
Application de voyage

Projet de test logiciel



Jules Braun, Allan Denoce et Lucie Perrin

17/01/2025

Table des matières

1	Introduction	1
2	Présentation de l'application	1
3	Choix d'implémentation	2
4	Manuel d'utilisation	3
5	Tests unitaires	3
6	Tests d'intégrations	3
7	Analyse de la couverture du code	3
8	Analyse par mutation	4
9	Conclusion	5

1 Introduction

Ce projet consistait à développer une application permettant à ses utilisateurs de planifier un séjour d'une certaine durée à une destination donnée incluant transports, hôtels et activités à proximité. L'utilisateur spécifie ses préférences et ses conditions puis l'application retourne un ou plusieurs forfaits correspondant aux données rentrées par l'utilisateur.

L'objectif de ce projet, en plus de réaliser cette application, était de réaliser des tests utilisant les différentes méthodes vu durant notre formation pour assurer la fiabilité de notre implémentation vis à vis des différentes exigences.

Au niveau de l'organisation, chaque membre du groupe a développé des parties différentes du projet mais c'est un membre différent de l'équipe qui s'est occupé de tester le code produit.

2 Présentation de l'application

Pour notre application, nous avons décidé de stocker les données dans des fichiers JSON et de même pour le résultat renseignant le voyage. L'utilisateur de l'application a donc la possibilité de consulter le fichier JSON de résultat et il lui est aussi possible de visualiser directement le résultat de sa requête en ligne de commande. En effet, notre application recueille les vœux de l'utilisateur en l'interrogeant en ligne de commande grâce à l'outil "Scanner" de Java puis affiche les propositions de voyages correspondant avec les éventuelles erreurs rencontrées. Les choix renseignés par l'utilisateur sont stockés dans les classes UserPreferences (le type de transport souhaité, le nombre minimal d'étoiles souhaitées pour l'hôtel et les critères de préférence pour le choix du transport, de l'hôtel et des activités) et TravelRequirements (les villes et les dates de départ et d'arrivée, la taille du rayon autour de l'hôtel pour les activités et le budget total).

En ce qui concerne la structure de notre application, nous avons développé, dans un premier temps, trois services essentiels : CorrespondingHotels, CorrespondingTransports et CorrespondingActivities. Ceux-ci permettent de récupérer les données stockées dans les fichiers JSON et de sélectionner les trajets, hôtels et activités qui correspondent à la demande du client. Les objets stockés sont des objets de classe Hotel, Transport ou Activity qui ont pour seul but de représenter les informations nécessaires. Ces services contiennent deux méthodes principales : l'une qui liste tous les transports (respectivement les hôtels et les activités) et l'autre qui liste tous les transports (respectivement les hôtels et les activités) qui correspondent au séjour voulu mais aussi aux préférences de l'utilisateur. La méthode listant tous les transports parse le fichier JSON correspondant en utilisant le FileManager en attribut de la classe.

Le service FileManager permet de récupérer les données d'un fichier JSON et de les stocker dans le type d'objet passé en paramètre. Il permet également d'écrire le fichier JSON correspondant aux propositions de voyages. Pour trouver les activités qui se trouvent dans un certain rayon, on utilise le service CoordinatesManager qui fait appel à l'api GeoCode pour récupérer les coordonnées de l'hôtel et des activités.

Les propositions de voyages sont quant à elles réalisées par le service CompleteTravel. Celui-ci fait appel aux trois services précédents pour récupérer toutes les propositions de transports, d'hôtels et d'activités et les regroupe dans un objet Travel. Cette classe permet de gérer le budget en le décrémentant au fur et à mesure des choix effectués.

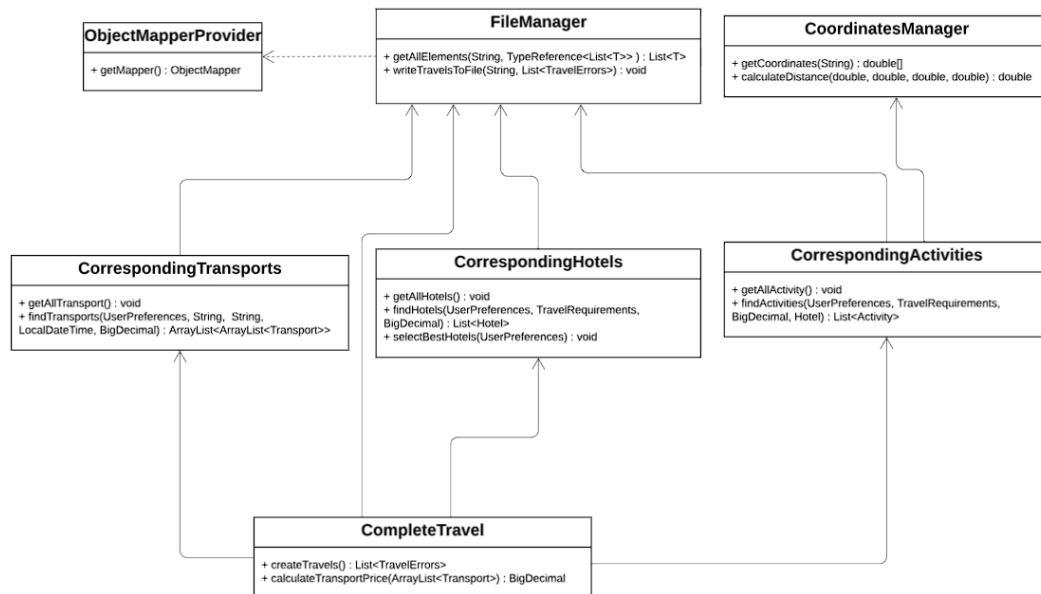


FIGURE 1 – Diagramme de classe de l'application

La figure 1 représente le diagramme de classe correspondant aux classes principales de notre code, sans les classes de données, afin de voir les interactions entre elles.

3 Choix d'implémentation

Au niveau de la création du Travel, il fallait faire un choix afin de s'occuper de ne pas dépasser le budget tout en choisissant un trajet possible. Pour la décision, le transport aller et retour est sélectionné en premier, s'il existe des transports respectant le budget, alors un hôtel va être cherché avec le budget restant. Un voyage peut ne pas contenir de transports ou d'hôtel ou d'activités ou les trois, mais sera quand même renvoyé. Vient ensuite le choix des activités avec le reste du budget. Le maximum d'activités est proposé, en tenant compte des préférences, de la distance des activités par rapport à l'hôtel et du budget restant.

Une erreur est renvoyée en cas de voyage non conforme et l'utilisateur est amené à changer ses valeurs d'entrée. La classe `TravelErrors` nous permet de stocker ces erreurs en ajoutant un champ spécifiant les erreurs en plus du champ `Travel`.

Chaque classe *Corresponding* s'occupe de gérer son propre budget ce qui évite la gestion de grosses structure de données dans **CompleteTravel** en renvoyant uniquement les éléments qui correspondent au voyage souhaité.

Pour les transports, il a été décidé de considérer que les transports qui avaient moins de 10 minutes d'écart pour faire l'escale ne sont pas choisis car le temps est trop court.

L'utilisateur est obligé de donner des préférences, s'il n'en choisit pas, alors elles sont mises par défaut.

4 Manuel d'utilisation

Afin de lancer l'application lorsque l'on est à la racine du projet, il faut suivre les étapes suivantes :

- Se placer dans le dossier projet et lancer la commande suivante : **mvn compile**
- Archiver le projet avec **mvn package**
- Exécuter l'archive avec **java -jar target/projet-1.0-SNAPSHOT-jar-with-dependencies.jar**

Pour lancer les tests il suffit de faire les commandes suivantes :

- Tests unitaire : **mvn clean test**
- Tests d'intégration : **mvn failsafe:integration-test** après avoir exécuter la commande pour les tests unitaires.

Pour analyser les différentes couverture, il faut faire les commandes suivantes :

- Pour voir le coverage : **mvn clean test jacoco:report**
- Pour voir les mutants : **mvn clean test org.pitest:pitest-maven:mutationCoverage**

5 Tests unitaires

Des tests unitaires ont été réalisés pour toutes les classes en dehors des classes de données, comme `Hotel.java`, qui sont composées exclusivement de getter et de setter.

Pour chacun des tests, nous avons suivi la règle AAA (**Arrange-Act-Assert**) consistant à d'abord initialiser les éléments nécessaires au test, avec des mocks lorsqu'il faut faire appel à des services externes à la classe comme les méthodes de nos classes `Manager`, puis à exécuter la méthode que nous testons et, enfin, à vérifier que les résultats obtenus correspondent aux spécifications.

6 Tests d'intégrations

Les tests d'intégration ont été effectués pour les activités, les hôtels et les transports. Les fichiers des tests d'intégration sont reconnaissables par le "IT" à la fin de leurs noms. Afin de réaliser les tests d'intégration, des fichiers tests en JSON ont été créés, tirés de data des vrais fichiers mais en quantité restreinte. Dans ces tests, nous vérifions le fonctionnement des interactions entre les databases et nos classes fournissant les informations nécessaires pour composer le voyage.

Il y a trois points qui ont été testés dans chacune de ces classes, si la classe récupère bien les informations de son fichier test, si la liste est vide venant d'un fichier vide et si la liste est vide en récupérant des données au mauvais format, par exemple des activités qui récupéré des hôtels.

7 Analyse de la couverture du code

A l'aide de l'outil `Jacoco`, nous avons généré des analyses de la couverture de notre code et, au fur et à mesure du projet, nous avons adapté nos tests afin d'avoir la meilleure couverture de

code possible (cf. figure 2 ci-dessous).




















Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes	
 projet	<div><div></div><div></div></div>	62%	<div><div></div><div></div></div>	71%	75	208	239	590	47	126	1	18	
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes	
 Main	<div><div></div><div></div></div>	0%	<div><div></div><div></div></div>	0%	27	27	143	143	2	2	1	1	
 CoordinatesManager	<div><div></div><div></div></div>	57%	<div><div></div><div></div></div>	83%	3	8	12	33	2	5	0	1	
 TravelRequirements	<div><div></div><div></div></div>	61%	<div><div></div><div></div></div>	n/a	7	15	14	30	7	15	0	1	
 UserPreferences	<div><div></div><div></div></div>	66%	<div><div></div><div></div></div>	50%	7	15	11	28	5	13	0	1	
 Travel	<div><div></div><div></div></div>	62%	<div><div></div><div></div></div>	n/a	5	11	10	22	5	11	0	1	
 Activity	<div><div></div><div></div></div>	62%	<div><div></div><div></div></div>	n/a	5	11	10	22	5	11	0	1	
 Hotel	<div><div></div><div></div></div>	62%	<div><div></div><div></div></div>	n/a	5	11	10	22	5	11	0	1	
 CompleteTravel	<div><div></div><div></div></div>	93%	<div><div></div><div></div></div>	100%	6	15	6	49	6	9	0	1	
 Transport	<div><div></div><div></div></div>	74%	<div><div></div><div></div></div>	n/a	4	13	8	26	4	13	0	1	
 FileManager	<div><div></div><div></div></div>	66%	<div><div></div><div></div></div>	100%	1	5	4	15	1	3	0	1	
 CorrespondingActivities	<div><div></div><div></div></div>	93%	<div><div></div><div></div></div>	100%	2	12	5	36	2	5	0	1	
 CorrespondingHotels	<div><div></div><div></div></div>	97%	<div><div></div><div></div></div>	100%	1	20	3	59	1	10	0	1	
 CorrespondingTransports	<div><div></div><div></div></div>	98%	<div><div></div><div></div></div>	100%	1	34	2	90	1	7	0	1	
 ObjectMapperProvider	<div><div></div><div></div></div>	80%	<div><div></div><div></div></div>	n/a	1	2	1	4	1	2	0	1	
 TravelErrors	<div><div></div><div></div></div>	100%	<div><div></div><div></div></div>	n/a	0	6	0	11	0	6	0	1	
 CorrespondingHotels.new TypeReference().{...}	<div><div></div><div></div></div>	100%	<div><div></div><div></div></div>	n/a	0	1	0	1	0	1	0	1	
 CorrespondingActivities.new TypeReference().{...}	<div><div></div><div></div></div>	100%	<div><div></div><div></div></div>	n/a	0	1	0	1	0	1	0	1	
 CorrespondingTransports.new TypeReference().{...}	<div><div></div><div></div></div>	100%	<div><div></div><div></div></div>	n/a	0	1	0	1	0	1	0	1	
Total		845 of 2,247		44 of 155	71%	75	208	239	590	47	126	1	18

FIGURE 2 – Couverture global du code

Pour la majorité des classes, seuls certains de nos getter et setter ne sont pas couverts par nos tests puisque ce sont des méthodes triviales pour lesquelles les tests n’apporteraient pas grand chose et puisqu’une grande partie de nos classes peuvent être entièrementinstanciées via leurs constructeurs.

Deux classes font exception :

- **CoordinatesManager** pour laquelle la méthode **CalculateDistance** n’a pas été testée : c’est une méthode qui prend la latitude et la longitude de deux points en paramètres et renvoie la distance entre ces deux points en suivant le modèle d’Aversine.
- **Main** pour laquelle nous avons fait le choix de réaliser des tests de recette puisqu’elle ne fait appel qu’à des méthodes des autres classes qui ont déjà été testées, à des **print** et à des **scanner**.

8 Analyse par mutation

Afin de s’assurer du bon fonctionnement de nos codes, on a décidé de perfectionner nos tests en analysant le rapport de mutations. Le but de ces tests est de vérifier la solidité du code avec de légers changements dans celui-ci, comme par exemple le changement des conditions dans des *if*.

Parmi les mutants qui ont pu être rencontrés, on retrouve notamment des conditions qui s’occupent de gérer le fait que le prix du billet de train est toujours moins cher que le budget du client. Comme les billets de trains sont bloqués en premier dans notre algorithme, si on utilise tout le budget dans le transport alors nous ne pourrions pas trouver d’hôtel pour loger durant la période du voyage. Les mutants se sont occupés de modifier la condition $<$ par la condition \leq . Il a donc fallu rajouter des tests afin de vérifier lorsque le prix du billet est égale au budget.

Il n'a malheureusement pas été possible de tester tous les mutants par manque de temps et ne trouvant pas de solution efficace pour les tuer comme on peut voir avec la figure 3.

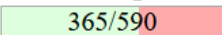
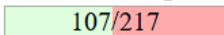
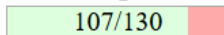
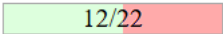
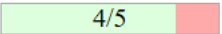
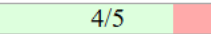
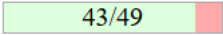
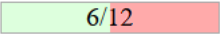
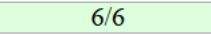
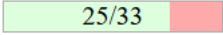
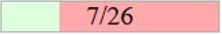
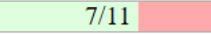
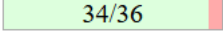
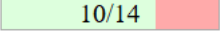
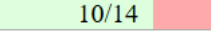
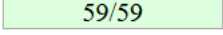
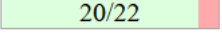
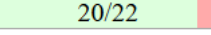
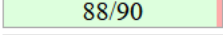
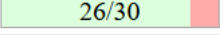
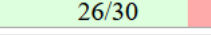
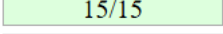
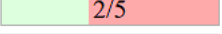
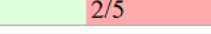
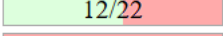
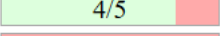
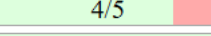
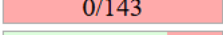
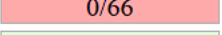
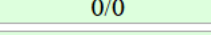
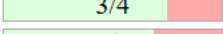
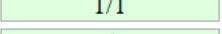
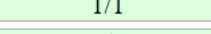
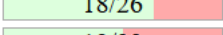
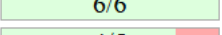
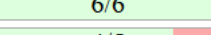
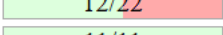
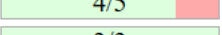
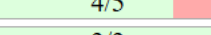
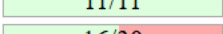
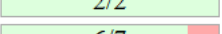
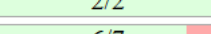
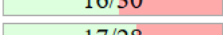
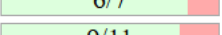
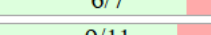
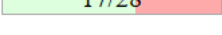
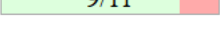
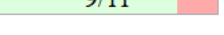
Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
projet	15	62% 	49% 	82% 
Breakdown by Class				
Name		Line Coverage	Mutation Coverage	Test Strength
Activity.java		55% 	80% 	80% 
CompleteTravel.java		88% 	50% 	100% 
CoordinatesManager.java		76% 	27% 	64% 
CorrespondingActivities.java		94% 	71% 	71% 
CorrespondingHotels.java		100% 	91% 	91% 
CorrespondingTransports.java		98% 	87% 	87% 
FileManager.java		100% 	40% 	40% 
Hotel.java		55% 	80% 	80% 
Main.java		0% 	0% 	100% 
ObjectMapperProvider.java		75% 	100% 	100% 
Transport.java		69% 	100% 	100% 
Travel.java		55% 	80% 	80% 
TravelErrors.java		100% 	100% 	100% 
TravelRequirements.java		53% 	86% 	86% 
UserPreferences.java		61% 	82% 	82% 

FIGURE 3 – Couverture global du code par les mutants

On peut voir que dans les classes *Corresponding*, la plupart des mutants ont été tués. Pour ce qui est de *CompleteTravel*, il n'y a que 50% des mutants qui ont été tués. Les mutants restants sont présents dans les getters et les setters. Pour ce qui est du *FileManager* et le *CoordinatesManager*, nous avons manqué de temps pour pouvoir tuer tout les mutants.

Les autres classes n'ont pas été testées car elles sont uniquement composées de getters et setters.

9 Conclusion

Nous avons réalisé une application de voyage fonctionnelle et correspondant aux exigences tout en ayant pris le temps de concevoir à notre manière les aspects "sous-spécifiés" du sujet. Nous n'avons pas eu le temps d'optimiser l'efficacité de nos tests vis à vis des mutants mais nos résultats sont tout de même plutôt bons puisque nos tests couvrent très largement notre code.

Nous avons trouvé le projet très intéressant car il permettait de se rendre compte de la réalisation complète d'une application en liant à la fois la conception, le développement et les tests. La charge de travail était correcte pour un projet à réaliser par groupe de trois puisque le projet permettait bien de se répartir le travail.