POLYTECH NICE SOPHIA

UNIVERSITÉ CÔTE D'AZUR

RENDU D1

SophiaTech Eats

TEAM I

Kobbi Fares – SA
Ben Mzoughia Youssef – SA
Younes Hammoud – OPS
Tchani Rajaa – PO
Amraoui Imane – QA

Sommaire:

I- Périmètre fonctionnel---------- 3

II - Glossaire----------4

III - Diagramme de cas d'utilisation---------- 5

IV - Diagramme de classes----------6

IV.1- Vue du diagramme de classes détaillé----------6

IV.2- Vue d'ensemble du diagramme de classes sous la forme d'un diagramme de packages---------- 6

V - Design patterns---------- 7

V.1- Design pattern----------7

V.1.1 – Builder----------7

V.1.2 – Facade:----------7

V.1.3 – Factory:---------- 7

V.2- Patterns envisagés mais non retenus---------- 8

2.5 Diagramme de séquence (Passer une commande avec paiement external)---------- 9

VII - Maquette----------10

VIII - Qualité de code et gestion du projet----------11

Tests et Qualité---------- 11

Types de Tests Supportés---------- 11

Couverture des tests:---------- 11

Vision de la Qualité du Code----------12

Gestion du Projet---------- 12

IX - Auto-évaluation----------12

I- Périmètre fonctionnel

Côté	Exigence	Etat	Remarques si besoin	Points forts	Points faible
Client	C1	DONE			
Client	C2	BLOCKED			
Client	C5	WIP			Temps de livraison non implémenté
Client	C6	BLOCKED			
Client	C7				
Restaurant	R2	WIP			Régime alimentaire non implémenté
Restaurant	R4	DONE	Le history suggestion est prioritaire mais pour une première utilisation on commence par le keyword suggestion toujours		Keyword Suggestion n'a pas trop de mots predefini pour etre plus precis
Restaurant	R5	DONE			
Paiement	P1	DONE			
Paiement	P2	WIP			Chaque commande contient son client. Toute les commandes enregistrer dans OrderManager
Paiement	P3	DONE			
Paiement	P6	DONE			

IA/Catalogue	AI2	DONE	Le service calcul le score aussi et le renvoie et le seuil fixé pour que ca passe c'est 0.75 (75%)		Pas de prise en compte des allergies et des toppings

Table 1.

Table 1. :Périmètre fonctionnel

II - Glossaire

Terme	Type	Définition
UserAccount	Classe / Entité	Représente un utilisateur générique de l'application (client ou restaurant). Contient les informations d'identification et de contact.
StudentAccount	Classe / Entité	Sous-type de UserAccount pour les étudiants. Contient le crédit étudiant (balance) utilisé pour les paiements internes.
OrderStatus	Énumération	Définit l'état actuel d'une commande (PENDING, CANCELLED, VALIDATED).
OrderBuilder	Pattern / Classe	Utilisé pour construire des instances de Order étape par étape (Pattern Builder).
RestaurantBuilder	Pattern / Classe	Permet de construire un objet Restaurant avec un processus fluide et sûr.

OrderFacade	Classe / Facade	Interface simplifiée pour gérer les commandes
RestaurantFacade	Classe / Facade	Fournit une interface unifiée pour les opérations liées aux restaurants
PaymentService	Service	Service applicatif centralisant le processus de paiement externe
IPaymentProcessor	Interface	Décrit le contrat de traitement des paiements. Implémentée par les processeurs internes et externes.
MockedExternalPaymentSystem	Classe	Simule un système de paiement externe pour les tests.
PaymentMethod	Énumération	Définit le mode de paiement choisi (EXTERNAL, INTERNAL).
TimeSlot	Classe	Représente un créneau horaire du restaurant
DishManager	Classe utilitaire	Gère les plats d'un restaurant (ajout, mise à jour).
OrderManager	Classe utilitaire	Centralise la gestion des commandes enregistrées dans le système.

Table 2. Glossaire

III - Diagramme de cas d'utilisation

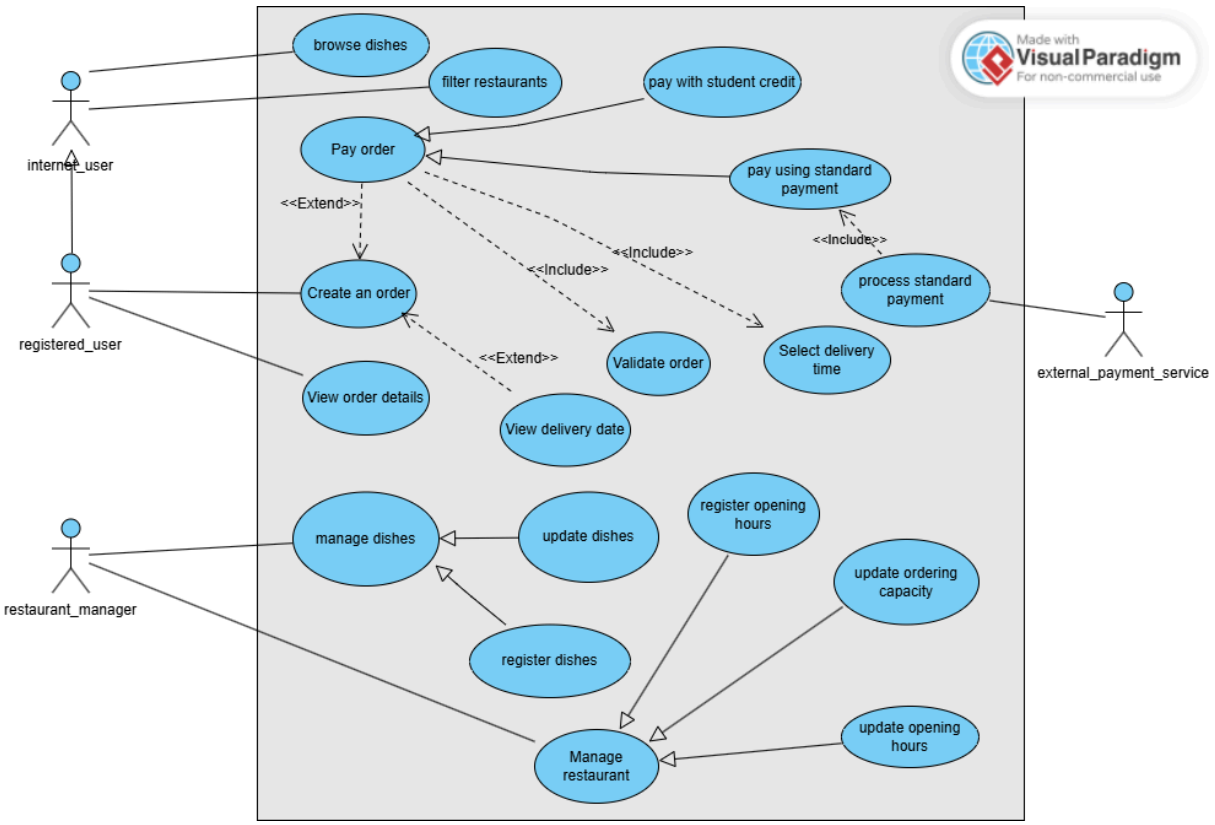


Figure 1. – Diagramme de cas d'utilisation

IV - Diagramme de classes

IV.1- Vue du diagramme de classes détaillé

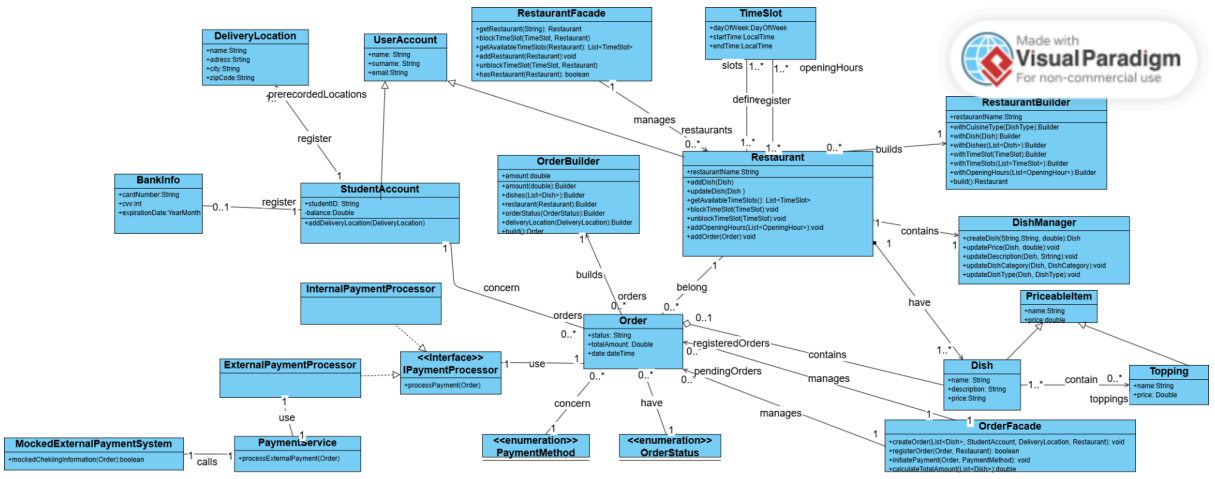


Figure 2. Diagramme de classe

IV.2- Vue d’ensemble du diagramme de classes sous la forme d'un diagramme de packages

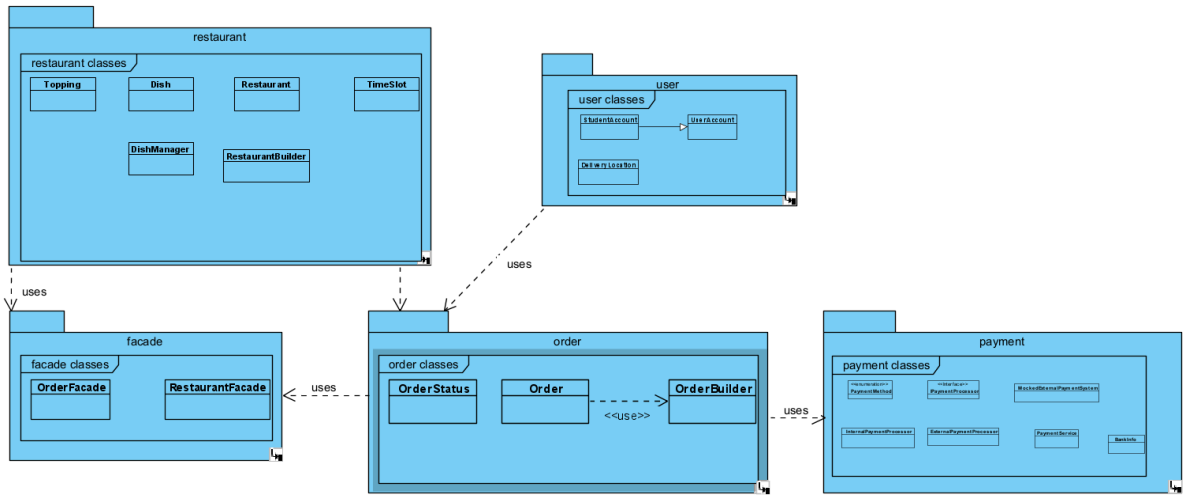


Figure 3. :Vue du diagramme de classe vue en packages

V - Design patterns

V.1- Design pattern

V.1.1 – Builder

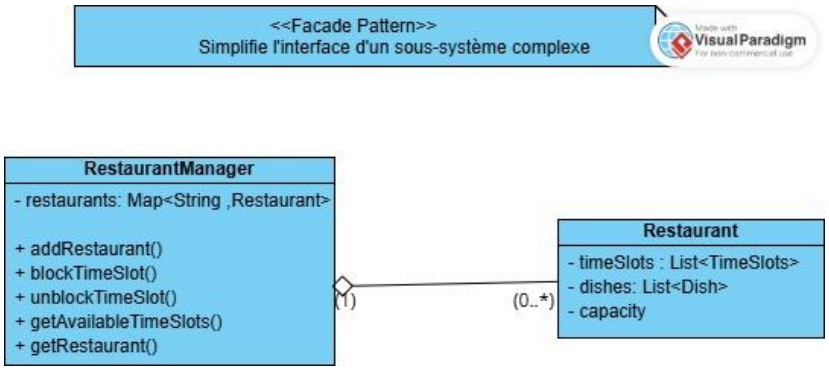
Nous avons choisi d’utiliser le **pattern Builder** pour plusieurs classes comportant un grand nombre d’attributs optionnels, comme **`Restaurant`**, **`Order`**
Ce pattern nous a permis de **rendre la création d’objets plus lisible et flexible**, tout en évitant les constructeurs trop longs et difficiles à maintenir.
Ce choix a également permis de **centraliser la validation** dans la méthode **`build()`** et de garantir une **immuabilité partielle** des objets créés.

V.1.2 – Facade:

Le pattern **Façade** a été utilisé pour **simplifier l’accès** à des sous-systèmes complexes.
Deux façades principales ont été créées :

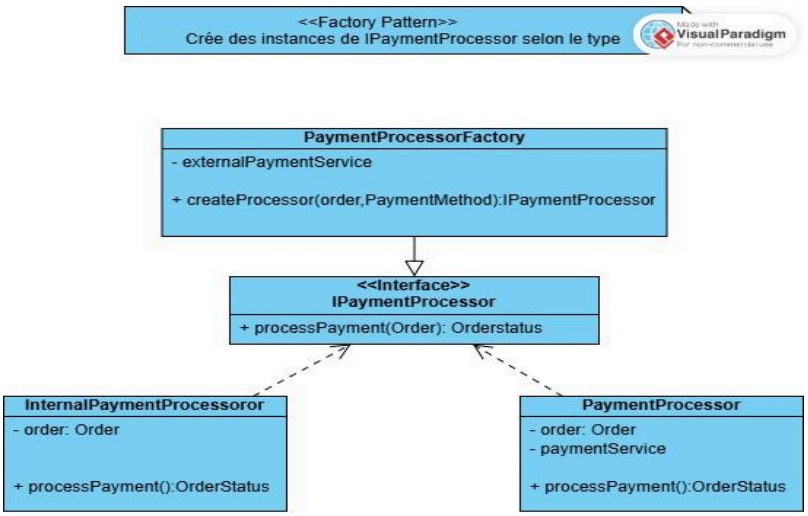
- **`RestaurantManager`**, pour la gestion des restaurants et des créneaux horaires.
- **`OrderManager`**, pour la création et le paiement des commandes.

Elles offrent une **interface unique** aux couches supérieures et **masquent la complexité interne**, ce qui rend le code **plus clair et plus facile à maintenir**.



V.1.3 – Factory:

Le **pattern Factory** a été mis en place pour la création des objets implémentant l'interface `IPaymentProcessor`.
La classe **`PaymentProcessorFactory`** décide quelle classe de paiement instancier (**`InternalPaymentProcessor`** ou **`PaymentProcessor`**) selon le **`PaymentMethod`**.
Ce pattern permet à **`OrderManager`** de ne pas connaître les détails de création des processeurs de paiement, facilitant ainsi l'extension du système



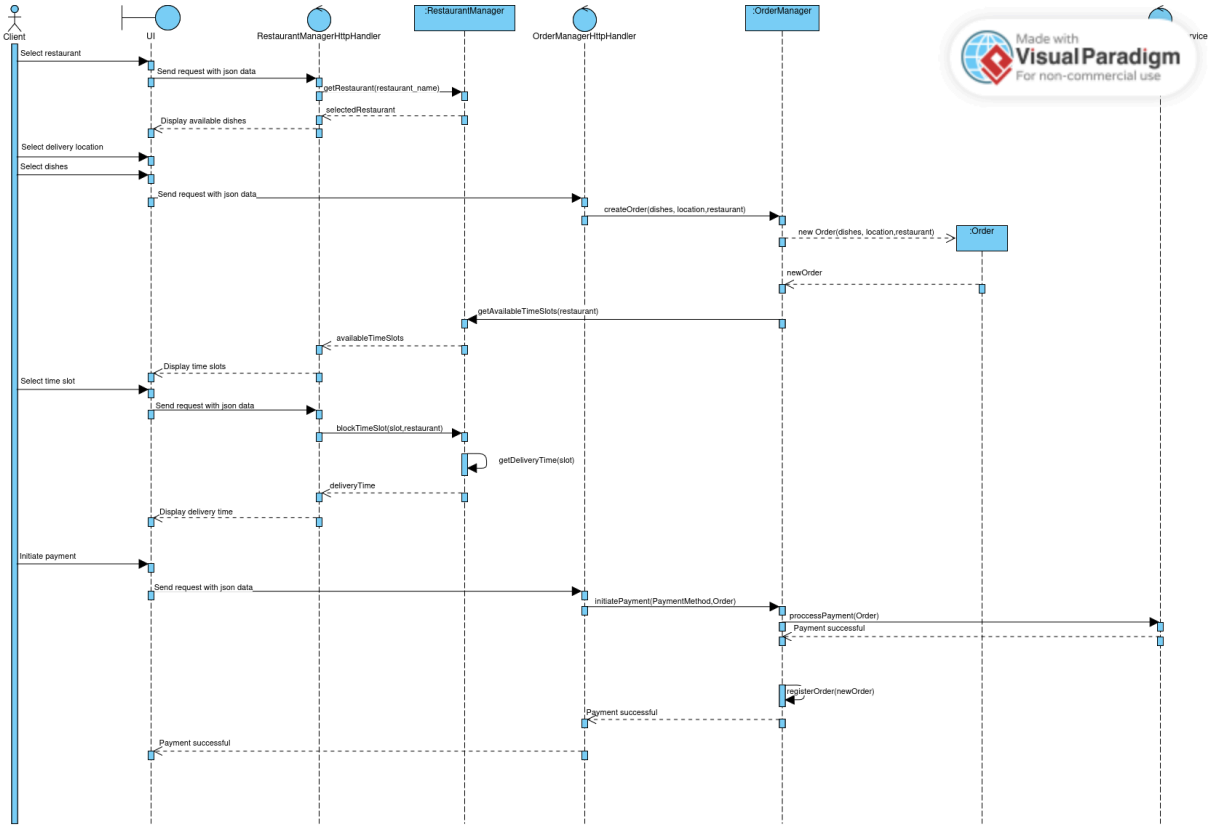
V.2- Patterns envisagés mais non retenus

Au cours du développement, plusieurs design patterns ont été envisagés mais finalement non implémentés.

- **Observer Pattern** : Nous avions envisagé de l'utiliser pour notifier les clients lorsque leur commande passait à l'état ***VALIDATED***.

- **Decorator Pattern** : Ce pattern a été considéré pour permettre l'**ajout dynamique** d'options sur les plats (par exemple sauces ou suppléments). Finalement, nous avons jugé que notre système de ***Toppings*** répondait déjà à la majorité des besoins.

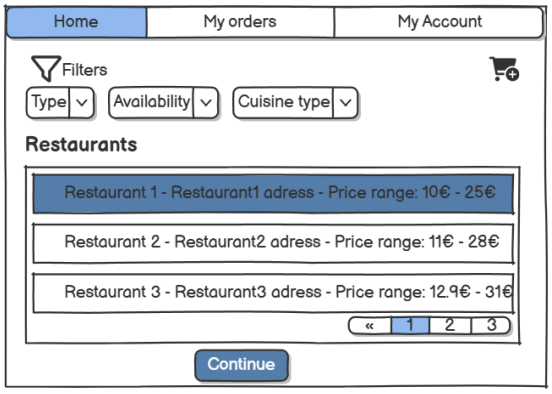
2.5 Diagramme de séquence (*Passer une commande avec paiement external*)

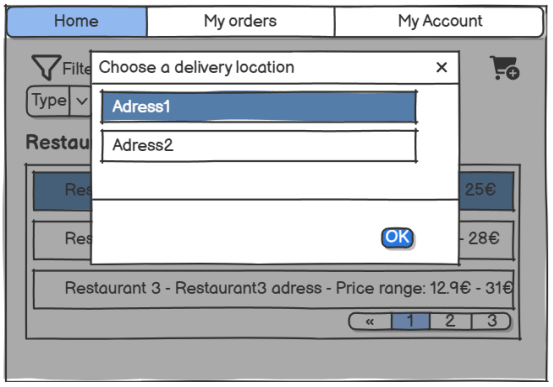


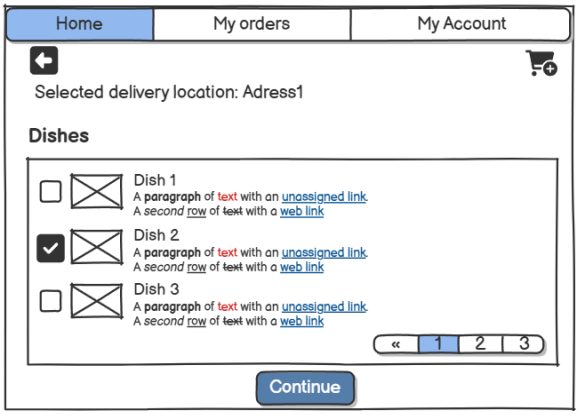
if the client takes too much time to pay or if the payment fails, then the time slot would be freed and the order would'nt be registered

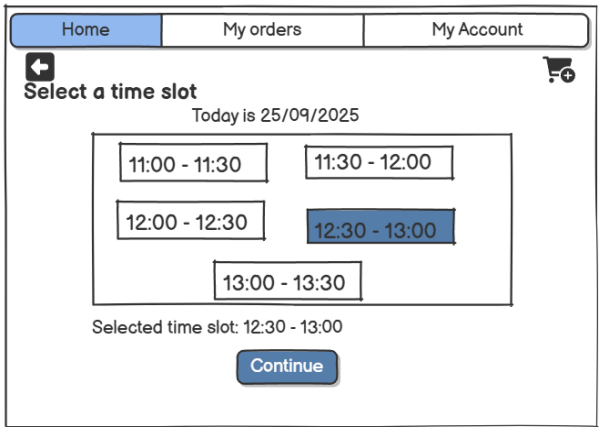
VII - Maquette

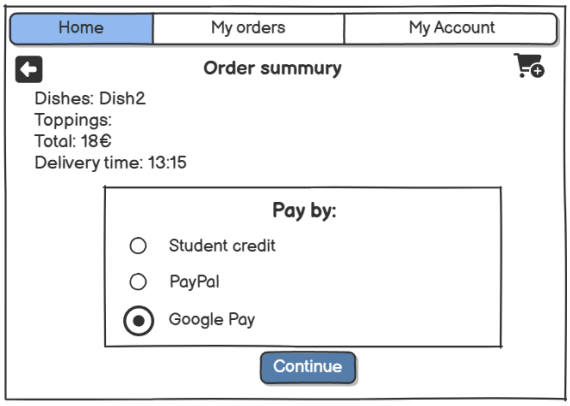
2.3.6 La maquette

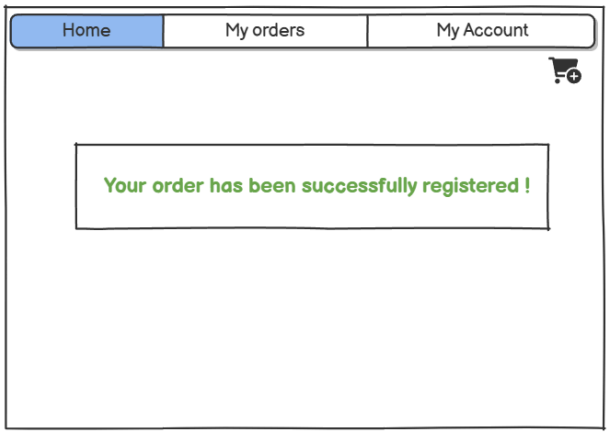












VIII - Qualité de code et gestion du projet

Tests et Qualité

Types de Tests Supportés

Le projet utilise principalement deux types de tests :

1. **Tests Unitaires** : Mis en œuvre avec **JUnit 5**, ils permettent de vérifier le bon fonctionnement isolé de composants spécifiques (classes, méthodes). L'utilisation de **Mockito** est également présente pour créer des objets "mock" (simulacres), facilitant ainsi l'isolation des unités testées de leurs dépendances externes (`OrderManagerTest.java`, `PaymentProcessorTest.java`).
2. **Tests d'Acceptation (BDD)** : Réalisés avec **Cucumber 7**, ces tests valident le comportement du système du point de vue de l'utilisateur. Ils sont écrits en langage Gherkin (fichiers `.feature`) et exécutés via Cucumber, assurant que le logiciel répond aux exigences fonctionnelles définies dans les scénarios.

Couverture des tests:

Element	Class, %	Method, %	Line, %	Branch, %
fr.unice.polytech	96% (31/32)	77% (159/204)	79% (393/495)	58% (136/232)
aigenerator	100% (4/4)	100% (11/11)	100% (29/29)	100% (2/2)
dishes	100% (5/5)	72% (21/29)	77% (41/53)	66% (8/12)
orderManagement	100% (4/4)	84% (27/32)	87% (68/78)	38% (14/36)
paymentProcessing	100% (8/8)	67% (19/28)	68% (48/70)	53% (16/30)
restaurants	85% (6/7)	74% (49/66)	74% (141/190)	66% (80/120)
users	100% (4/4)	84% (32/38)	88% (66/75)	50% (16/32)

Figure 4. Couverture des test

Pour la couverture presque l'ensemble des classes est testé, cependant il nous manque tout de même à peu près 20% des méthodes et des lignes de codes

Vision de la Qualité du Code

Notre approche pour la qualité du code du projet s'appuie sur :

- **Bonnes pratiques de conception** : L'utilisation de Design Patterns (comme le Builder Pattern dans `StudentAccount.java`, `Restaurant.java`, `Order.java`), et la séparation des responsabilités, `Restaurant.java` délègue la résponsabilité de la création des plats à `DishManager.java`.

Gestion du Projet

Notre gestion du projet intègre les éléments suivants :

- **Automatisation (CI)** : Le projet utilise **GitHub Actions** pour l'intégration continue. Un workflow est configuré pour déclencher automatiquement la compilation Maven et l'exécution des tests à chaque push de code, assurant ainsi une vérification constante de l'intégrité du projet.
- **Gestion des branches** : Pour garder une branche main stable, nous avons créé la branche develop possiblement instable. De cette branche develop sort des branches task/NomDeFonctionnalité et tests/FonctionnalitéATester, qui permettent un meilleur suivi de l'avancée du projet.
- **Suivi des tâches** : L'utilisation de **GitHub Issues** avec des modèles spécifiques pour les `user_story` et les `bug` (`README.md`) indique un processus structuré pour le suivi des exigences et des anomalies.

IX - Auto-évaluation

Nous avons tiré plusieurs enseignements de ce projet.
Une rédaction claire des *issues* nous aurait permis de mieux organiser le travail et de suivre efficacement l'avancement. Nous avons constaté qu'il est préférable de créer des *pull-requests* cohérentes et courtes : des PR trop longues ralentissent les revues et freinent la progression.

L’utilisation de la méthodologie **BDD** nous aurait aidés à relier les tests aux besoins fonctionnels et à maintenir une meilleure cohérence entre les exigences et le code.

Membre	AMRAOUI Imane	BEN MZOUGHIA Youssef	KOBBI Fares	TCHANI Rajaa	YOUNES Hammoud
Répartition	100	100	100	100	100

Table 3. : Auto-évaluation