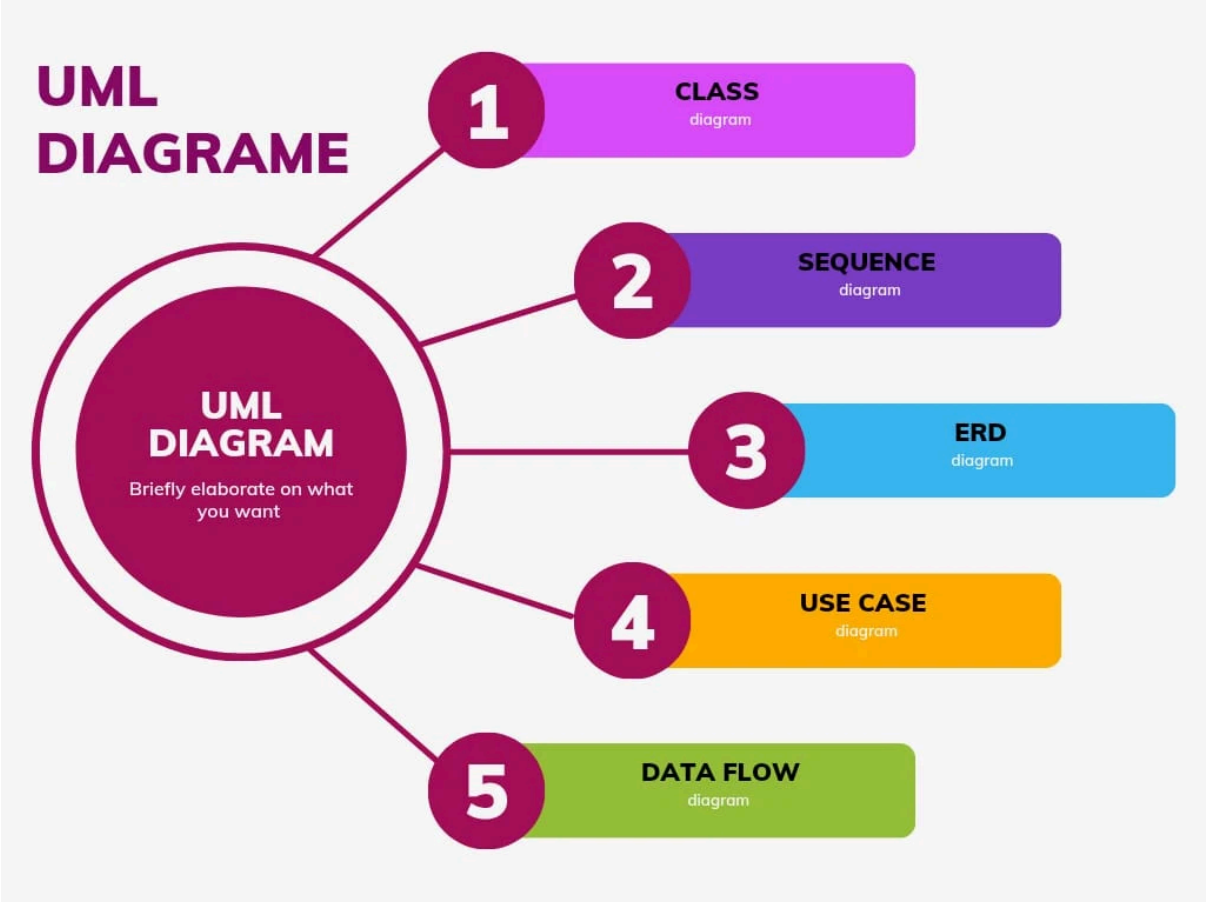
Kit d’exercices sur UML



- Vue Globale et Réalisme du Système
- PlantUML (Un must have !)

Accéder à l'adresse suivante PlantUML Web Server (https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) et copier coller le code suivant :

4. Système de Gestion de l'École
   - Contexte :
   - Tâches à réaliser :
   - Étapes à suivre :
   - Conseils pour la réalisation :

Correction :
   - Packages et Classes :
   - Relations :
   - Diagramme UMLet
   - Points informations utiles

Si les méthodes représentent des actions très différentes, il pourrait être intéressant de scinder l'interface en plusieurs interfaces plus petites et plus ciblées, chacune représentant un ensemble cohérent de responsabilités (principe de séparation des interfaces, ISP - Interface Segregation Principle).

Modélisation graphique
   - Explications
     - Packages, Interfaces et Classes
     - Relations, Compositions et Agrégations
     - Services Tiers
   - PlantUML (Un must have !)

Accéder à l'adresse suivante PlantUML Web Server (https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) et copier coller le code suivant :

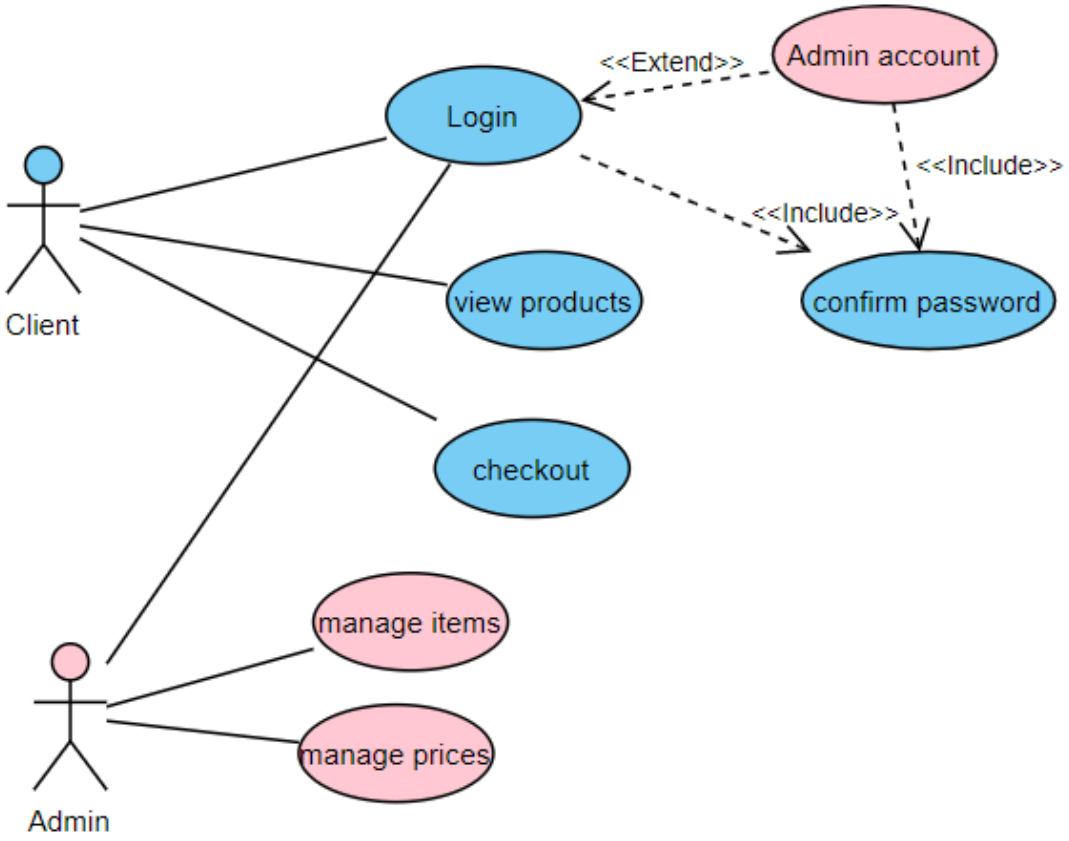
5. Système de Gestion de Projet
   - Contexte :
   - Tâches à réaliser :
   - Étapes à suivre :
   - Conseils pour la réalisation :

Chers étudiants,

Bienvenue à cette série d'exercices sur la modélisation UML (Unified Modeling Language). Dans ces exercices, nous allons explorer deux types de diagrammes fondamentaux : les diagrammes de cas d'utilisation (UML Use Case Diagrams) et les diagrammes de classes (UML Class Diagrams). Ces outils vous permettront de mieux comprendre et concevoir les systèmes logiciels de manière structurée et claire. Pour ces exercices, nous utiliserons le logiciel UMLet, un outil pratique et intuitif pour créer des diagrammes UML. Assurez-vous de bien suivre les instructions fournies et de modéliser chaque aspect avec précision.

Bonne modélisation !

LE DIAGRAMME DE CAS D'UTILISATION (USE CASE)



Chers étudiants,

La correction fournie ici est une parmi tant d'autres façons de structurer et d'implémenter vos projets en Java. Il existe de nombreuses approches possibles pour résoudre ces exercices, chacune avec ses propres avantages et inconvénients. L'important est de comprendre les concepts sous-jacents et de pouvoir expliquer clairement les choix que vous avez faits dans votre implémentation.
N'hésitez pas à explorer différentes solutions et à défendre vos choix techniques de manière réfléchie et argumentée. L'objectif est d'acquérir une solide compréhension des principes de la programmation orientée objet et de devenir des développeurs flexibles et adaptables.

Bonne continuation dans votre apprentissage !

1. Système de Gestion de Bibliothèque

Contexte :

Vous êtes un analyste junior chez **BiblioSoft Solutions**, une entreprise spécialisée dans le développement de logiciels pour les bibliothèques. Votre mission est de modéliser un nouveau système de gestion pour la bibliothèque municipale. Ce système doit permettre aux utilisateurs de rechercher des livres, emprunter des livres, retourner des livres, s'inscrire à la bibliothèque, et recevoir des notifications sur les nouveaux livres et les rappels de retour.

Les utilisateurs du système incluent les membres de la bibliothèque (utilisateurs) et le personnel de la bibliothèque (bibliothécaires). Vous devez créer un diagramme de cas d'utilisation pour représenter ces interactions.

Tâches à réaliser :

1. **Créer un diagramme de cas d'utilisation pour le système de gestion de bibliothèque en utilisant UMLet.**

Étapes à suivre :

1. **Ouvrir UMLet et créer un nouveau diagramme de cas d'utilisation.**
2. **Définir les acteurs principaux :**
    - **Utilisateur** : Un membre de la bibliothèque qui peut rechercher, emprunter, et retourner des livres, s'inscrire à la bibliothèque et recevoir des notifications.
    - **Bibliothécaire** : Un membre du personnel de la bibliothèque avec des responsabilités supplémentaires, comme la gestion des livres et la vérification des retours.
3. **Ajouter les cas d'utilisation suivants :**

- **Rechercher des livres** : L'utilisateur peut rechercher des livres dans le catalogue de la bibliothèque.
- **Emprunter des livres** : L'utilisateur peut emprunter des livres disponibles.
- **Retourner des livres** : L'utilisateur peut retourner les livres empruntés.
- **S'inscrire à la bibliothèque** : Une personne peut s'inscrire pour devenir membre de la bibliothèque.
- **Recevoir des notifications** : L'utilisateur peut recevoir des notifications sur les nouveaux livres et les rappels de retour.

4. **Relier les acteurs aux cas d'utilisation appropriés :**
   - L'utilisateur doit être lié à tous les cas d'utilisation mentionnés.
   - Le bibliothécaire doit également être lié à ces cas d'utilisation, en plus de pouvoir gérer le système.
5. **Ajouter une relation de généralisation entre "Utilisateur" et "Bibliothécaire" :**
   - Le bibliothécaire est un type d'utilisateur avec des responsabilités supplémentaires.

Conseils pour la réalisation :

Lorsque vous créez votre diagramme de cas d'utilisation, assurez-vous de bien comprendre les rôles et les responsabilités de chaque acteur.

Utilisez des noms clairs et descriptifs pour chaque cas d'utilisation afin de rendre le diagramme facile à comprendre. Pensez à la manière dont chaque action interagit avec le système global et représentez ces interactions avec précision. N'oubliez pas que la relation de généralisation montre que le bibliothécaire peut effectuer toutes les actions d'un utilisateur standard, ainsi que des tâches supplémentaires liées à la gestion de la bibliothèque. Prenez le temps de bien organiser votre diagramme pour qu'il soit propre et lisible.

Bonne chance !

Correction :

Acteurs :

- **Utilisateur** : Un membre de la bibliothèque.
- **Bibliothécaire** : Un utilisateur avec des responsabilités supplémentaires.

Cas d'utilisation :

- **Rechercher des livres** : L'utilisateur peut rechercher des livres dans le catalogue de la bibliothèque.
- **Emprunter des livres** : L'utilisateur peut emprunter des livres disponibles.
- **Retourner des livres** : L'utilisateur peut retourner les livres empruntés.
- **S'inscrire à la bibliothèque** : Une personne peut s'inscrire pour devenir membre de la bibliothèque.
- **Recevoir des notifications** : L'utilisateur peut recevoir des notifications sur les nouveaux livres et les rappels de retour.

Relations :

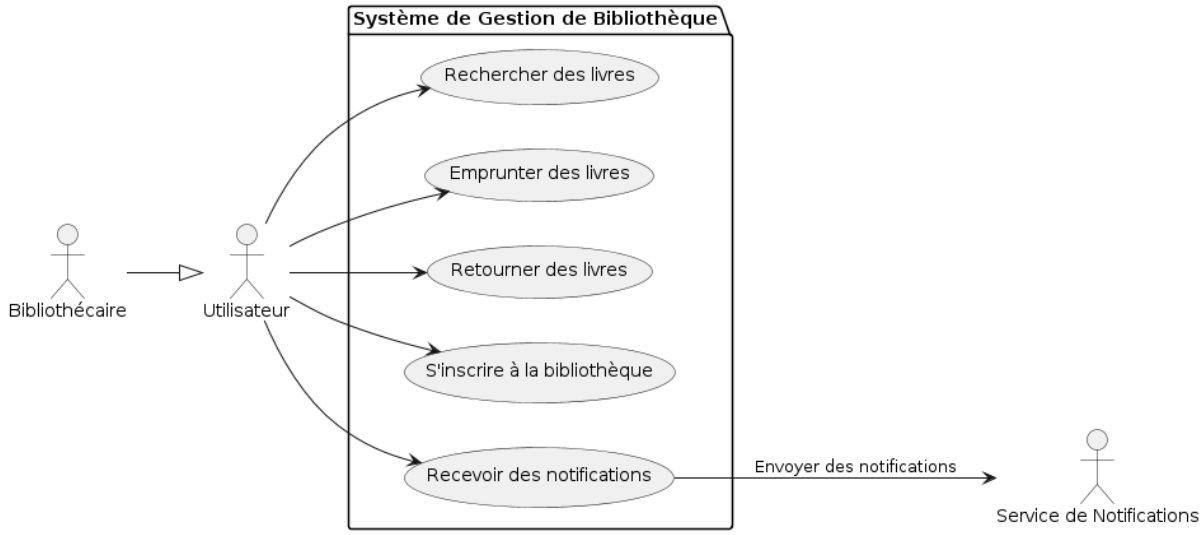
- **Utilisateur** :
    - Rechercher des livres
    - Emprunter des livres
    - Retourner des livres
    - S'inscrire à la bibliothèque
    - Recevoir des notifications
- **Bibliothécaire** : Hérite de l'acteur Utilisateur, donc peut effectuer toutes les actions d'un utilisateur, plus des actions supplémentaires liées à la gestion de la bibliothèque (non spécifiées dans cet exercice simple).

Diagramme UMLet

Pour créer le diagramme dans UMLet, suivez ces étapes :

1. **Créer les acteurs :**
    - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Utilisateur".
    - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Bibliothécaire".
2. **Créer les cas d'utilisation :**
    - Sélectionnez l'élément "Use Case" et placez-le sur le canevas. Ajoutez les noms des cas d'utilisation : "Rechercher des livres", "Emprunter des livres", "Retourner des livres", "S'inscrire à la bibliothèque", "Recevoir des notifications".
3. **Relier les acteurs aux cas d'utilisation :**
    - Utilisez l'élément "Line" pour relier chaque acteur aux cas d'utilisation pertinents. Par exemple, reliez "Utilisateur" à tous les cas d'utilisation, et "Bibliothécaire" à tous les cas d'utilisation.
4. **Ajouter la généralisation :**
    - Utilisez l'élément "Generalization" pour montrer que "Bibliothécaire" est un type de "Utilisateur". Reliez "Bibliothécaire" à "Utilisateur" avec une flèche indiquant la généralisation.

Modélisation graphique



PlantUML (Un must have !)

Chers étudiants,

Je tiens à vous informer de l'existence d'un outil en ligne très pratique pour ceux d'entre vous qui préfèrent coder plutôt que d'utiliser une interface graphique pour créer des diagrammes UML. Bien que UMLet soit un excellent outil pour concevoir vos diagrammes de manière visuelle, je vous recommande d'essayer PlantUML(https://plantuml.com/fr/).
PlantUML permet de générer des diagrammes UML (et pleins d'autres choses utiles pour les développeurs) directement à partir de texte. Cela peut être particulièrement utile si vous n'aimez pas les interfaces graphiques ou si vous souhaitez intégrer la génération de diagrammes dans votre workflow de développement.
En utilisant PlantUML, vous pouvez écrire du code simple pour définir vos diagrammes, ce qui peut également faciliter la version et la collaboration sur vos projets.

N'hésitez pas à explorer cette alternative pour voir si elle correspond mieux à vos préférences et à votre façon de travailler.

Accéder à l'adresse suivante PlantUML Web Server (https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) et copier coller le code suivant :

```
@startuml
left to right direction

actor Utilisateur
actor Bibliothécaire
actor "Service de Notifications" as NotificationService

package "Système de Gestion de Bibliothèque" {
    usecase "Rechercher des livres" as UC1
    usecase "Emprunter des livres" as UC2
    usecase "Retourner des livres" as UC3
    usecase "S'inscrire à la bibliothèque" as UC4
    usecase "Recevoir des notifications" as UC5
}

Utilisateur --> UC1
Utilisateur --> UC2
Utilisateur --> UC3
Utilisateur --> UC4
Utilisateur --> UC5

Bibliothécaire --|> Utilisateur

UC5 --> NotificationService : "Envoyer des notifications"

@enduml
```

2. Système de Gestion de Restaurant

Contexte :

Vous travaillez comme analyste junior chez **GourmetSoft Solutions**, une entreprise spécialisée dans le développement de logiciels pour la restauration. Votre mission est de modéliser un nouveau système de gestion pour un restaurant. Ce système doit permettre aux clients de réserver une table, commander des plats, payer l'addition, et recevoir des notifications sur les offres spéciales.

Les utilisateurs du système incluent les clients et le personnel du restaurant (serveurs et chefs). Vous devez créer un diagramme de cas d'utilisation pour représenter ces interactions.

Tâches à réaliser :

1. **Créer un diagramme de cas d'utilisation pour le système de gestion de restaurant en utilisant UMLet.**

Étapes à suivre :

1. **Ouvrir UMLet et créer un nouveau diagramme de cas d'utilisation.**
2. **Définir les acteurs principaux :**
    - **Client** : Un client du restaurant qui peut réserver une table, commander des plats, payer l'addition et recevoir des notifications.
    - **Serveur** : Un membre du personnel du restaurant qui prend les commandes, sert les plats, et gère les paiements.
    - **Chef** : Un membre du personnel du restaurant qui prépare les plats commandés.
3. **Ajouter les cas d'utilisation suivants :**
    - **Réserver une table** : Le client peut réserver une table dans le restaurant.

- **Commander des plats** : Le client peut commander des plats disponibles au menu.
- **Payer l'addition** : Le client peut payer l'addition après avoir consommé les plats.
- **Recevoir des notifications** : Le client peut recevoir des notifications sur les offres spéciales et les nouveaux plats.

4. **Relier les acteurs aux cas d'utilisation appropriés :**
   - Le client doit être lié à tous les cas d'utilisation mentionnés.
   - Le serveur doit être lié aux cas d'utilisation "Commander des plats" et "Payer l'addition".
   - Le chef doit être lié au cas d'utilisation "Commander des plats".

Conseils pour la réalisation :

Lorsque vous créez votre diagramme de cas d'utilisation, assurez-vous de bien comprendre les rôles et les responsabilités de chaque acteur. Utilisez des noms clairs et descriptifs pour chaque cas d'utilisation afin de rendre le diagramme facile à comprendre. Pensez à la manière dont chaque action interagit avec le système global et représentez ces interactions avec précision. Prenez le temps de bien organiser votre diagramme pour qu'il soit propre et lisible.

Bonne chance !

Correction :

Acteurs :

- **Client** : Un client du restaurant.
- **Serveur** : Un membre du personnel du restaurant.
- **Chef** : Un membre du personnel du restaurant.

Cas d'utilisation :

- **Réserver une table** : Le client peut réserver une table dans le restaurant.
- **Commander des plats** : Le client peut commander des plats disponibles au menu.
- **Payer l'addition** : Le client peut payer l'addition après avoir consommé les plats.
- **Recevoir des notifications** : Le client peut recevoir des notifications sur les offres spéciales et les nouveaux plats.

Relations :

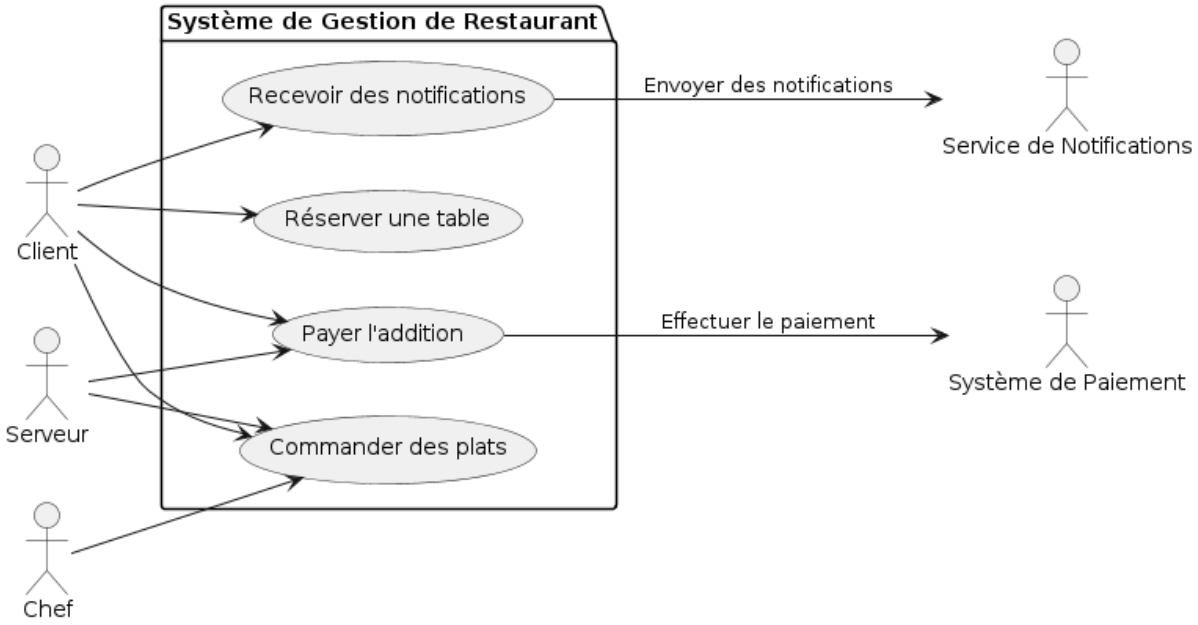
- **Client** :
    - Réserver une table
    - Commander des plats
    - Payer l'addition
    - Recevoir des notifications
- **Serveur** :
    - Commander des plats
    - Payer l'addition
- **Chef** :
    - Commander des plats

Diagramme UMLet

Pour créer le diagramme dans UMLet, suivez ces étapes :

1. **Créer les acteurs :**
    - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Client".
    - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Serveur".
    - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Chef".
2. **Créer les cas d'utilisation :**
    - Sélectionnez l'élément "Use Case" et placez-le sur le canevas. Ajoutez les noms des cas d'utilisation : "Réserver une table", "Commander des plats", "Payer l'addition", "Recevoir des notifications".
3. **Relier les acteurs aux cas d'utilisation :**
    - Utilisez l'élément "Line" pour relier chaque acteur aux cas d'utilisation pertinents. Par exemple, reliez "Client" à tous les cas d'utilisation, "Serveur" à "Commander des plats" et "Payer l'addition", et "Chef" à "Commander des plats".

Modélisation graphique



PlantUML (Un must have !)

Accéder à l'adresse suivante PlantUML Web Server (https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) et copier coller le code suivant :

```
@startuml
left to right direction

actor Client
actor Serveur
actor Chef
actor "Service de Notifications" as NotificationService
actor "Système de Paiement" as PaymentSystem

package "Système de Gestion de Restaurant" {
    usecase "Réserver une table" as UC1
    usecase "Commander des plats" as UC2
    usecase "Payer l'addition" as UC3
    usecase "Recevoir des notifications" as UC4
}

Client --> UC1
Client --> UC2
Client --> UC3
Client --> UC4

Serveur --> UC2
Serveur --> UC3

Chef --> UC2

UC3 --> PaymentSystem : "Effectuer le paiement"
UC4 --> NotificationService : "Envoyer des notifications"

@enduml
```

3.Système de Gestion de d'Hôpital

Contexte :

Vous travaillez comme analyste junior chez **MediSoft Solutions**, une entreprise spécialisée dans le développement de logiciels pour les hôpitaux. Votre mission est de modéliser un nouveau système de gestion pour un hôpital. Ce système doit permettre aux patients de prendre des rendez-vous, consulter leurs dossiers médicaux, et recevoir des notifications sur leurs rendez-vous. Le personnel médical (médecins et infirmières) doit pouvoir gérer les rendez-vous et consulter les dossiers médicaux des patients.

Tâches à réaliser :

1. **Créer un diagramme de cas d'utilisation pour le système de gestion d'hôpital en utilisant UMLet.**

Étapes à suivre :

1. **Ouvrir UMLet et créer un nouveau diagramme de cas d'utilisation.**
2. **Définir les acteurs principaux :**
    - **Patient** : Un patient de l'hôpital qui peut prendre des rendez-vous, consulter ses dossiers médicaux et recevoir des notifications.
    - **Médecin** : Un membre du personnel médical qui peut gérer les rendez-vous et consulter les dossiers médicaux des patients.
    - **Infirmière** : Un membre du personnel médical qui peut gérer les rendez-vous et consulter les dossiers médicaux des patients.
3. **Ajouter les cas d'utilisation suivants :**
    - **Prendre un rendez-vous** : Le patient peut prendre un rendez-vous médical.

- **Consulter les dossiers médicaux** : Le patient peut consulter ses dossiers médicaux.
- **Gérer les rendez-vous** : Le médecin et l'infirmière peuvent gérer les rendez-vous des patients.
- **Consulter les dossiers médicaux des patients** : Le médecin et l'infirmière peuvent consulter les dossiers médicaux des patients.
- **Recevoir des notifications** : Le patient peut recevoir des notifications sur ses rendez-vous.

4. **Relier les acteurs aux cas d'utilisation appropriés :**
   - Le patient doit être lié aux cas d'utilisation "Prendre un rendez-vous", "Consulter les dossiers médicaux", et "Recevoir des notifications".
   - Le médecin et l'infirmière doivent être liés aux cas d'utilisation "Gérer les rendez-vous" et "Consulter les dossiers médicaux des patients".

Conseils pour la réalisation :

Lorsque vous créez votre diagramme de cas d'utilisation, assurez-vous de bien comprendre les rôles et les responsabilités de chaque acteur. Utilisez des noms clairs et descriptifs pour chaque cas d'utilisation afin de rendre le diagramme facile à comprendre. Pensez à la manière dont chaque action interagit avec le système global et représentez ces interactions avec précision. Prenez le temps de bien organiser votre diagramme pour qu'il soit propre et lisible.

Bonne chance !

Correction :

Acteurs :

- **Patient** : Un patient de l'hôpital.
- **Médecin** : Un membre du personnel médical.
- **Infirmière** : Un membre du personnel médical.

Cas d'utilisation :

- **Prendre un rendez-vous** : Le patient peut prendre un rendez-vous médical.
- **Consulter les dossiers médicaux** : Le patient peut consulter ses dossiers médicaux.
- **Gérer les rendez-vous** : Le médecin et l'infirmière peuvent gérer les rendez-vous des patients.
- **Consulter les dossiers médicaux des patients** : Le médecin et l'infirmière peuvent consulter les dossiers médicaux des patients.
- **Recevoir des notifications** : Le patient peut recevoir des notifications sur ses rendez-vous.

Relations :

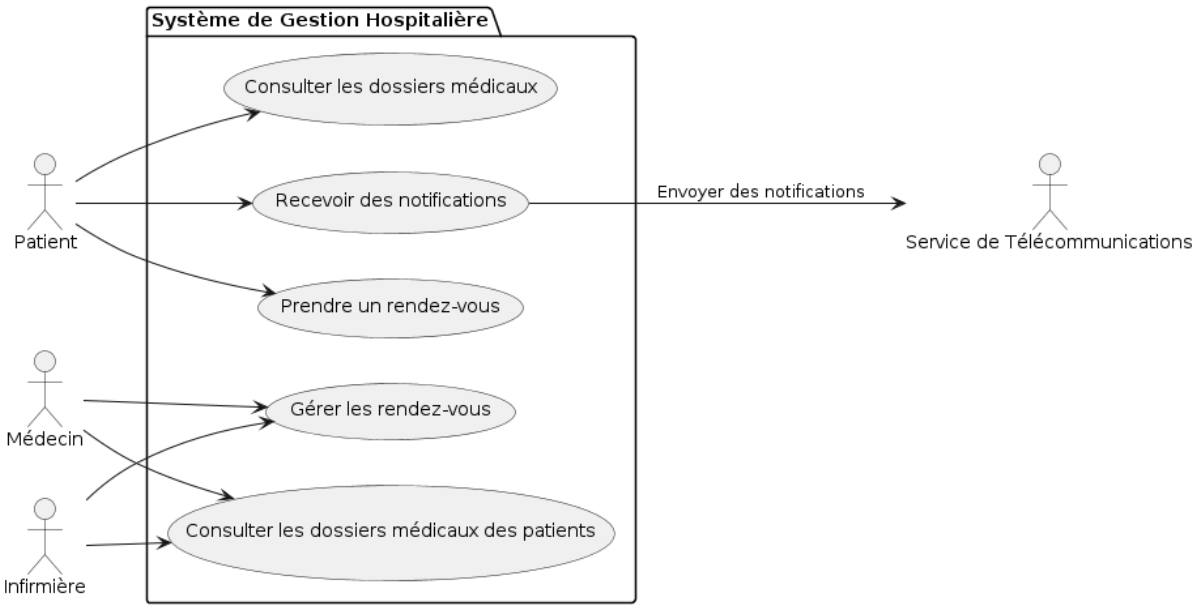
- **Patient** :
    - Prendre un rendez-vous
    - Consulter les dossiers médicaux
    - Recevoir des notifications
- **Médecin** :
    - Gérer les rendez-vous
    - Consulter les dossiers médicaux des patients
- **Infirmière** :
    - Gérer les rendez-vous
    - Consulter les dossiers médicaux des patients

Diagramme UMLet

Pour créer le diagramme dans UMLet, suivez ces étapes :

1. **Créer les acteurs :**
    - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Patient".
    - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Médecin".
    - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Infirmière".
2. **Créer les cas d'utilisation :**
    - Sélectionnez l'élément "Use Case" et placez-le sur le canevas. Ajoutez les noms des cas d'utilisation : "Prendre un rendez-vous", "Consulter les dossiers médicaux", "Gérer les rendez-vous", "Consulter les dossiers médicaux des patients", "Recevoir des notifications".
3. **Relier les acteurs aux cas d'utilisation :**
    - Utilisez l'élément "Line" pour relier chaque acteur aux cas d'utilisation pertinents. Par exemple, reliez "Patient" aux cas d'utilisation "Prendre un rendez-vous", "Consulter les dossiers médicaux", et "Recevoir des notifications". Reliez "Médecin" et "Infirmière" aux cas d'utilisation "Gérer les rendez-vous" et "Consulter les dossiers médicaux des patients".

Modélisation graphique



PlantUML (Un must have !)

Accéder à l'adresse suivante PlantUML Web Server (https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) et copier coller le code suivant :

```
@startuml
left to right direction

actor Patient
actor Médecin
actor Infirmière
actor "Service de Télécommunications" as TelecomService

package "Système de Gestion Hospitalière" {
    usecase "Prendre un rendez-vous" as UC1
    usecase "Consulter les dossiers médicaux" as UC2
    usecase "Recevoir des notifications" as UC3
    usecase "Gérer les rendez-vous" as UC4
    usecase "Consulter les dossiers médicaux des patients" as UC5
}

Patient --> UC1
Patient --> UC2
Patient --> UC3

Médecin --> UC4
Médecin --> UC5

Infirmière --> UC4
Infirmière --> UC5

UC3 --> TelecomService : "Envoyer des notifications"

@enduml
```

4. Système de Gestion de Projet

Contexte :

Vous êtes analyste chez **ProjectPlus Inc.**, une entreprise spécialisée dans les outils de gestion de projet pour les grandes organisations. Votre mission est de modéliser un nouveau système de gestion de projet. Ce système doit permettre aux utilisateurs de créer des projets, ajouter des tâches, assigner des tâches aux membres de l'équipe, suivre l'avancement des tâches, et générer des rapports de projet.

Les utilisateurs du système incluent les chefs de projet, les membres de l'équipe et les administrateurs système. Vous devez créer un diagramme de cas d'utilisation pour représenter ces interactions.

Tâches à réaliser :

1. **Créer un diagramme de cas d'utilisation pour le système de gestion de projet en utilisant UMLet.**

Étapes à suivre :

1. **Ouvrir UMLet et créer un nouveau diagramme de cas d'utilisation.**
2. **Définir les acteurs principaux :**
    - **Chef de Projet** : Un utilisateur qui peut créer des projets, ajouter et assigner des tâches, suivre l'avancement des tâches, et générer des rapports.
    - **Membre de l'Équipe** : Un utilisateur qui peut consulter les tâches assignées, mettre à jour l'état des tâches, et consulter les rapports de projet.
    - **Administrateur Système** : Un utilisateur qui peut gérer les comptes utilisateurs et configurer les paramètres du système.
3. **Ajouter les cas d'utilisation suivants :**

- **Créer un projet** : Le chef de projet peut créer un nouveau projet.
- **Ajouter des tâches** : Le chef de projet peut ajouter des tâches à un projet.
- **Assigner des tâches** : Le chef de projet peut assigner des tâches aux membres de l'équipe.
- **Suivre l'avancement des tâches** : Le chef de projet et les membres de l'équipe peuvent suivre l'avancement des tâches.
- **Générer des rapports** : Le chef de projet et les membres de l'équipe peuvent générer des rapports de projet.
- **Gérer les comptes utilisateurs** : L'administrateur système peut gérer les comptes utilisateurs.
- **Configurer les paramètres du système** : L'administrateur système peut configurer les paramètres du système.

4. **Relier les acteurs aux cas d'utilisation appropriés :**
   - Le chef de projet doit être lié à tous les cas d'utilisation excepté ceux de l'administrateur système.
   - Le membre de l'équipe doit être lié aux cas d'utilisation "Suivre l'avancement des tâches" et "Générer des rapports".
   - L'administrateur système doit être lié aux cas d'utilisation "Gérer les comptes utilisateurs" et "Configurer les paramètres du système".

Conseils pour la réalisation :

Lorsque vous créez votre diagramme de cas d'utilisation, assurez-vous de bien comprendre les rôles et les responsabilités de chaque acteur. Utilisez des noms clairs et descriptifs pour chaque cas d'utilisation afin de rendre le diagramme facile à comprendre. Pensez à la manière dont chaque action interagit avec le système global et représentez ces interactions avec précision. Prenez le temps de bien organiser votre diagramme pour qu'il soit propre et lisible.

Bonne chance !

Correction :

Acteurs :

- **Chef de Projet** : Un utilisateur qui peut créer des projets, ajouter et assigner des tâches, suivre l'avancement des tâches, et générer des rapports.
- **Membre de l'Équipe** : Un utilisateur qui peut consulter les tâches assignées, mettre à jour l'état des tâches, et consulter les rapports de projet.
- **Administrateur Système** : Un utilisateur qui peut gérer les comptes utilisateurs et configurer les paramètres du système.

Cas d'utilisation :

- **Créer un projet** : Le chef de projet peut créer un nouveau projet.
- **Ajouter des tâches** : Le chef de projet peut ajouter des tâches à un projet.
- **Assigner des tâches** : Le chef de projet peut assigner des tâches aux membres de l'équipe.
- **Suivre l'avancement des tâches** : Le chef de projet et les membres de l'équipe peuvent suivre l'avancement des tâches.
- **Générer des rapports** : Le chef de projet et les membres de l'équipe peuvent générer des rapports de projet.
- **Gérer les comptes utilisateurs** : L'administrateur système peut gérer les comptes utilisateurs.
- **Configurer les paramètres du système** : L'administrateur système peut configurer les paramètres du système.

Relations :

- **Chef de Projet** :
    - Créer un projet
    - Ajouter des tâches

    - Assigner des tâches
    - Suivre l'avancement des tâches
    - Générer des rapports
- **Membre de l'Équipe** :
    - Suivre l'avancement des tâches
    - Générer des rapports
- **Administrateur Système** :
    - Gérer les comptes utilisateurs
    - Configurer les paramètres du système

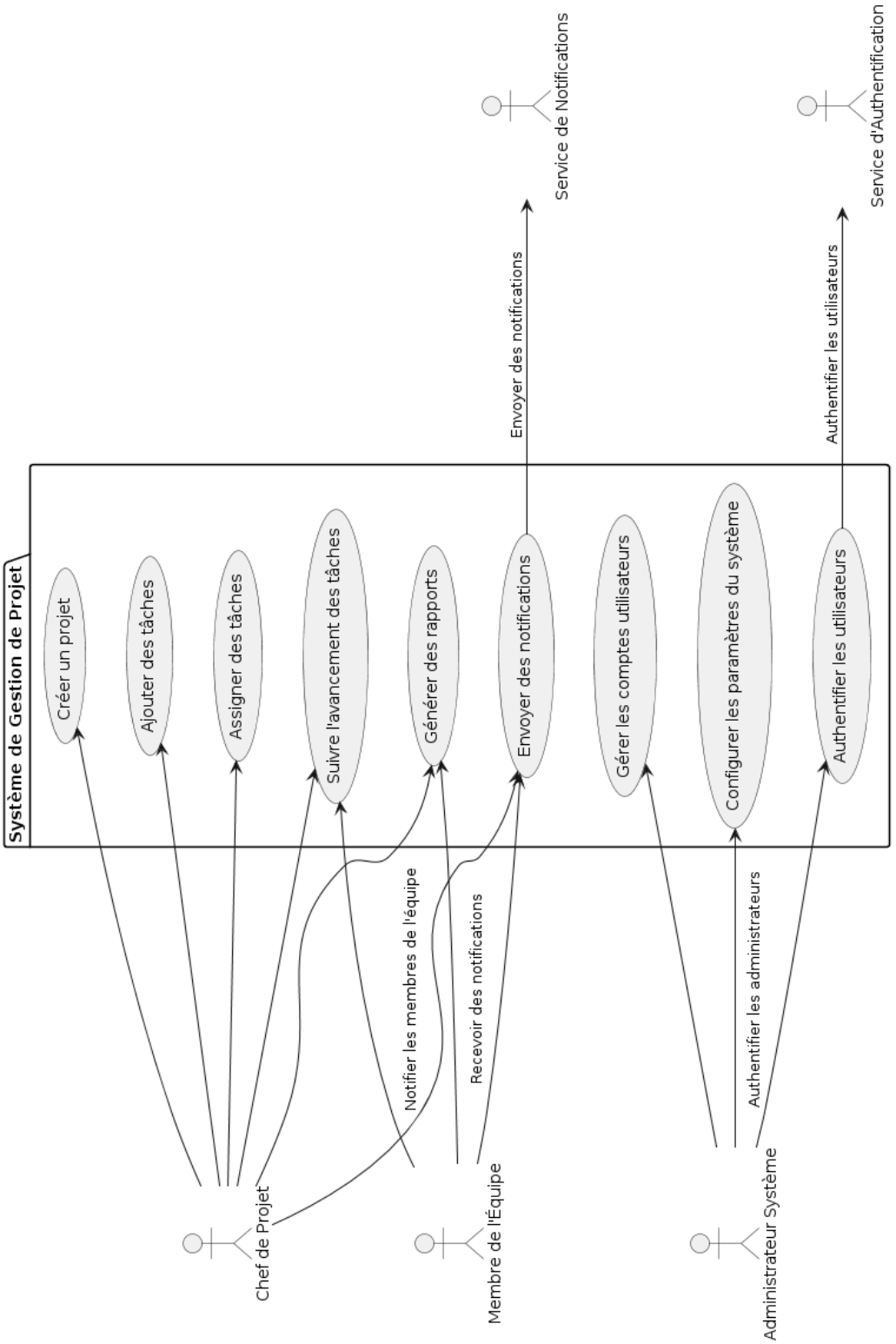
Diagramme UMLet

Pour créer le diagramme dans UMLet, suivez ces étapes :

1. **Créer les acteurs :**
    - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Chef de Projet".
    - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Membre de l'Équipe".
    - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Administrateur Système".
2. **Créer les cas d'utilisation :**
    - Sélectionnez l'élément "Use Case" et placez-le sur le canevas. Ajoutez les noms des cas d'utilisation : "Créer un projet", "Ajouter des tâches", "Assigner des tâches", "Suivre l'avancement des tâches", "Générer des rapports", "Gérer les comptes utilisateurs", "Configurer les paramètres du système".
3. **Relier les acteurs aux cas d'utilisation :**
    - Utilisez l'élément "Line" pour relier chaque acteur aux cas d'utilisation pertinents. Par exemple, reliez "Chef de Projet" à "Créer un projet", "Ajouter des tâches", "Assigner des tâches", "Suivre l'avancement des tâches", et "Générer des rapports". Reliez "Membre de l'Équipe" à "Suivre l'avancement des tâches" et "Générer des rapports". Reliez "Administrateur Système" à

"Gérer les comptes utilisateurs" et "Configurer les paramètres du système".

Modélisation graphique



PlantUML (Un must have !)

Accéder à l'adresse suivante PlantUML Web Server (https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) et copier coller le code suivant :

```
@startuml
left to right direction

actor "Chef de Projet" as ChefProjet
actor "Membre de l'Équipe" as MembreEquipe
actor "Administrateur Système" as AdminSysteme
actor "Service de Notifications" as NotificationService
actor "Service d'Authentification" as AuthService

package "Système de Gestion de Projet" {
    usecase "Créer un projet" as UC1
    usecase "Ajouter des tâches" as UC2
    usecase "Assigner des tâches" as UC3
    usecase "Suivre l'avancement des tâches" as UC4
    usecase "Générer des rapports" as UC5
    usecase "Gérer les comptes utilisateurs" as UC6
    usecase "Configurer les paramètres du système" as UC7
    usecase "Envoyer des notifications" as UC8
    usecase "Authentifier les utilisateurs" as UC9
}

ChefProjet --> UC1
ChefProjet --> UC2
ChefProjet --> UC3
ChefProjet --> UC4
ChefProjet --> UC5

MembreEquipe --> UC4
MembreEquipe --> UC5

AdminSysteme --> UC6
AdminSysteme --> UC7

UC8 --> NotificationService : "Envoyer des notifications"
```

```
UC9 --> AuthService : "Authentifier les utilisateurs"

ChefProjet --> UC8 : "Notifier les membres de l'équipe"
MembreEquipe --> UC8 : "Recevoir des notifications"
AdminSysteme --> UC9 : "Authentifier les administrateurs"

@enduml
```

5. Système de Commerce Électronique

Contexte :

Vous travaillez comme analyste chez **E-Shop Solutions**, une entreprise spécialisée dans les plateformes de commerce électronique. Votre mission est de modéliser un nouveau système de gestion pour une plateforme de commerce électronique. Ce système doit permettre aux utilisateurs de créer un compte, naviguer dans les produits, ajouter des produits au panier, passer des commandes, effectuer des paiements, et suivre les commandes. Les administrateurs doivent pouvoir gérer les produits, les utilisateurs et les commandes.

Tâches à réaliser :

1. **Créer un diagramme de cas d'utilisation pour le système de commerce électronique en utilisant UMLet.**

Étapes à suivre :

1. **Ouvrir UMLet et créer un nouveau diagramme de cas d'utilisation.**
2. **Définir les acteurs principaux :**
    - **Client** : Un utilisateur qui peut créer un compte, naviguer dans les produits, ajouter des produits au panier, passer des commandes, effectuer des paiements, et suivre les commandes.
    - **Administrateur** : Un utilisateur qui peut gérer les produits, les utilisateurs et les commandes.
3. **Ajouter les cas d'utilisation suivants :**
    - **Créer un compte** : Le client peut créer un nouveau compte utilisateur.
    - **Naviguer dans les produits** : Le client peut parcourir les produits disponibles sur le site.

- **Ajouter des produits au panier** : Le client peut ajouter des produits à son panier d'achat.
- **Passer une commande** : Le client peut passer une commande pour les produits dans son panier.
- **Effectuer un paiement** : Le client peut payer pour les produits commandés.
- **Suivre les commandes** : Le client peut suivre l'état de ses commandes.
- **Gérer les produits** : L'administrateur peut ajouter, modifier ou supprimer des produits.
- **Gérer les utilisateurs** : L'administrateur peut gérer les comptes utilisateurs.
- **Gérer les commandes** : L'administrateur peut gérer les commandes passées par les clients.

4. **Relier les acteurs aux cas d'utilisation appropriés :**
   - Le client doit être lié à tous les cas d'utilisation sauf ceux de l'administrateur.
   - L'administrateur doit être lié aux cas d'utilisation "Gérer les produits", "Gérer les utilisateurs", et "Gérer les commandes".

Conseils pour la réalisation :

Lorsque vous créez votre diagramme de cas d'utilisation, assurez-vous de bien comprendre les rôles et les responsabilités de chaque acteur. Utilisez des noms clairs et descriptifs pour chaque cas d'utilisation afin de rendre le diagramme facile à comprendre. Pensez à la manière dont chaque action interagit avec le système global et représentez ces interactions avec précision. Prenez le temps de bien organiser votre diagramme pour qu'il soit propre et lisible.

Bonne chance !

Correction :

Acteurs :

- **Client** : Un utilisateur qui peut créer un compte, naviguer dans les produits, ajouter des produits au panier, passer des commandes, effectuer des paiements, et suivre les commandes.
- **Administrateur** : Un utilisateur qui peut gérer les produits, les utilisateurs et les commandes.

Cas d'utilisation :

- **Créer un compte** : Le client peut créer un nouveau compte utilisateur.
- **Naviguer dans les produits** : Le client peut parcourir les produits disponibles sur le site.
- **Ajouter des produits au panier** : Le client peut ajouter des produits à son panier d'achat.
- **Passer une commande** : Le client peut passer une commande pour les produits dans son panier.
- **Effectuer un paiement** : Le client peut payer pour les produits commandés.
- **Suivre les commandes** : Le client peut suivre l'état de ses commandes.
- **Gérer les produits** : L'administrateur peut ajouter, modifier ou supprimer des produits.
- **Gérer les utilisateurs** : L'administrateur peut gérer les comptes utilisateurs.
- **Gérer les commandes** : L'administrateur peut gérer les commandes passées par les clients.

Relations :

- **Client** :
    - Créer un compte

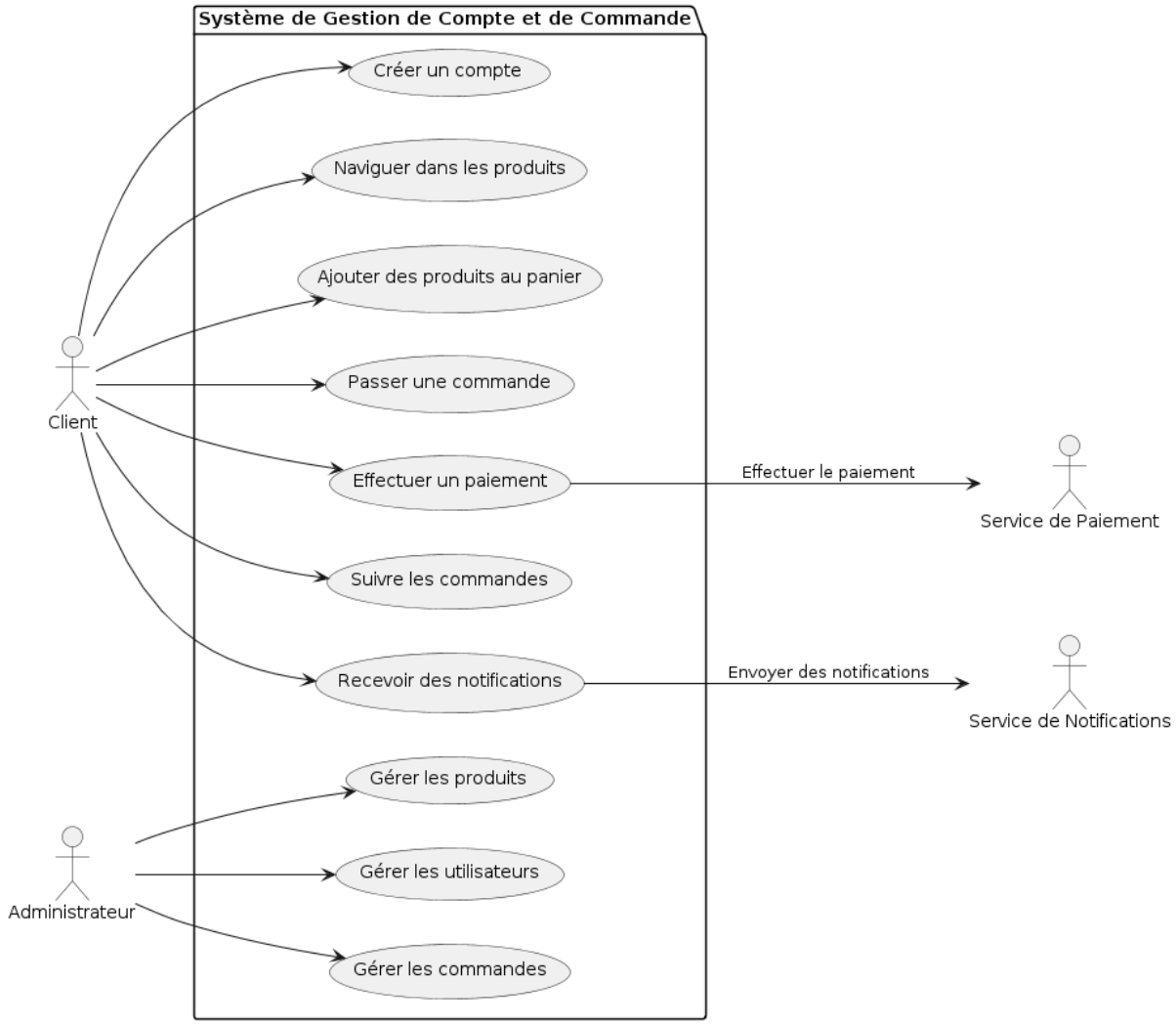
    - Naviguer dans les produits
    - Ajouter des produits au panier
    - Passer une commande
    - Effectuer un paiement
    - Suivre les commandes
- **Administrateur** :
    - Gérer les produits
    - Gérer les utilisateurs
    - Gérer les commandes

Diagramme UMLet

Pour créer le diagramme dans UMLet, suivez ces étapes :

1. **Créer les acteurs :**
    - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Client".
    - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Administrateur".
2. **Créer les cas d'utilisation :**
    - Sélectionnez l'élément "Use Case" et placez-le sur le canevas. Ajoutez les noms des cas d'utilisation : "Créer un compte", "Naviguer dans les produits", "Ajouter des produits au panier", "Passer une commande", "Effectuer un paiement", "Suivre les commandes", "Gérer les produits", "Gérer les utilisateurs", "Gérer les commandes".
3. **Relier les acteurs aux cas d'utilisation :**
    - Utilisez l'élément "Line" pour relier chaque acteur aux cas d'utilisation pertinents. Par exemple, reliez "Client" à tous les cas d'utilisation sauf ceux de l'administrateur. Reliez "Administrateur" à "Gérer les produits", "Gérer les utilisateurs", et "Gérer les commandes".

Modélisation graphique



PlantUML (Un must have !)

Accéder à l'adresse suivante PlantUML Web Server (https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) et copier coller le code suivant :

```
@startuml
left to right direction

actor Client
actor Administrateur
actor "Service de Paiement" as PaymentService
actor "Service de Notifications" as NotificationService

package "Système de Gestion de Compte et de Commande" {
    usecase "Créer un compte" as UC1
    usecase "Naviguer dans les produits" as UC2
    usecase "Ajouter des produits au panier" as UC3
    usecase "Passer une commande" as UC4
    usecase "Effectuer un paiement" as UC5
    usecase "Suivre les commandes" as UC6
    usecase "Recevoir des notifications" as UC7
    usecase "Gérer les produits" as UC8
    usecase "Gérer les utilisateurs" as UC9
    usecase "Gérer les commandes" as UC10
}

Client --> UC1
Client --> UC2
Client --> UC3
Client --> UC4
Client --> UC5
Client --> UC6
Client --> UC7

Administrateur --> UC8
Administrateur --> UC9
Administrateur --> UC10

UC5 --> PaymentService : "Effectuer le paiement"
```

```
UC7 --> NotificationService : "Envoyer des notifications"

@enduml
```

6. Système de Réservation de Vols

Contexte :

Vous travaillez comme analyste chez **AeroSoft Systems**, une entreprise spécialisée dans les systèmes de réservation de vols pour les compagnies aériennes. Votre mission est de modéliser un nouveau système de réservation de vols. Ce système doit permettre aux utilisateurs de rechercher des vols, réserver des billets, annuler des réservations, consulter les détails des vols, et recevoir des notifications. Les agents de voyage doivent pouvoir gérer les réservations et les utilisateurs.

Tâches à réaliser :

1. **Créer un diagramme de cas d'utilisation pour le système de réservation de vols en utilisant UMLet.**

Étapes à suivre :

1. **Ouvrir UMLet et créer un nouveau diagramme de cas d'utilisation.**
2. **Définir les acteurs principaux :**
    - **Client** : Un utilisateur qui peut rechercher des vols, réserver des billets, annuler des réservations, consulter les détails des vols, et recevoir des notifications.
    - **Agent de Voyage** : Un utilisateur qui peut gérer les réservations et les utilisateurs.
3. **Ajouter les cas d'utilisation suivants :**
    - **Rechercher des vols** : Le client peut rechercher des vols disponibles.
    - **Réserver des billets** : Le client peut réserver des billets de vol.
    - **Annuler des réservations** : Le client peut annuler ses réservations.

    - **Consulter les détails des vols** : Le client peut consulter les détails des vols réservés.
    - **Recevoir des notifications** : Le client peut recevoir des notifications sur ses vols.
    - **Gérer les réservations** : L'agent de voyage peut gérer les réservations des clients.
    - **Gérer les utilisateurs** : L'agent de voyage peut gérer les comptes des utilisateurs.
4. **Relier les acteurs aux cas d'utilisation appropriés :**
    - Le client doit être lié à tous les cas d'utilisation sauf ceux de l'agent de voyage.
    - L'agent de voyage doit être lié aux cas d'utilisation "Gérer les réservations" et "Gérer les utilisateurs".

Conseils pour la réalisation :

Lorsque vous créez votre diagramme de cas d'utilisation, assurez-vous de bien comprendre les rôles et les responsabilités de chaque acteur. Utilisez des noms clairs et descriptifs pour chaque cas d'utilisation afin de rendre le diagramme facile à comprendre. Pensez à la manière dont chaque action interagit avec le système global et représentez ces interactions avec précision. Prenez le temps de bien organiser votre diagramme pour qu'il soit propre et lisible.

Bonne chance !

Correction :

Acteurs :

- **Client** : Un utilisateur qui peut rechercher des vols, réserver des billets, annuler des réservations, consulter les détails des vols, et recevoir des notifications.
- **Agent de Voyage** : Un utilisateur qui peut gérer les réservations et les utilisateurs.

Cas d'utilisation :

- **Rechercher des vols** : Le client peut rechercher des vols disponibles.
- **Réserver des billets** : Le client peut réserver des billets de vol.
- **Annuler des réservations** : Le client peut annuler ses réservations.
- **Consulter les détails des vols** : Le client peut consulter les détails des vols réservés.
- **Recevoir des notifications** : Le client peut recevoir des notifications sur ses vols.
- **Gérer les réservations** : L'agent de voyage peut gérer les réservations des clients.
- **Gérer les utilisateurs** : L'agent de voyage peut gérer les comptes des utilisateurs.

Relations :

- **Client** :
    - Rechercher des vols
    - Réserver des billets
    - Annuler des réservations
    - Consulter les détails des vols
    - Recevoir des notifications
- **Agent de Voyage** :
    - Gérer les réservations

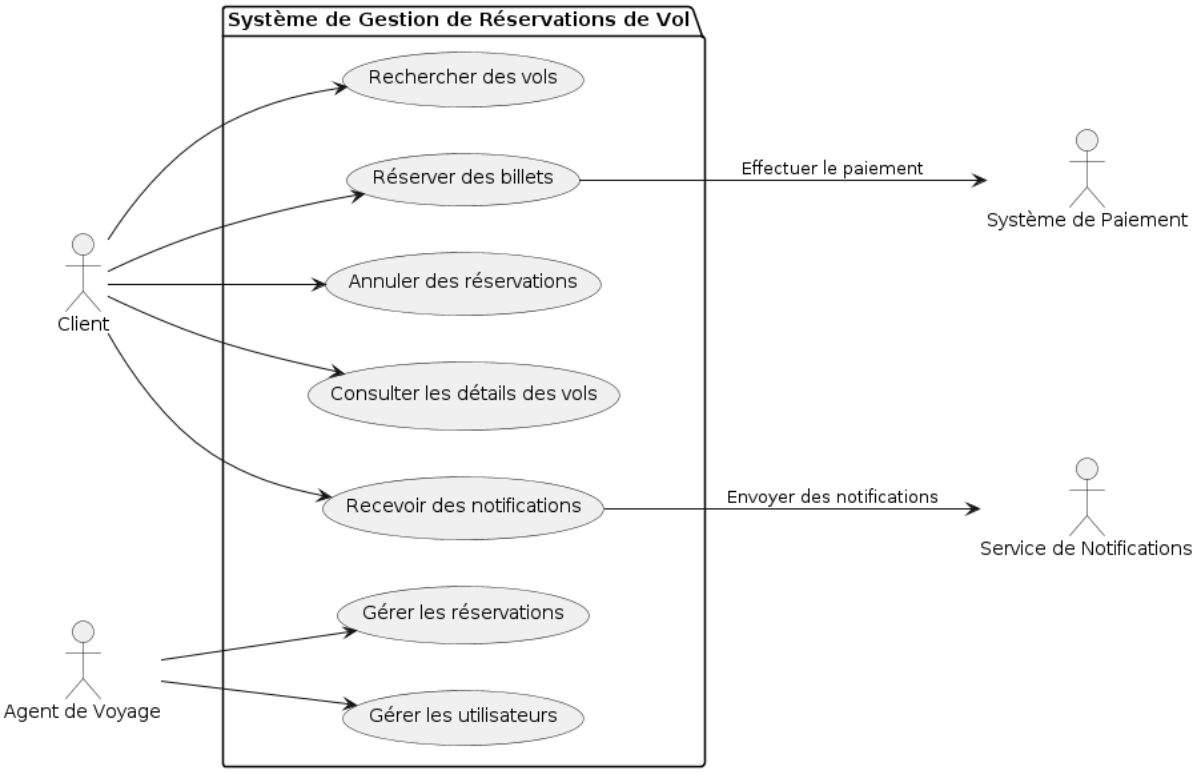
- Gérer les utilisateurs

Diagramme UMLet

Pour créer le diagramme dans UMLet, suivez ces étapes :

1. **Créer les acteurs :**
   - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Client".
   - Sélectionnez l'élément "Actor" et placez-le sur le canevas. Nommez-le "Agent de Voyage".
2. **Créer les cas d'utilisation :**
   - Sélectionnez l'élément "Use Case" et placez-le sur le canevas. Ajoutez les noms des cas d'utilisation : "Rechercher des vols", "Réserver des billets", "Annuler des réservations", "Consulter les détails des vols", "Recevoir des notifications", "Gérer les réservations", "Gérer les utilisateurs".
3. **Relier les acteurs aux cas d'utilisation :**
   - Utilisez l'élément "Line" pour relier chaque acteur aux cas d'utilisation pertinents. Par exemple, reliez "Client" à "Rechercher des vols", "Réserver des billets", "Annuler des réservations", "Consulter les détails des vols", et "Recevoir des notifications". Reliez "Agent de Voyage" à "Gérer les réservations" et "Gérer les utilisateurs".

Modélisation graphique

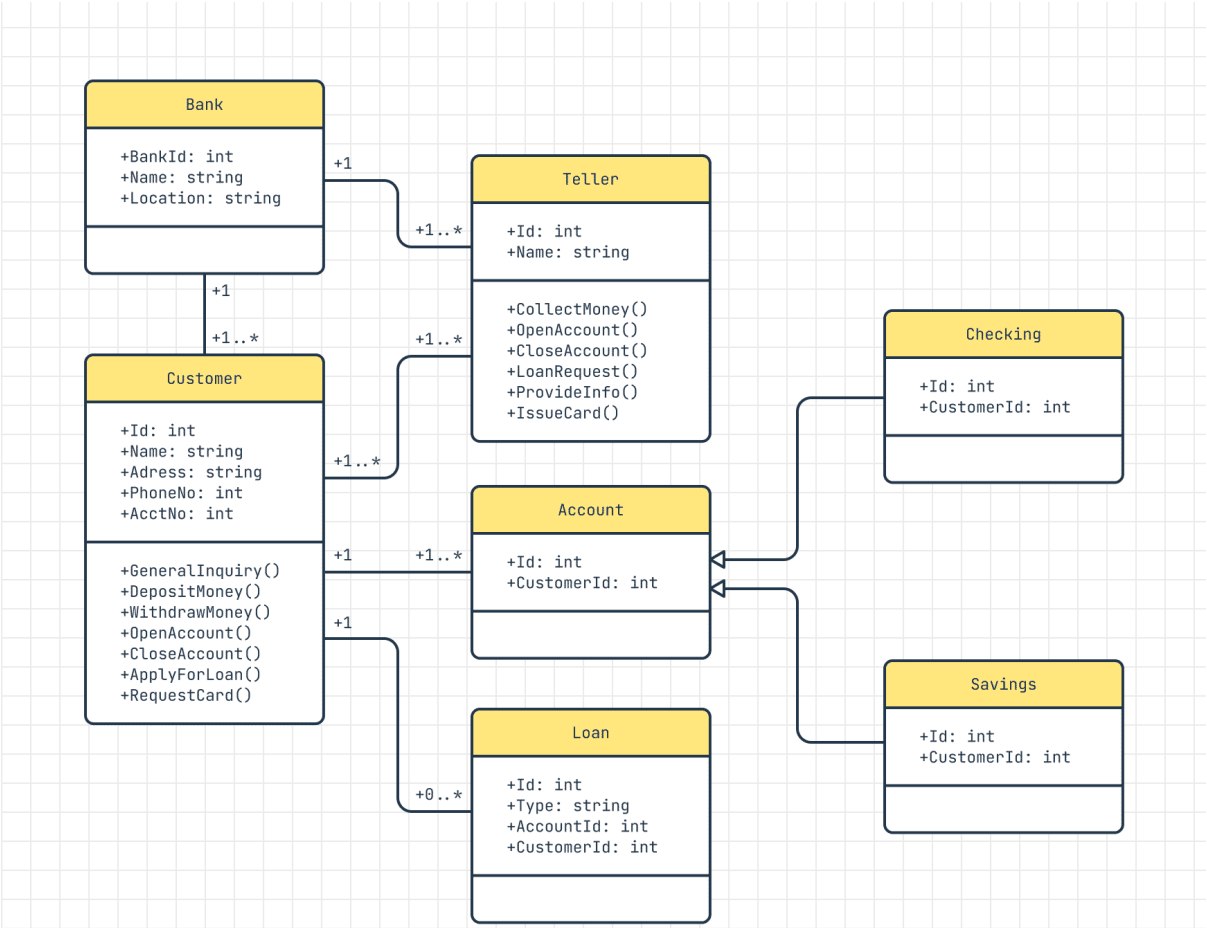


PlantUML (Un must have !)

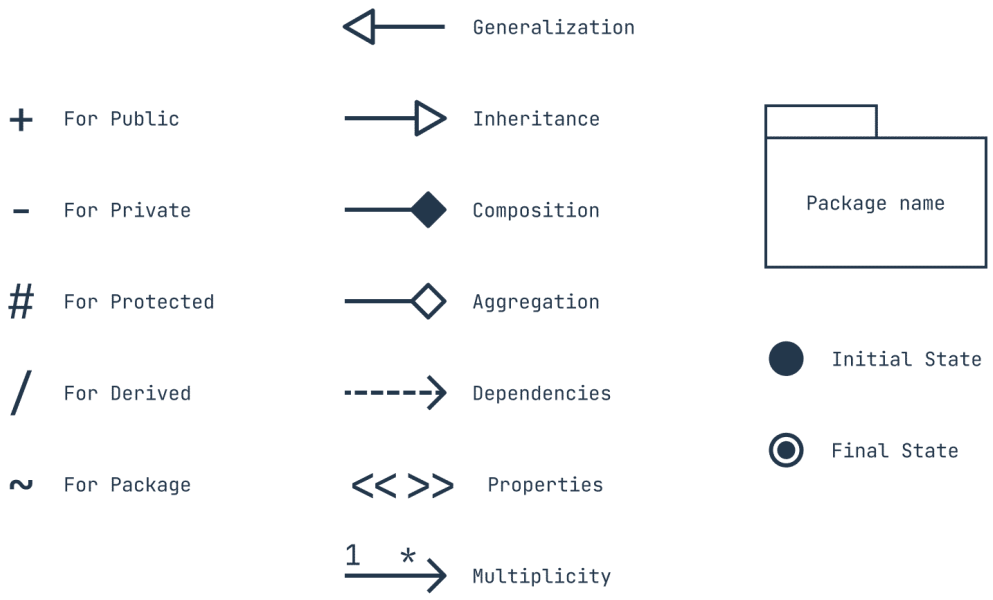
Accéder à l'adresse suivante PlantUML Web Server (https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) et copier coller le code suivant :

```
@startuml
left to right direction

actor Client
actor "Agent de Voyage" as AgentVoyage
actor "Service de Notifications" as NotificationService
actor "Système de Paiement" as PaymentService

package "Système de Gestion de Réservations de Vol" {
    usecase "Rechercher des vols" as UC1
    usecase "Réserver des billets" as UC2
    usecase "Annuler des réservations" as UC3
    usecase "Consulter les détails des vols" as UC4
    usecase "Recevoir des notifications" as UC5
    usecase "Gérer les réservations" as UC6
    usecase "Gérer les utilisateurs" as UC7
}

Client --> UC1
Client --> UC2
Client --> UC3
Client --> UC4
Client --> UC5

AgentVoyage --> UC6
AgentVoyage --> UC7

UC2 --> PaymentService : "Effectuer le paiement"
UC5 --> NotificationService : "Envoyer des notifications"


@enduml
```

LE DIAGRAMME DE CLASSE (CLASS DIAGRAM)



Class Diagram for a Banking System


Generalization
+ For Public
Inheritance
Package name
- For Private
Composition
For Protected
Aggregation
Initial State
/ For Derived
Dependencies
~ For Package
<< >> Properties
Final State
1 *
Multiplicity

1. Système de Bibliothèque

Contexte :

Vous travaillez pour **LibraryTech**, une entreprise spécialisée dans les systèmes de gestion de bibliothèques. Votre mission est de modéliser un système de gestion de bibliothèque. Ce système doit permettre de gérer les livres, les membres et les prêts de livres.

Tâches à réaliser :

1. **Créer un diagramme de classes pour le système de gestion de bibliothèque en utilisant UMLet.**

Étapes à suivre :

1. **Définir les classes principales :**
    - **Book** : Représente un livre dans la bibliothèque.
    - **Member** : Représente un membre de la bibliothèque.
    - **Loan** : Représente un prêt de livre à un membre.
2. **Ajouter les Attributes et Methods pour chaque classe :**
    - **Book** :
        - Attributes : title, author, ISBN, yearOfPublication, condition.
        - Methods : borrowBook(), returnBook().
    - **Member** :
        - Attributes : name, membershipNumber, address, telephone.
        - Methods : registerMember(), borrowBook(), returnBook().
    - **Loan** :
        - Attributes : dateBorrowed, dateReturned.
        - Methods : recordBorrowBook(), recordReturnBook().
3. **Définir les relations entre les classes :**
    - Un **Book** peut être associé à plusieurs **Loan**.
    - Un **Member** peut être associé à plusieurs **Loan**.

- Un **Loan** est associé à un seul **Book** et à un seul **Member**.

Conseils pour la réalisation :

Lorsque vous créez votre diagramme de classes, assurez-vous de bien définir les Attributes et les Methods de chaque classe. Utilisez des noms clairs et descriptifs pour les classes, les Attributes et les Methods afin de rendre le diagramme facile à comprendre. Pensez à la manière dont chaque classe interagit avec les autres et représentez ces relations avec précision. Prenez le temps de bien organiser votre diagramme pour qu'il soit propre et lisible.

Bonne chance !

Correction :

Classes :

- **Book** :
    - Attributes :
        - title : String
        - author : String
        - ISBN : String
        - yearOfPublication : int
        - état : String
    - Methods :
        - borrowBook()
        - returnBook()
- **Member** :
    - Attributes :
        - name : String
        - membershipNumber : String
        - address : String
        - telephone : String
    - Methods :
        - registerMember()
        - borrowBook()
        - returnBook()
- **Loan**:
    - Attributes :
        - dateBorrowed : Date
        - dateReturned : Date
    - Methods :
        - recordBorrowBook()
        - recordReturnBook()

Relations :

- Un **Book** peut être associé à plusieurs **Loan**(relation 1 à plusieurs).
- Un **Member** peut être associé à plusieurs **Loan**(relation 1 à plusieurs).
- Un **Loan** est associé à un seul **Book** (relation plusieurs à 1) et à un seul **Member** (relation plusieurs à 1).

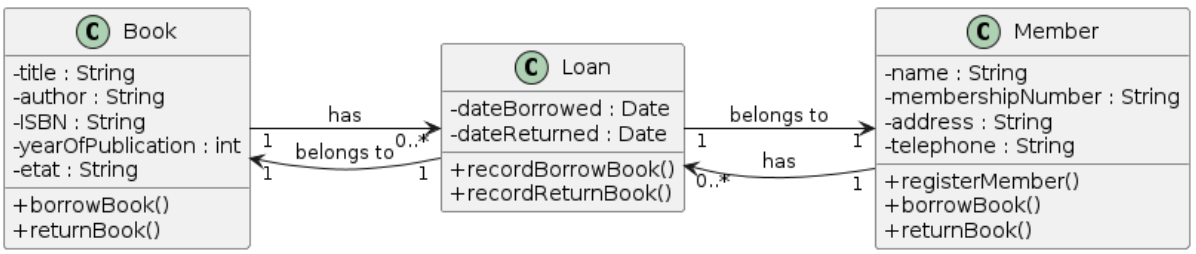
Diagramme UMLet

Pour créer le diagramme dans UMLet, suivez ces étapes :

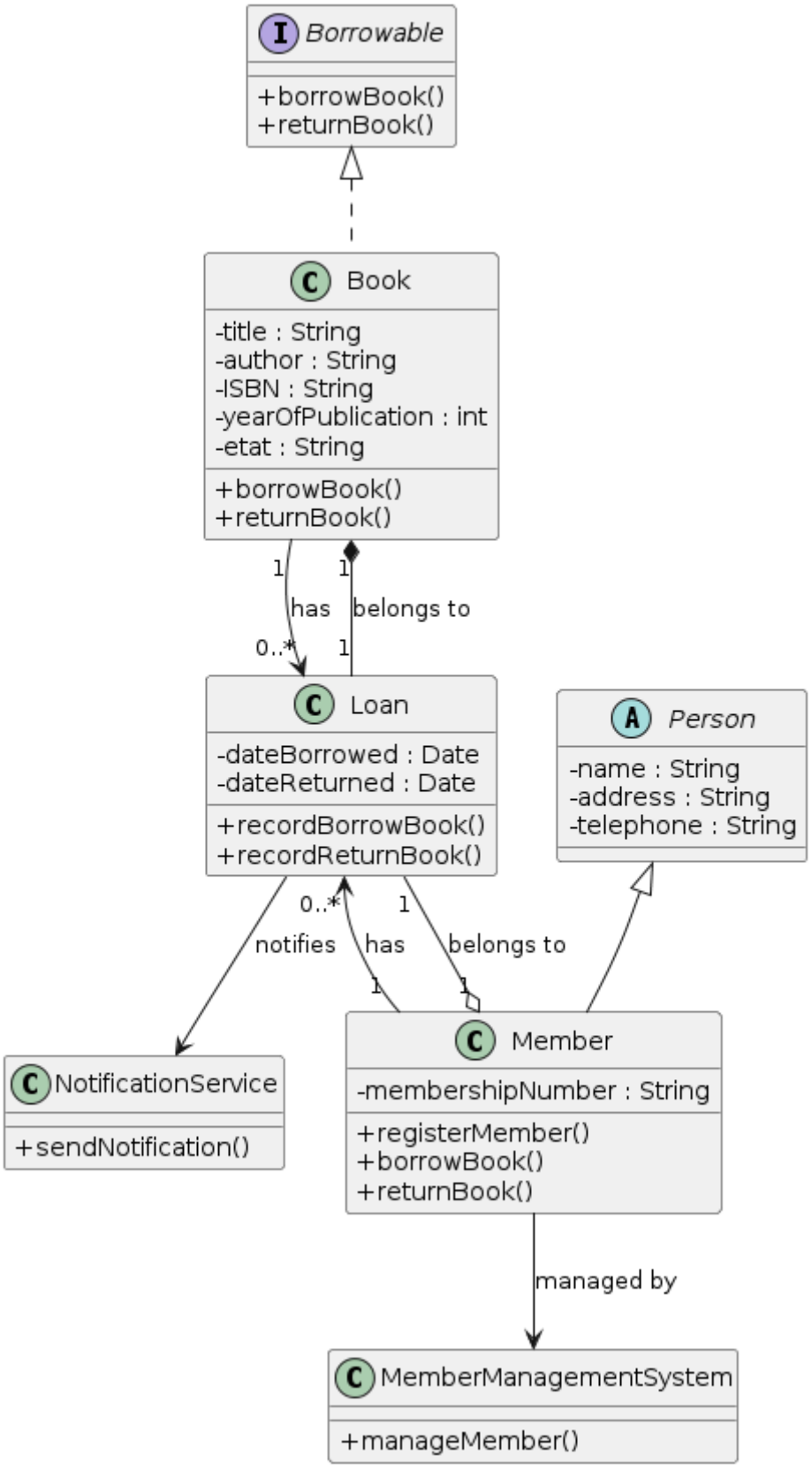
1. **Créer les classes :**
    - Sélectionnez l'élément "Class" et placez-le sur le canevas. Nommez-le "Book".
    - Ajoutez les Attributes et Methods à la classe "Book".
    - Répétez pour les classes "Member" et "Loan".
2. **Définir les relations :**
    - Utilisez l'élément "Line" pour relier les classes en fonction des relations définies. Par exemple, reliez "Book" à "Loan" avec une relation 1 à plusieurs, et "Member" à "Prêt" avec une relation 1 à plusieurs.

Modélisation graphique

Réponse attendu pour l'exercice de base :



Réponse enrichi pour l'exercice de base :



Points informations utiles

1. **Visibilité des Méthodes** :
    - Les méthodes d'une interface sont implicitement `public` en Java, (c'est-à-dire qu'une méthode dans une interface ne peut pas être `protected` ou `private`, elles doivent être `public)` , donc il n'est pas nécessaire de spécifier le modificateur de visibilité `+`.
2. **Granularité** :
    - Dans le corrigé de la version améliroer, les noms des méthodes sont très génériques. Il pourrait être bénéfique de rendre les méthodes plus spécifiques à leur contexte d'utilisation. Par exemple, `createUser`, `deleteDocument`, etc., pourraient être plus explicites si l'interface est destinée à un domaine particulier.
3. **Segmentation des Responsabilités** :

Si les méthodes représentent des actions très différentes, il pourrait être intéressant de scinder l'interface en plusieurs interfaces plus petites et plus ciblées, chacune représentant un ensemble cohérent de responsabilités (principe de séparation des interfaces, ISP - Interface Segregation Principle).

Explications des améliorations

1. **Classe `Person` abstraite :**
    - Cette classe abstraite contient des attributs communs comme `name`, `address`, et `telephone`. Elle est étendue par la classe `Member`.
2. **Interface `Borrowable` :**
    - Cette interface définit des méthodes communes `borrowBook` et `returnBook`, implémentées par la classe `Book`.
3. **Compositions et Agrégations :**
    - La relation entre `Book` et `Loan` est une composition, car un prêt appartient à un livre et ne peut pas exister sans lui.
    - La relation entre `Member` et `Loan` est une agrégation, car un prêt est lié à un membre mais ces membres peuvent exister indépendamment des prêts.
4. **Services Tiers :**
    - `NotificationService` : Ajouté pour envoyer des notifications (par exemple, rappel de retour de livres).
    - `MemberManagementSystem` : Ajouté pour gérer les membres de la bibliothèque.
5. **Relations directionnelles :**
    - Les relations directionnelles montrent clairement les interactions entre les différentes classes et services.

Ce diagramme offre une vue plus détaillée et réaliste du système de gestion de bibliothèque, en incluant des éléments pratiques et des relations complexes qui reflètent mieux un environnement de développement réel.

PlantUML (Un must have !)

Accéder à l'adresse suivante PlantUML Web Server (https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) et copier coller le code suivant :

Réponse attendu pour l'exercice de base :

```
@startuml

left to right direction
skinparam classAttributeIconSize 0

class Book {
  - title : String
  - author : String
  - ISBN : String
  - yearOfPublication : int
  - etat : String
  + borrowBook()
  + returnBook()
}

class Member {
  - name : String
  - membershipNumber : String
  - address : String
  - telephone : String
  + registerMember()
  + borrowBook()
  + returnBook()
}

class Loan {
  - dateBorrowed : Date
  - dateReturned : Date
  + recordBorrowBook()
  + recordReturnBook()
}

Book "1" --> "0..*" Loan : has
Member "1" --> "0..*" Loan : has
Loan "1" --> "1" Book : belongs to
Loan "1" --> "1" Member : belongs to

@enduml
```

Réponse enrichi pour l'exercice de base :

```
@startuml
skinparam classAttributeIconSize 0

interface Borrowable {
  + borrowBook()
  + returnBook()
}

abstract class Person {
  - name : String
  - address : String
  - telephone : String
}

class Book implements Borrowable {
  - title : String
  - author : String
  - ISBN : String
  - yearOfPublication : int
  - etat : String
  + borrowBook()
  + returnBook()
}

class Member extends Person {
  - membershipNumber : String
  + registerMember()
  + borrowBook()
  + returnBook()
}

class Loan {
  - dateBorrowed : Date
  - dateReturned : Date
  + recordBorrowBook()
  + recordReturnBook()
}

class NotificationService {
  + sendNotification()
}

class MemberManagementSystem {
  + manageMember()
}

Book "1" --> "0..*" Loan : has
Member "1" --> "0..*" Loan : has
Loan "1" --* "1" Book : belongs to
```

```
Loan "1" --o "1" Member : belongs to

Loan --> NotificationService : "notifies"
Member --> MemberManagementSystem : "managed by"

@enduml
```

2. Système de Gestion de Restaurant

Contexte :

Vous travaillez pour **RestaurantPro**, une entreprise spécialisée dans les systèmes de gestion de restaurants. Votre mission est de modéliser un système de gestion pour un restaurant. Ce système doit permettre de gérer les tables, les commandes, les plats et les serveurs.

Tâches à réaliser :

1. **Créer un diagramme de classes pour le système de gestion de restaurant en utilisant UMLet.**

Étapes à suivre :

1. **Définir les classes principales :**
    - **Table** : Représente une table dans le restaurant.
    - **Order** : Représente une commande passée par un client.
    - **Dish** : Représente un plat du menu.
    - **Waiter** : Représente un serveur du restaurant.
2. **Ajouter les Attributes et Methods pour chaque classe :**
    - **Table** :
        - Attributes : numberTable, numberPlaces, isOccupied.
        - Methods : occupyTable(), freeTable().
    - **Order** :
        - Attributes : orderNumber, orderDate, status.
        - Methods : addDish(), deleteDish(), validateOrder().
    - **Dish** :
        - Attributes : name, description, price.
        - Methods : modifyDish(), deleteDish().
    - **Waiter** :
        - Attributes : name, identifier.
        - Methods : takeOrder(), serveOrder().

3. **Définir les relations entre les classes :**
    - Une **Table** peut être associée à plusieurs **Order**.
    - Une **Order** peut inclure plusieurs **Dish**.
    - Un **Waiter** peut gérer plusieurs **Order** .
    - Une **Order** est associée à une seule **Table** et à un seul **Waiter**.

Conseils pour la réalisation :

Lorsque vous créez votre diagramme de classes, assurez-vous de bien définir les Attributes et les Methods de chaque classe. Utilisez des noms clairs et descriptifs pour les classes, les Attributes et les Methods afin de rendre le diagramme facile à comprendre. Pensez à la manière dont chaque classe interagit avec les autres et représentez ces relations avec précision. Prenez le temps de bien organiser votre diagramme pour qu'il soit propre et lisible.

Bonne chance !

Correction :

Classes :

- **Table** :
  - Attributes :
    - numberTable : int
    - numberPlaces : int
    - isOccupied : boolean
  - Methods :
    - occupyTable()
    - freeTable()
- **Order** :
  - Attributes :
    - orderNumber : int
    - orderDate : Date
    - status : String
  - Methods :
    - addDish()
    - deleteDish()
    - validateOrder()
- **Dish** :
  - Attributes :
    - name : String
    - description : String
    - price : double
  - Methods :
    - modifyDish()
    - deleteDish()
- **Waiter** :
  - Attributes :
    - name : String
    - identifier : String

    - Methods :
        - takeOrder()
        - serveOrder()

Relations :

- Une **Table** peut être associée à plusieurs **Order** (relation 1 à plusieurs).
- Une **Order** peut inclure plusieurs **Dish** (relation 1 à plusieurs).
- Un **Waiter** peut gérer plusieurs **Order** (relation 1 à plusieurs).
- Une **Order** est associée à une seule **Table** (relation plusieurs à 1) et à un seul **Waiter** (relation plusieurs à 1).

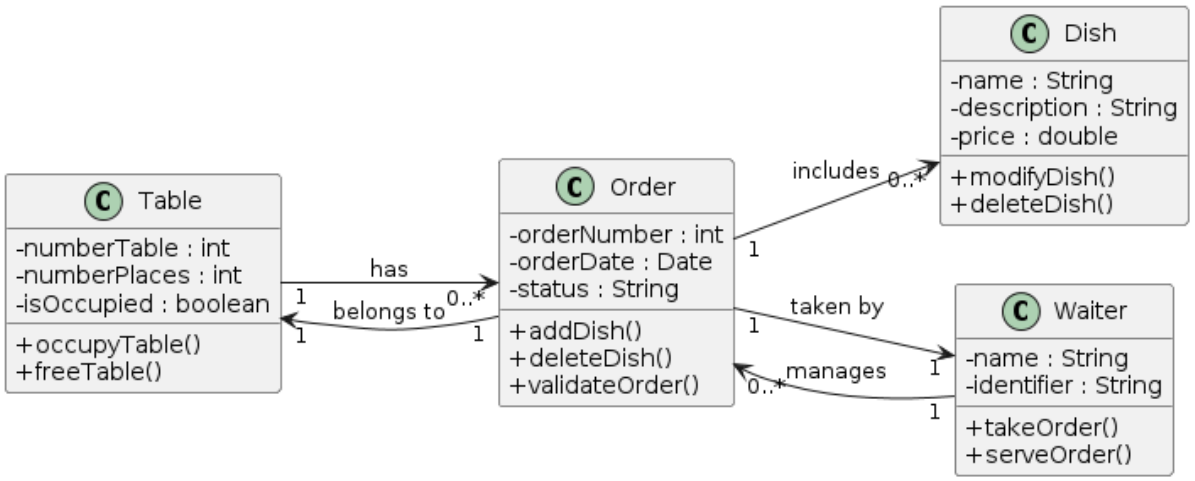
Diagramme UMLet

Pour créer le diagramme dans UMLet, suivez ces étapes :

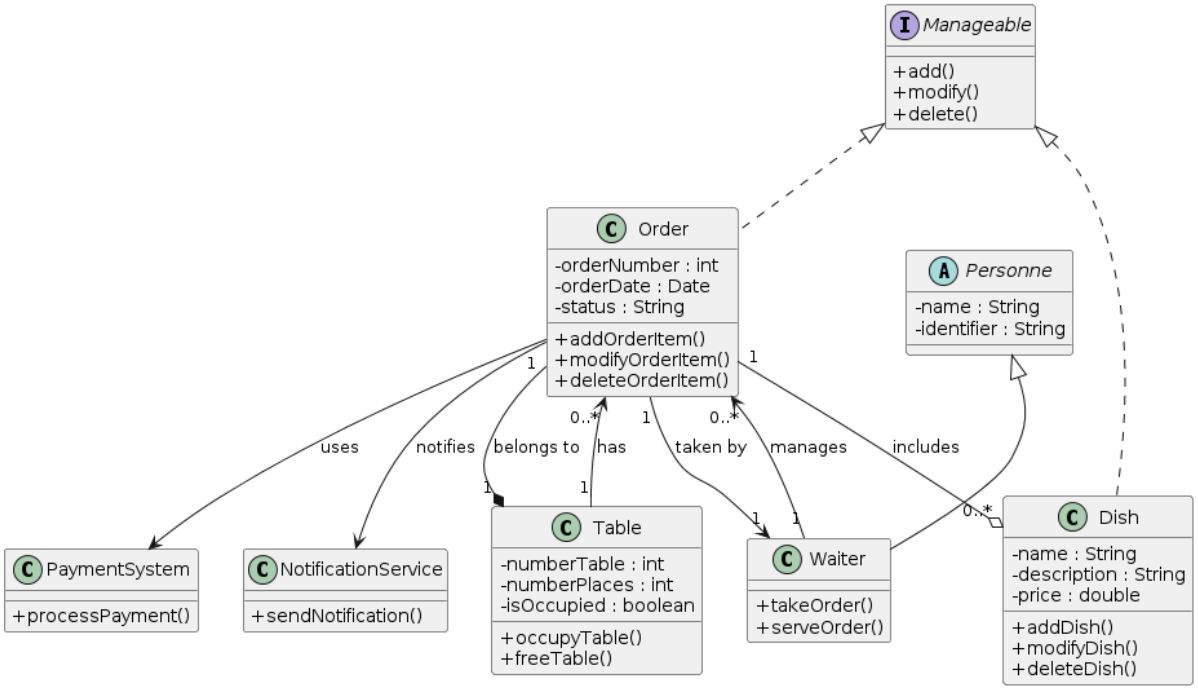
1. **Créer les classes :**
    - Sélectionnez l'élément "Class" et placez-le sur le canevas. Nommez-le "Table".
    - Ajoutez les Attributes et Methods à la classe "Table".
    - Répétez pour les classes "Order", "Dish", et "Waiter".
2. **Définir les relations :**
    - Utilisez l'élément "Line" pour relier les classes en fonction des relations définies. Par exemple, reliez "Table" à "Order" avec une relation 1 à plusieurs, "Order" à "Dish" avec une relation 1 à plusieurs, et "Waiter" à "Order" avec une relation 1 à plusieurs.

Modélisation graphique

Réponse attendu pour l'exercice de base :



Réponse enrichi pour l'exercice de base :



Points informations utiles

4. **Visibilité des Méthodes** :
    - Les méthodes d'une interface sont implicitement `public` en Java, (c'est-à-dire qu'une méthode dans une interface ne peut pas être `protected` ou `private`, elles doivent être `public)` , donc il n'est pas nécessaire de spécifier le modificateur de visibilité `+`.
5. **Granularité** :
    - Dans le corrigé de la version améliroer, les noms des méthodes sont très génériques. Il pourrait être bénéfique de rendre les méthodes plus spécifiques à leur contexte d'utilisation. Par exemple, `createUser`, `deleteDocument`, etc., pourraient être plus explicites si l'interface est destinée à un domaine particulier.
6. **Segmentation des Responsabilités** :

Si les méthodes représentent des actions très différentes, il pourrait être intéressant de scinder l'interface en plusieurs interfaces plus petites et plus ciblées, chacune représentant un ensemble cohérent de responsabilités (principe de séparation des interfaces, ISP - Interface Segregation Principle).

Explications des Améliorations

Packages et Classes

Classe `Personne` Abstraite :

- **Description :** La classe abstraite `Personne` contient des attributs communs tels que `name` et `identifier`. Cette abstraction permet de centraliser les attributs communs à plusieurs types de personnes, réduisant ainsi la redondance.
- **Héritage :** La classe `Waiter` (serveur) étend `Personne`, héritant de ces attributs communs.

Interface `Manageable` :

- **Description :** L'interface `Manageable` définit des méthodes génériques `add()`, `modify()`, et `delete()` pour les opérations de gestion.
- **Implémentation :** Les classes `Order` (commande) et `Dish` (plat) implémentent cette interface, spécialisant les méthodes pour leur contexte spécifique.

Compositions et Agrégations

- **Composition entre `Table` et `Order` :** La relation entre `Table` et `Order` est une composition, ce qui signifie qu'une commande (order) appartient à une table (table) et ne peut pas exister sans elle. Si une table est supprimée, toutes les commandes associées sont également supprimées.
- **Agrégation entre `Order` et `Dish` :** La relation entre `Order` et `Dish` est une agrégation. Une commande inclut plusieurs plats (dishes), mais ces plats peuvent exister indépendamment des commandes.

Services Tiers

- **PaymentSystem :** Cette classe est ajoutée pour gérer les paiements des commandes. Elle permet de centraliser et d'abstraire la logique de paiement, facilitant ainsi la maintenance et l'extension du système.
- **NotificationService :** Cette classe est ajoutée pour envoyer des notifications, telles que des notifications de commande prête. Elle permet de gérer les communications de manière centralisée.

Relations Directionnelles

- **Description :** Les relations directionnelles dans le diagramme montrent clairement les interactions entre les différentes classes et services, indiquant quelle classe utilise ou dépend de quelle autre classe ou service.
- **Exemple :** La classe `Order` utilise `PaymentSystem` pour traiter les paiements et `NotificationService` pour envoyer des notifications.

Vue Globale et Réalisme du Système

- **Description :** Ce diagramme offre une vue plus détaillée et réaliste du système de gestion de restaurant. En incluant des éléments pratiques comme les services tiers et des relations complexes, il reflète mieux un environnement de développement réel.
- **Avantages :** Cette approche améliore la compréhension des interactions et des dépendances entre les différentes parties du système, facilitant ainsi le développement, la maintenance et l'évolution du système.

PlantUML (Un must have !)

Accéder à l'adresse suivante PlantUML Web Server (https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) et copier coller le code suivant :

Réponse attendu pour l'exercice de base :

```
@startuml

left to right direction
skinparam classAttributeIconSize 0

class Table {
  - numberTable : int
  - numberPlaces : int
  - isOccupied : boolean
  + occupyTable()
  + freeTable()
}

class Order {
  - orderNumber : int
  - orderDate : Date
  - status : String
  + addDish()
  + deleteDish()
  + validateOrder()
}

class Dish {
  - name : String
  - description : String
  - price : double
  + modifyDish()
  + deleteDish()
}

class Waiter {
  - name : String
  - identifier : String
  + takeOrder()
  + serveOrder()
}

Table "1" --> "0..*" Order : has
```

```
Order "1" --> "0..*" Dish : includes
Waiter "1" --> "0..*" Order : manages
Order "1" --> "1" Table : belongs to
Order "1" --> "1" Waiter : taken by

@enduml
```

Réponse enrichi pour l'exercice de base :

```
@startuml
skinparam classAttributeIconSize 0

interface Manageable {
  + add()
  + modify()
  + delete()
}

abstract class Personne {
  - name : String
  - identifier : String
}

class Table {
  - numberTable : int
  - numberPlaces : int
  - isOccupied : boolean
  + occupyTable()
  + freeTable()
}

class Order implements Manageable {
  - orderNumber : int
  - orderDate : Date
  - status : String
  + addOrderItem()
  + modifyOrderItem()
  + deleteOrderItem()
}

class Dish implements Manageable {
  - name : String
  - description : String
  - price : double
  + addDish()
  + modifyDish()
  + deleteDish()
}
```

```
class Waiter extends Personne {
  + takeOrder()
  + serveOrder()
}

class PaymentSystem {
  + processPayment()
}

class NotificationService {
  + sendNotification()
}

Table "1" --> "0..*" Order : has
Order "1" --o "0..*" Dish : includes
Waiter "1" --> "0..*" Order : manages
Order "1" --* "1" Table : belongs to
Order "1" --> "1" Waiter : taken by
Order --> PaymentSystem : "uses"
Order --> NotificationService : "notifies"

@enduml
```

3.Système de Gestion d'Hôpital

Contexte :

Vous travaillez pour **HealthCare IT Solutions**, une entreprise spécialisée dans les systèmes de gestion pour les hôpitaux. Votre mission est de modéliser un système de gestion d'hôpital. Ce système doit permettre de gérer les patients, les médecins, les rendez-vous et les traitements.

Tâches à réaliser :

1. **Créer un diagramme de classes pour le système de gestion d'hôpital en utilisant UMLet.**

Étapes à suivre :

1. **Définir les classes principales :**
    - **Patient** : Représente un patient de l'hôpital.
    - **Doctor** : Représente un médecin de l'hôpital.
    - **Appointment** : Représente un rendez-vous médical.
    - **Treatment** : Représente un traitement médical prescrit à un patient.
2. **Ajouter les Attributes et Methods pour chaque classe :**
    - **Patient** :
        - Attributes : name, patientNumber, address, telephone, dateOfBirth.
        - Methods : takeAnAppointment(), cancelAnAppointment().
    - **Doctor** :
        - Attributes : name, specialty, identifier.
        - Methods : consultPatient(), prescribeTreatment().
    - **Appointment** :
        - Attributes : appointmentDate, appointmentTime, status.

        - Methods : scheduleAnAppointment(), cancelAnAppointment().
    - **Treatment** :
        - Attributes : description, startDate, endDate.
        - Methods : startTreatment(), endTreatment().
3. **Définir les relations entre les classes :**
    - Un **Patient** peut avoir plusieurs **Appointment** .
    - Un **Doctor** peut avoir plusieurs **Appointment** .
    - Un **Appointment** est associé à un seul **Patient** et à un seul **Doctor** .
    - Un **Patient** peut avoir plusieurs **Treatment**.
    - Un **Treatment** est associé à un seul **Patient**.

Conseils pour la réalisation :

Lorsque vous créez votre diagramme de classes, assurez-vous de bien définir les Attributes et les Methods de chaque classe. Utilisez des noms clairs et descriptifs pour les classes, les Attributes et les Methods afin de rendre le diagramme facile à comprendre. Pensez à la manière dont chaque classe interagit avec les autres et représentez ces relations avec précision. Prenez le temps de bien organiser votre diagramme pour qu'il soit propre et lisible.

Bonne chance !

Correction :

Classes :

- **Patient** :
    - Attributes :
        - name : String
        - patientNumber : String
        - address : String
        - telephone : String
        - dateOfBirth : Date
    - Methods :
        - takeAnAppointment()
        - cancelAnAppointment()
- **Doctor** :
    - Attributes :
        - name: String
        - spécialité : String
        - identifiant : String
    - Methods :
        - consultPatient()
        - prescribeTreatment()
- **Appointment** :
    - Attributes :
        - appointmentDate : Date
        - appointmentTime : Time
        - status : String
    - Methods :
        - scheduleAnAppointment()
        - cancelAnAppointment()
- **Traitement** :
    - Attributes :
        - description : String

        - startDate : Date
        - endDate : Date
    - Methods :
        - startTreatment()
        - endTreatment()

Relations :

- Un **Patient** peut avoir plusieurs **Appointment** (relation 1 à plusieurs).
- Un **Doctor** peut avoir plusieurs **Appointment** (relation 1 à plusieurs).
- Un **Appointment** est associé à un seul **Patient** (relation plusieurs à 1) et à un seul **Doctor** (relation plusieurs à 1).
- Un **Patient** peut avoir plusieurs **Treatment** (relation 1 à plusieurs).
- Un **Treatment** est associé à un seul **Patient** (relation plusieurs à 1).

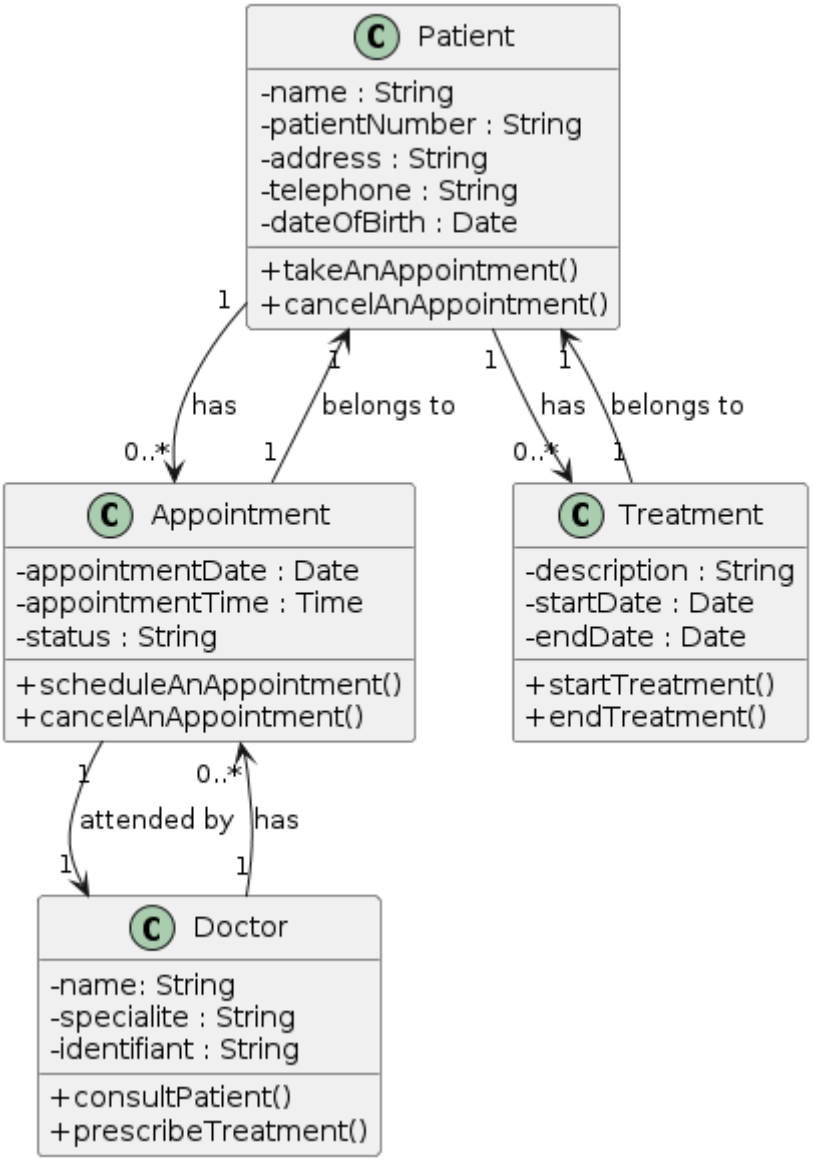
Diagramme UMLet

Pour créer le diagramme dans UMLet, suivez ces étapes :

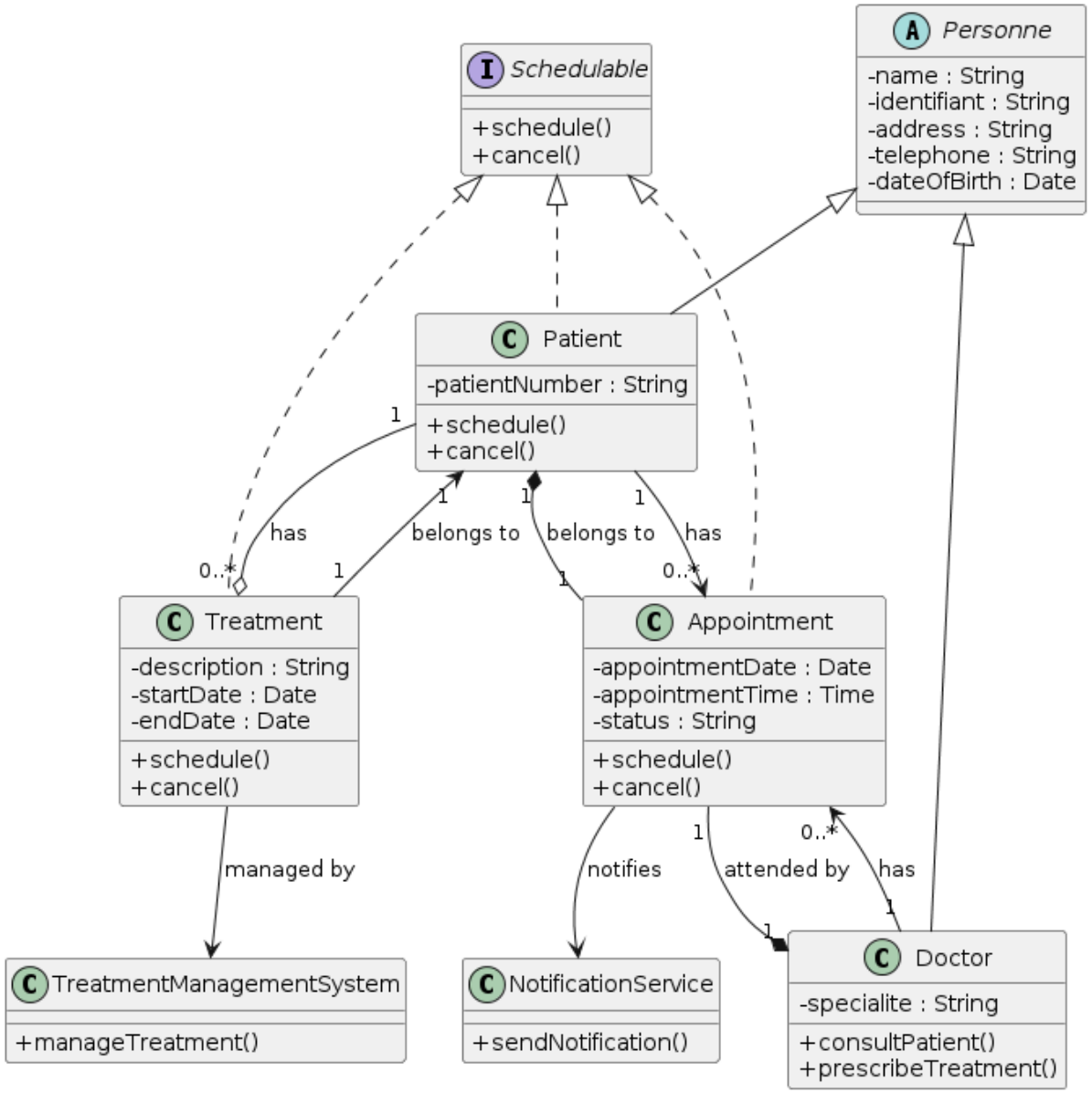
1. **Créer les classes :**
    - Sélectionnez l'élément "Class" et placez-le sur le canevas. Nommez-le "Patient".
    - Ajoutez les Attributes et Methods à la classe "Patient".
    - Répétez pour les classes "Doctor", "Appointment", et "Treatment".
2. **Définir les relations :**
    - Utilisez l'élément "Line" pour relier les classes en fonction des relations définies. Par exemple, reliez "Patient" à "Appointment" avec une relation 1 à plusieurs, "Doctor" à "Rendez-vous" avec une relation 1 à plusieurs, et "Patient" à "Treatment" avec une relation 1 à plusieurs.

Modélisation graphique

Réponse attendu pour l'exercice de base :



Réponse enrichi pour l'exercice de base :



Points informations utiles :

7. **Visibilité des Méthodes** :
    - Les méthodes d'une interface sont implicitement `public` en Java, (c'est-à-dire qu'une méthode dans une interface ne peut pas être `protected` ou `private`, elles doivent être `public)` , donc il n'est pas nécessaire de spécifier le modificateur de visibilité `+`.
8. **Granularité** :
    - Dans le corrigé de la version améliroer, les noms des méthodes sont très génériques. Il pourrait être bénéfique de rendre les méthodes plus spécifiques à leur contexte d'utilisation. Par exemple, `createUser`, `deleteDocument`, etc., pourraient être plus explicites si l'interface est destinée à un domaine particulier.
9. **Segmentation des Responsabilités** :

Si les méthodes représentent des actions très différentes, il pourrait être intéressant de scinder l'interface en plusieurs interfaces plus petites et plus ciblées, chacune représentant un ensemble cohérent de responsabilités (principe de séparation des interfaces, ISP - Interface Segregation Principle).

Explications des Améliorations

Packages et Classes

Classe `Personne` Abstraite :

- **Description :** La classe abstraite `Personne` contient des attributs communs tels que `name`, `identifiant`, `address`, `telephone`, et `dateOfBirth`. Cette abstraction permet de centraliser les attributs communs à plusieurs types de personnes, réduisant ainsi la redondance.
- **Héritage :** Les classes `Patient` (patient) et `Doctor` (médecin) étendent `Personne`, héritant de ces attributs communs.

Interface `Schedulable` :

- **Description :** L'interface `Schedulable` définit des méthodes génériques `schedule()` et `cancel()` pour les opérations de planification.
- **Implémentation :** Les classes `Patient`, `Appointment` (rendez-vous) et `Treatment` (traitement) implémentent cette interface, unifiant ainsi les méthodes de planification et d'annulation pour différents types d'entités programmables.

Compositions et Agrégations

- **Composition :** La relation entre `Appointment` et `Patient` est une composition, car un rendez-vous appartient à un patient spécifique et ne peut pas exister sans lui. De même, la relation entre `Appointment` et `Doctor` est une composition, car un rendez-vous appartient à un médecin spécifique.
- **Agrégation :** La relation entre `Patient` et `Treatment` est une agrégation, car un traitement est associé à un patient mais peut

exister indépendamment d'un patient spécifique. Un patient peut avoir plusieurs traitements, mais les traitements peuvent aussi être définis et utilisés indépendamment.

Services Tiers

- **NotificationService :** Cette classe est ajoutée pour envoyer des notifications, telles que des rappels de rendez-vous. Elle permet de gérer les communications de manière centralisée.
- **TreatmentManagementSystem :** Cette classe est ajoutée pour gérer les traitements des patients. Elle centralise la gestion des traitements, facilitant ainsi la maintenance et l'extension du système.

Relations Directionnelles

- **Description :** Les relations directionnelles dans le diagramme montrent clairement les interactions entre les différentes classes et services, indiquant quelle classe utilise ou dépend de quelle autre classe ou service.
- **Exemple :** La classe `Appointment` utilise `NotificationService` pour envoyer des notifications et `Treatment` utilise `TreatmentManagementSystem` pour gérer les traitements.

Vue Globale et Réalisme du Système

- **Description :** Ce diagramme offre une vue plus détaillée et réaliste du système de gestion de rendez-vous et de traitements médicaux. En incluant des éléments pratiques comme les services tiers et des relations complexes, il reflète mieux un environnement de développement réel.
- **Avantages :** Cette approche améliore la compréhension des interactions et des dépendances entre les différentes parties du système, facilitant ainsi le développement, la maintenance et l'évolution du système.

En adoptant ces améliorations, le diagramme devient non seulement plus structuré et cohérent, mais il reflète également une conception orientée objet efficace et adaptable à un environnement de développement réel, tel que requis pour une application en Java

PlantUML (Un must have !)

Accéder à l'adresse suivante PlantUML Web Server (https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) et copier coller le code suivant :

Réponse attendu pour l'exercice de base :

```
@startuml

left to right direction
skinparam classAttributeIconSize 0

class Patient {
  - name : String
  - patientNumber : String
  - address : String
  - telephone : String
  - dateOfBirth : Date
  + takeAnAppointment()
  + cancelAnAppointment()
}

class Doctor {
  - name: String
  - specialite : String
  - identifiant : String
  + consultPatient()
  + prescribeTreatment()
}

class Appointment {
  - appointmentDate : Date
  - appointmentTime : Time
  - status : String
  + scheduleAnAppointment()
  + cancelAnAppointment()
}

class Treatment {
  - description : String
  - startDate : Date
  - endDate : Date
  + startTreatment()
  + endTreatment()
}
```

```
Patient "1" --> "0..*" Appointment : has
Doctor "1" --> "0..*" Appointment : has
Appointment "1" --> "1" Patient : belongs to
Appointment "1" --> "1" Doctor : attended by
Patient "1" --> "0..*" Treatment : has
Treatment "1" --> "1" Patient : belongs to

@enduml
```

Réponse enrichi pour l'exercice de base :

```
@startuml

skinparam classAttributeIconSize 0

interface Schedulable {
  + schedule()
  + cancel()
}

abstract class Personne {
  - name : String
  - identifiant : String
  - address : String
  - telephone : String
  - dateOfBirth : Date
}

class Patient extends Personne {
  - patientNumber : String
  + takeAnAppointment()
  + cancelAnAppointment()
}

class Doctor extends Personne {
  - specialite : String
  + consultPatient()
  + prescribeTreatment()
}

class Appointment implements Schedulable {
  - appointmentDate : Date
  - appointmentTime : Time
  - status : String
  + scheduleAnAppointment()
  + cancelAnAppointment()
```

```
  + schedule()
  + cancel()
}

class Treatment {
  - description : String
  - startDate : Date
  - endDate : Date
  + startTreatment()
  + endTreatment()
}

class NotificationService {
  + sendNotification()
}

class TreatmentManagementSystem {
  + manageTreatment()
}

Patient "1" --> "0..*" Appointment : has
Doctor "1" --> "0..*" Appointment : has
Appointment "1" --* "1" Patient : belongs to
Appointment "1" --* "1" Doctor : attended by
Patient "1" --o "0..*" Treatment : has
Treatment "1" --> "1" Patient : belongs to

Appointment --> NotificationService : "notifies"
Treatment --> TreatmentManagementSystem : "managed by"

@enduml
```

4. Système de Gestion de l'École

Contexte :

Vous travaillez pour **EduManage**, une entreprise spécialisée dans les systèmes de gestion pour les institutions éducatives. Votre mission est de modéliser un système de gestion scolaire qui permet de gérer les étudiants, les enseignants, les cours et les inscriptions.

Tâches à réaliser :

1. **Créer un diagramme de classes pour le système de gestion scolaire en utilisant UMLet.**

Étapes à suivre :

1. **Définir les packages principaux :**
   - `School`
   - `Persons`
   - `Courses`
   - `Registration`
2. **Définir les classes principales et interfaces :**
   - `Persons` :
     - **Person** (Classe abstraite)
     - **Student** (Hérite de `Person`)
     - **Teacher** (Hérite de `Person`)
   - `Courses` :
     - **Course**
     - **Subject** (Interface)
   - `Registration` :
     - **Registration**
3. **Ajouter les Attributes et Methods pour chaque classe avec différentes visibilités et concepts avancés :**
   - **Person** (Classe abstraite) :

        - Attributes :
            - #name : String
            - #address : String
            - #telephone : String
        - Methods :
            - +getName() : String
            - +setName(name : String) : void
    - **Student** :
        - Attributes :
            - -studentNumber : String
            - -dateBirth : Date
        - Methods :
            - +registerCourse() : void
            - +unregisterCourse() : void
    - **Teacher** :
        - Attributes :
            - -speciality: String
            - -identifier : String
        - Methods :
            - +teachCourse() : void
            - +assessStudent() : void
    - **Course** :
        - Attributes :
            - +courseName : String
            - +courseCode : String
            - +description : String
        - Methods :
            - +addCourse() : void
            - +removeCourse() : void
    - **Registration** :
        - Attributes :
            - +registrationDate : Date
            - #status : String

        - Methods :
            - +validateRegistration() : void
            - +cancelRegistration() : void
    - **Subject** (Interface) :
        - Methods :
            - +getDescription() : String
4. **Définir les relations entre les classes :**
    - Un **Student** peut être inscrit à plusieurs **Course** (Association).
    - Un **Teacher** peut enseigner plusieurs **Course** (Association).
    - Un **Course** peut avoir plusieurs **Student** inscrits (Aggregation) et plusieurs **Teacher** (Aggregation).
    - Une **Registration** est associée à un seul **Student** et un seul **Course** (Composition).

Conseils pour la réalisation :

Lorsque vous créez votre diagramme de classes, assurez-vous de bien définir les visibilités des Attributes et des Methods pour chaque classe. Utilisez les concepts avancés d'UML tels que les packages, les classes abstraites, les interfaces, l'héritage, l'agrégation et la composition pour rendre votre modèle plus réaliste et structuré. Pensez à la manière dont chaque classe interagit avec les autres et représentez ces relations avec précision. Prenez le temps de bien organiser votre diagramme pour qu'il soit propre et lisible.

Bonne chance !

Correction :

Packages et Classes :

- `Persons`
  - **Personne** (Classe abstraite)
    - Attributes :
      - #name : String
      - #address : String
      - #telephone : String
    - Methods :
      - +getName() : String
      - +setName(nom : String) : void
  - **Student** (Hérite de `Personne`)
    - Attributes :
      - -studentNumber : String
      - -dateBirth : Date
    - Methods :
      - +registerCourse() : void
      - +unregisterCourse() : void
  - **Teacher** (Hérite de `Personne`)
    - Attributes :
      - -speciality : String
      - -identifier : String
    - Methods :
      - +teachCourse() : void
      - +assessStudent() : void
- `Courses`
  - **Course**
    - Attributes :
      - +courseName : String
      - +courseCode : String
      - +description : String

        - Methods :
            - +addCourse() : void
            - +removeCourse() : void
    - **Subject** (Interface)
        - Methods :
            - +getDescription() : String
- `Registrations`
    - **Registration**
        - Attributes :
            - +registrationDate : Date
            - #status : String
        - Methods :
            - +validateRegistration() : void
            - +cancelRegistration() : void

Relations :

- Un **Student** peut être inscrit à plusieurs **Course** (Association).
- Un **Teacher** peut enseigner plusieurs **Course** (Association).
- Un **Course** peut avoir plusieurs **Student** inscrits (Aggregation) et plusieurs **Teacher** (Aggregation).
- Une **Registration** est associée à un seul **Student** et un seul **Course** (Composition).

Diagramme UMLet

Pour créer le diagramme dans UMLet, suivez ces étapes :

1. **Créer les packages :**
    - Sélectionnez l'élément "Package" et placez-le sur le canevas. Nommez-le "Persons".
    - Répétez pour "Courses" et "Registrations".
2. **Créer les classes et interfaces :**

- Sélectionnez l'élément "Class" et placez-le dans le package "Persons". Nommez-le "Person" et indiquez qu'il s'agit d'une classe abstraite.
- Ajoutez les classes "Student" et "Teacher" dans le package "Persons".
- Ajoutez les classes "Course" et l'interface "Subject" dans le package "Courses".
- Ajoutez la classe "Registrations" dans le package "Registration".

3. **Définir les relations :**
   - Utilisez l'élément "Line" pour relier les classes en fonction des relations définies. Par exemple, reliez "Student" à "Course" avec une association, "Teacher" à "Course" avec une association, "Course" à "Student" et "Teacher" avec une agrégation, et "Registration" à "Teacher" et "Course" avec une composition.

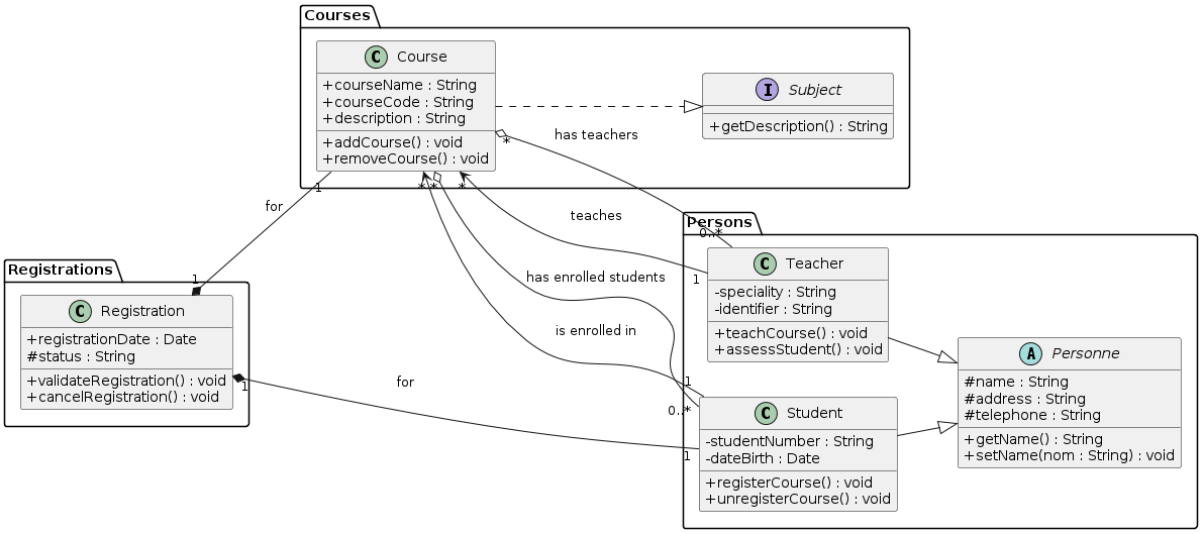
Points informations utiles

10. **Visibilité des Méthodes** :
    - Les méthodes d'une interface sont implicitement `public` en Java, (c'est-à-dire qu'une méthode dans une interface ne peut pas être `protected` ou `private`, elles doivent être `public)` , donc il n'est pas nécessaire de spécifier le modificateur de visibilité `+`.
11. **Granularité** :
    - Dans le corrigé de la version améliroer, les noms des méthodes sont très génériques. Il pourrait être bénéfique de rendre les méthodes plus spécifiques à leur contexte d'utilisation. Par exemple, `createUser`, `deleteDocument`, etc., pourraient être plus explicites si l'interface est destinée à un domaine particulier.
12. **Segmentation des Responsabilités** :

Si les méthodes représentent des actions très différentes, il pourrait être intéressant de scinder l'interface en plusieurs interfaces plus petites et plus ciblées, chacune représentant un ensemble cohérent de responsabilités (principe de séparation des interfaces, ISP - Interface Segregation Principle).

Modélisation graphique

Réponse attendu pour l'exercice de base :



Explications

Packages, Interfaces et Classes

- Classe `Personne` abstraite : Contient des attributs communs comme `name`, `address`, `telephone`, et est étendue par les classes `Student` et `Teacher`.
- Classes `Student` et `Teacher` : Héritent de `Personne` et ajoutent des attributs spécifiques (`studentNumber`, `speciality`). Elles implémentent des méthodes spécifiques (`registerCourse`, `teachCourse`).
- Interface `Subject` : Définie pour fournir un comportement commun à tous les cours, mais n'est pas utilisée directement ici.

Relations, Compositions et Agrégations

Relations :

- Entre `Student` et `Course` : Un étudiant peut être inscrit à plusieurs cours (`is enrolled in`).
- Entre `Teacher` et `Course` : Un enseignant peut enseigner plusieurs cours (`teaches`).

Agrégations :

- Entre `Course` et `Student` : Un cours peut avoir plusieurs étudiants inscrits (`has enrolled students`).
- Entre `Course` et `Teacher` : Un cours peut avoir plusieurs enseignants (`has teachers`).

Compositions :

- Entre `Registration` et `Student` : Une inscription est associée à un seul étudiant (`for`).
- Entre `Registration` et `Course` : Une inscription est associée à un seul cours (`for`).

Services Tiers

- Aucun service tiers inclus, mais des services comme un `NotificationService` pourraient être ajoutés pour gérer les notifications.

Ce diagramme offre une vue claire et organisée du système de gestion des personnes, des cours et des inscriptions. En adoptant ces améliorations, le système devient plus structuré, plus cohérent et mieux adapté à la modélisation des interactions complexes entre les entités.

PlantUML (Un must have !)

Accéder à l'adresse suivante PlantUML Web Server (https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) et copier coller le code suivant :

Réponse attendu pour l'exercice de base :

```
@startuml

left to right direction
skinparam classAttributeIconSize 0

package Persons {
    abstract class Personne {
        #name : String
        #address : String
        #telephone : String
        + getName() : String
        + setName(nom : String) : void
    }

    class Student {
        - studentNumber : String
        - dateBirth : Date
        + registerCourse() : void
        + unregisterCourse() : void
    }
    Student --|> Personne

    class Teacher {
        - speciality : String
        - identifier : String
        + teachCourse() : void
        + assessStudent() : void
    }
    Teacher --|> Personne
}

package Courses {
    class Course {
        + courseName : String
        + courseCode : String
        + description : String
        + addCourse() : void
        + removeCourse() : void
```

```
    }

    interface Subject {
        + getDescription() : String
    }
    Course ..|> Subject
}

package Registrations {
    class Registration {
        + registrationDate : Date
        #status : String
        + validateRegistration() : void
        + cancelRegistration() : void
    }
}


Student "1" --> "*" Course : is enrolled in
Teacher "1" --> "*" Course : teaches

Course -[hidden]--> Student : spacer
Course "*" o-- "0..*" Student : has enrolled students
Course "*" o-- "0..*" Teacher : has teachers


Registration "1" *-- "1" Student : for
Registration "1" *-- "1" Course : for

@enduml
```

5. Système de Gestion de Projet

Contexte :

Vous travaillez pour **ProjectHub**, une entreprise spécialisée dans les systèmes de gestion de projets. Votre mission est de modéliser un système de gestion de projet qui permet de gérer les projets, les tâches, les employés et les rapports de progression.

Tâches à réaliser :

1. **Créer un diagramme de classes pour le système de gestion de projet en utilisant UMLet.**

Étapes à suivre :

1. **Définir les packages principaux :**
    - `Project`
    - `Tasks`
    - `Employees`
    - `Reports`
2. **Définir les classes principales et interfaces :**
    - `Project`:
        - **Project**
    - `Tasks`:
        - **Task**
        - **ComplexTask** (Hérite de `Task`)
        - **SubTask** (Hérite de `Task`)
        - **ITask** (Interface)
    - `Employees`:
        - **Employee**
        - **Manager** (Hérite de `Employee`)
    - `Reports` :
        - **Report**

3. **Ajouter les Attributes et Methods pour chaque classe avec différentes visibilités et concepts avancés :**
    - **Project**:
        - Attributes :
            - +name : String
            - #budget : double
            - -projectCode : String
            - -startDate : Date
            - -EndDate : Date
        - Methods :
            - +addTask() : void
            - +removeTask() : void
    - **Task**(Implémente `ITask`) :
        - Attributes :
            - +TaskName : String
            - #description : String
            - -estimatedDuration : int
            - -status : String
        - Methods :
            - +assignEmployee() : void
            - #markComplete() : void
    - **ComplexTask**(Hérite de `Task`) :
        - Attributes :
            - +subtasks : List<Subtasks>
        - Methods :
            - +addSubTask() : void
            - +removeSubTask() : void
    - **Subtask** (Hérite de `Task`)
        - Methods :
            - +completedTask() : void
    - **ITask** (Interface) :
        - Methods :
            - +getStatus() : String

- **Employee** :
  - Attributes :
    - +name: String
    - -identifier : String
    - #post : String
  - Methods :
    - +takeTask() : void
    - #submitReport() : void
- **Manager** (Hérite de `Employee`)
  - Methods :
    - +superviseProject() : void
    - +approveTask() : void
- **Report** :
  - Attributes :
    - +reportDate : Date
    - #content : String
    - -note : String
  - Methods :
    - +createReport() : void
    - #approveReport() : void

4. **Définir les relations entre les classes :**
   - Un **Project** peut contenir plusieurs **Tasks** (Aggregation).
   - Une **ComplexTask** peut contenir plusieurs **Subtask** (Composition).
   - Une **Task** peut être assignée à plusieurs **Employee** (Association).
   - Un **Employee** peut rédiger plusieurs **Report** (Aggregation).
   - Un **Report** est associé à une seule **Task** et un seul **Employee** (Composition).
   - Un **Manager** supervise plusieurs **Project** (Association).

Conseils pour la réalisation :

Lorsque vous créez votre diagramme de classes, assurez-vous de bien définir les visibilités des Attributes et des Methods pour chaque classe. Utilisez les concepts avancés d'UML tels que les packages, les interfaces, l'héritage, l'implémentation, l'agrégation et la composition pour rendre votre modèle plus réaliste et structuré. Pensez à la manière dont chaque classe interagit avec les autres et représentez ces relations avec précision. Prenez le temps de bien organiser votre diagramme pour qu'il soit propre et lisible.

Bonne chance !

Correction :

Packages et Classes :

- `Project`
  - **Project**
    - Attributes :
      - +name : String
      - #budget : double
      - -projectCode : String
      - -startDate : Date
      - -EndDate : Date
    - Methods :
      - +addTask() : void
      - +removeTask() : void
- `Tasks`
  - **Task**(Implémente `ITask`)
    - Attributes :
      - +TaskName : String
      - #description : String
      - -estimatedDuration : int
      - -status : String
    - Methods :
      - +assignEmployee() : void
      - #markComplete() : void
  - **ComplexTask** (Hérite de `Task`)
    - Attributes :
      - +subtasks : List<Subtasks>
    - Methods :
      - +addSubTask() : void
      - +removeSubTask() : void
  - **SubTask** (Hérite de `Task`)
    - Methods :

            - +completedTask() : void
        - **ITask** (Interface)
            - Methods :
                - +getStatus() : String
- `Employees`
    - **Employee**
        - Attributes :
            - +name : String
            - -identifier : String
            - #position : String
        - Methods :
            - +takeTask() : void
            - #submitReport() : void
    - **Manager** (Hérite de `Employee`)
        - Methods :
            - +superviseProject() : void
            - +approveTask() : void
- `Reports`
    - **Report**
        - Attributes :
            - +reportDate : Date
            - #content : String
            - -note : String
        - Methods :
            - +createReport() : void
            - #approveReport() : void

Relations :

- Un **Project** peut contenir plusieurs **Tasks** (Aggregation).
- Une **ComplexTask** peut contenir plusieurs **Subtask** (Composition).
- Une **Task** peut être assignée à plusieurs **Employee** (Association).
- Un **Employee** peut rédiger plusieurs **Report** (Aggregation).

- Un **Report** est associé à une seule **Task** et un seul **Employee** (Composition).
- Un **Manager** supervise plusieurs **Project** (Association).

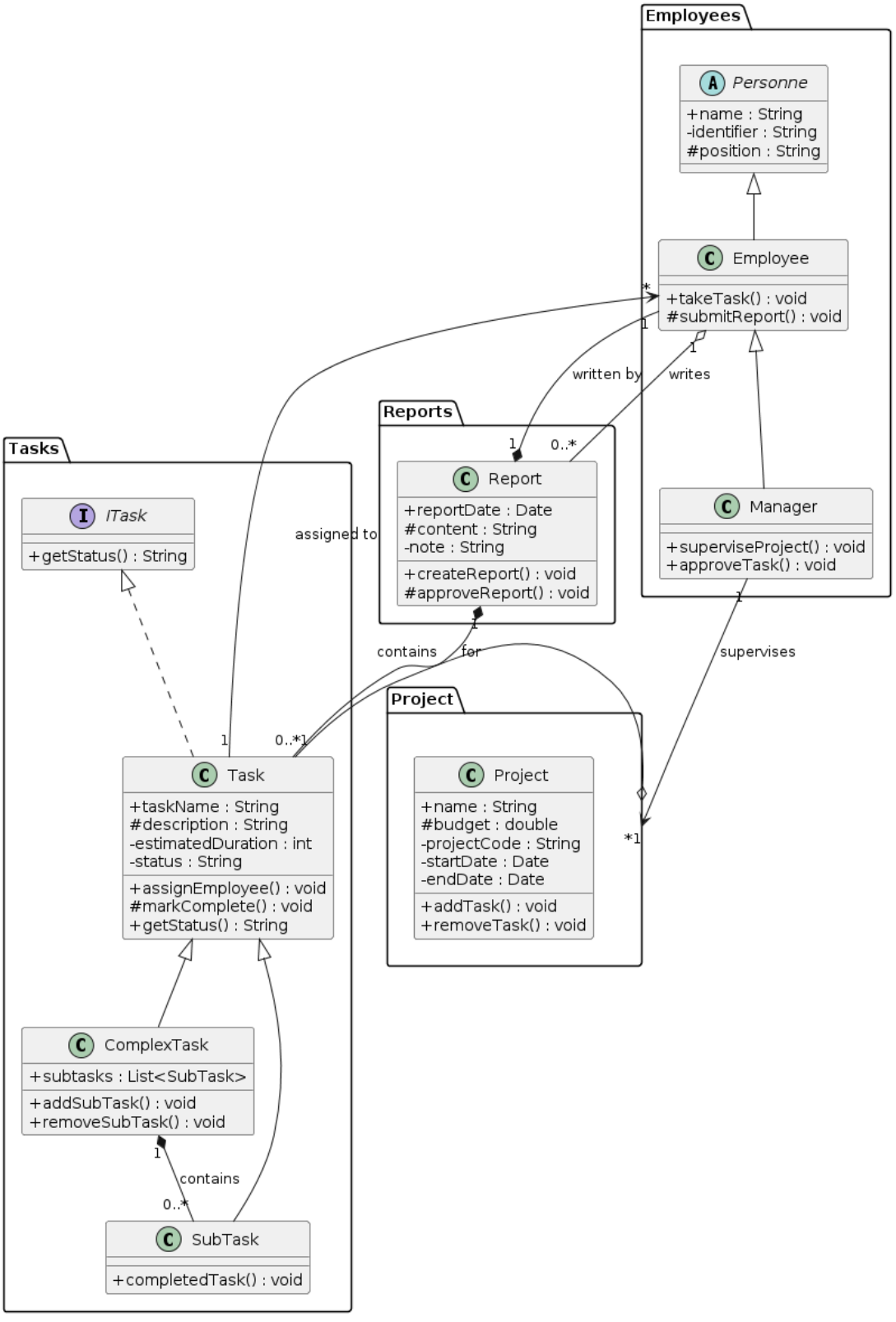
Diagramme UMLet

Pour créer le diagramme dans UMLet, suivez ces étapes :

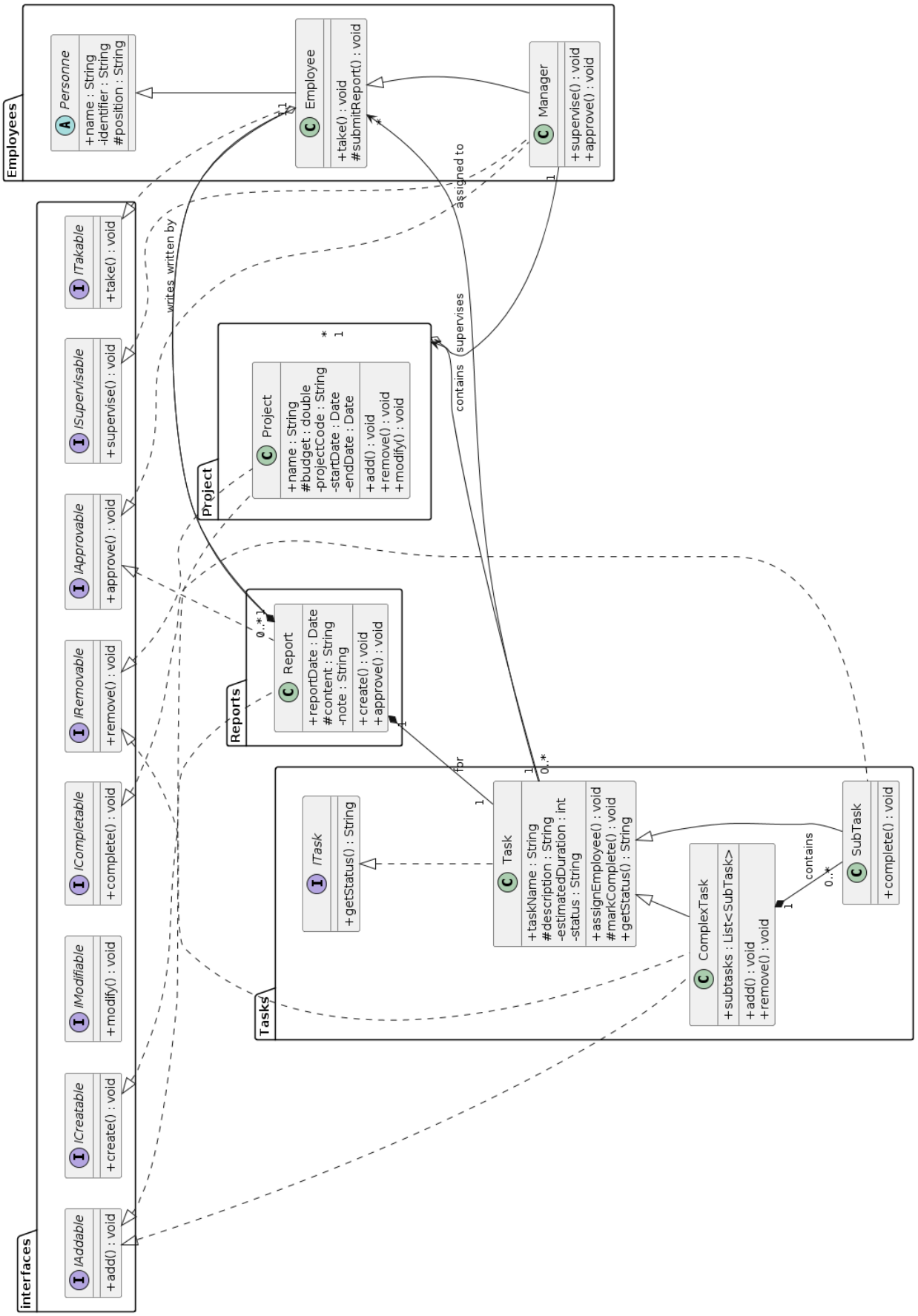
1. **Créer les packages :**
    - Sélectionnez l'élément "Package" et placez-le sur le canevas. Nommez-le "Projects".
    - Répétez pour "Tasks", "Employees" et "Report".
2. **Créer les classes et interfaces :**
    - Sélectionnez l'élément "Class" et placez-le dans le package "Projects". Nommez-le "Project".
    - Ajoutez les classes "Task", "ComplexTask", "Subtask" et l'interface "ITask" dans le package "Tasks".
    - Ajoutez les classes "Employee" et "Manager" dans le package "Employees".
    - Ajoutez la classe "Report" dans le package "Reports".
3. **Définir les relations :**
    - Utilisez l'élément "Line" pour relier les classes en fonction des relations définies. Par exemple, reliez "Project" à "Task" avec une agrégation, "ComplexTask" à "Subtask" avec une composition, et "Report" à "Task" et "Employee" avec une composition.

Modélisation graphique

Réponse attendu pour l'exercice de base :



Réponse enrichi pour l'exercice de base :



Points informations utiles

13. **Visibilité des Méthodes** :
    - Les méthodes d'une interface sont implicitement `public` en Java, (c'est-à-dire qu'une méthode dans une interface ne peut pas être `protected` ou `private`, elles doivent être `public)` , donc il n'est pas nécessaire de spécifier le modificateur de visibilité `+`.
14. **Granularité** :
    - Dans le corrigé de la version améliroer, les noms des méthodes sont très génériques. Il pourrait être bénéfique de rendre les méthodes plus spécifiques à leur contexte d'utilisation. Par exemple, `createUser`, `deleteDocument`, etc., pourraient être plus explicites si l'interface est destinée à un domaine particulier.
15. **Segmentation des Responsabilités** :

Si les méthodes représentent des actions très différentes, il pourrait être intéressant de scinder l'interface en plusieurs interfaces plus petites et plus ciblées, chacune représentant un ensemble cohérent de responsabilités (principe de séparation des interfaces, ISP - Interface Segregation Principle).

Explications des Améliorations

Classes et Interfaces

Classe `Personne` Abstraite :

- **Description** : La classe abstraite `Personne` contient des attributs communs tels que `name`, `identifier`, et `position`. Cette abstraction permet de centraliser les attributs communs à plusieurs types de personnes, réduisant ainsi la redondance.
- **Héritage** : Les classes `Employee` (employé) et `Manager` (gestionnaire) étendent `Personne`, héritant de ces attributs communs.

Interface `IAddable` :

- **Description** : L'interface `IAddable` définit la méthode générique `add()` pour ajouter des éléments.
- **Implémentation** : Les classes `Project` (projet) et `ComplexTask` (tâche complexe) implémentent cette interface, unifiant ainsi la méthode d'ajout pour différents types d'entités.

Interface `IRemovable` :

- **Description** : L'interface `IRemovable` définit la méthode générique `remove()` pour supprimer des éléments.
- **Implémentation** : Les classes `Project` (projet) et `ComplexTask` (tâche complexe) implémentent cette interface, unifiant ainsi la méthode de suppression pour différents types d'entités.

Interface `ICreatable` :

- **Description** : L'interface `ICreatable` définit la méthode générique `create()` pour créer des éléments.
- **Implémentation** : La classe `Report` (rapport) implémente cette interface, unifiant ainsi la méthode de création.

Interface `IModifiable` :

- **Description** : L'interface `IModifiable` définit la méthode générique `modify()` pour modifier des éléments.
- **Implémentation** : La classe `Project` (projet) implémente cette interface, unifiant ainsi la méthode de modification.

Interface `ICompletable` :

- **Description** : L'interface `ICompletable` définit la méthode générique `complete()` pour marquer des éléments comme terminés.
- **Implémentation** : La classe `SubTask` (sous-tâche) implémente cette interface, unifiant ainsi la méthode de complétion.

Interface `IApprovable` :

- **Description** : L'interface `IApprovable` définit la méthode générique `approve()` pour approuver des éléments.
- **Implémentation** : Les classes `Manager` (gestionnaire) et `Report` (rapport) implémentent cette interface, unifiant ainsi la méthode d'approbation.

Interface `ITakable` :

- **Description** : L'interface `ITakable` définit la méthode générique `take()` pour prendre en charge des tâches.
- **Implémentation** : La classe `Employee` (employé) implémente cette interface, unifiant ainsi la méthode de prise en charge.

Interface `ISupervisable` :

- **Description** : L'interface `ISupervisable` définit la méthode générique `supervise()` pour superviser des projets.
- **Implémentation** : La classe `Manager` (gestionnaire) implémente cette interface, unifiant ainsi la méthode de supervision.

Compositions et Agrégations

Composition :

- **Relation entre `Project` et `Task`** : La relation entre `Project` et `Task` est une composition, car une tâche appartient à un projet spécifique et ne peut pas exister sans lui.
- **Relation entre `ComplexTask` et `SubTask`** : La relation entre `ComplexTask` et `SubTask` est une composition, car une sous-tâche appartient à une tâche complexe spécifique et ne peut pas exister sans elle.

Agrégation :

- **Relation entre `Employee` et `Report`** : La relation entre `Employee` et `Report` est une agrégation, car un rapport est associé à un employé mais peut exister indépendamment d'un employé spécifique. Un employé peut écrire plusieurs rapports, mais les rapports peuvent aussi être définis et utilisés indépendamment.

Services Tiers

- **NotificationService** : Cette classe serait ajoutée pour envoyer des notifications, telles que des rappels de tâches ou des messages d'approbation. Elle permet de gérer les communications de manière centralisée.
- **TaskManagementSystem** : Cette classe serait ajoutée pour gérer les tâches des employés. Elle centralise la gestion des tâches, facilitant ainsi la maintenance et l'extension du système.

Relations Directionnelles

Description : Les relations directionnelles dans le diagramme montrent clairement les interactions entre les différentes classes et services, indiquant quelle classe utilise ou dépend de quelle autre classe ou service.

Exemple :

- La classe `Task` utilise `Employee` pour assigner des tâches.
- La classe `Manager` utilise `Project` pour superviser les projets.

Vue Globale et Réalisme du Système

Description : Ce diagramme offre une vue plus détaillée et réaliste du système de gestion des tâches et des projets. En incluant des éléments pratiques comme les services tiers et des relations complexes, il reflète mieux un environnement de développement réel.

Avantages : Cette approche améliore la compréhension des interactions et des dépendances entre les différentes parties du système, facilitant ainsi le développement, la maintenance et l'évolution du système.

En adoptant ces améliorations, le diagramme devient non seulement plus structuré et cohérent, mais il reflète également une conception orientée objet efficace et adaptable à un environnement de développement réel, tel que requis pour une application en Java. Regrouper les interfaces dans des packages spécifiques améliore encore la lisibilité et la maintenabilité du code.

PlantUML (Un must have !)

Accéder à l'adresse suivante PlantUML Web Server (https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) et copier coller le code suivant :

Réponse attendu pour l'exercice de base :

```
@startuml
skinparam classAttributeIconSize 0

package Project {
    class Project {
        + name : String
        # budget : double
        - projectCode : String
        - startDate : Date
        - endDate : Date
        + addTask() : void
        + removeTask() : void
    }
}

package Tasks {
    interface ITask {
        + getStatus() : String
    }

    class Task implements ITask {
        + taskName : String
        # description : String
        - estimatedDuration : int
        - status : String
        + assignEmployee() : void
        # markComplete() : void
        + getStatus() : String
    }

    class ComplexTask extends Task {
        + subtasks : List<SubTask>
        + addSubTask() : void
        + removeSubTask() : void
    }

    class SubTask extends Task {
        + completedTask() : void
```

```
    }
}

package Employees {
    abstract class Personne {
        + name : String
        - identifier : String
        # position : String
    }

    class Employee extends Personne {
        + takeTask() : void
        # submitReport() : void
    }

    class Manager extends Employee {
        + superviseProject() : void
        + approveTask() : void
    }
}

package Reports {
    class Report {
        + reportDate : Date
        # content : String
        - note : String
        + createReport() : void
        # approveReport() : void
    }
}


Project "1" o-- "0..*" Task : contains
Employee "1" o-- "0..*" Report : writes


ComplexTask "1" *-- "0..*" SubTask : contains
Report "1" *-- "1" Task : for
Report "1" *-- "1" Employee : written by


Task "1" --> "*" Employee : assigned to
Manager "1" --> "*" Project : supervises
@enduml
```

Réponse enrichi pour l'exercice de base :

```
@startuml

skinparam classAttributeIconSize 0

package interfaces {
    interface IAddable {
        + add() : void
    }

    interface ICreatable {
        + create() : void
    }

    interface IModifiable {
        + modify() : void
    }

    interface IRemovable {
        + remove() : void
    }

    interface ICompletable {
        + complete() : void
    }

    interface IApprovable {
        + approve() : void
    }

    interface ITakable {
        + take() : void
    }

    interface ISupervisable {
        + supervise() : void
    }
}

package Project {
    class Project implements interfaces.IAddable, interfaces.IRemovable {
        + name : String
        # budget : double
        - projectCode : String
        - startDate : Date
        - endDate : Date
        + add() : void
        + remove() : void
```

```
        + modify() : void
    }
}

package Tasks {
    interface ITask {
        + getStatus() : String
    }

    class Task implements ITask {
        + taskName : String
        # description : String
        - estimatedDuration : int
        - status : String
        + assignEmployee() : void
        # markComplete() : void
        + getStatus() : String
    }

    class ComplexTask extends Task implements interfaces.IAddable,
interfaces.IRemovable {
        + subtasks : List<SubTask>
        + add() : void
        + remove() : void
    }

    class SubTask extends Task implements interfaces.ICompletable {
        + complete() : void
    }
}

package Employees {
    abstract class Personne {
        + name : String
        - identifier : String
        # position : String
    }

    class Employee extends Personne implements interfaces.ITakable {
        + take() : void
        # submitReport() : void
    }

    class Manager extends Employee implements interfaces.IApprovable,
interfaces.ISupervisable {
        + supervise() : void
        + approve() : void
    }
}

package Reports {
```

```
    class Report implements interfaces.ICreatable, interfaces.IApprovable {
        + reportDate : Date
        # content : String
        - note : String
        + create() : void
        + approve() : void
    }
}

Project "1" o-- "0..*" Task : contains
Employee "1" o-- "0..*" Report : writes

ComplexTask "1" *-- "0..*" SubTask : contains
Report "1" *-- "1" Task : for
Report "1" *-- "1" Employee : written by

Task "1" --> "*" Employee : assigned to
Manager "1" --> "*" Project : supervises

@enduml
```

6. Système de Gestion de la Vente en Ligne

Contexte :

Vous travaillez pour **E-Commerce Solutions**, une entreprise spécialisée dans les systèmes de gestion de la vente en ligne. Votre mission est de modéliser un système de gestion de la vente en ligne qui permet de gérer les clients, les produits, les commandes et les paiements.

Tâches à réaliser :

1. **Créer un diagramme de classes pour le système de gestion de la vente en ligne en utilisant UMLet.**

Étapes à suivre :

1. **Définir les packages principaux :**
   - `Customers`
   - `Products`
   - `CustomersOrders`
   - `Payments`
2. **Définir les classes principales et interfaces :**
   - `Customers` :
     - **Customer**
     - **Address** (Classe)
     - **Contact** (Interface)
   - `Products` :
     - **Product**
     - **Stock**
   - `CustomersOrders` :
     - **CustomerOrder**
     - **OnlineOrderableProduct**(Classe)

- `Paiements` :
  - **Payment**
  - **PaymentMode** (Interface)

3. **Ajouter les Attributes et Methods pour chaque classe avec différentes visibilités et concepts avancés :**
   - **Customer** :
     - Attributes :
       - +name : String
       - #address : Adresse
       - -email : String
       - -phone : String
     - Methods :
       - +makeOrder() : void
       - +cancelOrder() : void
   - **Address** :
     - Attributes :
       - +street : String
       - +city : String
       - +postCode : String
     - Methods :
       - +getCompleteAddress() : String
   - **Product** :
     - Attributes :
       - +productName : String
       - #price : double
       - -stock : Stock
     - Methods :
       - +addProduct() : void
       - #updateProduct() :Void
   - **Stock** :
     - Attributes :
       - +quantityAvailable : int
       - +quantityReserved : int

        - Methods :
            - +addStock() : void
            - +removeStock() : void
    - **CustomerOrder** :
        - Attributes :
            - +orderNumber : String
            - #orderDate : Date
            - -status : String
            - -OnlineOrderProduct : List<OnlineOrderProduct>
        - Methods :
            - +validateOrder() : void
            - +cancelOrder() : void
    - **OnlineOrderableProduct** :
        - Attributes :
            - +quantity : int
            - #unitPrice : double
        - Methods :
            - +calculateAmount() : double
    - **Payment** :
        - Attributes :
            - +amount : double
            - #paymentDate : Date
            - -paymentMode : paymentMode
        - Methods :
            - +makePayment() : void
            - +cancelPayment() : void
    - **PaymentMode** (Interface) :
        - Methods :
            - +makeTransaction() : void
    - **BankCard** (Implémente `PaymentMode`) :
        - Methods :
            - +makeTransaction() : void
    - **PayPal** (Implémente `PaymentMode`) :

        - Methods :
            - +makeTransaction() : void

Relations :

- Un **Customer** peut passer plusieurs **CustomerOrder** (Association).
- Une **CustomerOrder** peut contenir plusieurs **OnlineOrderableProduct** (Composition).
- Un **Product** est associé à un seul **Stock** (Composition).
- Un **Stock** est associé à un seul **Product** (Composition).
- Une **CustomerOrder** est associée à un seul **Payment** (Composition).
- Un **Payment** utilise un seul **PaymentMode** (Association).

Conseils pour la réalisation :

Dans ce diagramme de classes, nous avons introduit des concepts avancés tels que la composition, l'implémentation d'interfaces, et les agrégations pour modéliser les relations entre les classes. Assurez-vous de comprendre chaque concept et son utilisation appropriée dans le contexte du système de gestion de la vente en ligne. Utilisez les visibilités des Attributes et Methods pour contrôler l'accès aux différentes parties du système. Organisez vos classes de manière logique en les regroupant dans des packages pertinents. N'hésitez pas à consulter la documentation d'UML pour plus d'informations sur ces concepts. Bonne modélisation !

Correction :

Packages et Classes :

- `Customers`
  - **Customer**
    - Attributes :
      - +name : String
      - #address : Adresse
      - -email : String
      - -phone : String
    - Methods :
      - +makeOrder() : void
      - +cancelOrder() : void
  - **Address** :
    - Attributes :
      - +street : String
      - +city : String
      - +postCode : String
    - Methods :
      - +getCompleteAddress() : String
  - **Contact** :
    - Methods :
      - +getCompleteAddress() : String
- `Products`
  - **Produit**
    - Attributes :
      - +productName : String
      - #price : double
      - -stock : Stock
    - Methods :
      - +addProduct() : void
      - #updateProduct() : void

- **Stock**
  - Attributes :
    - +quantityAvailable : int
    - +quantityReserved : int
  - Methods :
    - +addStock() : void
    - +removeStock() : void

- `CustomersOrders`
  - **CustomerOrder**
    - Attributes :
      - +orderNumber : String
      - #orderDate : Date
      - -status : String
      - -OnlineOrderProduct : List<OnlineOrderProduct>
    - Methods :
      - +validateOrder() : void
      - +cancelOrder() : void
  - **OnlineOrderableProduct**
    - Attributes :
      - +quantity : int
      - #unitPrice : double
    - Methods :
      - +calculateAmount() : double

- `Payments`
  - **Payment**
    - Attributes :
      - +amount : double
      - #paymentDate : Date
      - -paymentMode : paymentMode
    - Methods :
      - +makePayment() : void
      - +cancelPayment() : void
  - **ModePaiement** (Interface)
    - Methods :

            - +makeTransaction() : void
    - **CarteBancaire** (Implémente `PaymentMode`)
        - Methods :
            - +makeTransaction() : void
    - **PayPal** (Implémente `PaymentMode`)
        - Methods :
            - +makeTransaction() : void

Relations :

- Un **Customer** peut passer plusieurs **CustomerOrder** (Association).
- Une **CustomerOrder** peut contenir plusieurs **OnlineOrderableProduct** (Composition).
- Un **Product** est associé à un seul **Stock** (Composition).
- Un **Stock** est associé à un seul **Product** (Composition).
- Une **CustomerOrder** est associée à un seul **Payment** (Composition).
- Un **Payment** utilise un seul **PaymentMode** (Association).

Diagramme UMLet

Pour créer le diagramme dans UMLet, suivez ces étapes :

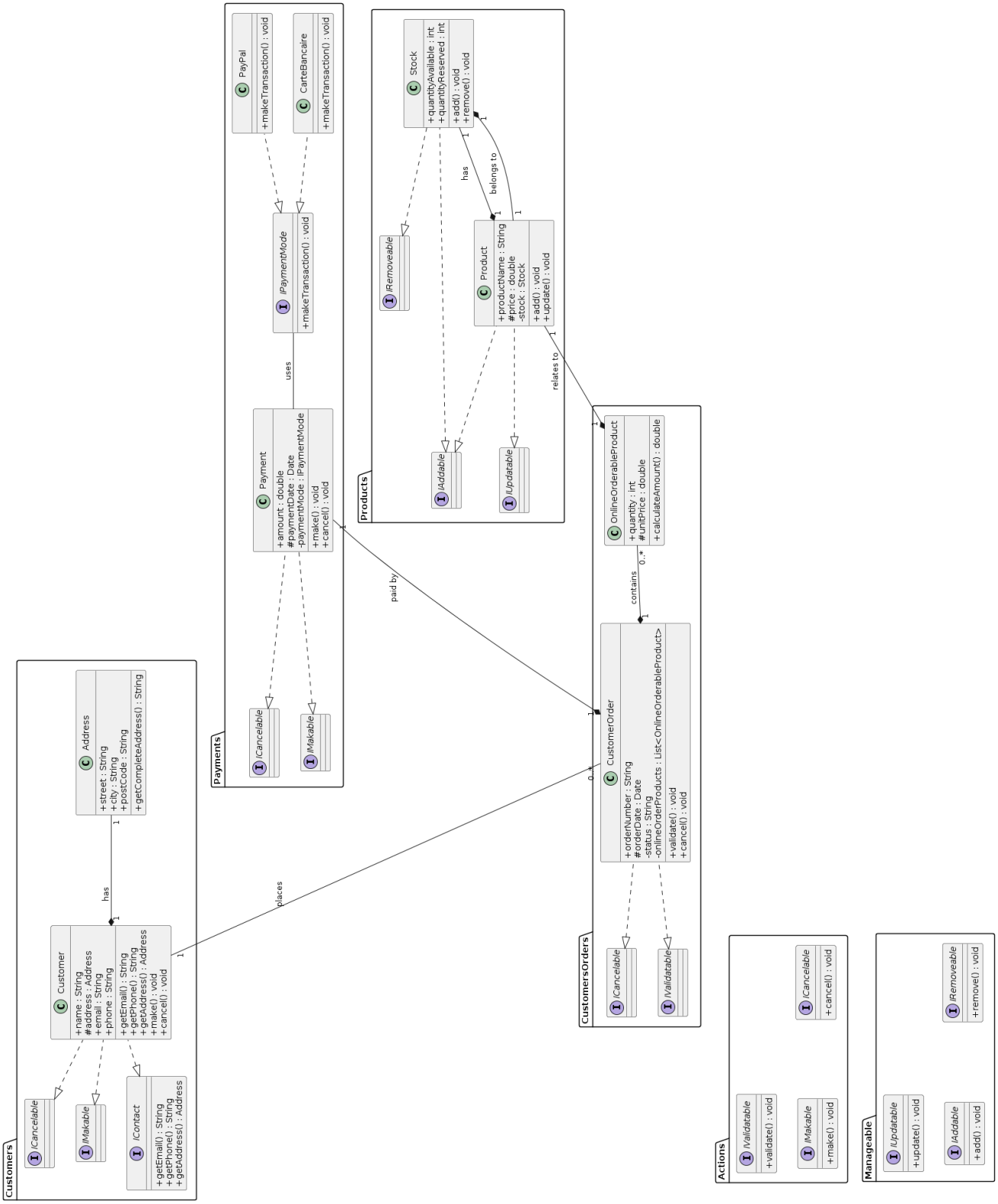
1. **Créer les packages :**
    - Sélectionnez l'élément "Package" et placez-le sur le canevas. Nommez-le "Customers".
    - Répétez pour "Products", "CustomersOrders" et "Payments".
2. **Créer les classes et interfaces :**
    - Sélectionnez l'élément "Class" et placez-le dans le package "Customers". Nommez-le "Customer".
    - Ajoutez les classes "Address" et "Contact" dans le package "Customers".
    - Ajoutez la classe "Product" dans le package "Products" et la classe "Stock" dans le package "Products".

- Ajoutez les classes "CustomerOrder" et "OnlineOrderableProduct" dans le package "CustomersOrders".
- Ajoutez les classes "Payment", "PaymentMode", "BankCard" et "PayPal" dans le package "Payments".

3. **Définir les relations :**
   - Utilisez l'élément "Line" pour relier les classes en fonction des relations définies. Par exemple, reliez "Customer" à "CustomerOrder" avec une association, "CustomerOrder" à "OnlineOrderableProduct" avec une composition, et "Payment" à "CustomerOrder" avec une composition.

Modélisation graphique

Réponse enrichi avec la segmentation des responsabilités pour l'exercice de base :



Points informations utiles

16. **Visibilité des Méthodes** :
    - Les méthodes d'une interface sont implicitement `public` en Java, (c'est-à-dire qu'une méthode dans une interface ne peut pas être `protected` ou `private`, elles doivent être `public)` , donc il n'est pas nécessaire de spécifier le modificateur de visibilité `+`.
17. **Granularité** :
    - Dans le corrigé de la version améliroer, les noms des méthodes sont très génériques. Il pourrait être bénéfique de rendre les méthodes plus spécifiques à leur contexte d'utilisation. Par exemple, `createUser`, `deleteDocument`, etc., pourraient être plus explicites si l'interface est destinée à un domaine particulier.
18. **Segmentation des Responsabilités** :
    - Si les méthodes représentent des actions très différentes, il pourrait être intéressant de scinder l'interface en plusieurs interfaces plus petites et plus ciblées, chacune représentant un ensemble cohérent de responsabilités (principe de séparation des interfaces, ISP - Interface Segregation Principle).

Explications des Améliorations

Classes et Interfaces

Classe Abstraite Customer :

- **Description** : La classe `Customer` contient des attributs communs tels que `name`, `address`, `email` et `phone`. Cette abstraction permet de centraliser les attributs communs à tous les clients, réduisant ainsi la redondance.
- **Implémentation** : La classe `Customer` implémente les interfaces `IContact`, `IMakable`, et `ICancelable`, unifiant les méthodes de contact, de création et d'annulation des commandes.

Interface IContact :

- **Description** : L'interface `IContact` définit des méthodes pour obtenir les informations de contact.
- **Implémentation** : La classe `Customer` implémente cette interface pour fournir les informations de contact comme l'email, le téléphone et l'adresse.

Interface IAddable :

- **Description** : L'interface `IAddable` définit la méthode générique `add()` pour ajouter des éléments.
- **Implémentation** : Les classes `Product` et `Stock` implémentent cette interface, unifiant ainsi la méthode d'ajout pour différents types d'entités.

Interface IUpdatable :

- **Description** : L'interface `IUpdatable` définit la méthode générique `update()` pour mettre à jour des éléments.

- **Implémentation** : La classe `Product` implémente cette interface, unifiant ainsi la méthode de mise à jour des produits.

Interface IRemoveable :

- **Description** : L'interface `IRemoveable` définit la méthode générique `remove()` pour supprimer des éléments.
- **Implémentation** : La classe `Stock` implémente cette interface, unifiant ainsi la méthode de suppression de stock.

Interface IMakable :

- **Description** : L'interface `IMakable` définit la méthode générique `make()` pour créer des éléments.
- **Implémentation** : Les classes `Customer` et `Payment` implémentent cette interface, unifiant ainsi la méthode de création des commandes et des paiements.

Interface IValidatable :

- **Description** : L'interface `IValidatable` définit la méthode générique `validate()` pour valider des éléments.
- **Implémentation** : La classe `CustomerOrder` implémente cette interface, unifiant ainsi la méthode de validation des commandes.

Interface ICancelable :

- **Description** : L'interface `ICancelable` définit la méthode générique `cancel()` pour annuler des éléments.
- **Implémentation** : Les classes `Customer`, `CustomerOrder` et `Payment` implémentent cette interface, unifiant ainsi la méthode d'annulation des commandes et des paiements.

Interface IPaymentMode :

- **Description** : L'interface `IPaymentMode` définit la méthode générique `makeTransaction()` pour effectuer des transactions.

- **Implémentation** : Les classes `CarteBancaire` et `PayPal` implémentent cette interface, unifiant ainsi la méthode de transaction pour différents modes de paiement.

Compositions et Agrégations

Composition :

- **Relation entre `CustomerOrder` et `OnlineOrderableProduct`** : Une commande client (`CustomerOrder`) contient plusieurs produits commandables en ligne (`OnlineOrderableProduct`), mais ces produits n'existent pas indépendamment de la commande.
- **Relation entre `Product` et `Stock`** : Un produit est associé à un stock spécifique et ne peut pas exister sans ce stock.
- **Relation entre `CustomerOrder` et `Payment`** : Une commande client est associée à un paiement spécifique et ne peut pas exister sans ce paiement.
- **Relation entre `Customer` et `Address`** : Une adresse est une partie intégrante du client et ne peut pas exister indépendamment du client.

Agrégation :

- **Relation entre `Customer` et `CustomerOrder`** : Un client peut passer plusieurs commandes, mais les commandes peuvent exister indépendamment des clients spécifiques.
- **Relation entre `CustomerOrder` et `OnlineOrderableProduct`** : Une commande peut inclure plusieurs produits, mais les produits peuvent exister indépendamment des commandes spécifiques.

Services Tiers

- **NotificationService** : Cette classe peut être ajoutée pour envoyer des notifications, telles que des confirmations de commande ou des notifications de paiement. Elle permet de gérer les communications de manière centralisée.
- **OrderManagementSystem** : Cette classe peut être ajoutée pour gérer les commandes des clients. Elle centralise la gestion des commandes, facilitant ainsi la maintenance et l'extension du système.

Relations Directionnelles

- **Description** : Les relations directionnelles dans le diagramme montrent clairement les interactions entre les différentes classes et services, indiquant quelle classe utilise ou dépend de quelle autre classe ou service.
- **Exemple** :
    - La classe `Customer` utilise `Address` pour stocker l'adresse du client.
    - La classe `CustomerOrder` utilise `Product` pour inclure des produits dans les commandes.
    - La classe `Payment` utilise `IPaymentMode` pour effectuer les transactions de paiement.

Vue Globale et Réalisme du Système

- **Description** : Ce diagramme offre une vue plus détaillée et réaliste du système de gestion des clients, des produits, des commandes et des paiements. En incluant des éléments pratiques comme les services

tiers et des relations complexes, il reflète mieux un environnement de développement réel.

- **Avantages** : Cette approche améliore la compréhension des interactions et des dépendances entre les différentes parties du système, facilitant ainsi le développement, la maintenance et l'évolution du système.

PlantUML (Un must have !)

Accéder à l'adresse suivante [PlantUML Web Server](https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) (https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000) et copier coller le code suivant :

Réponse enrichi avec la segmentation des responsabilités pour l'exercice de base :

```
@startuml
left to right direction
skinparam classAttributeIconSize 0

package Manageable {
    interface IAddable {
        +add() : void
    }

    interface IUpdatable {
        +update() : void
    }

    interface IRemoveable {
        +remove() : void
    }
}

package Actions {
    interface IMakable {
        +make() : void
    }

    interface IValidatable {
        +validate() : void
```

```
    }

    interface ICancelable {
        +cancel() : void
    }
}

package Customers {
    interface IContact {
        +getEmail() : String
        +getPhone() : String
        +getAddress() : Address
    }

    class Customer implements IContact, IMakable, ICancelable {
        +name : String
        #address : Address
        +email : String
        +phone : String
        +getEmail() : String
        +getPhone() : String
        +getAddress() : Address
        +make() : void
        +cancel() : void
    }

    class Address {
        +street : String
        +city : String
        +postCode : String
        +getCompleteAddress() : String
    }
}

package Products {
    class Product implements IAddable, IUpdatable {
        +productName : String
        #price : double
        -stock : Stock
        +add() : void
        +update() : void
    }

    class Stock implements IAddable, IRemoveable {
        +quantityAvailable : int
        +quantityReserved : int
        +add() : void
        +remove() : void
    }
}
```

```
package CustomersOrders {
    class CustomerOrder implements IValidatable, ICancelable {
        +orderNumber : String
        #orderDate : Date
        -status : String
        -onlineOrderProducts : List<OnlineOrderableProduct>
        +validate() : void
        +cancel() : void
    }

    class OnlineOrderableProduct {
        +quantity : int
        #unitPrice : double
        +calculateAmount() : double
    }
}

package Payments {
    class Payment implements IMakable, ICancelable {
        +amount : double
        #paymentDate : Date
        -paymentMode : IPaymentMode
        +make() : void
        +cancel() : void
    }

    interface IPaymentMode {
        +makeTransaction() : void
    }

    class CarteBancaire implements IPaymentMode {
        +makeTransaction() : void
    }

    class PayPal implements IPaymentMode {
        +makeTransaction() : void
    }
}

Customer "1" -- "0..*" CustomerOrder : places
CustomerOrder "1" *-- "0..*" OnlineOrderableProduct : contains
Product "1" *-- "1" Stock : has
Stock "1" *-- "1" Product : belongs to
CustomerOrder "1" *-- "1" Payment : paid by
Payment -- IPaymentMode : uses
OnlineOrderableProduct "1" *-- "1" Product : relates to
Customer "1" *-- "1" Address : has

@enduml
```