
SophiaTech Eats

Take your time to enjoy

Équipe L

Product owner	:	Stanislas Mélanie
DevOps	:	Ponce Yvann
Quality A	:	Baidouri Lamya
Software Architecture	:	Chhilif Anas

Année : 2023-2024

1. Périmètre fonctionnel	2
1.1. Hypothèses de travail	2
1.2. Extensions choisies et éléments spécifiques.....	3
1.3. Points non implémentés.....	4
2. Conception	4
2.1. Glossaire	4
2.2. Diagramme de cas d'utilisation et User Stories	4
2.2.1. Diagramme de cas d'utilisations	4
2.2.2. User Stories des extensions	4
2.2.3. Critères d'acceptation	5
2.3. Diagramme de classes - Architecture.....	6
2.4. Diagramme de séquence.....	7
3. Design Patterns.....	8
3.1. Design Patterns en détail.....	8
3.2. Autres Design Patterns	10
4. Qualité des codes et gestion de projet.....	10
5. Rétrospective et auto-évaluation	12

1. Périmètre fonctionnel

1.1. Hypothèses de travail

Nous acceptons les hypothèses évoquées dans l'étude de cas du projet :

- Connectivité Internet stable : il est supposé que les utilisateurs disposent d'une connectivité Internet stable et fiable lors de l'accès au système.

- Disponibilité du personnel de livraison : La disponibilité du personnel de livraison est supposée suffisante pour répondre aux demandes de livraison. Bien que le module ne traite pas directement de l'embauche du personnel de livraison, il est supposé que la main-d'œuvre nécessaire est disponible.
- Intégration cohérente de l'API : l'intégration avec des systèmes de paiement externes, tels que PayPal et Google Pay, est supposée simple et cohérente et suit toujours le même principe.
- Gestionnaires de restaurant coopératifs : on suppose que les directeurs de restaurant s'impliquent activement dans le système et maintiendront avec précision leurs offres de menu, y compris en éliminant les offres lorsque les stocks sont faibles.
- Lieux de livraison précis : Les utilisateurs doivent fournir des lieux de livraison pré-identifiés sur le campus. Cette hypothèse garantit que les livraisons sont effectuées dans les zones désignées correctes.

Nous ajoutons ces hypothèses-ci :

- Nous supposons que l'accès à une commande groupée est disponible tant que l'initiateur principal n'a pas payé et confirmé la commande.
- Le créneau choisi pour une commande doit être dans minimum 2h, en raison du temps de préparation et de livraison.
- Le créneau de l'afterwork est précisé par le restaurateur à la création de l'afterwork.

1.2. Extensions choisies et éléments spécifiques

Pour l'implémentation de notre version 2 du projet, en plus de l'extension requise concernant la diversité des commandes, nous avons choisi :

- les ristournes des restaurateurs [EX2]
- le système de recommandation [EX7]

Concernant la diversité des commandes, nous avons rajouté un type de menu "AfterWork", géré comme un menu classique lors de la commande, mais avec une fonction de création qui légère différemment, prenant en compte également le nombre de personnes concerné, et qui passe directement la commande en "closed" sans passer par les étapes de paiement, livraison, etc.

Au sujet des ristournes, via l'historique de commande de l'utilisateurs on récupère le nombre de commande déjà faite pour le restaurant concernant, et c'est lors de la création de la commande, via les infos du restaurant sur les ristournes proposées, que l'application ou non de la ristourne se fait.

Le système de notation des entités est indépendant du reste. Lorsqu'un utilisateur décide de mettre une note il appelle cette classe en donnant le nom de l'entité et la note attribué. Cette classe détient toutes les notes attribuées et permet de les récupérer. Le RatingManager n'a aucune dépendance avec le reste du projet.

Enfin, concernant les fonctionnalités spécifiques de notre projet, nos notifications peuvent être gérées de manière transparente selon divers types de notifications avec des comportements supplémentaires, comme le formatage spécifique des e-mails ou l'ajout de mécanismes de livraison.

1.3. Points non implémentés

Le buffet n'est pas totalement implémenté. Nous avons commencé par la prise en compte des afterwork dans le système de commande, et débiter le buffet. Le fait qu'il ne soit pas terminé est en partie dû à un problème de communication entre nous à la fin du projet, qu'il était trop tard pour corriger.

2. Conception

2.1. Glossaire

Réduit aux termes non présents dans le glossaire initial :

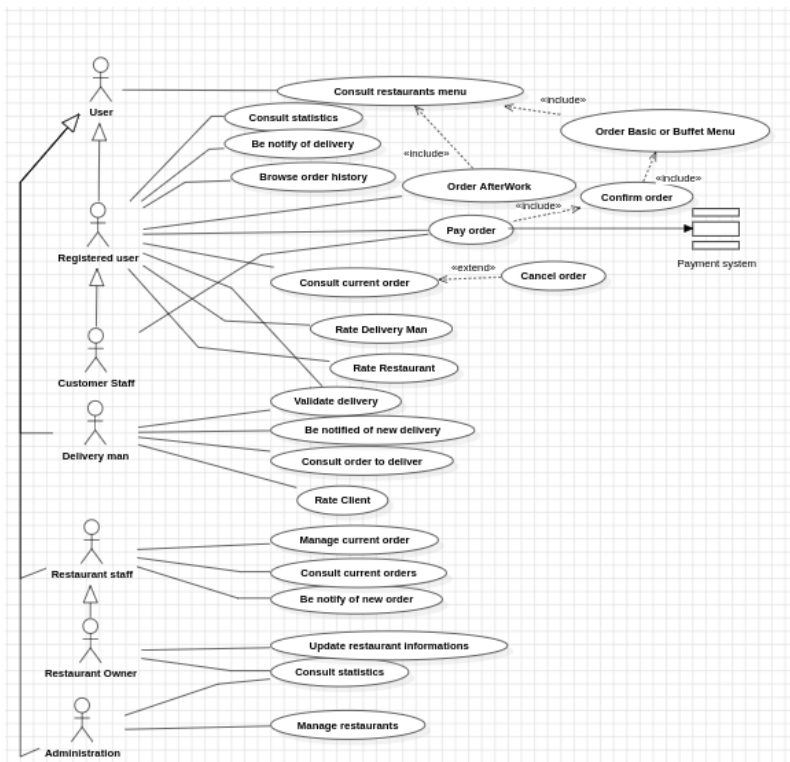
Buffet : commande pour x personnes, pour laquelle la personne qui la passe doit faire partie du staff de l'université + la personne qui reçoit la commande est différente

Afterwork : une commande non payée ni livrée (on peut la voir comme une réservation) pour x personnes

Ristourne : réduction en fonction du nombre de commandes directes

2.2. Diagramme de cas d'utilisation et User Stories

2.2.1. Diagramme de cas d'utilisations



2.2.2. User Stories des extensions

[US]V2/AfterWork : Ajouter le type de menu AfterWork à la carte du restaurant
En tant que manager du restaurant X

Je veux ajouter le type de menu “AfterWork” pour un nombre nb maximum de personnes à la carte de mon restaurant X
Afin que les clients puissent le consulter et le commander

[US]V2/AfterWork : Passer une commande pour un AfterWork

En tant qu'utilisateur connecté

Je veux passer une commande de type “AfterWork” pour 10 personnes au restaurant X
Afin de notifier le restaurant X de ma commande

[US]V2/Buffer : Ajouter le type de menu Buffet à la carte du restaurant

En tant que manager du restaurant X

Je veux ajouter le type de menu “Buffet” pour un nombre nb de personnes à la carte de mon restaurant X
Afin que les clients puissent le consulter et le commander

[US]V2/Buffer : Passer une commande pour un buffet

En tant qu'utilisateur connecté et membre du staff de l'université

Je veux passer une commande de type buffet
Afin de la faire livrer à un tiers

[US]V2/Ristournes : Bénéficier d'une ristourne

En tant qu'utilisateur connecté

Je veux passer ma 10ème commande simple dans le restaurant X
Afin de bénéficier de ma ristourne

[US]V2/Notation : Noter le restaurant

En tant qu'utilisateur qui a été livré suite à une commande passée dans le restaurant X

Je veux noter le restaurant X suivant différents critères

Afin que mon avis soit ajouté dans les avis du restaurant X

[US]V2/Notation : Noter le livreur

En tant qu'utilisateur qui a été livré par le livreur Y suite à une commande passée dans le restaurant X

Je veux noter le livreur Y suivant différents critères

Afin que mon avis soit ajouté dans les avis du livreur Y

[US]V2/Notation : Noter le client

En tant que livreur qui a livré la commande au client X

Je veux noter le client suivant différents critères

Afin que mon avis soit ajouté dans les avis du client

2.2.3. Critères d'acceptation

Dans le cas du système de recommandation, les critères d'acceptation sont les suivants :

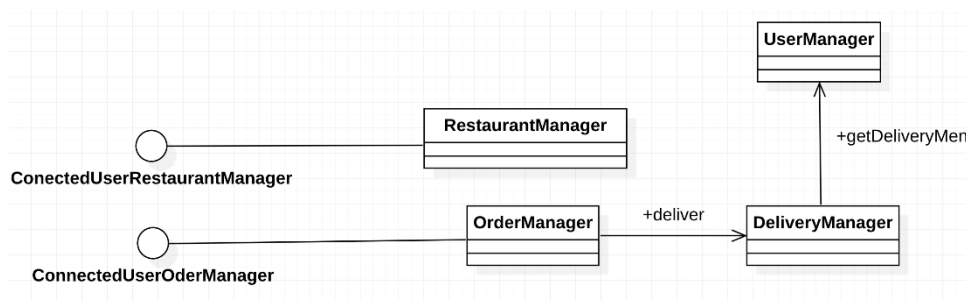
- L'entité concerné a bien la note correspondante à ce que l'utilisateur a renseigné. Cette note a été enregistrée dans le RatingManager et peut être récupérée en questionnant cette classe avec le nom de l'entité

2.3. Diagramme de classes / Architecture

L'architecture de STEats a été conçu selon les principes SOLID de la programmation orienté objet. Les communications entres les classes ont été réduites au maximum, un seul appel nécessitant de la synchronicité réside dans cette architecture. Il permet de vérifier la disponibilité des menus et les horaires du restaurant.

Chaque classe a une responsabilité unique : [Gestion des commandes](#) - [Gestion des utilisateurs](#) – [Notation des acteurs](#) – [Envoi des notifications](#)

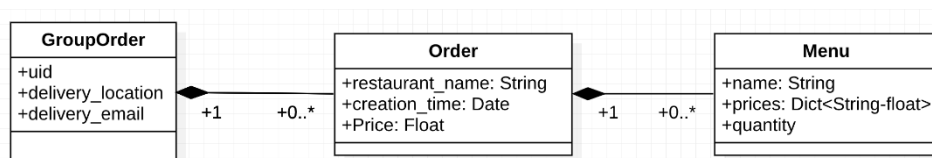
Fonctions principales



Les fonctions core de l'application sont partagées en 4 classes. L'appel de OrderManager vers DeliveryManager est unidirectionnel. Une fois la commande envoyée pour la livraison, plus aucune communication entre les deux classes n'est nécessaire. UserManager permet aux utilisateurs de se connecter à l'application et de récupérer les interfaces correspondantes à leur rôle. Il renvoie aussi au DeliveryManager la liste des livreurs. Cet appel pourrait être remplacé par un pattern observer qui notifierait le DeliveryManager à la création d'un nouveau livreur.

Commandes groupées

Toutes les commandes sont réalisées par la création d'un objet GroupOrder. Cette order général détient les informations de livraison ainsi que les commandes de chaque acteur y participant.

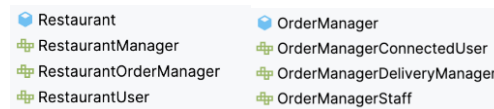


L'objet Order contient quant à lui l'ensemble des menu commandés par l'utilisateur.

Chaque Order contient un objet State qui selon le pattern State, control l'état de la commande et les étapes à passer jusqu'à sa complétion. Pour la gestion des commandes [AfterWork](#) ou la commande ne nécessite pas de paiement, l'état est directement déclaré comme commande confirmé à la commande de celle-ci.

Gestion des utilisateurs

Les utilisateurs de l'application sont tous enregistrés dans UserManager. Lors de leur connexion au système, l'interface correspondant à leur rôle leur est retournée. De ce fait les utilisateurs n'ont accès qu'au méthodes leur étant autorisés.



Des interfaces sont aussi utilisées pour les communications entre les classes. Elles ne sont ainsi seulement dépendantes des méthodes utilisées, respectant donc le **principe de ségrégation des interfaces** (I) de SOLID.

Livraison

Une fois la commande envoyée au service de livraison, l'OrderManager n'en est plus responsable. Si l'utilisateur ne trouve pas la commande en questionnant l'OrderManager, il sollicite le DeliveryManager. Cette séparation des responsabilités permet un couplage faible entre les classes. La prise en charge de la livraison peut être effectuée de manière **asynchrone** pour l'OrderManager.

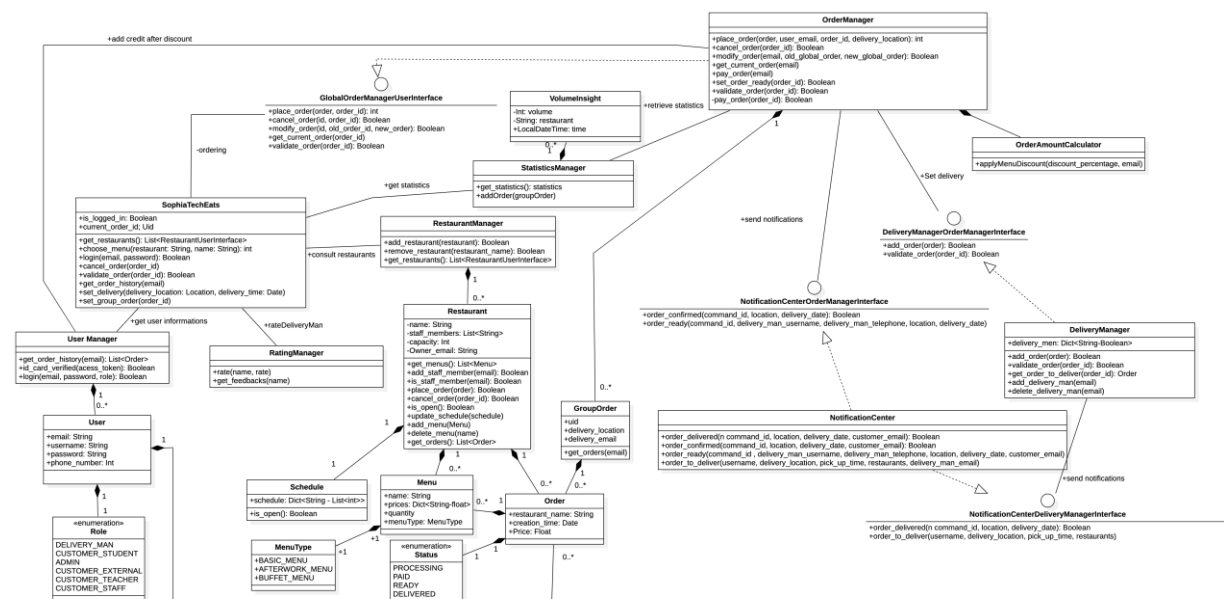
Fonctions secondaires

Toutes les fonctions secondaires sont réalisées par des appels pouvant être asynchrone. En effet, les classes principales ne sont plus sollicités et n'ont pas besoin du retour de ces méthodes pour continuer leur service.

La sauvegarde des commandes dans le profile utilisateur est faites au moment de la livraison ou le DeliveryManager envoi au UserManager la commande à enregistrer.

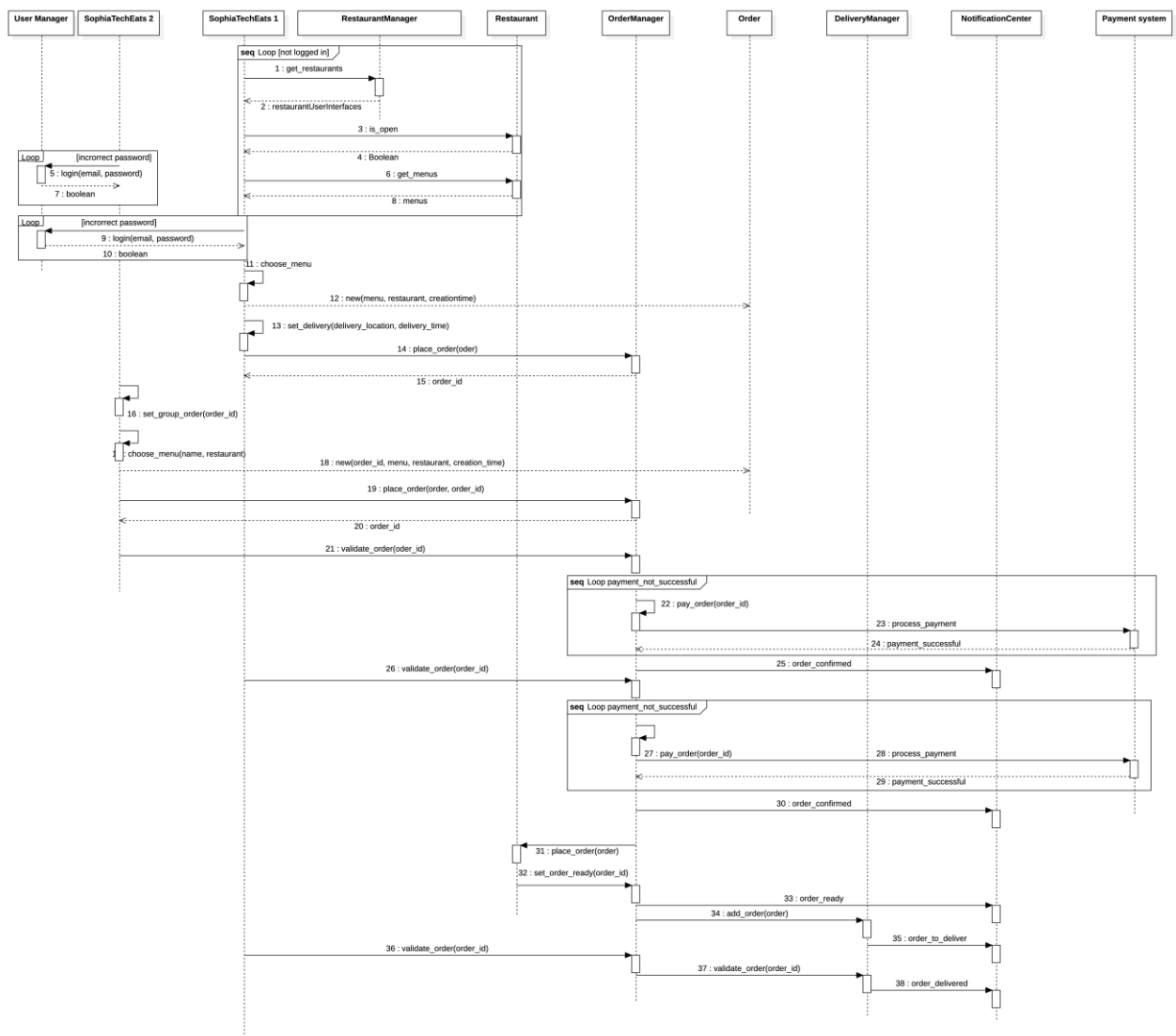
Ce même principe est utilisé pour générer les statistiques où le GroupOrder est envoyé au StatisticsManager.

Diagramme de classe ci-dessous :



2.4. Diagramme de séquence

Le diagramme de séquence ci-dessous représente une prise de commande simple. Il parcourt le chemin qu'un utilisateur est amené à faire pour pouvoir commander un menu.



3. Design Patterns

3.1. Design Patterns en détail

Patron de Conception d'État (State Pattern)

Contexte: Le patron de conception d'état est souvent utilisé dans le développement logiciel pour permettre à un objet de changer son comportement lorsque son état interne change. Ce pattern est particulièrement utile dans la modélisation des systèmes où l'objet peut être dans un nombre fini d'états distincts et où son comportement dépend de son état actuel.

Application dans le Projet : Dans le cadre de la gestion d'un système de commandes pour un restaurant, le State Pattern permet de représenter clairement et efficacement les différents états d'une commande (par exemple, CreatedState, PaidState, DeliveredState, etc.) et de gérer les transitions entre ces états de manière ordonnée.

Rôles des Éléments :

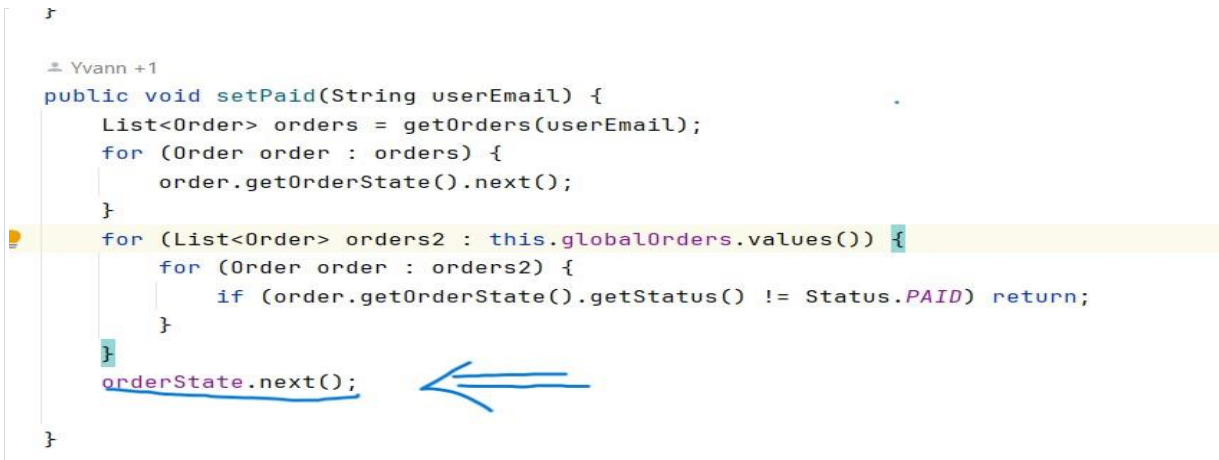
- **IOrderState (Interface d'État de Commande) :** Une interface qui définit les actions possibles pour les différents états. Chaque état concret du système de commande implémentera cette interface.

- Concrete States (États Concrets) : Des classes telles que CreatedState, ReadyState, DeliveredState, etc., qui implémentent l'interface IOrderState. Chaque classe représente un état spécifique d'une commande et définit le comportement de la commande dans cet état.
- Context (Contexte) : Un objet OrderState qui maintient une instance d'une sous-classe de IOrderState représentant l'état courant de la commande. Le contexte délègue toutes les actions spécifiques à l'état à l'instance de IOrderState.

Avantages : L'utilisation du State Pattern dans ce contexte offre plusieurs avantages :

- Encapsulation des Comportements : Les comportements spécifiques à chaque état sont encapsulés dans leurs classes respectives, ce qui rend le code plus modulaire et facile à gérer.
- Facilité de Maintenance et d'Extension : Ajouter de nouveaux états ou modifier les comportements existants est simplifié, car cela nécessite des modifications dans des classes d'état spécifiques plutôt que dans une structure conditionnelle complexe.
- Clarté du Flux de Travail : Le flux des états d'une commande est plus compréhensible et peut être suivi de manière visuelle à travers le diagramme UML, améliorant la communication au sein de l'équipe de développement et avec les parties prenantes.

Exemple d'Utilisation: Un exemple de changement d'état pourrait être illustré par le passage d'une commande de CreatedState à PaidState. Lorsque le paiement de la commande est confirmé, le contexte (OrderState) change son état interne de CreatedState à PaidState, et le comportement de la commande est mis à jour pour refléter ce nouveau statut.



```

Yvann +1
public void setPaid(String userEmail) {
    List<Order> orders = getOrders(userEmail);
    for (Order order : orders) {
        order.getOrderState().next();
    }
    for (List<Order> orders2 : this.globalOrders.values()) {
        for (Order order : orders2) {
            if (order.getOrderState().getStatus() != Status.PAID) return;
        }
    }
    orderState.next();
}

```

Patron de Conception (Decorator):

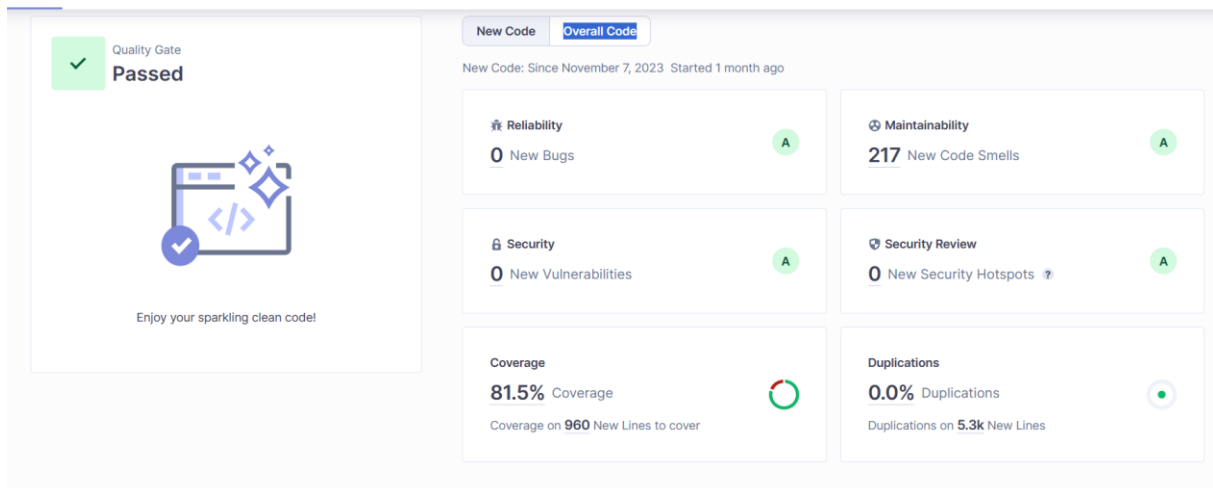
Dans le cadre de notre système de gestion des notifications, nous avons commencé à poser les bases d'une implémentation flexible en utilisant le pattern Decorator. Ce modèle de conception nous permet d'ajouter de nouvelles fonctionnalités aux notifications sans modifier les classes existantes, ce qui favorise l'extensibilité et la maintenabilité du code. Nous avons défini une interface **Notification** qui décrit l'opération de base de notification, et à travers **NotificationDecoratorInterface**, nous avons créé un cadre permettant l'extension dynamique de cette fonctionnalité. Les décorateurs concrets, tels que **EmailNotificationDecorator**, enrichissent les objets de notification avec des comportements supplémentaires, comme le formatage spécifique des e-mails ou l'ajout de mécanismes de livraison. Ainsi, notre **NotificationCenter** peut gérer de manière transparente divers types de notifications et les étendre au besoin, sans perturber l'architecture existante.

3.2. Autres Design Patterns

Patron de Conception (Observer):

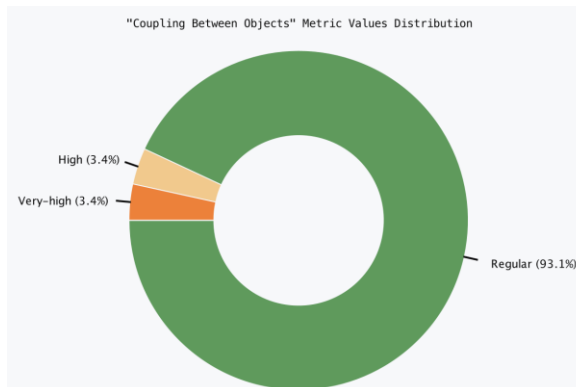
L'hypothèse d'utiliser un modèle d'Observer dans notre système de gestion de restaurant envisage de suivre la variation des capacités des établissements par créneau horaire. Cette approche permettrait de notifier automatiquement chaque restaurant des changements de disponibilité suite à des commandes ou annulations. L'idée est que, lorsqu'un restaurant démarre la préparation d'une commande, le système, via l'Observer, ajuste la capacité disponible du restaurant pour le créneau concerné.

4. Qualité des codes et gestion de projet

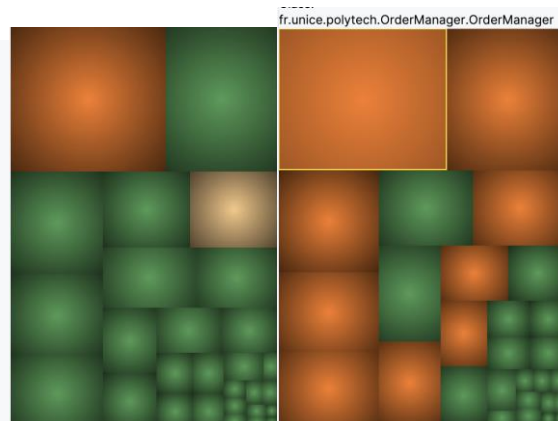


Le rapport SonarQube indique un taux de coverage de 81.5%, cette couverture est réalisée entière grâce aux tests d'intégrations qui couvrent toutes les user stories de l'application. Chaque user story est associé à un ensemble de test d'intégration ce qui nous permet de valider le comportement de l'application. Les code smells restant n'ont pas un grand impact sur la qualité du code. Notre projet assure une conformité par rapport au quality gates conseillés par SonarQube.

Coupling between object



Long method



Grâce au plugin présenté en cours (**MetricsTree**) nous remarquons que les classes de l'application n'ont pas de fort couplage entre elles, seulement la classe `OrderManager` est impactée. Cette classe est la partie la plus complexe de l'application. C'est un résultat qui correspond à ce qui était attendu. On remarque aussi que les méthodes sont généralement trop longues dans les classes principales.

DevOps

Parmi les 3 principales missions du DevOps :

- CI (Continuous Integration)
- CD (Continuous Delivery)
- Monitoring

Nous nous sommes concentrés sur l'intégration continue puisque la livraison n'était pas prise en charge sur ce projet.

L'objectif a été d'accompagner l'équipe de développement pour qu'elle puisse produire rapidement tout en respectant certains standards de qualité.

Nous sommes donc partis sur une branching strategy Trunk Base. Cette strategy permet aux équipes d'intégrer leur travail rapidement et de ne pas avoir de dérive entre les branches.

Le critère le plus important à la réalisation de cette stratégie est la conservation du build. Il est capital de ne pas casser le build pour ne pas bloquer les membres de l'équipe.

La seule condition à pouvoir regrouper son travail avec le tronc commun est de pouvoir build et de passer tous les tests.

Un Job, validate essential, a été mis en place pour vérifier ces points à chaque push sur une branche. Ce job doit s'exécuter avec succès pour envisager un merge avec le tronc commun.

Le scanner SonarQube est utilisé sur le projet et analyse le code à chaque fois qu'une personne joint son travail au tronc commun. Le scanner doit valider des standards de qualité pour permettre la mise en release de l'application.

La gestion des commits est réalisé grâce au système de création de branche à la suite d'une issue. En effet, une branche est liée à une issue ce qui permet de retrouver tous les commits liés à celle-ci.

5. Rétrospective et auto-évaluation

Mélanie Stanislas	Yvann Ponce	Anas Chhilif	Lamya Baidouri
PO	Devops	Architecte	QA
100	100	100	100

Notre équipe a adopté une approche équilibrée pour la répartition des tâches, assurant ainsi que chaque membre contribue efficacement selon son rôle tout en s'appuyant sur l'expertise des autres. Pour chaque User Story, nous avons généralement des scénarios écrits par le Product Owner (PO). Chaque issue est traitée dans une branche spécifique dédiée à une User Story.

En ce qui concerne notre utilisation de Git, nous avons créé une branche distincte pour chaque User Story. Sur ces branches, nous avons développé toutes les fonctionnalités relatives avant de fusionner les versions finales sur la branche principale, nommée "Intégration".

Pour maintenir une communication fluide, nous avons utilisé un groupe Discord. Cette plateforme nous permet d'avoir des discussions synchrones entre les différents membres de l'équipe. Elle est également utile pour se tenir au courant des changements et des mises à jour qui surviennent au cours du projet.

Project Owner : “En tant que PO du projet, j’ai veillé à la bonne compréhension du sujet par l’équipe. Mon rôle a été en majorité de lire et relire le sujet du projet, de mettre en place une TodoList des différentes tâches à faire pour chaque semaine. L’écriture des User Story a été un travail qui a demandé beaucoup de temps. J’ai veillé à jouer en partie de rôle de leader, même si nous étions tous au même niveau concernant les décisions à prendre, j’étais celle qui relançait les sujets et veillait à l’organisation. J’ai appris via cette expérience qu’être PO demande beaucoup de rigueur, d’organisation et de management. Si je devais revenir en arrière, j’établirais une meilleure communication sur l’avancement du projet et des réunions chaque semaine pour être à jour et plus organisés.”

Architect : “En tant que SA du projet, j’ai conceptualisé avec l’aide de mon équipe les besoins techniques de notre projet, ainsi que les solutions qu’on peut introduire pour résoudre les problèmes qu’on peut rencontrer. J’ai commencé par faire le diagramme de cas d’utilisation, puis j’ai discuté avec l’équipe pour que l’on mette en place une architecture flexible et découplé. J’ai aussi assuré l’ajout des design patterns et la maintenance de robustesse de l’architecture lors des changements des spécifications.”

Quality Assurance : “En tant que QA, j’ai contrôlé la qualité du code produit et la pertinence des tests réalisés. Je me suis ainsi assuré du bon fonctionnement des use cases demandées.”

DevOps : “En tant que devops, j’ai accompagné l’équipe dans ses besoins en assurant via différents outils le bon déroulement du projet et la bonne coopération dans l’équipe.”