Rapport Conception logicielle

The cookie Factory

Octobre - Novembre 2020

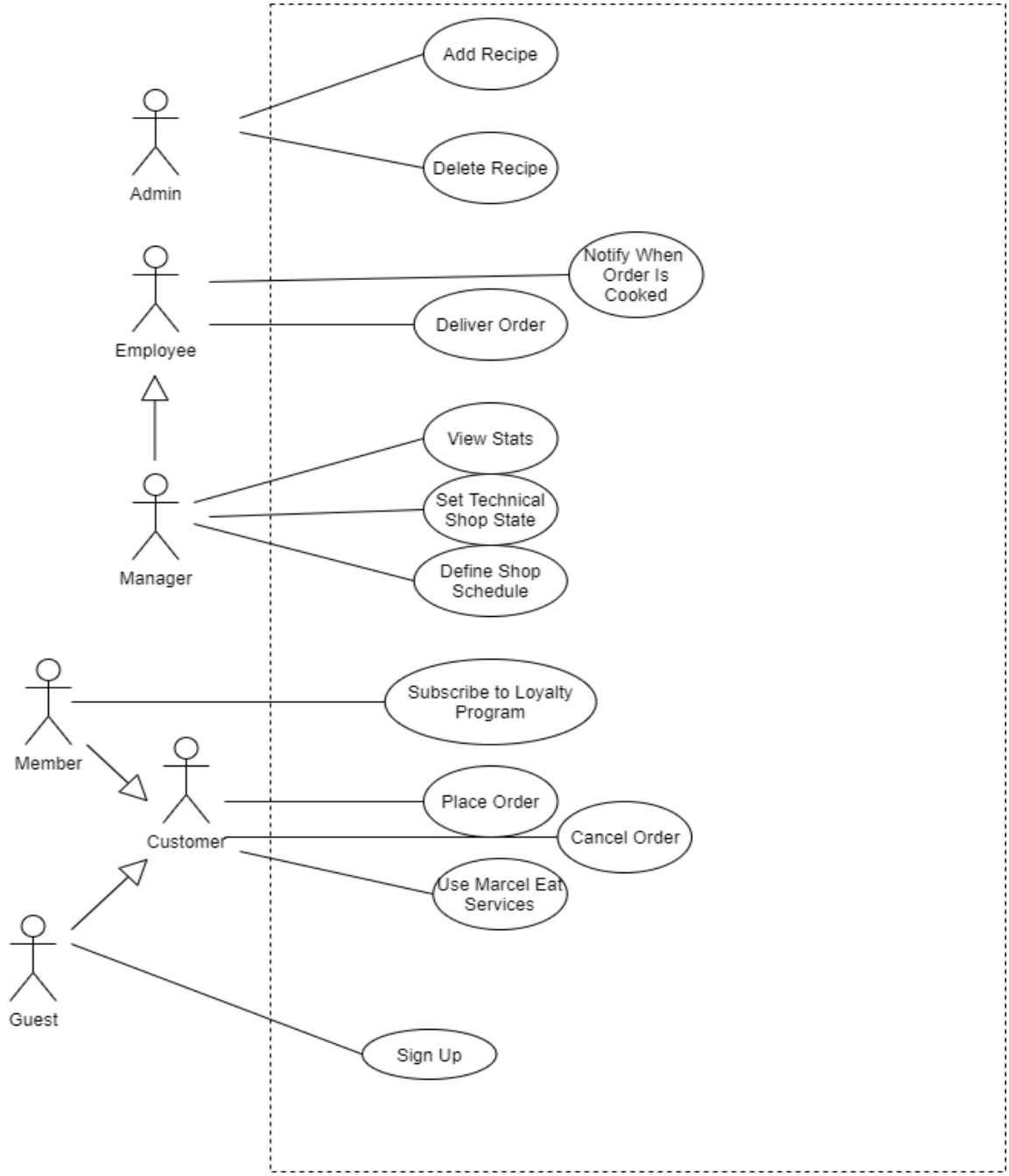
Équipe S

BERTOLOTTO Loïc
FACQ Antoine
LECAVELIER Maëva
MAZURIER Alexandre
POUEYTO Clément

Université Côte d'Azur
Polytech Nice Sophia
SI4-Conception Logicielle

Sommaire

Diagramme de cas d'utilisation



Description des actions des différents acteurs

Customer

L'acteur *Customer* est l'entité qui représente tous les clients interagissant avec notre système. Ils peuvent être des clients enregistrés, comme des invités commandant un cookie une seule fois, sans avoir de compte-client.

Cet acteur peut donc passer une commande auprès de notre *Factory*, avec une liste d'articles souhaités, une heure de récupération/livraison et préciser s'il souhaite récupérer sa commande via le *click'N'collect* ou se faire livrer grâce à *Marcel Eat* (à condition qu'il soit dans un rayon de moins de 10 kilomètres par rapport au shop d'achat choisi).
Il peut au choix commander une recette prédéfinie, une recette modifiée, ou un cookie entièrement personnalisé.
Cet acteur peut également demander à être livré en dernière minute ou annuler sa commande (si celle-ci n'a pas déjà été mise en préparation).

Guest

L'acteur *Guest* est un client qui ne possède pas de compte client. Il a donc la possibilité de s'inscrire si un compte avec son e-mail n'existe pas déjà.

User

L'acteur *User* est un client qui possède un compte. Ce dernier a donc la possibilité de s'inscrire à un programme de fidélité qui lui applique une réduction de 10% sur sa prochaine commande tous les 30 cookies commandés sur le site.

Employee

Cet acteur peut passer une commande au statut *Delivered* si celle-ci est prête au moment où il scanne le ticket associé à la commande

Manager

Manager est un employé avec un statut lui permettant d'effectuer des actions supplémentaires.
En effet, ce dernier va pouvoir modifier les horaires du magasin au sein duquel il travaille, modifier le statut du shop au sein duquel il travaille (ouvert, fermé pour cause de panne, …), vérifier la quantité de stock pour chaque ingrédient utilisé dans la préparation de délicieux cookies et également consulter les statistiques de ventes de son shop.

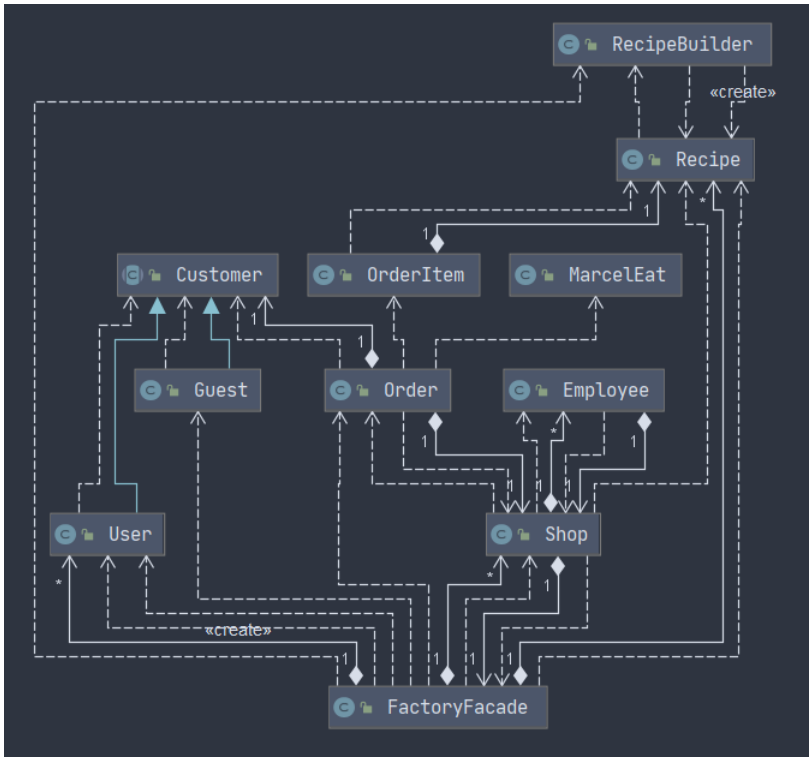
Admin

L'admin peut interagir avec le système afin de pouvoir ajouter des recettes ou bien en supprimer.

Diagrammes de classes obtenus par rétro-ingénierie

Nous avons généré des diagrammes de classes spécifiques, montrant les principales implémentations de notre projet.

Vue générale



Dans ce diagramme, nous observons la structure générale, et les dépendances entre les principaux objets de notre système. On observe que l'élément central de notre système, est la classe *FactoryFacade*.

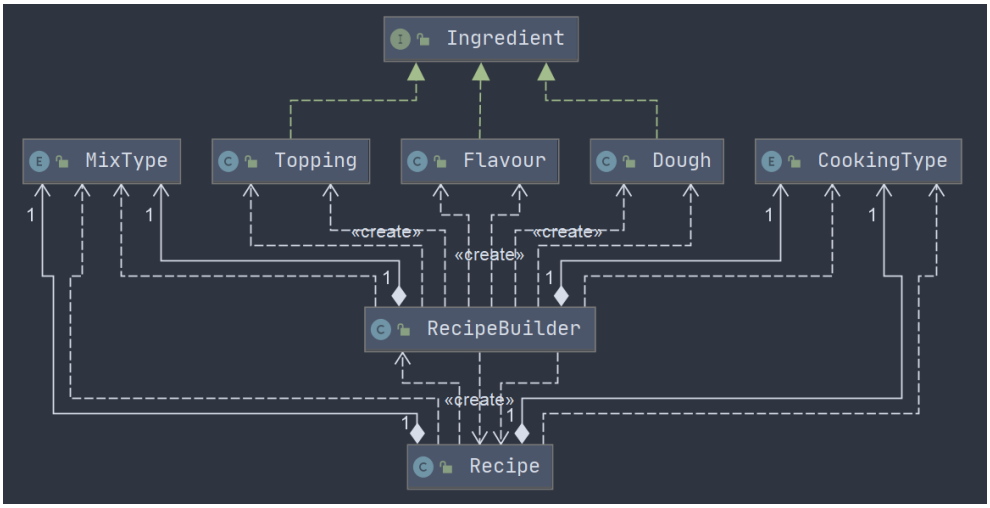
En effet cette façade est notre point d'entrée des principales requêtes. La façade possède un accès aux informations de tous les magasins ainsi que tous les utilisateurs, afin de pouvoir traiter ou rediriger les requêtes d'un acteur.

On remarque qu'un client (*Customer*) peut commander en tant qu'utilisateur invité ou membre.

Un client possède une commande en cours (ou non). Il ne peut passer qu'une commande à la fois tant que sa commande n'a pas été récupérée.

Le modèle de commande est composé de plusieurs recettes associées à une quantité.

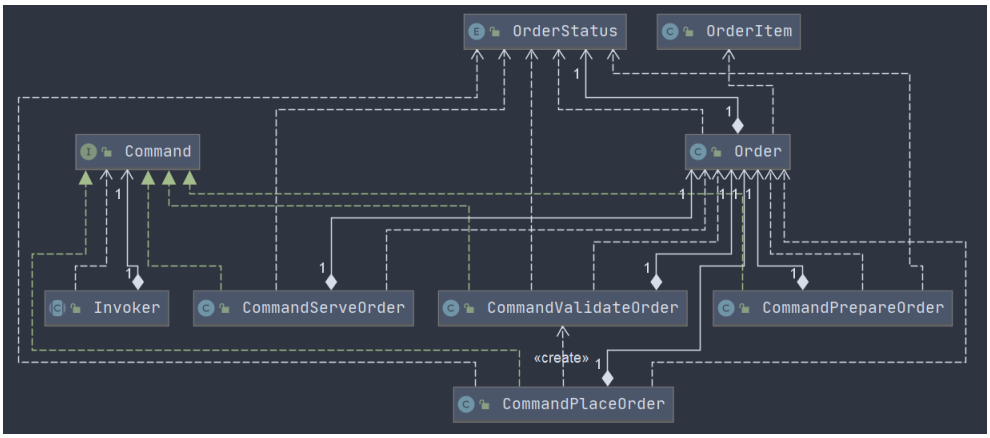
Zoom sur Recipe



Chaque recette est créée par la classe *RecipeBuilder*. Cette classe est destinée à la création des recettes de cookies, quelles qu'elles soient (recettes préexistantes, personnalisées, et étendues)

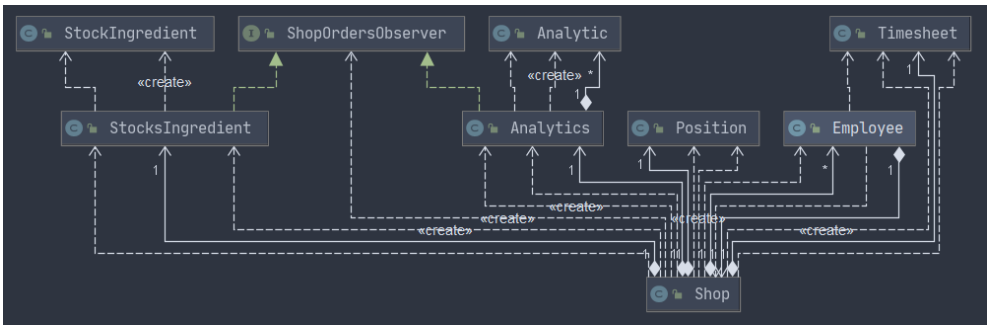
Le *RecipeBuilder* a aussi la responsabilité d'instancier les *ingrédients* demandés dans la recette. Nous détaillerons l'utilisation de ce patron de conception dans la partie 3.

Zoom sur Order



Une commande est encapsulée dans l'interface Command qui est implémentée par les classes *CommandServeOrder*, *CommandValidateOrder*, *CommandPrepareOrder* et *CommandPlaceOrder*. Ces classes représentent les différentes actions dans la vie d'une commande. La classe *Invoker* possède une Command et est implémentée par les classes manipulant les commandes.

Zoom sur shop



Un magasin est composé des horaires d'ouverture *TimeSheet*, d'une *position* et d'employés.

Un magasin possède des statistiques ainsi que des stocks d'ingrédients qui sont mis à jour automatiquement par un Observer de la classe Shop lorsqu'une nouvelle commande est passée dans un magasin.

Chaque magasin possède une liste d'employés.

Patrons de conception utilisés

Pattern Builder (RecipeBuilder) :

Une recette de cookie est composée d'un ensemble d'ingrédients de différents types (type de la pâte, saveur, type de cuisson, la garniture, dose …). Cela rend le processus de création d'une recette fastidieux et complexe car il y a de nombreuses représentations différentes. Ainsi en séparant le modèle de la classe de sa construction, on simplifie son implémentation notamment en réduisant le nombre de paramètres passés dans le constructeur.

De plus certains arguments sont optionnels comme la garniture ou l'arôme.

C'est pourquoi nous avons opté pour l'utilisation du pattern Builder qui est un pattern de construction. Celui-ci répond aux besoins cités car il suffit uniquement d'appeler les méthodes nécessaires qui constitueront la recette finale.

Cela permet par la suite de rajouter facilement d'autres fonctionnalités comme la taille d'un cookie.

De plus, il s'agit du Builder qui s'occupe de la création des objets "ingrédients" en fonction des paramètres passés ce qui évite l'accumulation de lignes répétitives pour créer ces ingrédients.

Ainsi en séparant le rôle de création et le rôle de modèle, on gagne en efficacité et en lisibilité du code.

Nous avions réfléchi au choix de "l'Abstract Factory" mais il n'a pas été retenu car il était plus lourd à mettre en place et n'était pas pour autant nécessaire. En effet la classe Builder (de recette) se charge elle-même de créer les ingrédients qui sont des objets basiques et qui n'ont pas besoin d'interface de création.

Pattern Observer (ShopOrdersObserver)

Lorsqu'une commande est mise en préparation et considérée comme validée, il est important de notifier nos stocks ainsi que nos statistiques afin de les actualiser (diminution des stocks à chaque commande mise en préparation, ajout des différents cookies commandés dans les statistiques pour déterminer le BestOf cookie par exemple).

Nous avons donc implémenté un pattern observer entre les commandes mise en préparation dans le shop et les objets représentants les stocks et les statistiques.

Nous avons un objet "ShopPrepareOrders" qui contient les commandes du shop qui sont mise en préparation. Cet objet contient deux observers : Analytics et StocksIngredient.

A chaque fois qu'une commande est ajoutée à ShopPrepareOrders, la méthode notifyObservers est appelée qui va ensuite appeler la méthode update de tous ses observers.

Le patron permet donc de simplifier l'arrivée de nouvelles commandes au sein d'un magasin.

Pattern Command (Order)

La vie d'une commande passe par plusieurs étapes avant d'être livrée au client. Il y a la phase de la création, du paiement, de la préparation et enfin de la livraison. Chaque étape joue un rôle clé dans le bon

fonctionnement du système d'information et chaque étape est réalisée par des personnes différentes qui exécuteront des actions différentes sur la commande. Ainsi nous avons souhaité découper toutes les actions possibles et les séparer par étape de la commande. C'est pourquoi nous avons choisi le patron comportemental Command qui permet d'encapsuler un objet "Order" dans une interface "Command" implémentée par les différentes étapes d'une commande (Command placed, validated, cooked, delivered).

Chaque "Command" redéfinie sa méthode d'exécution afin de faire avancer l'état de la commande et cela sans contrainte de manipulation et de temps, cela permet aussi de simplifier les vérifications de commandes irréalisables ou bien de commande non payée.

De plus, par exemple, dans le cadre du projet cela permet à un client d'annuler sa commande tant qu'elle n'a pas été cuisinée, ainsi en fonction de l'état de la commande le patron s'occupera des actions à exécuter.

Les classes de clients et employés manipulant l'objet hériteront à leur tour d'une classe leur permettant d'interagir avec l'objet à leur manière. L'intégrité des données est conservée et assurée à chaque étape de la commande ce qui empêche les erreurs jusqu'au moment de la livraison.

Nous avions hésité à ajouter le patron de comportement State à notre modèle de traitement de commande cependant, cela aurait substituer une partie du rôle du patron Command. De plus l'état d'une commande ne changeait pas son comportement mais l'utilisation qu'on en a, et l'utilisation d'un Context aurait été une complication inutile. C'est pourquoi nous avons décidé que ce patron n'était pas adapté à notre projet.

Pattern Facade (Factory facade et Marcel Eat)

Dans ce projet de nombreuses classes interagissent entre elles lors d'appels de méthodes et cela peut devenir complexe lorsqu'on ne sait pas quelle classe appeler pour déclencher une certaine action. Pour pallier ce problème nous avons créé une classe FactoryFacade qui utilise le patron structurel Facade afin de regrouper des méthodes générales qui lient différents packages sur un même point d'entrée.

Cela permet de créer une interface pour l'utilisateur afin de limiter ses actions au strict nécessaire.

Dans notre cas, la création d'une commande débute par l'appel de la méthode par la façade car celle-ci fait appel à la création d'une recette, d'un magasin, des stocks disponibles, etc… Afin de simplifier l'utilisation des méthodes l'ensemble des requêtes générales est contenu dans cette classe.

Dans le cadre de la simulation de l'API de MarcelEat nous avons créé une classe simulant un appel d'API reprenant le principe du patron Facade. La classe regroupe également les appels nécessaires à l'exécution du service.

Rétrospective sur le projet

En nous basant sur notre premier diagramme de classe UML, nous avons développé la structure de base de notre application. A cette étape, le T-Shirt Sizing a eu une importance cruciale pour pondérer nos User Stories et développer les fonctionnalités associées au cours des différents sprints.

Pour le découpage, celles-ci étaient regroupées dans les Epics suivantes :

- Commander un cookie avec une recette personnalisée (ou non)
- Gestion des cookies best of (Statistiques des magasins et à l'échelle nationale)
- Magasin viable pour commande (stock, horaire, pannes, position, etc…)
- Client: cookie pot, inscription, loyalty program.
- Retrait de commande (lieu, date, heure, magasin prêt)
- MarcelEat : appel d'API (livraison de commandes)

Ainsi, les User Stories associées à ces Epics constituaient notre fil directeur. Nous nous sommes donc réparti les issues créées en conséquence en fonction de leur importance, en veillant à limiter l'interdépendance de celles-ci. En effet, nous nous sommes efforcés d'écrire des User Stories autonomes pour accélérer au maximum la phase de développement, et ainsi éviter de dépendre d'une fonctionnalité pour en implémenter une autre.

Nous avons réparti les User Stories sur GitHub en utilisant des Issues et des Milestones par "sprint".

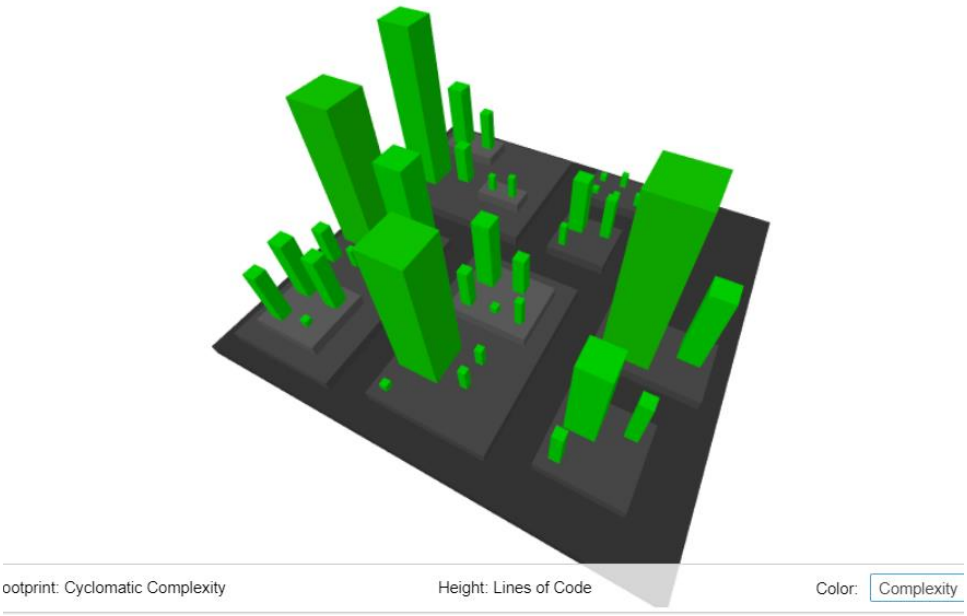
Chaque fin de sprint correspondait à une phase de stabilité dans le fonctionnement du projet (et à un merge sur la branche master). Nous avons réalisé en priorité les User Stories les plus importantes en piochant dans presque toutes les Epics afin de respecter le schéma du MVP. Par exemple dès le début du projet nous étions capables de créer une commande avec un cookie simple, qui une fois payé était prêt à être distribué dans un magasin.

Chaque issue a été décrite avec des labels représentant le T-Shirt Sizing choisi (importance et estimation du temps).

L'implémentation des différents patrons de conception (listés ci-dessus) au fur et à mesure ont permis de rendre les différentes parties du code concernées plus fonctionnelles et plus proche du métier ciblé. Notamment, la construction d'une recette et la gestion du cycle de vie d'une commande ont été implémentés en vue de soulager les classes Shop et Recipe, qui aurait auquel cas bien trop de responsabilités.

L'utilisation de SonarQube (tel qu'enseigné en QGL) a accompagné le développement de notre application en termes de qualité. En effet, cet outil nous a été utile pour analyser l'efficacité de nos tests unitaires et nous a averti des parties du code trop légèrement couvertes. Nous avons atteint plus de 80% de couverture par les tests unitaires, le pourcentage restant représente les méthodes que nous avons jugés

inutile de tester (getter et setter basiques, toString …). Avec SonarQube nous avons remarqué que notre complexité dans les différentes classes était équitablement répartie grâce à l'utilisation des patrons de conception qui ont permis un découplage des classes.



Représentation 3D du projet par classe en fonction du nombre de ligne décrivant la complexité

Pour les prochains développements, nous veillerons dès le début à orienter nos tests fonctionnels sur des scénarios plus complets (au début du développement de l'application, ces derniers n'étaient pas assez verticaux). Ceci permettrait d'emblée d'être plus confiant sur notre conception de l'application par rapport aux spécifications.

Auto-évaluation

Nous considérons ensemble avoir fourni un effort égal dans la réalisation de ce travail de conception logicielle. Grâce à l'outil git et à diagrams.net, nous avons pu travailler simultanément et à part égale tant sur la partie développement que sur la partie conception jusqu'à la rédaction de ce rapport.

Ainsi, nous attribuons 100 points d'implication à chaque membre de notre équipe.

En effet, tous les membres du groupe ont pu toucher à toutes les parties de ce projet, cependant certains membres se sont plus occupés des tests, du code ou bien de la conception de l'application, ce qui peut expliquer les quelques différences de lignes de code sur GitHub. Ainsi la quantité de travail fournie par chacun d'entre nous est quasiment égale.