

SophiaTech EATS

Rendu-D1

TeamJ

- LACROIX Baptiste
- MAÏSTRE Antoine
- TLILI Abderrahmen
- TOUPENCE Tom

2024-2025

Table des matières

1 . Périmètre fonctionnel.....	2
i . Hypothèses de travail.....	2
ii . Limites identifiées.....	2
iii . Stratégies choisies et éléments spécifiques.....	2
2 . Conception UML.....	3
i . Glossaire.....	3
ii . Diagramme de cas d'utilisation.....	5
iii . Diagramme de classe.....	6
iv . Design Patterns.....	6
v . Diagramme de séquence.....	9
vi . Maquette.....	9
3 . Qualité des codes et gestion de projets.....	10
4 . Rétrospective et Auto-évaluation.....	12

1. Périmètre fonctionnel

i . Hypothèses de travail

Livraison & temps de préparation:

- Livraison, nous n'avons pas pris en compte le temps de livraison entre le moment où la commande est prête et le moment où elle se fait livrer. Donc nous partons du principe qu'il est compris dans le temps de préparation.

Système de paiement:

- Nous avons implémenté une stratégie de paiement qui pour le moment ne fait rien a part faire un affichage. Donc pour le moment tous les paiements sont validés automatiquement.

ii . Limites identifiées

Annulation de commande:

- Le système ne prend pas en compte le cas où un restaurant annule une commande après validation.

iii . Stratégies choisies et éléments spécifiques

Paieement:

Dans ce projet, nous avons adopté le **patron de conception Factory** (Factory Design Pattern) pour gérer les différents modes de paiement. Cette stratégie permet de créer une structure flexible et évolutive où il est possible d'ajouter de nouveaux moyens de paiement sans modifier le code existant, ce qui facilitera grandement l'évolution future du projet.

1. **Stratégie choisie** : En utilisant le Factory Design Pattern, nous avons centralisé la logique de création des instances de chaque méthode de paiement (ex : carte de crédit, PayPal, virement bancaire) dans une seule "Factory" dédiée. Cela permet d'instancier dynamiquement chaque méthode de paiement en fonction des besoins sans changer le code métier principal.
2. **Éléments spécifiques** :

- La Factory, en fonction du type de paiement spécifié, renvoie une instance de la classe correspondante. Par exemple, si le type de paiement est "carte de crédit", la Factory renvoie une instance de la classe **CreditCardPayment**.
- Chaque méthode de paiement implémente une interface commune qui garantit que les méthodes essentielles, comme **pay**, sont présentes dans toutes les classes.

3. Originalité de la proposition :

- Cette approche offre une grande **modularité** et **extensibilité**. Elle permet d'ajouter de nouvelles méthodes de paiement (par exemple, crypto-monnaie ou Apple Pay) sans avoir à modifier le code central du projet.
- Du point de vue de la **scalabilité**, ce choix est judicieux, car il simplifie l'ajout de nouveaux types de paiements et diminue les risques d'erreurs en centralisant la création des instances.

4. Limites courantes de la mise en œuvre :

- La mise en œuvre actuelle nécessite de modifier la Factory elle-même si l'on veut ajouter un nouveau type de paiement, ce qui, bien que limité, pourrait être encore simplifié par une approche plus dynamique (ex : en utilisant un registre d'extensions).
- Dans certaines situations, des dépendances supplémentaires peuvent apparaître si une méthode de paiement particulière exige une bibliothèque externe, augmentant la complexité potentielle de la gestion des dépendances.

2 . Conception UML

i . Glossaire

GitHub & Project Management

- **Issue Template** : Un modèle prédéfini pour créer des issues (problèmes ou tâches) sur GitHub, facilitant la communication des détails importants pour la gestion des problèmes et des demandes de fonctionnalités.
- **Continuous Integration (Intégration Continue)** : Pratique de développement qui consiste à automatiser les tests et la construction du code à chaque modification (comme un push), assurant la qualité et la stabilité du code.
- **Workflow** : Série d'actions automatisées sur GitHub, comme les tests ou les déploiements, définie dans un fichier YAML (`maven.yml` dans ce projet) pour automatiser les processus de développement.

Java & Application Structure

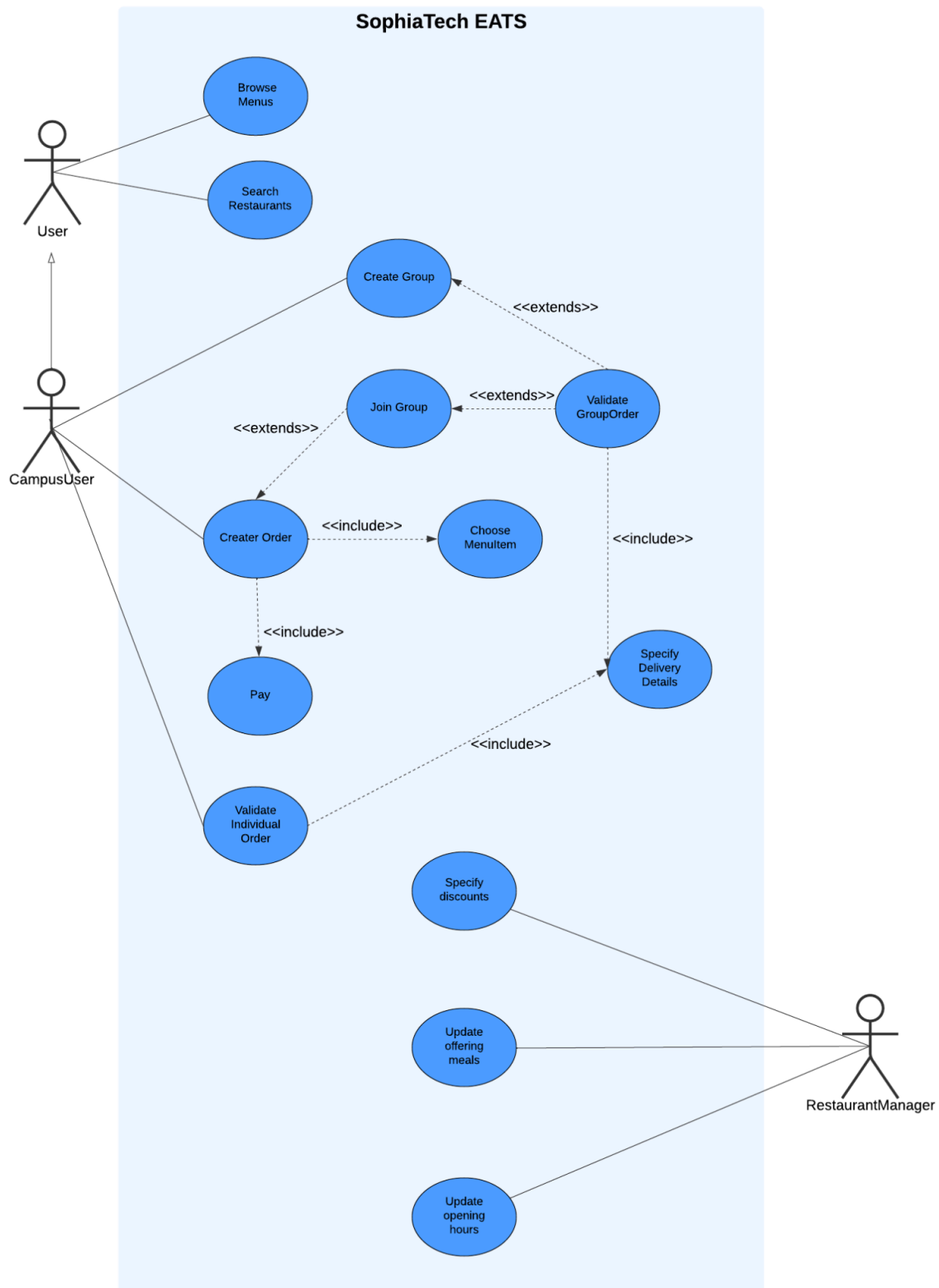
- **Dependency (Dépendance)** : Paquet logiciel ou bibliothèque que le projet utilise et qui est géré dans `pom.xml` avec Maven. Les dépendances permettent de réutiliser du code tiers sans le réécrire.
- **Unit Test (Test Unitaire)** : Test automatisé vérifiant le bon fonctionnement d'une unité de code (comme une méthode ou une classe individuelle) de manière isolée.
- **Integration Test (Test d'Intégration)** : Test qui vérifie l'interaction entre différentes parties de l'application, par exemple la gestion des paiements ou des commandes dans ce projet.
- **Cucumber** : Outil de test basé sur le BDD (Behavior-Driven Development) qui utilise des scénarios en langage naturel (Gherkin) pour écrire des tests. Les fichiers `.feature` définissent ces scénarios de test.

Restaurant & Order Management Domain

- **Order Management (Gestion des Commandes)** : Processus de traitement, de suivi et de validation des commandes. Dans votre projet, il comprend la gestion des `IndividualOrder`, `GroupOrder`, et leur `OrderStatus`.

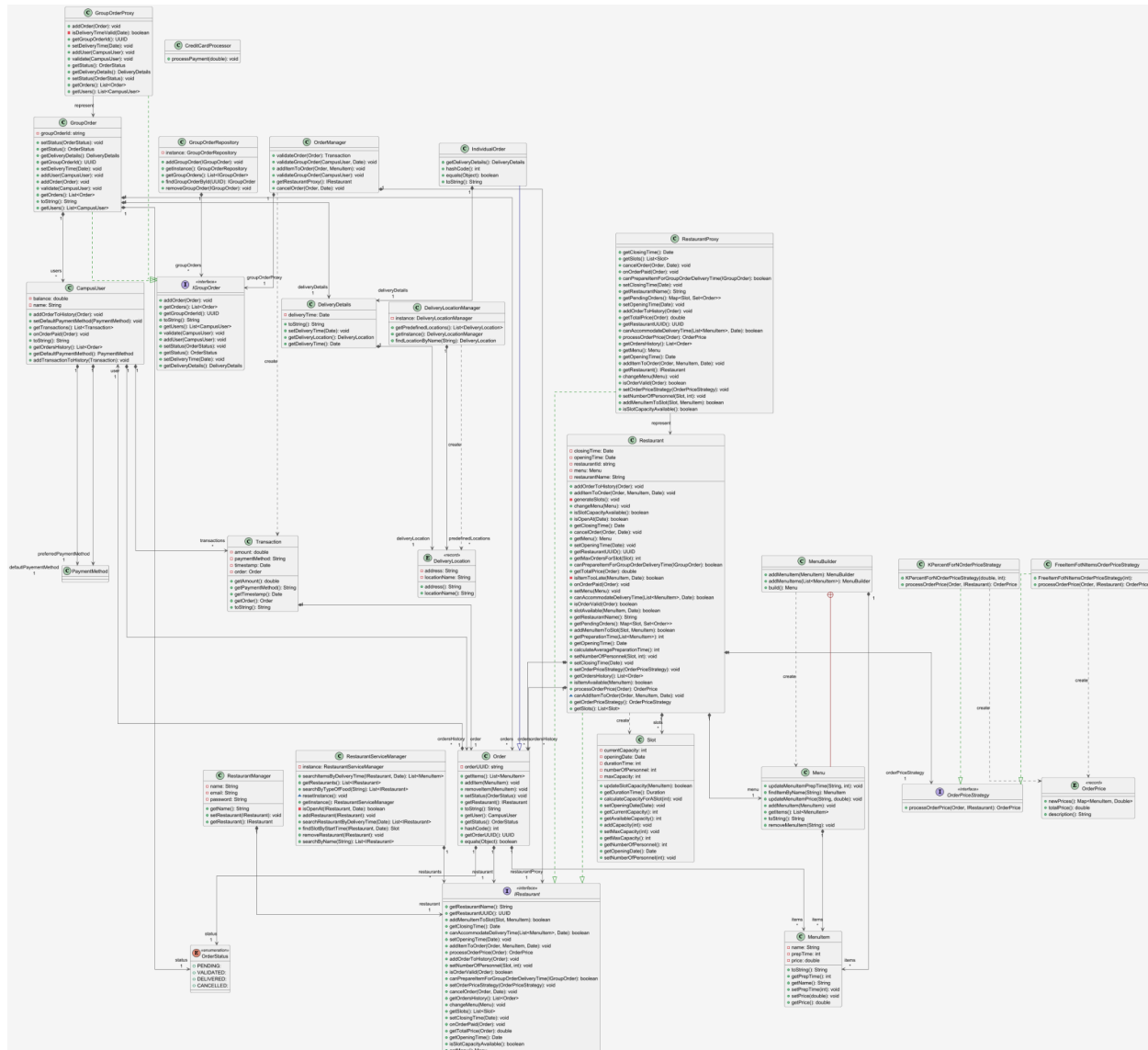
- **Payment Processor (Processeur de Paiement)** : Composant permettant de traiter les paiements de manière sécurisée. Dans ce projet, cela inclut plusieurs types de paiements (*CreditCardProcessor*, *PayPalProcessor*, etc.).
- **Delivery Details (Détails de Livraison)** : Informations concernant l'adresse et les conditions de livraison, gérées dans *DeliveryDetails* et *DeliveryLocation*.
- **Strategy Pattern (Patron de Stratégie)** : Patron de conception utilisé dans *OrderPriceStrategy* et *OrderPriceStrategyFactory* pour permettre de modifier dynamiquement la manière dont le prix d'une commande est calculé en fonction des besoins.

ii . Diagramme de cas d'utilisation



iii . Diagramme de classe

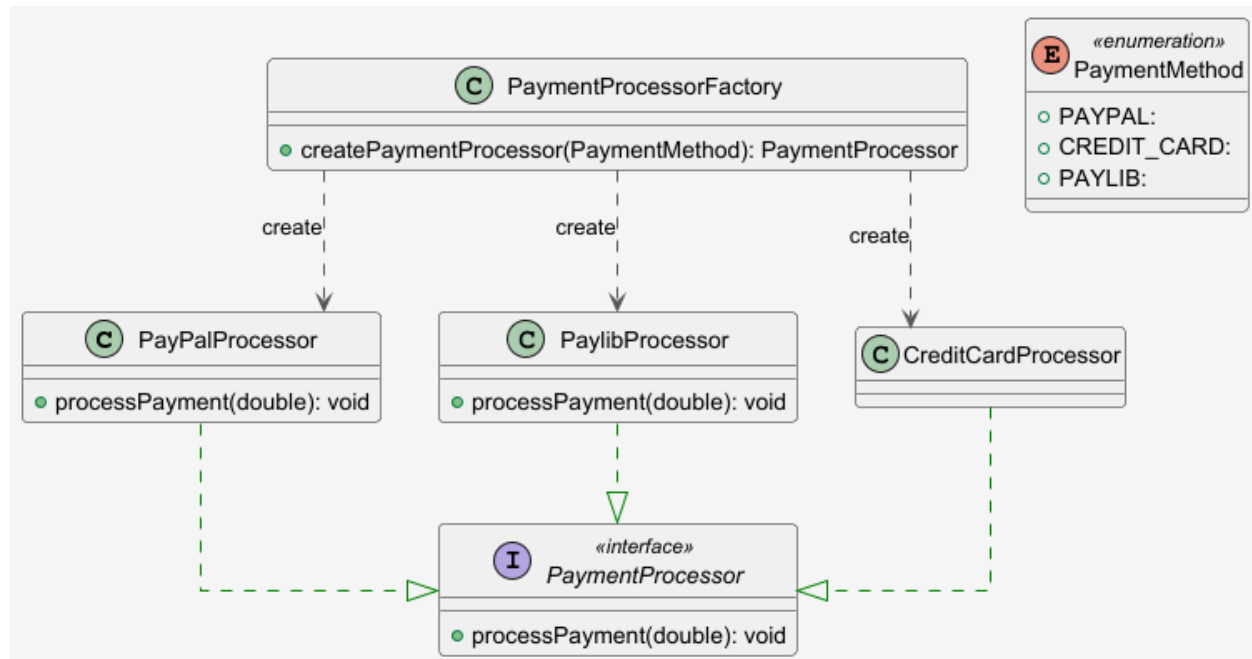
Pour mieux visualiser le diagramme de classe, il sera disponible sur notre repository github dans le répertoire doc/class_diagram/



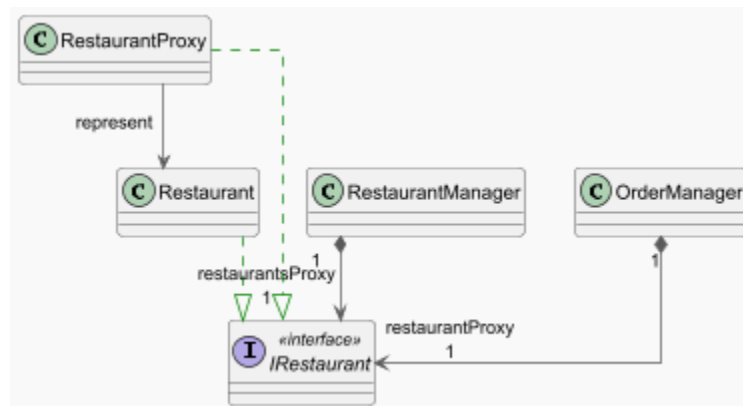
iv . Design Patterns

Nous avons retiré le contenu des classes pour que ce soit plus lisible pour plus de détails, voir le diagramme de classe complet, remis dans la section diagramme de classe.

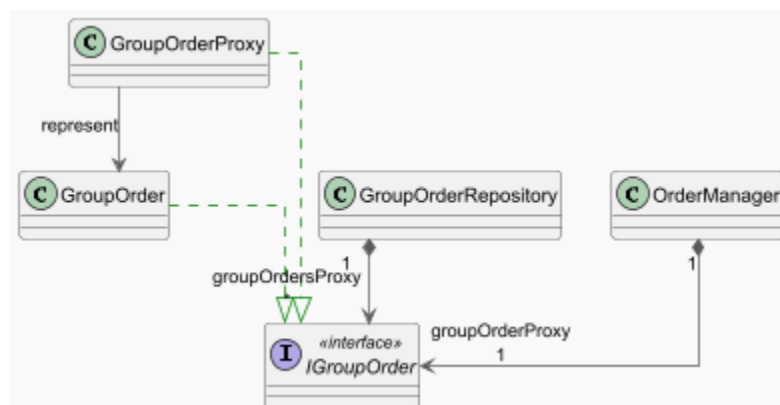
Design pattern Factory, pour les paiements :



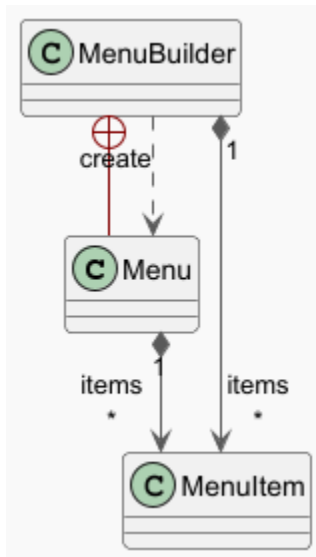
Design pattern Proxy, pour le restaurant :



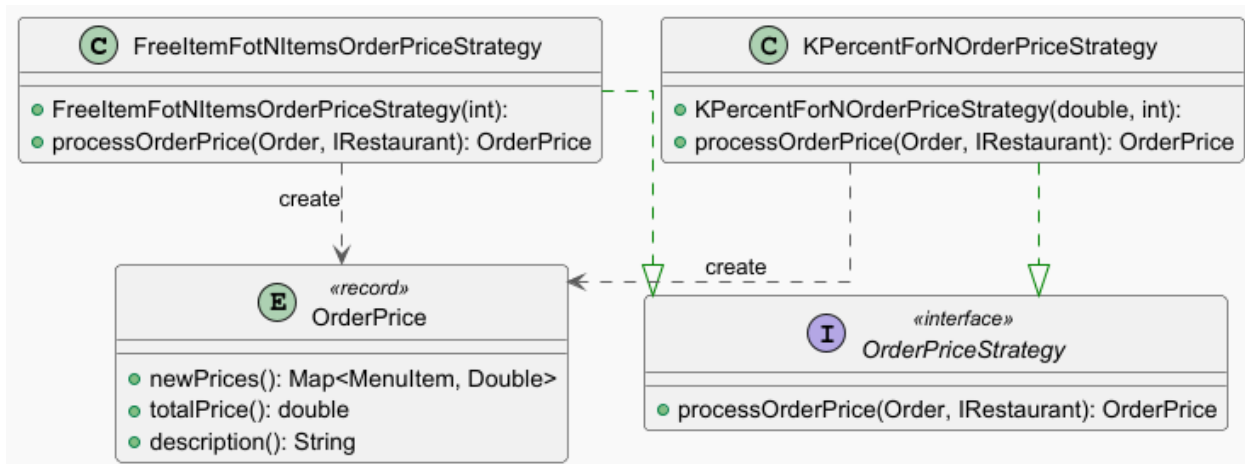
Design pattern Proxy, pour la gestion des commandes de groupe :



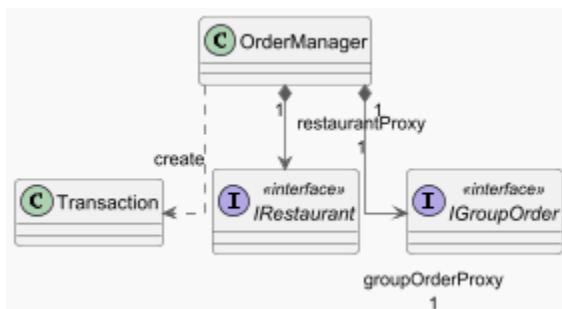
Design pattern Builder, pour la création des menus :



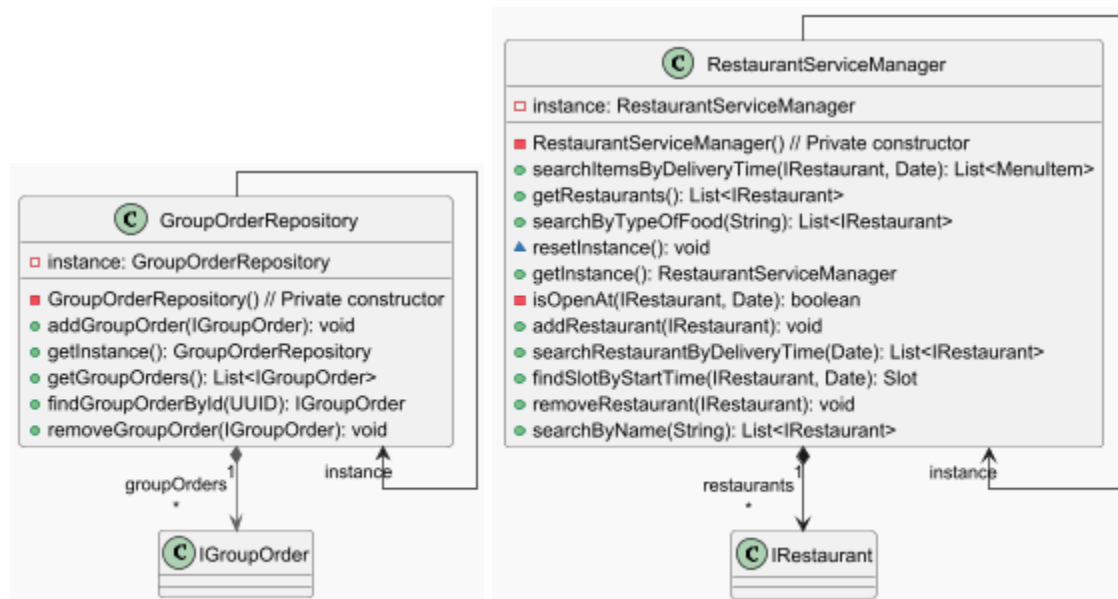
Design pattern Strategy, pour les réductions appliquées par le restaurant aux commandes :



Design pattern Facade, c'est cette classe qui va appeler toutes les autres classes :

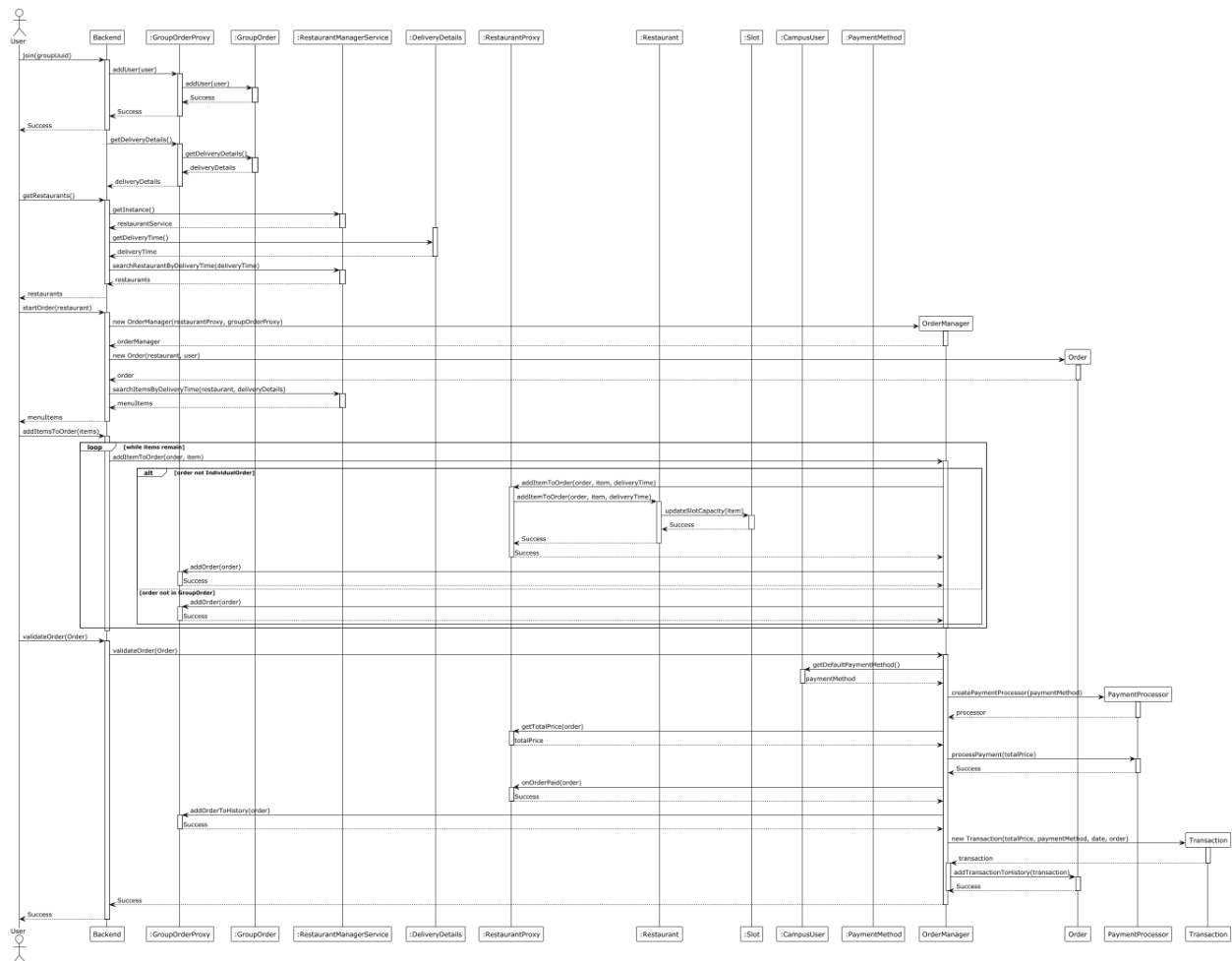


Design pattern Singleton



On aurait pu rajouter un builder pour les orders mais par manque de temps nous ne l'avons pas implémenté.

v. Diagramme de séquence



vi. Maquette

Name (job title)	N°Commande	Nom Menu	Status
Giacomo Guizzoni	012515	TODO	<input type="checkbox"/>
Marco Belton	012515	TODO	<input checked="" type="checkbox"/>
Marish Macloschian	012515	TODO	<input type="checkbox"/>
Valerie Liberty	012515	TODO	<input checked="" type="checkbox"/>

Chosir Menu Valider

Restaurant name

search

Item One
Item Two
Item Three

Annuler Valider

Your Cart

Item	Price	Quantity
Summary (1 item)		
<input checked="" type="checkbox"/> Menu TODO	\$20.00	1
Subtotal: \$20.00		
Subtotal	\$20.00	
Shipping	-	
Ext. Taxes	-	
Total	\$20.00	

Annuler Valider

Name (job title)	N°Commande	Nom Menu	Status
Giacomo Guizzoni	012515	TODO	<input type="checkbox"/>
Marco Belton	012515	TODO	<input checked="" type="checkbox"/>
Marish Macloschian	012515	TODO	<input type="checkbox"/>
Valerie Liberty	012515	TODO	<input checked="" type="checkbox"/>
Me	012515	TODO	<input checked="" type="checkbox"/>

Chosir Menu Valider

3 . Qualité des codes et gestion de projets

Types de tests









Au cours du développement d'un BACK-END, il était nécessaire de tester notre code au fur et à mesure que l'on avance.

C'est pourquoi nous avons effectué 2 principaux tests : les tests **unitaires** (JUnit, Mock..) et les tests **Cucumber**.

Nous avons parcouru l'ensemble des points exigés par l'étude de cas à base de scénarios avec les tests Cucumber. Ils nous ont permis de tester la validité des méthodes et de pouvoir gérer de temps en temps des cas d'erreur pour s'assurer de la pertinence de notre méthode et du respect de notre conception.

Ils ont été accompagnés par des tests unitaires (avec des mock) pour regarder plus en détail certaines méthodes et rendre notre code plus qualitatif.

Voilà le coverage qu'occupent nos tests:

Element ^	Class, %	Method, %	Line, %	Branch, %
✓  fr.unice.polytech.equipe.j	100% (30/30)	87% (188/214)	85% (443/517)	73% (160/219)
>  delivery	100% (3/3)	90% (9/10)	95% (23/24)	100% (6/6)
>  order	100% (7/7)	85% (49/57)	78% (111/142)	62% (35/56)
>  payment	100% (7/7)	92% (12/13)	94% (32/34)	100% (11/11)
>  restaurant	100% (9/9)	88% (92/104)	86% (231/267)	73% (106/144)
>  slot	100% (1/1)	92% (13/14)	95% (23/24)	100% (2/2)
>  user	100% (2/2)	76% (10/13)	86% (20/23)	100% (0/0)
 TimeUtils	100% (1/1)	100% (3/3)	100% (3/3)	100% (0/0)

Ainsi, nous jugeons nos tests pertinents et qualitatifs car on a 87% des méthodes qui sont implémentées dans les tests..

Vision de notre code

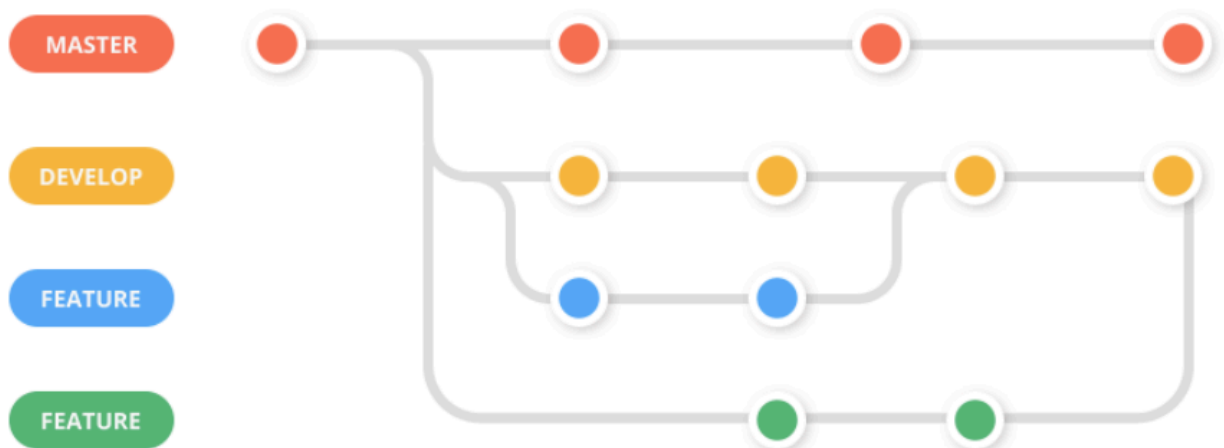
Nous estimons avoir produit un code de qualité, nous avons pris du temps via review de pull requests et divers refactoring et pouvons assurer que les points suivant ont été respectés :

- Les différents éléments des spécifications ont été identifiés et répartis logiquement dans les classes et packages.

- Les responsabilités sont clairement séparées sans ambiguïtés
- Nous nous sommes appliqués dans l'utilisation des design patterns afin que ceux-ci soient utilisés conventionnellement.
- Le code est lisible, il est possible de comprendre le déroulé de celui-ci simplement en regardant les méthodes appelées dans les tests cucumber.
- Le code est correctement testé dans sa grande majorité que ce soit avec les scénarios ou les tests unitaires.
- Le code est modulable via le polymorphisme et le principe d'injection de dépendances, de plus les dépendances bidirectionnelles ont été minimisées. Ainsi les mock tests ont pu être réduits considérablement.
- Bien que les noms des méthodes soient pertinents, les méthodes complexes sont également documentées pour assurer une cohésion entre les membres de l'équipe.

Gestion du projet

Pour la gestion du projet, nous avons divisé les tâches par issues, en rédigeant les Users Stories et en classant les tâches par milestones. Pour l'organisation des branches, nous avons utilisé le modèle suivant :



Pour s'organiser dans la progression des tâches, on a utilisé un tableau Kanban. Chaque feature correspondait à une issue. De plus, nous avons imposé des conventions de commit tel que dans le message de commit, il est nécessaire de mettre : "<type de commit>: <objet du commit> #<numéro de l'issue>". Enfin, à chaque tentative de merge d'une feature dans la branche dev, il était nécessaire à chaque personne de review la pull request et d'approuver pour pouvoir merge (une approval était suffisante).

4 . Rétrospective et Auto-évaluation

Prise de recul

Ce projet nous a permis de prendre un recul sur notre conception d'application, sur l'organisation en équipe et sur la compréhension des exigences imposées.

En effet, ce qui a été plutôt bien mené, c'est notre gestion du projet. Grâce à Github, nous avons fait en sorte d'avoir une gestion du projet structurée, où chaque personne du groupe devait être impliquée dans chaque changement pour chaque feature. Cela nous a fait gagner du temps mais aussi de l'expérience en travail de groupe.

Pour la conception UML, c'est le point qui nous a le plus fait défaut et donc le domaine où nous avons le plus appris. En effet, nous avons fait des premières erreurs sur la modélisation du diagramme de classe principalement avec la représentation des acteurs en leur incluant des méthodes par exemple. C'était l'un des gros points noirs de notre modélisation et donc nous étions amené à prendre encore plus de recul sur ce qui était demandé et les façons de pratiquer la conception logicielle. Désormais, nous comprenons également l'importance de modéliser ses idées sous forme de diagrammes avant même de commencer à coder. Cela nous permet de commencer à programmer avec les idées claires et à éliminer un maximum de questions qui auraient pu arriver sur le tas.

Enfin, nous avons appris de nombreux Design Patterns. Une partie du groupe n'avait jamais eu l'occasion d'utiliser ces patrons donc ce fut un point intéressant à explorer, même si nous n'avons pas implémenter l'entièreté des Design Patterns que l'on a regardé. C'est une façon de programmer qui nous aide beaucoup en termes d'architecture donc c'est un des points où nous avons beaucoup appris.

Remplissage des tâches

- **Product Owner (LACROIX Baptiste) :** En tant que Product Owner, j'ai assuré le bon déroulement de notre projet en créant les issues correspondantes aux User Stories (US). J'ai également attribué des rôles spécifiques à chaque membre de l'équipe, facilitant ainsi la collaboration. Pour garantir que toutes les fonctionnalités demandées par le client étaient implémentées correctement, j'ai élaboré un scénario complet intégrant toutes les US.

- **DevOps (TLILI Abderrahmen)** : Dans mon rôle Ops, j'ai géré le CI/CD avec GitHub Actions pour les tests et déploiements automatisés, configuré le pom.xml pour les dépendances, examiné les PRs en faisant des suggestions, développé certaines des fonctionnalités principales et veillé à l'alignement avec la vision du projet.
- **Architecte (MAÏSTRE Antoine)** : Dans ce projet, mon rôle d'architecte est de définir l'architecture globale, en veillant à son alignement avec notre vision et nos exigences techniques. J'établis les standards de codage, les modèles architecturaux et les bonnes pratiques, afin de guider l'équipe vers la cohérence et la qualité. J'assure également la cohésion et l'intégration des différents modules, garantissant une interaction fluide et gérant les dépendances entre les composants. En plus, je révise les conceptions et le code pour maintenir l'alignement avec l'architecture prévue et j'apporte des conseils sur les décisions techniques complexes pour garder le projet structuré et évolutif.
- **Quality Assurance (TOUPENCE Tom)** : J'ai totalement découvert l'aspect des tests. N'ayant jamais utilisé de tests unitaires ou bien Cucumber, j'ai pu apprendre sur le tas. Ainsi, j'ai vérifié pour chaque tests Cucumber que l'on évitait d'effectuer des actions dans les Given, que l'on respectait bien le découpage de scénarios pour chaque Users Story. Ensuite, j'ai assuré d'avoir un score de coverage convenable pour nous confirmer que l'on testait bien les méthodes. Enfin, j'ai veillé à une qualité de code optimale avec la meilleure lisibilité malgré parfois quelques défauts dû à l'implémentation des acteurs que l'on a veillé à corriger au maximum.

Répartition des points

Répartition des points	Tom	Antoine	Baptiste	Abderrahmen
Tom	24	25	26	25
Antoine	25	25	25	25
Baptiste	25	23	29	23
Abderrahmen	27	22	32	19
<u>TOTAL :</u>	101	95	112	92