Cookie On Demand

Rendu du projet de conception logicielle

Equipe 5

Natan BEKELE
Théo BONNET
Fiacre TETEVI-TOGNI
Walid LARABI

Table des matières

Contexte

Nous avons mis en place un service “*Cookies on Demand*” (CoD) pour une franchise, qui permet aux clients de sélectionner un magasin, de faire leur sélection parmi une gamme de cookies disponibles, de personnaliser certains cookies, d’en passer la commande et de spécifier la date et l‘heure à laquelle ils viendront les chercher.

Pour notre client, ce qui était important, c’est l’assurance de toujours obtenir sa recette de cookies préférée, sortant du four, en temps et en heure. sans de mauvaise surprise comme une rupture de stock disponible en moins de deux heures.

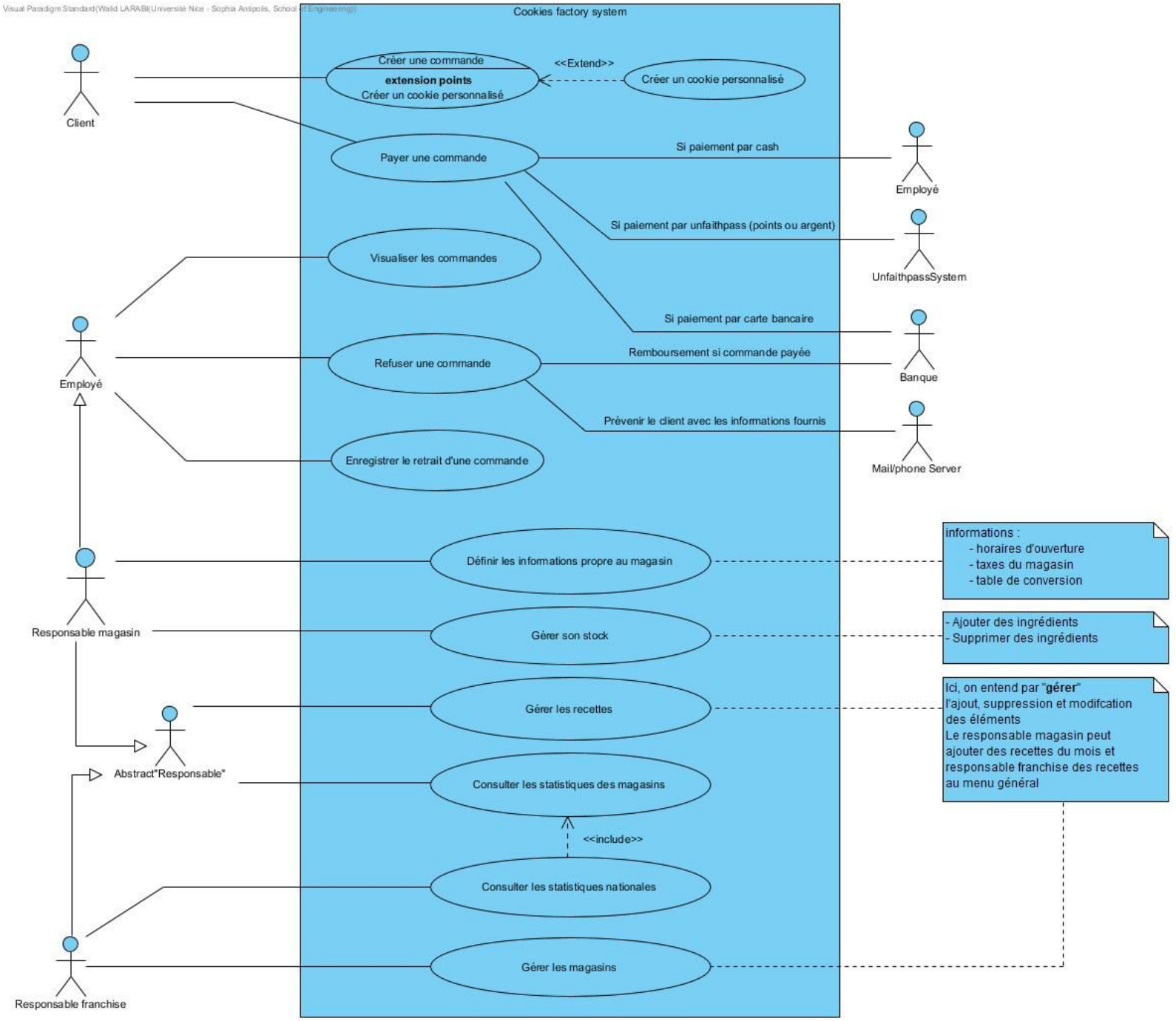
Le système permet au responsables d’un magasin de gérer le stock propre au magasin, en ajoutant, retirant des ingrédients, de modifier les prix de certains ingrédients, ou leur marges. il permet également de refuser certains commandes, de consulter des statistiques par magasin, ou au niveau national.

Notre système supporte plusieurs façons et moyens de paiement. On peut payer sur le comptoir pour des petites commandes (moins de 50 cookies par défaut, mais que le manager peut modifier), on peut payer en ligne. On peut payer par cash, par carte bancaire, par carte UnfaithPass (en argent / en points), et le système est extensible pour y greffer de nouveaux moyens de paiement *tout en respectant le open-close principle et nous obligeant à faire bon usage de l'abstraction et du polymorphisme.*

Ce qui n’est pas supporté par le système actuellement, c’est le fait d’inscrire les fournisseurs et leurs prix, ça ne fait pas partie de notre périmètre, cependant comme mentionné ci-dessus, on permet au manager d’ajuster les prix / marges sur ses ingrédients. Donc à l’extérieur de notre système, le manager calcule les nouveaux prix des ingrédients et les modifie sur le système.

De l’analyse fonctionnelle

Diagramme de cas d’utilisation couvrant le périmètre final de votre projet



L’avancée fonctionnelle et tests de validation

Thème	Critères d’acceptation	Avancée fonctionnelle	fichier de test de validation
Boutique	Je peux changer les horaires d'ouverture de ma boutique	Le manager de chaque magasin peut ajouter des portions d’horaires d’ouverture, en supprimer La suppression n’est pas possible si une ou plusieurs commandes sont prévu dans la portion concernée	features/manageStore/ **manageWorkingHours.feature**
Boutique	Je ne peux pas commander en dehors des horaires d'ouverture	Les horaires sont d’abord vérifier avant la validation de la commande et sont refusés s’ils sont trop rapproché de l’heure actuelle ou s’ils sont en dehors des horaires d’ouverture	features/ordering/ **creatingNewOrder.feature**
Boutique	Je peux mettre ma "recette du mois" à disposition	Chaque magasin dispose d’un menu qui lui est propre, ou le manager peut ajouter une ou plusieurs recettes propres au magasin (en plus des recettes de la franchise) et en supprimer.	features/manageStore/ **manageStoreMenu.feature**
Boutique	Je peux déclarer et gérer mes fournisseurs	Nous n’avons pas ajouter une partie pour déclarer et gérer les fournisseurs, cependant, nous avons ajouté une fonctionnalité pour modifier les prix/marges des ingrédients, et ça se calcule en interne en utilisant des méthodes de gestion de stock (CUMP). Nous n’avons pas trouvé logique d’avoir un même ingrédient avec des prix différents. Les prix seront ajustés par le manager avec la fonctionnalité de changement de prix	features\manageStore\ **manageIngredients.feature**
Boutique	Je peux retirer ma commande	Si la commande est prête, je peux venir la retirer à condition qu’elle soit payé. Ensuite la commande passe dans un état indiquant qu’elle a été retirée.	features/ordering/ **creatingNewOrder.feature**
Boutique	Je peux payer ma commande au comptoir	Je peux choisir de payer ma commande au comptoir. Une fois au comptoir je peux utiliser n’importe quelle type de paiement.	features/payment/ **payOrder.feature**
Boutique	Je peux régler la table de conversion euros - points du système d'infidélité pour ma boutique	Le manager de chaque magasin peut modifier les taux de conversion points-euro et euros-points. Le manager recevra certainement de la part des responsables unfaithpass les taux, et lui il les rentre.	features/manageStore/ **manageUnfaithPassPointsConverter.feature**
Commande	Je sais que la commande a été payée	Le fait que la commande a été payé est une inforamtion directement disponible sur notre commande	features/ordering/ **creatingNewOrder.feature**
Commande	Je peux commander sans compte sur la plateforme CoD	On peut commander sans compte en donnant son numéros de téléphone. C’est en réalité le seul moyen que l’on a vu que l’intérêt d’avoir un compte ne c’est pas montré.	features/ordering/ **creatingNewOrder.feature**
Commande	Il n'est pas possible de commander quelque chose qu'on ne sait pas fabriquer	Les contraintes de fabrication sont définis dans le cookie factory, qui fabrique des recettes. On définit le minimum et le maximum de chaque type d’ingrédient et si on souhaite créer une recette qui ne respecte pas les contraintes de fabrication, nous avons un échec. On a également le fait que les stocks sont vérifiés avant dès que l’on veut ajouter un cookies à sa commande.	features\manageRecipe\ **recipesCreation.feature** features/ordering/ **creatingNewOrder.feature**

Commande	Je peux passer une commande avec des cookies classique et personnalisés	On peut créer des cookies et les ajouter dans notre commande. On peut également ajouté des cookies normaux. Ensuite on peut la valider pour qu'elle soit à faire (sauf si le prix est trop élevé auquelle cas il faut la payer avant de la faire faire.	features/ordering/ **creatingNewOrder.feature**
Commande	Je peux payer ma commande par carte bancaire	On peut payer pour une commande directement en ligne ou au comptoir avec une carte bancaire	features/payment/ **payOrder.feature**
Commande	Je peux calculer un prix TTC à partir des ingrédients, des marges et des taxes.	On utilise cette technique pour calculer le prix sur nos cookies personnalisé, sur les autres, le prix est choisi par le gérant directement et n'est pas fonction des aliments.Cela permet une meilleur flexibilité pour le gérant sur le prix de ses cookies.	features/ordering/ **creatingNewOrder.feature**
Commande	Les cookies personnalisés respectent les contraintes de fabrication	Les contraintes de fabrication sont définis dans le cookie factory, qui fabrique des recettes. On définit le minimum et le maximum de chaque type d'ingrédient et si on souhaite créer une recette qui ne respecte pas les contraintes de fabrication, nous avons un échec	features\manageRecipe\ **recipesCreation.feature**
Fidélité	Je peux bénéficier des 10% tous les 30 cookies	On a implémenté ça mais le décompte n'est pas présent car cela comporte des incohérence si c'est introduit sur un système déjà en place donc on a préféré laissé le client le signaler et l'employé le vérifier	features/ordering/ **creatingNewOrder.feature**
Fidélité	Je peux payer au comptoir avec ma carte d'infidélité (en argent ou en points)	Le client choisit de payer au comptoir et il peut payer avec sa carte d'infidélité.	features/payment/ **payOrder.feature**
Fidélité	Je peux cumuler des points sur le système d'infidélité avec mes achats CoD	Tant que le client a eu une carte de fidélité après chaque achats il est crédité d'un certain points de fidélité défini par le store.	features/payment/ **payOrder.feature**
Fidélité	Je peux bénéficier d'un bonus direct (p. ex. "1 acheté, 1 offert") en consommant des points de ma carte d'infidélité	Les offres n'ont pas été implémentés dans le système. ça se fera à l'extérieur de notre système	x
Recette	La marque peut ajouter des ingrédients inconnus pour le moment	Le manager de chaque magasin peut ajouter a son stock de nouveaux ingrédients avec une quantité. Si l'ingrédient existe déjà dans le stock, on ajoute la quantité, et si il est inconnu, on l'ajoute dans le stock	features\manageStore\ **manageStock.feature**
Recette	Je peux gérer mes marges sur les cookies (ingrédients, recettes personnalisés)	Dans notre cookie factory, on peut définir les prix des recettes normal à la création, et la marge spéciale pour les recettes personnalisés et son prix final est calculé en additionnant le prix et marges des ingrédients et la marge spéciale	features\manageRecipe\ **recipesCreation.feature**
Statistiques	Je peux collecter des statistiques par boutique (p. ex. nombre de cookies vendu par jour, % de cookies personnalisés)	Chaque store peut calculer le nombre d cookie vendu par jour et le pourcentages de cookie personnalisé..	features**displayStat.feature**
Statistiques	Je peux agréger des statistiques nationalement pour TCF	TCF peut agréger tous les statistiques que les stores peuvent calculer.	features/**displayStat.feature**

Vers architecture et implémentation

un diagramme de classes complet (rétro-généré)

La taille de l'image étant **très grande**, nous avons mis le document contenant notre diagramme sur un lien externe

Lien sur google drive:
https://drive.google.com/file/d/1gYsntMP_LCigG_2AwjGtbRi4Bl3N3pSx/view?usp=sharing

Mais également sur notre dépôt github, sous le répertoire :
doc/class_diagram.pdf

Les patrons de conception retenus

pourquoi et comment ont été appliqués

Le patron Simple factory : pour la création des cookies

Nous avons utilisé Simple Factory pour instancier les objets cookies, en effet, la simple factory n'est pas seulement un wrapper autour du new, mais elle fait des calculs et utilise une logique métier pour paramétrer la création des cookies (calcul des prix..) et vérifier les règles de création des cookies s'ils sont respectés. (les règles sont modifiables)

Ce n'est pas complètement un patron de conception, parce qu'il n'utilise pas tout le pouvoir de polymorphisme, mais le besoin ne s'est pas manifesté

Comment ça été utilisé? :
Notre Cookie Factory est directement une classe factory concrète (et n'implémente pas un Creator). Si jamais nous aurons besoin d'avoir un factory utilisant une logique différente pour créer des cookies, nous implémenterons le patron complètement
La factory contient en variables le minimum et le maximum nombre possible par type d'ingrédient, l'ensemble de ces variables constitue les règles/contraintes de fabrication, et elles sont modifiables grâce à des méthodes disponibles pour le responsable.

Ce que nous avons gagné en utilisant ce pattern, c'est d'éviter la duplication, encapsuler la logique de création de cookie et avoir une logique uniforme pour la création de cookie partout dans le code.

Le patron Stratégie : pour le paiement

Pour les moyens paiement, c'était un défi de supporter plusieurs moyens de paiement, notamment sur notre projet ou on avait 4 moyens de paiement possibles. Même si les algorithmes pour payer sont différents par exemple pour la carte bancaire, on envoie des requêtes à la banque, alors que la carte de fidélité, on envoie des requêtes au système unfaithpass mais l'interface basique est la même, mettre une somme et exécuter le paiement.

Nous avons donc opté pour le patron de conception stratégie, parce que pour chaque moyen de paiement, nous avons un algorithme différent qui exécute la même tâche *Pay* (). Et c'était important pour nous de pouvoir changer ces algorithmes en runtime en fonction de comment le client veut payer.

Ce qui nous a apporté ce patron, c'est la possibilité d'étendre, en ajoutant de nouveau moyens de paiement sans modifier le code présent (ouvert à l'extension, fermé à la modification). Et la possibilité de changer les moyens de paiement en runtime grâce au polymorphisme.

Comment ça été utilisé?
Par conséquent, au lieu de disperser la logique des méthodes de paiement dans l'ensemble de notre système, nous avons créer une interface simple, PaymentMethod, avec une méthode unique acceptant un coût en paramètre. Nous allons ensuite créer trois classes de stratégies concrètes: CashPayment, CreditCardPayment et UnfaithPassPayment.
Etant donné que les classes UnfaithpassMoneyPayment et UnfaithpassPointsPayment partagent les mêmes champs (tels qu'un QR Code), nous allons créer une superclasse abstraite nommée UnfaithpassPayment pour gérer ces champs. Chacune des sous-classes de cartes concrètes remplacera ensuite la méthode executeTransaction pour exécuter une logique spécifique à la carte, telle que contacter une banque, le système unfaithpass .. etc

Le patron a été couplé avec une Factory simple pour la création des moyens de paiement valide (vérification de la numéro de la carte si c'est une carte valide visa ou mastercard avec les expressions régulières correspondantes.) par exemple.

Le patron Stratégie : pour les Statistiques

Chaque classe concrète statistique hérite de la classe abstraite Stat qui est composée d'une méthode compute prenant une liste de commande pour calculer des statistiques.
Avec du recul nous nous sommes rendu compte que ceci n'était pas la bonne solution. Notre solution aura un nombre de classe proportionnel au nombre de différentes statistiques que le store veut calculé, donc ce n'est pas optimal.
En effet on aurait pu créer quelques methode génériques qui prennent un liste de commande et un attribut de commande sur lequel calculer la statistique en question. Ces méthodes peuvent être des méthodes comme somme, maximum ou moyenne..
En utilisant la réflexivité on pourra donc regarder dynamiquement le type de l'attribut donné en paramètre et appliquer la logique de la fonction sur la chaque attribut de la liste de commande.
Ceci nous aurait permis de ne pas utilisé le patron de conception et limiter le nombre de classe nécessaires sans ajouter de couplage aux autres classes.

Le patron État : Pour les commandes

Nous avons décidé d'implémenter les commandes en utilisant un patron de conception État,

La possibilité de changer de comportement est un élément clé également pour le patron de conception. Les commandes se comportent différemment en fonction de leur état comme par exemple, une commande qui est cours de création, et on décide de l'annuler, tout ce qu'on aura a faire c'est de remettre les ingrédients qui étaient réservés dans le stock, alors que si la commande a été validée (sans qu'elle soit faite) et est annulée par la suite, on remet les ingrédients réservés dans le stock et on rembourse le client.

La possibilité de garder un contexte (comme la liste des items, le prix restant à payer..) renforce le choix

Comment ça été utilisé?

Nous avons donc les différentes classes qui implémentent OrderState, qui contient 3 méthodes importants, getState() pour avoir l'état actuel, nextState() pour aller au prochain état et Cancel() pour annuler une commande.
L'état est contenu dans un contexte qui dispose de d'autre informations concernant la commande notamment ce qui reste à payer, la liste des items, s'il a une réduction..

Les commandes peuvent se trouver dans différents états durant leur cycle de vie. Quand un client commence sa commande, la commande commence vide et elle est dans l'état "OnCreation", on ajoute à fur et à mesure les ingrédients tout en vérifiant qu'il y en a assez dans le stock, et à la fin on valide, et l'état passe à l'état "Todo", une fois la commande est faite, elle passe a "Done", ensuite "Collected" quand elle est récupérée.

Les patrons de conception non-retenus

pourquoi semblaient t'ils pertinents mais ils n'ont pas été appliqués

Le patron Decorator : pour la création des cookies

Le patron de conception Decorator était applicable pour notre façon de faire les cookies, c'est à dire, on aurait pu avoir un composant, par exemple un ingrédient, qu'on décore avec d'autre ingrédients, et l'ensemble des couches nous aurait donné le prix total à la fin
Mais nous avons estimé que c'était trop compliqué pour notre problème de fabrication de cookie. Étant donné qu'ils sont de même type, il suffisait seulement de passer au constructeur du cookie une liste d'ingrédients (ou Addons comme on les appelle dans le pattern Decorator) et pour calculer le prix, on itère sur la liste en additionnant le prix de chaque ingrédient et nous avons notre prix final. Si les ingrédients (decorators) étaient significativement différent (par exemple la fonction de prix ou autres..), l'utilisation des décorateurs ici aurait été justifiée, mais dans notre cas, nous avons seulement des propriétés qui diffère, donc utiliser des décorateurs n'est pas justifiée.

Le patron Observer : pour le menu

Le patron de conception Observer aurait été intéressant au niveau du stock, qui aurait pu être observable, et le menu observer. C'est à dire que à chaque fois que nous avons un ingrédient s'épuise, ou qui devient disponible, le menu sera prévenu, et on aura en temps réel la liste des recettes possibles.
Nous n'avons pas retenu ce patron parce que c'est trop lourd pour peu d'intérêt, le seul observeur possible dans notre spec aurait été le menu, nous n'avions pas d'autre systèmes qui devait observer notre stock, et nous n'avons pas souhaité faire de l'over-engineering.

Le patron Etat : pour le paiement

Le patron de conception État est similaire au patron Stratégie. Nous avons un comportement différent selon l'état ou on se trouve. Cette idée aurait été une bonne idée pour les paiement. ça nous aurait ouvert la possibilité d'avoir une méthode payer() différente selon l'état (ou autrement dit, le moyen de paiement) avec lequel on se trouve. Mais le choix n'a pas été retenu parce qu'à la différence de Stratégie, l'état est contenu dans un contexte comme une variable d'instance alors que nous avons pas vraiment besoin d'un contexte parce que nous n'avons pas de variables à stocker, et que le patron Stratégie est passé directement en argument à la méthode et, grâce au polymorphisme, on appelle la méthode nécessaire sans se soucier du type concret de l'objet passé.

Rétrospective

La partie conception

Ce que nous avons retenu de notre conception, c'est qu'il y a plusieurs façons de faire la même chose, sauf que toujours, ce qu'on gagne d'une façon, on le perd de l'autre.

Le piège que nous avons eu c'est que nous avons mis beaucoup de temps pour faire la conception, nous avons essayé d'être super précis, alors qu'il ne fallait pas. L'exercice était de faire abstraction sur les détails et concevoir l'essentiel à partir des concepts métiers.

Donc pour une prochaine fois, nous essayons de passer moins de temps sur la conception. On essayera de prendre une décision rapide, et ainsi, découvrir les détails cachés à fur et à mesure de l'avancement.

La partie qualité logicielle

Tests unitaires

Faire les tests unitaires c'est une procédure qui permet de vérifier le bon fonctionnement des petites composantes/modules indépendamment du reste du programme, ça permet de détecter les erreurs et les localiser rapidement.
Nous avons utilisé pour nos tests unitaires le framework JUnit4.
Durant notre phase de production, chaque fonction avant qu'elle soit mis sur notre dépôt, est testée. ça nous a permis de voir lors des modifications d'un programme les éventuelles régressions (donc on doit soit réécrire le code pour correspondre aux nouvelles attentes, soit corriger les erreurs du code)
Les tests unitaires garantissent qu'on développe un produit **vérifié** (**V**&V)

Dans notre projet, les tests unitaires couvrent une grande partie de notre code, mais on estime qu'il faut en faire plus

Tests fonctionnelles - Approche *Behaviour Driven Development*

L'adoption de l'approche BDD était bénéfique au projet, c'est une méthode agile tournée vers la valeur que nous produisons pour le client et la compréhension et le respect de ses exigences. Les fonctionnalités sont montrés au client via des scénarios compréhensible par tout les parties prenantes (des scénarios en langage naturel)
Au niveau technique, nous avons utilisé le framework cucumber avec gherkin qui est un langage utilisé pour écrire les scénarios et étapes (steps)

Le but est de créer ensemble un produit qui répond exactement aux attentes du client et avoir un produit **valide** (V&**V**)

L'avantage était le fait qu'on pouvait exprimer certain tests fonctionnelles avec un langage naturelle, ce qui implique la participation de tous les parties prenantes, et de montrer le bon fonctionnement de chaque fonctionnalité de bout en bout.

L'inconvénient était plus au niveau technique ou nous avons fait beaucoup d'efforts pour créer des scénarios réutilisables (paramétrables), mais ça ne marchait que par fichier feature et stepdef, nous avons eu des difficultés pour factoriser certains steps qui étaient communes à différents scénarios de différents features, certains solutions se sont présenté comme l'utilisation des singletons pour faire un contexte de test mais c'est le mal (notamment si on souhaite paralléliser les tests, donc le contexte sera écrit et lu dans n'importe quel ordre..)

Les tests fonctionnelles semblaient parfois être *overkill* en plus des tests unitaires, mais la différence c'est que les tests fonctionnelles couvraient de bout en bout les fonctionnalités et étaient exprimés en langage naturel.

L'erreur que nous avons commis dans cette partie c'est que nous avons écrit les scénarios tardivement, alors qu'il fallait écrire les scénarios en langage naturel dès le début pour le faire valider par le client.

Répartition du travail

Au niveau du travail, nous mettons 100 points à chaque membre du groupe.
En effet, nous avons réussi à mettre en place des réunions pour répartir les tâches et débloquer certains situations, nous étions tous présents et impliqués dans le projet. Certains avaient plus de facilités pour le code que d'autre mais d'autre ont fait plus d'efforts sur la partie fonctionnelle du projet.