Guide d’étapes clés : Testez l'implémentation d'une nouvelle fonctionnalité Java

Comment utiliser ce document ?

Ce guide vous propose un découpage du projet en étapes. Vous pouvez suivre ces étapes selon vos besoins. Dans chacune, vous trouverez :

- des recommandations pour compléter la mission ;
- les points de vigilance à garder en tête ;
- une estimation de votre avancement sur l’ensemble du projet (attention, celui-ci peut varier d’un apprenant à l’autre).

Suivre ce guide vous permettra ainsi :

- d’organiser votre temps ;
- de gagner en autonomie ;
- d’utiliser les cours et ressources de façon efficace ;
- de mobiliser une méthodologie professionnelle que vous pourrez réutiliser.

Gardez en tête que votre progression sur les étapes n’est qu’une estimation, et sera différente selon votre vitesse de progression.

Soyez conscient que ce guide d’étapes clés est **obligatoire à lire**. Ce guide contient les informations essentielles sur la gestion de projet (par exemple, les user stories) dont vous avez besoin pour mener à bien le projet. Cependant, vous ne devez pas suivre exactement toutes les étapes.

Étape 1 : Initialisez votre projet GitHub

10 % de progression

Une fois cette étape réalisée :

- vous aurez mis en place le versionning avec Git.

Recommandations :

- Forker le repository https://github.com/OpenClassrooms-Student-Center/parkingsystem en un repository public sur votre compte GitHub nommé <nom-prenom-tester-Java>
- Cloner ce nouveau repository sur votre poste de travail en local.
- Créez et pushez une nouvelle branche
    - Créez une nouvelle branche de dev avec `git checkout -b dev`
    - Faites un push avec `git push origin dev`
- Vous travaillerez sur cette branche tout au long du projet, pensez à faire un commit à chaque étape. En fin de projet, il faudra fusionner cette branche sur la branche `main`.

Ressources utiles :

- Cours OpenClassrooms : Gérez du code avec git et github
- Cours OpenClassrooms : Apprenez à utiliser la ligne de commande dans un terminal
- GitHub Learning Lab : tutoriels officiels pour GitHub.

Étape 2 : Résolvez le bug existant sur le calcul du prix

20 % de progression

Une fois cette étape réalisée :

- vos tests unitaires existants s'exécuteront sans erreur.

Vous allez commencer par corriger le code pour que les tests qui échouent actuellement s'exécutent avec succès. Vous pouvez exécuter les tests avec la commande `mvn test`.

Voici les informations communiquées par l'équipe dans l'outil de gestion de projet :

> *Le système dysfonctionne lorsque les voitures sont garées dans le garage depuis plus de 24 heures : on obtient des durées négatives.*

> *Par exemple : pour une voiture qui arrive à 10 heures et repart le lendemain à 9 heures, la durée indiquée par le système est de -1h. Il faut résoudre ce problème.*

Recommandations :

- Exécutez les tests de la classe FareCalculatorServiceTest du package com.parkit.parkingsystem dans src/test/java, en lançant les commandes suivantes :
  cd parkingsystem
  mvn test
- Constatez l'échec des tests.
- Analysez la classe FareCalculatorServiceTest du package com.parkit.parkingsystem dans src/test/java et la classe FareCalculatorService du package com.parkit.parkingsystem dans src/main/java pour comprendre d'où vient le bug.
- Pour corriger le bug, une solution est d'utiliser l'heure d'entrée en millisecondes grâce à ticket.getInTime().getTime() et de faire la même chose sur la sortie pour calculer la durée précise.
- Exécutez à nouveau les tests et confirmez que le bug est bien résolu.

Points de vigilance :

- `ticket.getInTime().getTime()` renvoie des millisecondes, attention à la convertion en minutes pour permettre aux tests de fonctionner.

Étape 3 : Implémentez la fonctionnalité des 30 minutes gratuites

35 % de progression

Une fois cette étape réalisée :

- le parking sera gratuit pour des durées de stationnement inférieures à 30 minutes ;
- vous aurez implémenté les tests unitaires pour couvrir cette fonctionnalité.

Le calcul du tarif étant désormais opérationnel, vous pouvez implémenter de nouvelles fonctionnalités. La user story suivante correspond à votre prochaine tâche :

> ***User Story*** *: En tant qu'utilisateur, je veux pouvoir me garer pour une courte durée sans avoir à payer.*
>
> ***Description*** *: Dans ce scénario d'utilisation, l'utilisateur entre dans le parking après avoir renseigné sa plaque d'immatriculation. En sortant, il la renseigne à nouveau. S'il est resté moins de 30 minutes, les frais de parking devraient être gratuits (0$).*

Recommandations :

- Vous allez suivre la méthode du **TDD (Test Driven Development)** en implémentant d'abord les tests unitaires puis le code.
- Rédigez les tests unitaires pour la fonctionnalité
  - Écrivez un test unitaire vérifiant la user story dans le cas d'une voiture :
    Nom de la méthode de test : `calculateFareCarWithLessThan30minutesParkingTime`
    Description du test : ce test doit appeler la méthode `calculateFare` avec un ticket concernant une voiture, datant de moins de 30 minutes et vérifier que le prix calculé est égal à 0.
  - Écrivez un test unitaire vérifiant la user story dans le cas d'une moto.
    Nom de la méthode de test : `calculateFareBikeWithLessThan30minutesParkingTime`
    Description du test : ce test doit appeler la méthode `calculateFare` avec un ticket concernant une moto, datant de moins de 30 minutes et vérifier que le prix calculé est égal à 0.
  - Constatez que les 2 tests échouent.
- Développez la fonctionnalité
  - Mettez à jour la méthode `calculateFare` de la classe `FareCalculatorService` pour retourner un prix à 0 si la durée dans le parking est inférieure à 30 minutes.
  - Constatez que les 2 tests réussissent.

Points de vigilance :

- Veillez à ce que la “duration” soit à la bonne échelle de temps pour votre test.

Étape 4 : Développez la fonctionnalité des 5 % de remise

55 % de progression

Une fois cette étape réalisée :

- vous aurez implémenté la fonctionnalité des 5% de remise pour les utilisateurs récurrents ;
- ainsi que les tests unitaires pour couvrir cette fonctionnalité.

Cette fonctionnalité est un peu plus complexe que la précédente. Prenez bien connaissance des informations fournies dans la User Story :

> ***User Story*** *: En tant qu’utilisateur, je veux obtenir une réduction quand j’utilise le parking régulièrement.*
>
> ***Description*** *:*
>
> 1. *Quand un utilisateur entre dans le garage, il renseigne sa plaque d’immatriculation.*
> 2. *Le système vérifie si cette plaque d’immatriculation a déjà été utilisée.*
> 3. *Si c’est le cas, le système renvoie le message suivant : "Heureux de vous revoir ! En tant qu’utilisateur régulier de notre parking, vous allez obtenir une remise de 5%", puis le système reprend son fonctionnement habituel.*
> 4. *Quand l’utilisateur sort du parking, il bénéficie d’une réduction de 5% par rapport au tarif normal.*

Recommandations :

- D'abord, rédigez les tests unitaires pour la fonctionnalité.
    - Tout comme pour l'étape précédente, rédigez d'abord les tests vérifiant qu'un véhicule muni d'un ticket de réduction paiera bien 95% du tarif plein.
    - Écrivez un test unitaire le vérifiant dans le cas d'une voiture.
      Nom de la méthode de test : `calculateFareCarWithDiscount`
      Description du test : ce test doit appeler la méthode `calculateFare` avec un ticket concernant une voiture et avec le paramètre discount à true, puis vérifier que le prix calculé est est bien de 95% du tarif plein. La durée du ticket doit être de plus de 30 minutes.
    - Écrivez un test unitaire le vérifiant dans le cas d'une moto.
      Nom de la méthode de test : `calculateFareBikeWithDiscount`
      Description du test : ce test doit appeler la méthode `calculateFare` avec un ticket concernant une moto et avec le paramètre discount à true, puis vérifier que le prix calculé est est bien de 95% du tarif plein. La durée du ticket doit être de plus de 30 minutes.
    - Constatez que ces 2 tests échouent.
- Puis, développez la fonctionnalité.
    - Pour coder la fonctionnalité, il va d'abord falloir appliquer la réduction lorsqu'un ticket de discount est présent. Puis il faudra définir à quel moment un ticket discount doit être donné à la voiture, en comptant le nombre de fois où la voiture est déjà passée.
    - Appliquez la réduction aux véhicules munis d'un ticket discount
    - Modifiez la méthode existante `public void calculateFare(Ticket ticket)` de la classe `FareCalculatorService` pour lui ajouter un paramètre de type booléen ce qui donnera `public void calculateFare(Ticket ticket, boolean discount)` puis implémentez la réduction de 5% si le paramètre discount vaut true.
    - Ajoutez une nouvelle méthode `public void calculateFare(Ticket ticket)` qui appellera calculateFare(Ticket ticket, boolean discount) avec le paramètre discount à false.
    - Constatez que vos 2 tests s'exécutent avec succès.
- Enfin, donnez un ticket de discount aux véhicules récurrents.
    - Ajoutez une nouvelle méthode `getNbTicket` à la classe TicketDAO pour compter combien de tickets sont enregistrés pour un véhicule.
    - Modifiez la méthode `processIncomingVehicle` de la classe ParkingService pour afficher le message de bienvenue. Puis modifiez la méthode `processExitingVehicle` de la même classe

pour appeler la méthode `calculateFare` avec un paramètre discount à true si ce n'est pas le premier passage (utilisez `getNbTicket` pour le savoir).

Points de vigilance :

- La méthode calculateFare (à 1 paramètre) appelle la méthode calculateFare (à 2 paramètres) avec le booleen à true au lieu de false.
- Attention : ce code sera testé lors des étapes 5 et 6.

Étape 5 : Testez unitairement la classe ParkingService grâce aux Mocks

75 % de progression

Une fois cette étape réalisée :

- vous aurez une couverture de code supérieure à 90 % (instructions) pour la classe ParkingService.

À cette étape, le coverage global des instructions de tout le projet devrait être entre 65 et 70%, ce qui n'est pas encore suffisant. De plus, un rapide tour d'horizon de la classe de test ParkingServiceTest nous permet de constater que la classe ParkingService est faiblement testée unitairement. La classe ParkingService fait appel à d'autres classes comme TicketDAO, ParkingSpotDAO qui requierent une base de données ou encore InputReaderUtil qui requiert une interaction avec l'utilisateur. Pour tester unitairement cette classe, il est donc nécessaire de mocker ces appels.

L'objectif de cette étape est donc de coder des tests unitaires grâce aux mocks (données de tests) pour la classe ParkingService.

Recommandations :

- D'abord, faites un état des lieux.
    - Observez la couverture de code de la classe ParkingService :
        - Exécutez les tests avec la commande `mvn verify`
        - Consultez le rapport Jacoco : Rapport Jacoco
- Puis, complétez le test de sortie d'un véhicule.
    - Complétez le test existant : `processExitingVehicleTest`

    - Ce test doit également mocker l'appel à la méthode `getNbTicket()` implémentée lors de l'étape précédente.
- Implémentez les 5 nouveaux tests
    1. `testProcessIncomingVehicle` : test de l'appel de la méthode `processIncomingVehicle()` où tout se déroule comme attendu.
    2. `processExitingVehicleTestUnableUpdate` : exécution du test dans le cas où la méthode `updateTicket()` de ticketDAO renvoie false lors de l'appel de `processExitingVehicle()`
    3. `testGetNextParkingNumberIfAvailable` : test de l'appel de la méthode `getNextParkingNumberIfAvailable()` avec pour résultat l'obtention d'un spot dont l'ID est 1 et qui est disponible.
    4. `testGetNextParkingNumberIfAvailableParkingNumberNotFound` : test de l'appel de la méthode `getNextParkingNumberIfAvailable()` avec pour résultat aucun spot disponible (la méthode renvoie null).
    5. `testGetNextParkingNumberIfAvailableParkingNumberWrongArgument` : test de l'appel de la méthode `getNextParkingNumberIfAvailable()` avec pour résultat aucun spot (la méthode renvoie null) car l'argument saisi par l'utilisateur concernant le type de véhicule est erroné (par exemple, l'utilisateur a saisi 3).

- Enfin, générez le rapport Jacoco de la classe ParkingService
    - Assurez-vous, via le rapport Jacoco, d'avoir une couverture de test supérieure à 90% (missed instructions) sur la classe `ParkingService` pour vous assurer de la bonne implémentation des fonctionnalités.
    - Si ce n'est pas le cas, c'est sûrement qu'une partie de votre code mérite d'être mieux couverte en tests.

Étape 6 : Testez la classe ParkingService avec la base de données (test d'intégration)

100 % de progression

Une fois cette étape réalisée :

- vous aurez une couverture de code globale des instructions de 70% minimum.

Vous pouvez être satisfait des tests unitaires que vous avez créés. La dernière étape est d'implémenter les tests d'intégration. Ceux-ci vont vous permettre de vérifier que toutes les briques de votre code collaborent bien ensemble, y compris lors des interactions avec la base de données.

Le projet possède déjà la configuration nécessaire pour vous permettre d'exécuter des tests qui feront appel à une base de données de test. La classe de test `ParkingDataBaseIT` est prête à l'usage.

Voici les informations communiquées par votre équipe à ce sujet :

> *Il y a des tests d'intégration que je n'ai pas eu le temps de terminer. Je les ai marqués dans le code avec des "TODO" en commentaires. Il faudra bien compléter ces tests.*

Recommandations :

- D'abord, faites un état des lieux en exécutant les tests et en générant le rapport Jacoco avec `mvn verify`.
- Implémentez les TODOs de la classe de test `ParkingDatabaseIT` présents dans les méthodes `testParkingACar` et `testParkingLotExit`. Il s'agit d'implémenter les asserts nécessaires.
- Ajoutez un nouveau test d'intégration pour la fonctionnalité de remise de 5% : `testParkingLotExitRecurringUser`. Il doit tester le calcul du prix d'un ticket via l'appel de `processIncomingVehicle` et `processExitingVehicle` dans le cas d'un utilisateur récurrent.
- Assurez-vous, via le rapport Jacoco, d'avoir une couverture de test globale supérieure à 70%.

Point de vigilance :

- `mvn test` n'exécute que les tests unitaires. Pour les tests d'intégration, il faut utiliser `mvn verify`.

Projet terminé !