MGL7460 Énoncé du TP 2

Version 0.5

Sommaire

Le but de ce TP est de:

1. Implanter en Java le code "serveur" d'une mini boutique en ligne, partant d'une description textuelle des fonctionnalités souhaitées, et d'un ensemble d'interfaces Java fournies par le professeur.
2. Développer vos propres classes de test en JUnit en utilisant les bonnes pratiques, et en évitant les mauvaises, concernant les tests unitaires
3. Pratiquer les "pipelines" de build dans l'environnement gitlab
4. Pratiquer l'analyse statique de code en utilisant l'outil SonarQube.

Le travail sera évalué sur la base de:

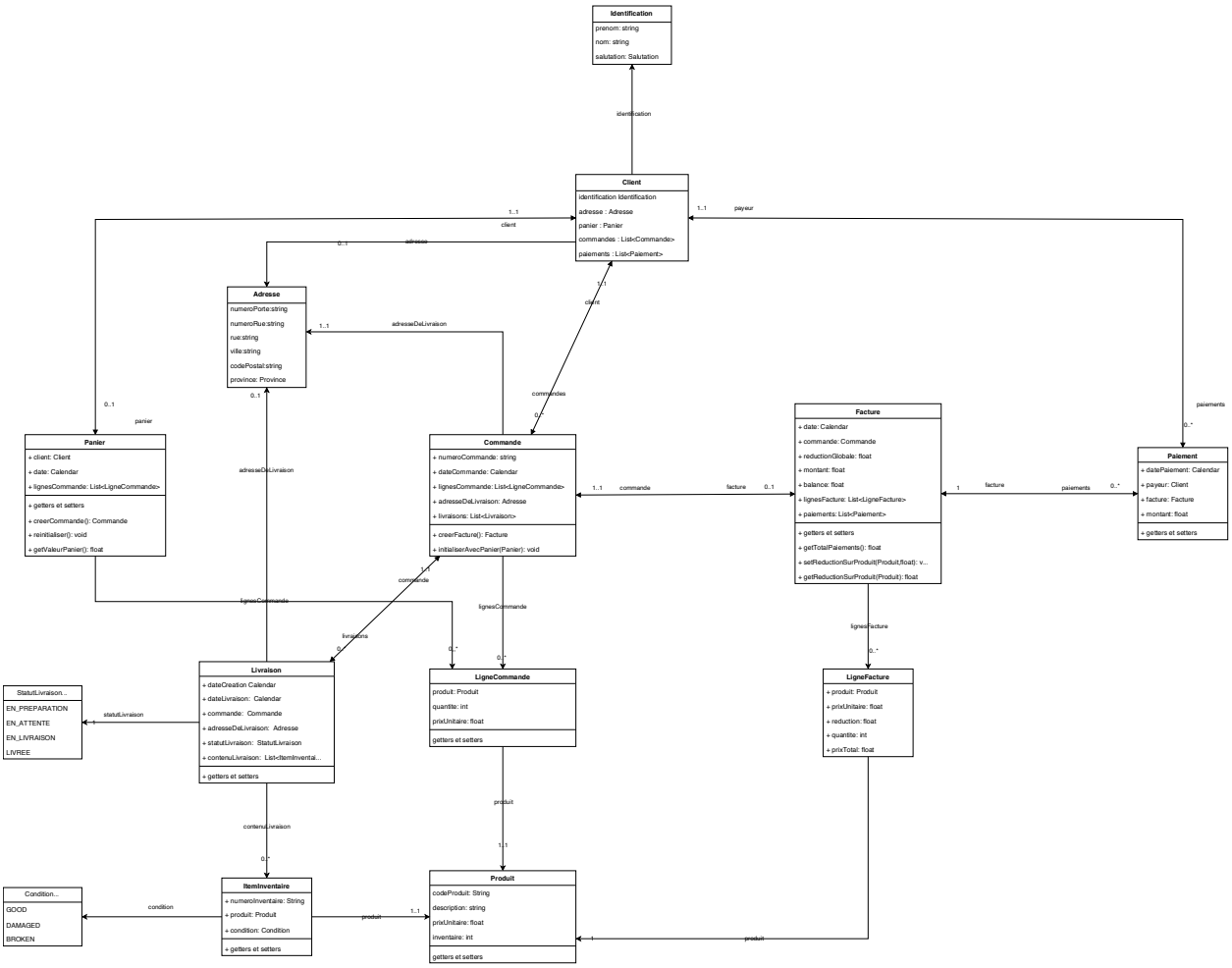
1. Une implantation des classes métiers qui "passe" une batterie de tests développée par le professeur, mais qui ne vous sera pas communiquée
2. La qualité de vos classes de test selon les bonnes pratiques et les tests smells couverts en cours
3. Le succès de la pipeline
4. Le résultat de l'exécution de SonarQube sur votre code
5. Un mini-rapport écrit

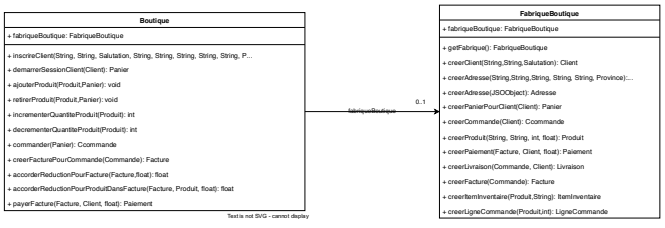
Objectif technique de l'exercice

Le but de l'exercice est de développer le back-end d'une simple boutique en-ligne. Le modèle objet est montré ci-bas.

En haut du modèle, vous avez les classes métiers, et en bas, vous avez deux classes :

1. **FabriqueBoutique** : c'est une classe qui implante plus ou moins fidèlement le patron *Abstract Factory* : cette classe contient une méthode de création par classe métier, **à l'exception de** : a) les classes métiers du type énumération (par ex., **StatutLivraison**, **Condition**, **Salutation**), et b) les classes métiers du type « record » de Java, qui correspondent à des 'struct' de C/C++, avant juste des champs de données, sans méthodes (par ex. **Identification**, **LigneFacture**)
2. **Boutique** : c'est une classe qui implante plus ou moins fidèlement le patron *Facade*. Vous y retrouverez les méthodes qui corresponderaient à des *routes*, dans le cadre d'applications web. On peut les imaginer comme des « call-backs » à différents boutons du type « submit », du côté « front-end »





Voici une vue sommaire de l’application :

1. Au premier usage de la boutique, le client s’inscrit en remplissant un formulaire. En appuyant sur le piton (« submit » ou « register » ou peu importe), on appelle la méthode « `inscrireClient(String prenom, String nom, Salutation salutation, String numeroPorte, String numeroRue, String nomRue, String ville, String codePostal, Province province)` » qui retourne un nouvel objet **`Client`**.
2. Étant donné un client, fraichement inscrit, ou proprement authentifié, on peut démarrer une séance de magasinage dans notre boutique. Ceci correspond à la méthode

« `demarrerSessionClient(Client cl) : Panier` » qui, concrètement, crée un panier vide associé au client.

3. Durant son magasinage, la cliente va faire des recherches sur les produits, et à l'occasion, ajouter un produit au panier. Cette « opération coté client » va « déclencher coté serveur » l'appel à la méthode « `ajouterProduit(Produit prod, Panier pan) : void` » « coté serveur ». Ceci va rajouter au panier une « ligne » correspondant au produit, avec une quantité de 1.
4. Le ou la cliente va pouvoir modifier la quantité en incrémentant ou décrémentant la quantité, au besoin, résultant en l'appel des méthodes « `incrementerQuantiteProduit(Produit prod, Panier pan) : int` » et « `decrementerQuantiteProduit(Produit prod, Panier pan) : int` », respectivement, qui retourne le nouveau nombre d'exemplaires du produit dans le panier.
5. Le ou la cliente peut aussi retirer un produit du panier coté client, ce qui va provoquer l'appel de la méthode « `retirerProduit(Produit prod, Panier pan) : void` » « coté serveur ».
6. Quand la ou le client est satisfait.e, elle appuie sur le piton 'commander' qui va créer une commande à partir du contenu actuel du panier. Cela est fait en appelant (coté serveur) la méthode « `commander(Panier pan) : Commande` », qui crée la commande correspondante, avec des lignes de commandes correspondant au contenu du panier, et pour le client « propriétaire » du panier. Pour l'adresse de livraison de la commande, on utilise l'adresse du client de la commande. Dans la vraie vie, on devrait être capable de spécifier un.e bénéficiaire/destinataire de la commande différent.e de la personne qui a commandé, et de spécifier une autre adresse de livraison. Ici, on va supposer que la personne destinataire/bénéficiaire est toujours le client qui a passé la commande. La classe Commande permet de modifier l'adresse de livraison (voir méthode « `void setAdresseDeLivraison(Adresse ad)` » de **`Commande`**), mais on n'offrira pas cette possibilité au niveau de notre classe **`Boutique`**

Dans la vraie vie, le processus de magasinage devrait se terminer ici, et une « application séparée » permettrait de créer les facteurs, accorder des réductions, et effectuer des paiements. Mais pour simplifier, pour les fins de cet exercice, c'est la même classe **`Boutique`** qui assure ces fonctions-là :

1. La méthode « `creerFacturePourCommande(Commande com) : Facture` » permet de créer une facture pour la commande. La facture va contenir les détails, dont l'identification de chaque produit commandé, son prix unitaire (prix du « catalogue produit »), sa quantité, le pourcentage de réduction pour ce produit, le cas échéant, et le cout total du produit. La somme des coûts totaux des différents produits peut elle-même être assujettie à une « réduction globale » sur le montant total de la facture—le cas échéant. Quand on créer une Facture à partir d'une Commande, il n'y a pas de réduction, ni pour les produits individuels (on utilise les prix de liste), ni pour la somme des coûts des produits (pas de réduction globale).
2. L'usager (agent.e de service à la clientèle?) peut accorder une réduction globale sur une facture. Ceci est fait en appelant la méthode

« `accorderReductionPourFacture(Facture fact, float reduction) : float` », qui modifie la reduction globale, et retourne le nouveau montant de la **`Facture`**.

3. L'usager peut aussi accorder une réduction sur une *ligne spécifique* de la Facture, correspondant à un produit spécifique, en appelant la méthode « `accorderReductionPourProduitDansFacture(Facture facture, Produit produit, float reduction) : float` », qui va modifier la **`LigneFacture`** correspondante, et retourner le nouveau montant de la **`Facture`**.
4. L'usager peut enregistrer un paiement sur uneFacture. Cela est fait grâce à la méthode « `payerFacture(Facture facture, Client payeur, float montant) : Paiement` », qui va créer un objet Paiement indiquant un paiement fait par « payeur » (qui peut être différent.e de la personne cliente qui a passé la commande de la facture, par exemple, je peux payer la facture d'une commande faite par mon fils).

Voilà, en ce qui concerne la classe **`Boutique`**.

Comme pour le TP1, vous allez partir d'un ensemble d'interfaces Java représentant les entités en question, et votre travail de développement va consister à :

1. Créer et coder les classes d'implémentation correspondantes en Java
2. Créer les classes de tests unitaires correspondantes.

Notez que :

1. Au niveau du modèle de classe UML (voir plus haut), plutôt que représenter ces interfaces comme des « interfaces UML », i.e. avec juste des méthodes et pas d'attributs, je les ai représentées par des vraies classes où j'ai mis les « attributs sous-entendus » aux méthodes getters/setters présentes dans l'interface Java, comme attributs, et les autres méthodes « non-triviales » dans la section méthodes. C'est pour cela que la plupart des classes ont dans la partie méthode « getters et setters ».
2. Votre source de vérité, évidemment, est le code (les interfaces) Java, et non le modèle de classes ci-haut.
3. Comme pour le TP1, je vous garantis que les interfaces sont « workables », puisque ce que je vous remets comme interfaces Java consiste en un projet fonctionnel duquel j'ai retiré les classes d'implantation.
    1) Je fais de mon mieux pour m'assurer que la documentation des interfaces est « complete and accurate »
    2) Il se peut que j'en échappe, alors n'hésitez pas à poser des questions
4. **Concernant les tests unitaires : n**otez que la classe **`Boutique`** n'exerce pas (ne couvre pas) *toutes* les fonctionnalités offertes par les interfaces Java que je vais vous fournir. Donc :
    1) Il ne suffit pas de tester juste les méthodes de la classe **`Boutique`**
    2) Vous devez tester les principales fonctionnalités du code développé, y compris des méthodes non appelées à partir de la classe **`Boutique`**

Vos tâches

Si vous les acceptez, consisteront en :

1) Réaliser les classes d'implantation correspondant aux interfaces Java fournies.
2) Développer vos classes de tests unitaires
3) Mettre en place une pipeline dans gitlab pour compiler et tester le code
4) Faire vérifier votre code par SonarQube et apporter les correctifs nécessaires
5) Rédiger un cours rapport, selon gabarit fourni par le professeur, à inclure dans votre dépôt gitlab.

Barème

Critère	Poids
Du code qui passe (réussit) vos tests en Java	20%
Du code qui passe (réussit) mes tests	20%
Une pipeline gitlab fonctionnelle	15%
Qualité de codage et documentation (selon professeur)	15%
Qualité de codage selon SonarQube	20%
Qualité du rapport écrit	10%