Rapport BACK-END

SophiaTech Eats

ÉQUIPE Q

Membres de l'équipe :
- Tom BUILLOT– OPS
- Gabriel LOIRAT – SA
- Yannis TAIEB – QA
- Alexandre LARGUECH – PO
- Boubaker AMDYOUN – SA

Polytech Nice Sophia
Année universitaire 2025–2026
Date : 25 octobre 2025

Table des matières

Résumé

Ce projet consiste à concevoir et modéliser l'application **SophiaTech Eats**, un système de commande de repas destiné aux étudiants du campus. Le rapport présente les besoins identifiés, les diagrammes UML (cas d'utilisation, classes, séquence) ainsi que les premières maquettes de l'interface utilisateur.

1 Suivi des exigences du projet

Côté	Exigence	État	Remarques	Points forts / faibles
Client	C1	**Done**		
Client	C2	**Done**		
Client	C5	**Done**		
Client	C6	**Done**		
Client	C7	**Done**		
Restaurant	R2	**Done**		
Restaurant	R4	**Partiel**	Une variable enregistre l'historique mais il manque l'implémentation de l'auto-complétion	
Restaurant	R5	**Partiel**	On peut ajouter un créneau horaire mais pas en supprimer un depuis un compte restaurant	
Paiement	P1	**Done**		
Paiement	P2	**Done**		
Paiement	P3	**Done**		
Paiement	P6	**Done**		
IA/Catalogue	AI2	**Done**	L'IA génère une description de plat qui peut être acceptée ou modifiée par le restaurateur avant mise en ligne.	

2 Glossaire complémentaire

- **MockAI** : Système dont le but est de copier le fonctionnement d'une intelligence artificielle.
- **Repository** : Lieu de stockage et d'accès centralisé aux données de l'application (ex. commandes, restaurants).
- **Adresse** : Localisation de livraison utilisée lors de la validation d'une commande.
- **Menu** : Ensemble des plats disponibles dans l'offre d'un établissement.
- **Plat (Meal)** : Élément du menu avec nom, prix, catégorie, type et préférences diététiques ; construit via un *Builder*.
- **Planning** : Agenda opérationnel décrivant les horaires d'ouverture et les périodes de service.
- **Façade applicative (ApplicationService)** : Point d'entrée simplifié coordonnant les services de bout en bout.

3 Diagramme de cas d'utilisation

3.1 Description

Le système **SophiaTech Eats** met en relation plusieurs acteurs aux rôles complémentaires :

- **Visiteur** : parcourt l'offre de plats et explore les restaurants sans obligation de créer une commande.
- **Client** : sélectionne un restaurant, choisit un lieu de livraison, ajoute des plats à une commande, confirme cette dernière et procède au paiement.
- **Restaurant** : gère la carte (plats disponibles), configure ses créneaux horaires et définit la capacité de commandes acceptées.
- **Service de paiement externe** : valide les transactions financières lorsque le solde de crédit étudiant est insuffisant ou pour tout paiement en ligne.

Cas d'utilisation principaux :

- Parcourir l'offre : consulter les plats disponibles et filtrer les restaurants (Visiteur/Client).
- Créer une commande : choisir un restaurant, un lieu de livraison et ajouter un ou plusieurs plats (Client).
- Confirmer et payer une commande : valider la commande puis procéder au paiement via crédit étudiant ou service externe (Client + Service de paiement).
- Utiliser le crédit étudiant (optionnel) : permettre le règlement partiel ou total de la commande via l'allocation offerte (Client).
- Enregistrer la commande : archiver la commande après paiement et la rendre visible pour le restaurant (Système).
- Gérer l'offre et les créneaux : ajouter, modifier ou supprimer des plats, définir les créneaux horaires et les capacités de service (Restaurant).

3.2 Diagramme

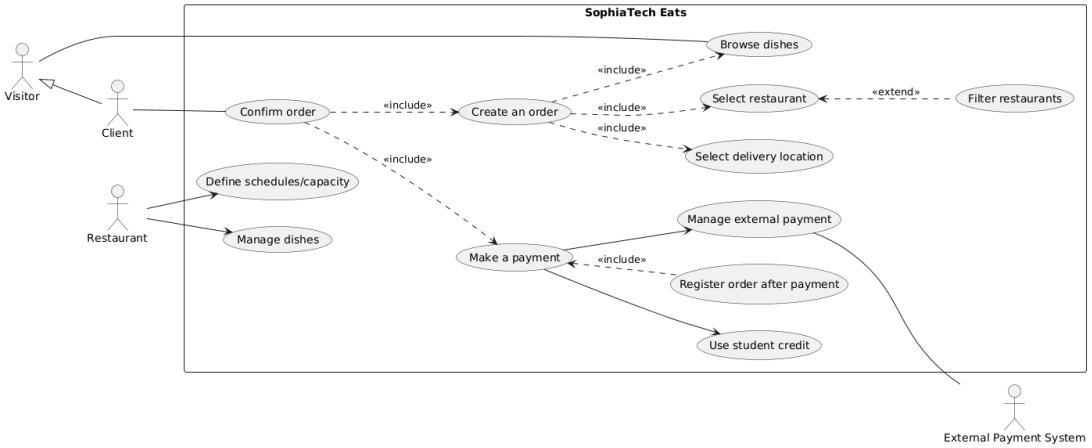


FIGURE 1 – Diagramme de cas d'utilisations

4 Diagramme de classes

4.1 Description

Le diagramme de classes modélise les entités : Client, Restaurant, Plat, Commande, Paiement, CréneauHoraire.

Note : pour que le restaurant ne puisse pas savoir si la commande a été payée avec du crédit, un getter opèrera une vérification pour savoir si la personne qui demande le moyen de paiement utilisé est légitime à le demander. D'autre part, le code postal est stocké dans un String pour conserver les zéros en début de code.

4.2 Diagrammes

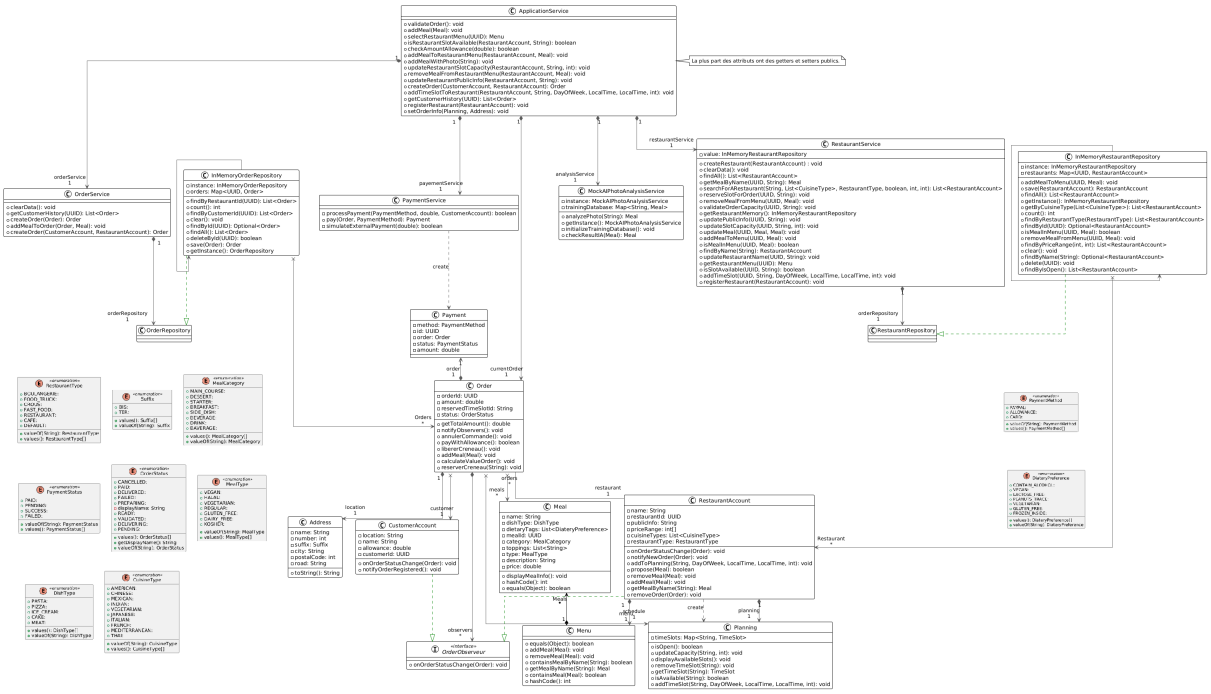


FIGURE 2 – Diagramme de classes

4.3 Architecture modulaire et découpage en packages

Le projet adopte une **architecture en couches** clairement séparées, respectant les principes de conception logicielle et facilitant la **maintenabilité** ainsi que l'**évolutivité** du système de livraison de repas universitaires.

- **Package `domain`** : constitue le cœur métier indépendant et contient trois sous-packages :
  - **enum** : centralise tous les *vocabulaires contrôlés* (types de cuisine, établissements, préférences diététiques, statuts de commande et de paiement, méthodes de paiement).
  - **model** : regroupe les *entités riches* du domaine (`RestaurantAccount`, `Menu`, `Meal`, `Order`, `Payment`, `Planning`, `CustomerAccount`, `Address`) avec leur logique métier encapsulée, ainsi que le **pattern Observer** pour notifier les changements d'état.
  - **exception** : isole les *erreurs métier spécifiques* (capacité dépassée, allocation insuffisante, erreurs de validation, conflits de planning), permettant une gestion fine des cas d'exception.
- **Package `repository`** : implémente le **pattern Repository** pour abstraire la persistance.
  - Les **interfaces** (`RestaurantRepository`, `OrderRepository`) définissent les contrats d'accès aux données (recherche par type de cuisine, prix, disponibilité).
  - Le sous-package `impl` fournit des *implémentations en mémoire* pour le développement et les tests, facilitant l'injection de dépendances et un futur remplacement par une base de données réelle.

— **Package `service`** : expose la **logique applicative** et orchestre les cas d'usage.
  — `RestaurantService`, `OrderService` et `PaymentService` coordonnent les opérations métier complexes.
  — `ApplicationService` joue le rôle de **façade unifiée**, simplifiant l'accès aux services depuis l'UI ou l'API.
  — `MockAIPhotoAnalysisService` simule l'intégration avec un service externe d'analyse d'images par IA.

Cette architecture garantit :
— une **séparation stricte des responsabilités**,
— une **testabilité élevée** via l'injection de dépendances,
— une **évolutivité facilitée**, en isolant les changements technologiques (persistance, services externes) du cœur métier stable.

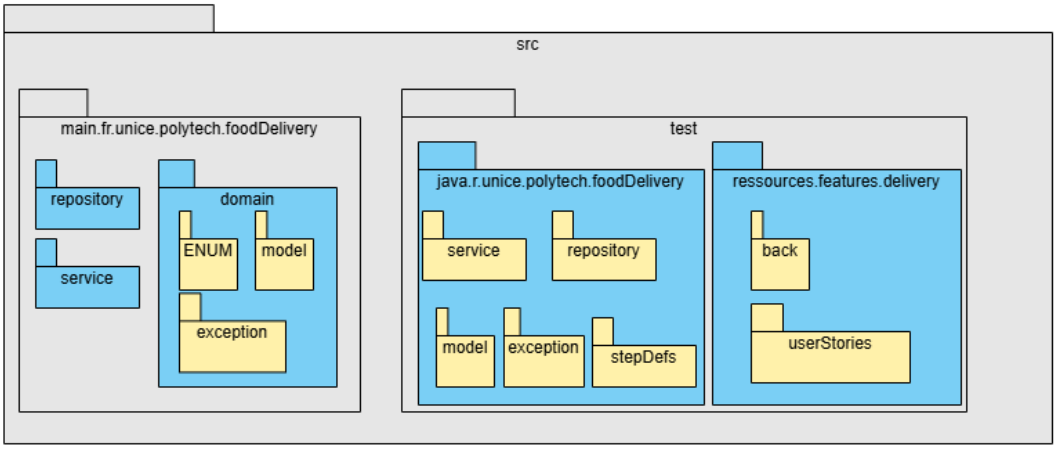


FIGURE 3 – Diagramme de packages

5 Design Patterns

Afin d'améliorer la lisibilité, la maintenabilité et l'extensibilité de notre solution, nous avons implémenté plusieurs *design patterns* classiques (GoF). Chaque pattern est présenté ci-dessous avec ses classes associées, son rôle logique dans notre architecture, la justification de son usage et ses bénéfices.

Builder

Classes : `Meal`, `Meal.Builder`

Rôle logique : Permettre de construire un repas (`Meal`) avec plusieurs attributs optionnels (toppings, description, tags) sans avoir de constructeurs à rallonge.

Pourquoi : Rendre la création d'objets plus lisible, réduire les erreurs, et garantir la cohérence des instances.

Bénéfices : Lisibilité, extensibilité, sécurité des instanciations.

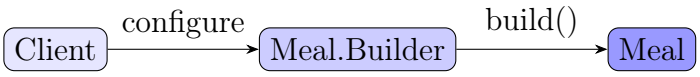


FIGURE 4 – Pattern Builder appliqué à `Meal`

Observer

Classes : `Order`, `OrderObserver`, `CustomerAccount`, `RestaurantAccount`

Rôle logique : Lier automatiquement une commande (`Order`) aux acteurs concernés (client et restaurant) lorsqu'elle change d'état.

Pourquoi : Découpler la logique métier des réactions liées (notifications, mise à jour d'états).

Bénéfices : Extensibilité (ajouter de nouveaux observateurs facilement), meilleure séparation des responsabilités.

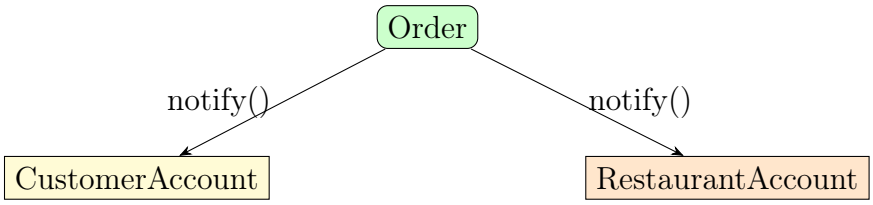


FIGURE 5 – Observer appliqué à `Order`

Repository (+ Singleton)

Classes : `OrderRepository`, `RestaurantRepository`, `InMemoryOrderRepository`, `InMemoryRestaurantRepository`

Rôle logique : Fournir une interface claire pour accéder et stocker les données de commandes et de restaurants. Chaque repository est un **Singleton** afin de centraliser les données en mémoire et éviter la duplication.

Pourquoi : Séparer la logique métier de la persistance, simplifier les tests grâce à une implémentation en mémoire, garantir une instance unique partagée.

Bénéfices : Testabilité, modularité, extensibilité future (migration vers une base de données).

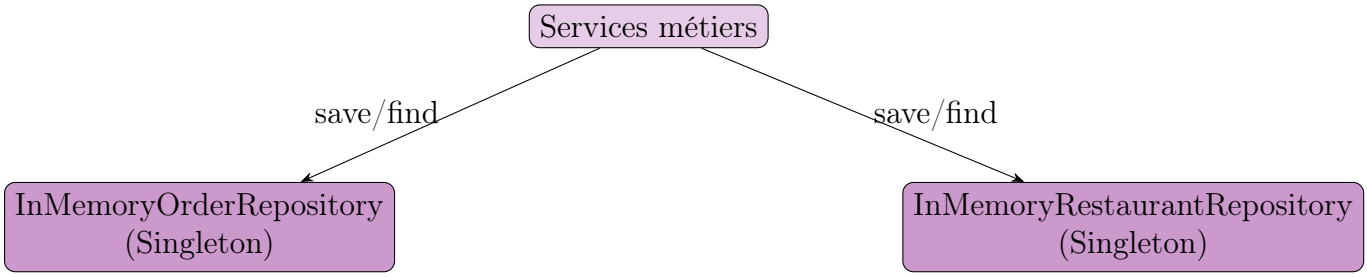


FIGURE 6 – Repositories en mémoire implémentés en Singleton

Façade + Service Layer

Classes : `ApplicationService`, `RestaurantService`, `OrderService`, `PaymentService`

Rôle logique : Offrir un point d'entrée unique (`ApplicationService`) qui orchestre les services métiers (commande, restaurant, paiement).

Pourquoi : Simplifier l'accès aux fonctionnalités, réduire le couplage entre l'interface et la logique métier.

Bénéfices : Lisibilité des cas d'usage, meilleure cohésion de l'API, évolutivité.

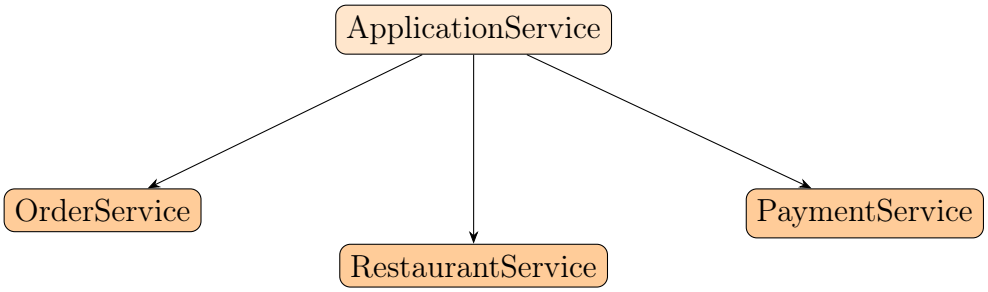


FIGURE 7 – Façade orchestrant les services métiers

Singleton (Service IA)

Classe : `MockAIPhotoAnalysisService`

Rôle logique : Centraliser l'analyse automatique des photos dans un service unique partagé.

Pourquoi : Éviter de multiplier les instances d'un service externe simulé (mock IA) et simplifier les tests.

Bénéfices : Simplicité, centralisation, meilleure isolation pour les scénarios de tests.

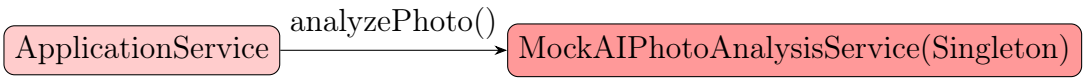


FIGURE 8 – Service IA mock implémenté en Singleton

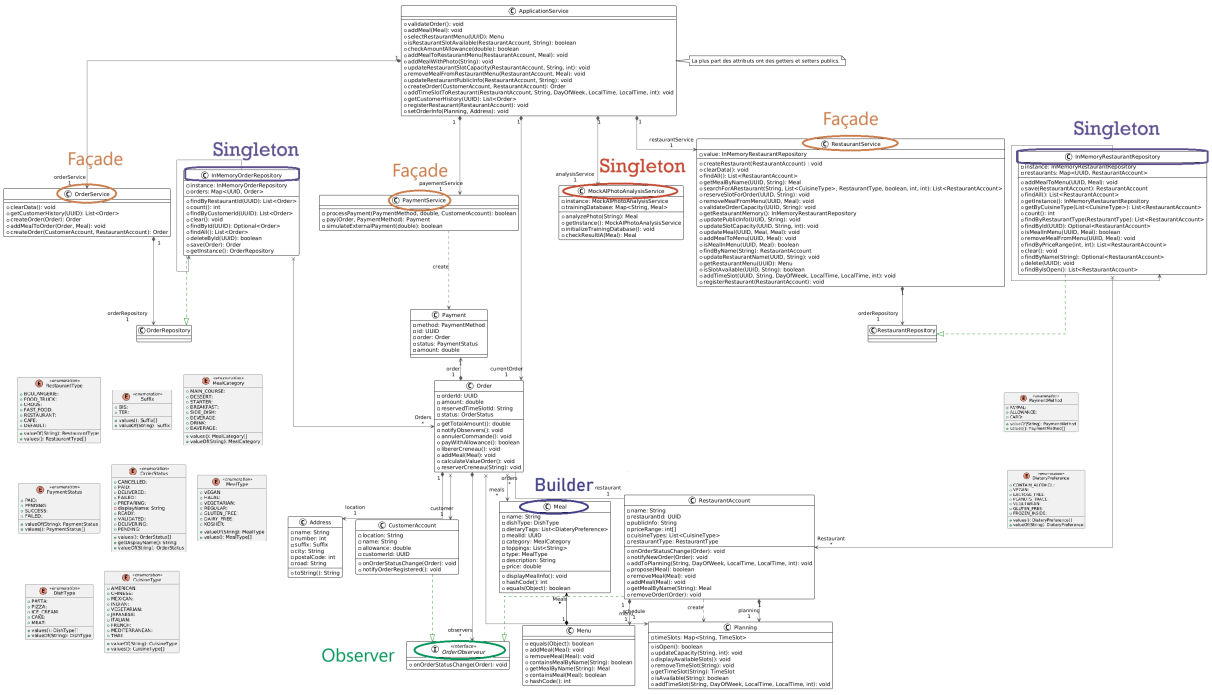


FIGURE 9 – Design patterns dans le diagramme de classes

6 Diagramme de séquence

Entités principales

Le système **SophiaTech Eats** repose sur plusieurs composantes clés :

- **Customer (acteur)** : initie la commande, choisit les plats, définit la livraison et procède au paiement.
- **Application Service (cœur de l'application)** : coordonne les interactions entre comptes, commandes et paiement.
- **Customer Account** : gère les informations personnelles, l'adresse de livraison et le solde de l'allocation mensuelle.
- **Restaurant Account** : gère les plats proposés et reçoit les commandes confirmées.
- **Order** : représente la commande en cours (plats, horaire, lieu de livraison).
- **Payment Service (externe)** : valide les paiements lorsque l'*allowance* est insuffisante.

Scénario principal : création, planification et paiement d'une commande

1. Le **Customer** choisit un premier plat et initie la création d'un **Order** via l'**Application Service**.
2. Le **Customer** ajoute un ou plusieurs **Meals** supplémentaires à cet **Order**.
3. Le **Customer** valide la commande et sélectionne un **créneau horaire** ainsi qu'un **lieu de livraison**.
4. L'**Application Service** vérifie l'**allowance** dans le **Customer Account** :

— Cas 1 – Solde suffisant : le montant est directement déduit.
— Cas 2 – Solde insuffisant : l'**Application Service** sollicite le **Payment Service** externe pour compléter le paiement.

5. Une fois le paiement confirmé, l'**Order** est enregistré dans le système.
6. L'**Application Service** notifie le **Customer** de la confirmation et informe le **Restaurant Account** afin de préparer la commande.

6.1 Diagramme

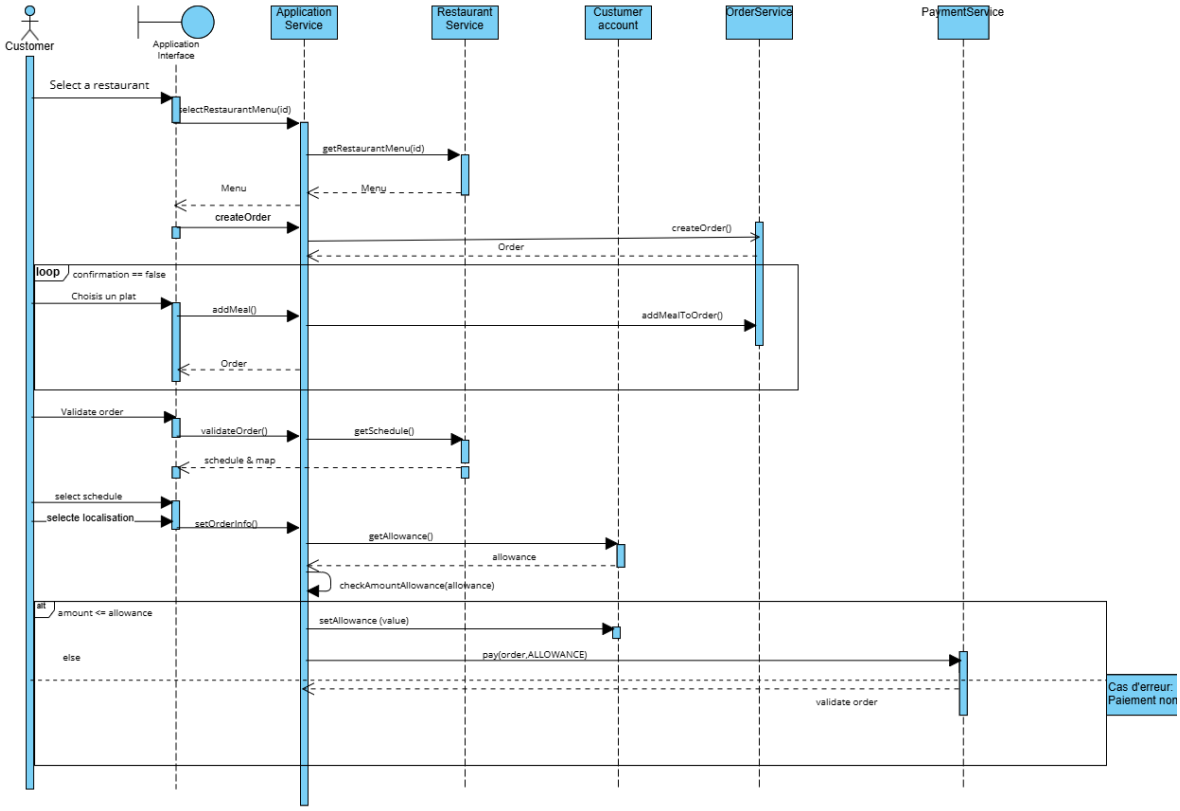
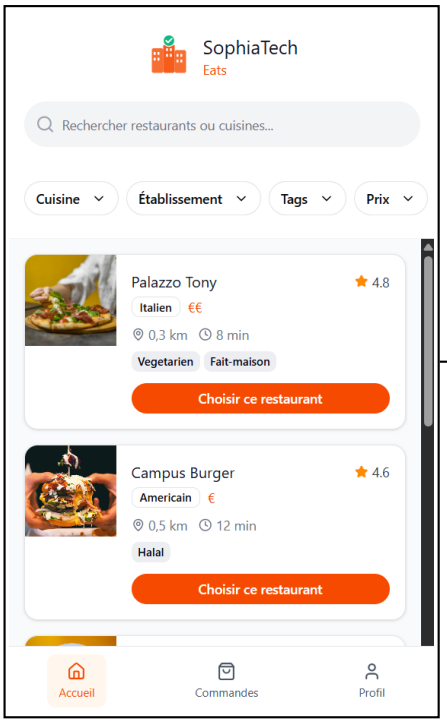


FIGURE 10 – Diagramme de séquence

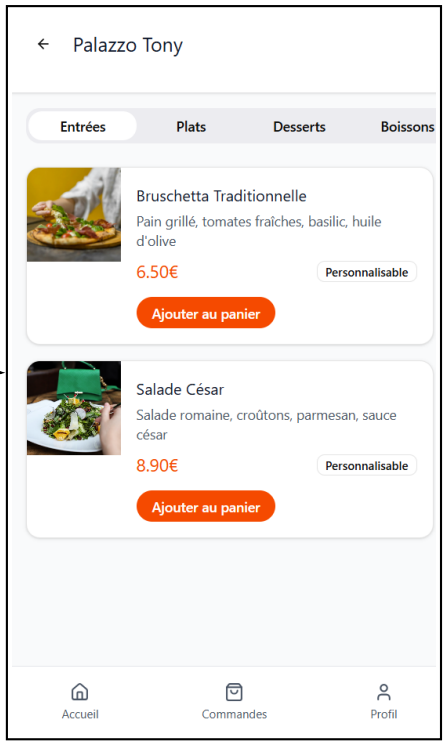
7 Maquettes

7.1 Description

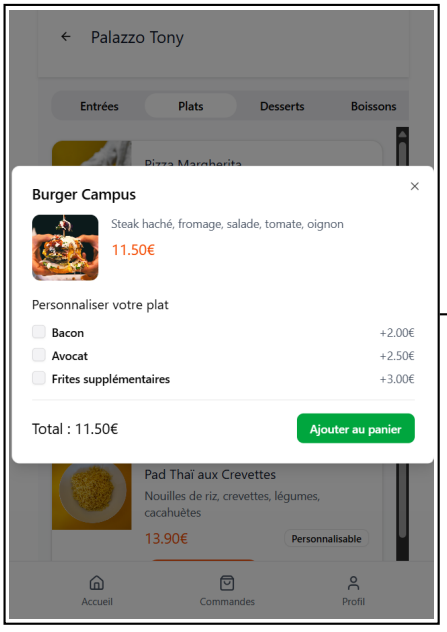
Les maquettes illustrent le parcours utilisateur : accueil, sélection des plats, panier, paiement, choix du lieu/créneau, confirmation.



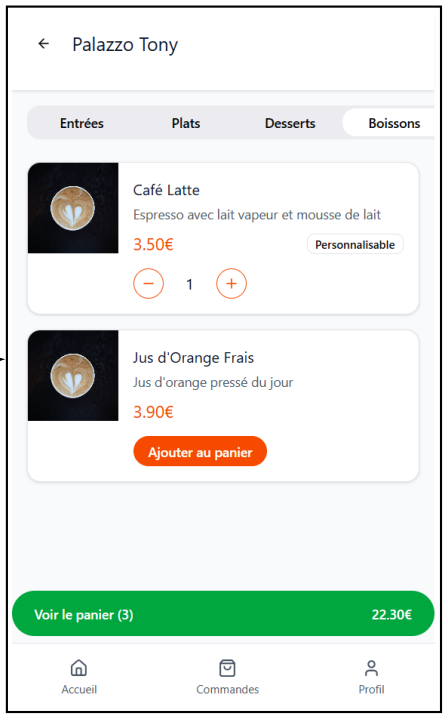
Chosir un restaurant et recherche



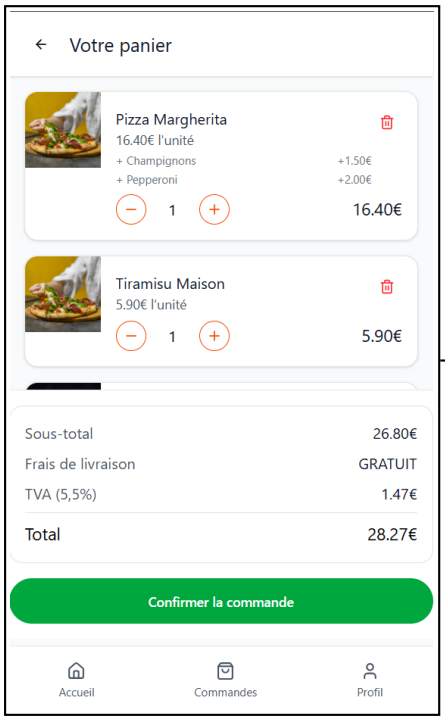
Parcourir l'offre du restaurant



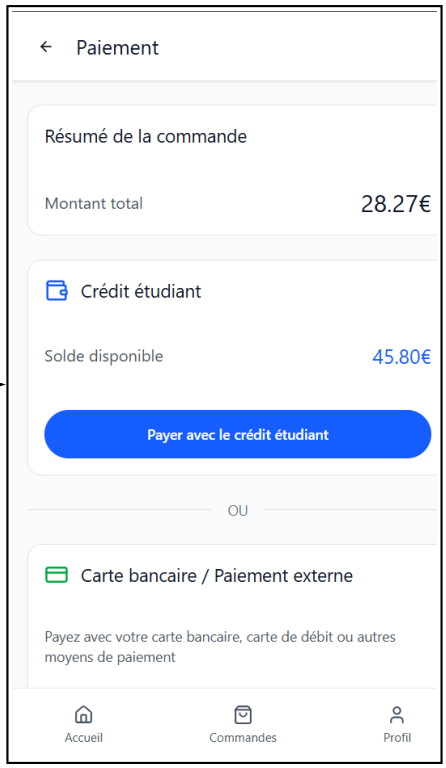
Ajouter un plat et option de supplement



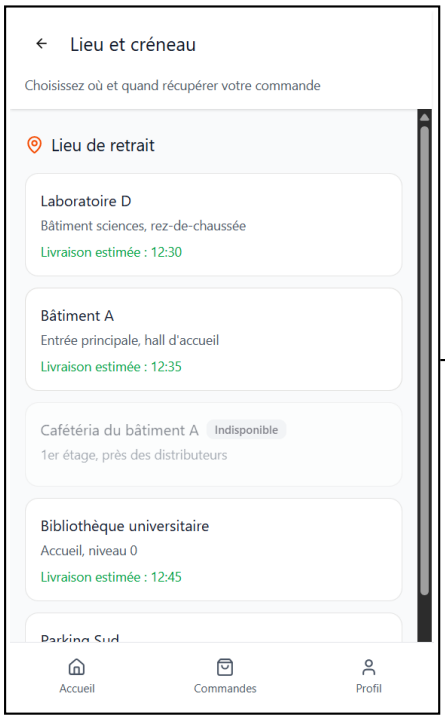
Ajouter un autre article



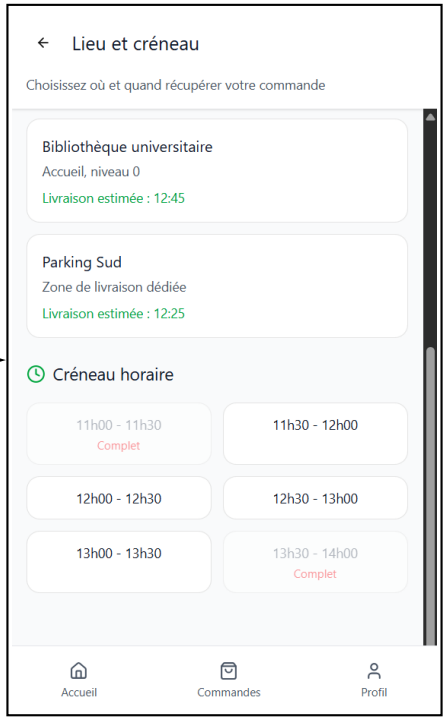
Vérification et confirmation



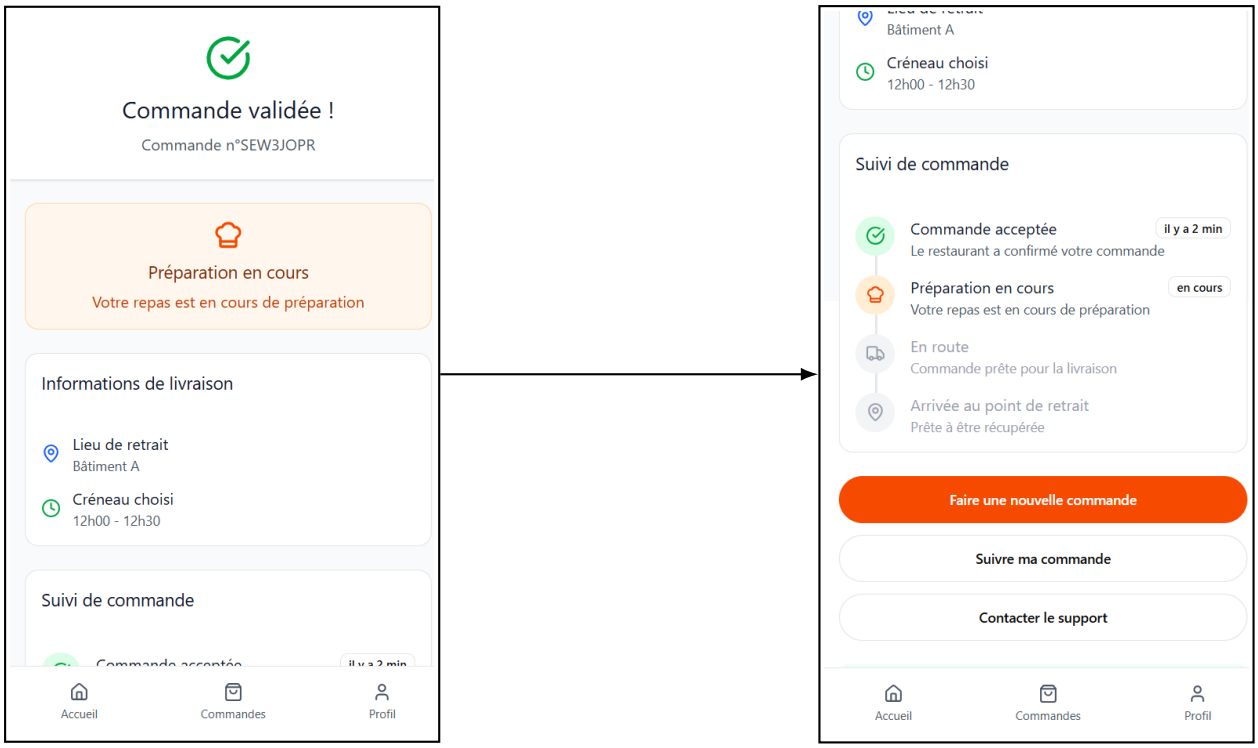
Paiement par crédit étudiant ou autre



Choir le site de livraison



Choisir le créneau



Commande validée

Suivi de commande

8 Qualité des codes et gestion de projets

Tests, qualité et couverture

Nous avons mis en place des tests Cucumber qui suivent les scénarios que nous avons préalablement définis. Les tests unitaires ont été réalisés avec JUnit.

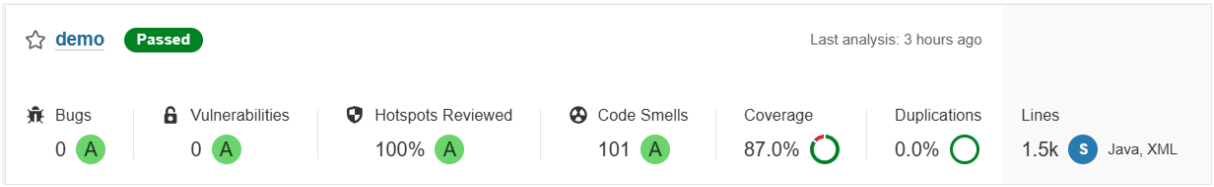


FIGURE 11 – Rapport SonarQube

Vision de la qualité du code

La qualité du code repose sur plusieurs principes mis en œuvre tout au long du développement. La séparation des responsabilités a été assurée à travers une architecture modulaire, distinguant clairement les couches de données, de logique métier et de présentation.

Des patrons de conception tels que *Observer*, *Singleton* et *Builder* ont été employés pour renforcer la maintenabilité et réduire le couplage entre les classes. Le code présente

une structure globalement stable et évolutive et dans l'ensemble, la qualité du code est jugée satisfaisante, à la fois sur le plan fonctionnel et sur la capacité du projet à être compris et repris par un autre développeur.

Gestion du projet et automatisation

Nous avons utilisé la méthode GitLab Flow pour la créer les branches. Nous avons utilisé le Kanban de GitHub pour travailler et créé des milestones, user-stories et issues associées pour répartir les tâches.

9 Auto-évaluation

Points positifs et réussites

- **Objectifs atteints** dans le périmètre fonctionnel.
- **Dépôt Git structuré** et organisation via *Kanban*.
- Implémentation de patrons de conception : *Observer*, *Singleton*, *Builder*.
- Tests **Cucumber** validant la prise de commande de bout en bout.
- Collaboration renforcée et montée en compétence collective.

Difficultés rencontrées

- Manque de clarification commune au démarrage du projet.
- Répartition du travail inégale, entraînant quelques retards.
- Diagramme de classes sous-estimé, nécessitant plusieurs itérations.
- Ajustements architecturaux tardifs, compliquant l'intégration finale.

Leçons apprises et axes d'amélioration

- Importance d'une **phase de conception approfondie**.
- Nécessité d'une **communication régulière et structurée**.
- Utilité d'*user stories précises* pour équilibrer la charge de travail.
- Les **tests automatisés** sont essentiels pour la stabilité du code.

Répartition des points

Membre	**Points**
Tom BOUILLOT	105
Boubaker AMDYOUN	110
Gabriel LOIRAT	95
Alexandre LARGUECH	95
Yannis TAIEB	95
Total	**500**