BORDEAUX INP Enseirb-Matmeca

I3GL – Test Du Logiciel

Projet Holitrip
Application de Planification de Séjours

Réalisé par :

Mohamed Dyae Chellaf

Mohamed Taha Sandi

Mohamed Reda El Khader

Mathieu Morel

Département Informatique

2025/2026

Table des matières

1 Architecture de l'application

1.1 Vue d'ensemble

L'application **Holitrip** permet de planifier des séjours complets incluant transports (directs ou avec correspondances), hôtels et activités selon les préférences utilisateur. L'architecture repose sur le principe de **séparation des préoccupations** avec une distinction claire entre interfaces et implémentations, facilitant l'injection de dépendances et les tests unitaires avec doublures.

1.2 Services et implémentations

Interface	**Implémentation**	**Responsabilité**
`TransportService`	`JsonTransportService`	Recherche de transports avec algorithme BFS pour correspondances (max 3 legs, 60min min., homogénéité mode)
`HotelService`	`JsonHotelService`	Recherche d'hôtels par ville, note minimale et prix
`ActivityService`	`JsonActivityService`	Recherche d'activités avec filtrage par catégorie et distance depuis l'hôtel
`GeocodingService`	`ApiGeocodingService`	Conversion adresses → coordonnées GPS via API geocode.maps.co
`DistanceService`	`HaversineDistanceService`	Calcul de distance à vol d'oiseau (formule d'Haversine)
`PackageService`	`PackageBuilder`	Assemblage des forfaits complets selon critères utilisateur

TABLE 1 – Services de l'application

1.3 Modèles de données

Les données métier sont dans le package `fr.univ.holitrip.model` :

— `Transport` : trajet unitaire (villes, dates, mode TRAIN/PLANE, prix)
— `Trip` : voyage complet (liste de transports pour gérer les correspondances)
— `Hotel` : hébergement (nom, adresse, note 1-5 étoiles, prix/nuit)
— `Activity` : activité (nom, adresse, date, catégorie, prix)
— `Package` : forfait complet (aller + retour + hôtel + activités + erreurs)
— `Coordinates` : coordonnées GPS (latitude, longitude)

1.4 Diagramme de classes

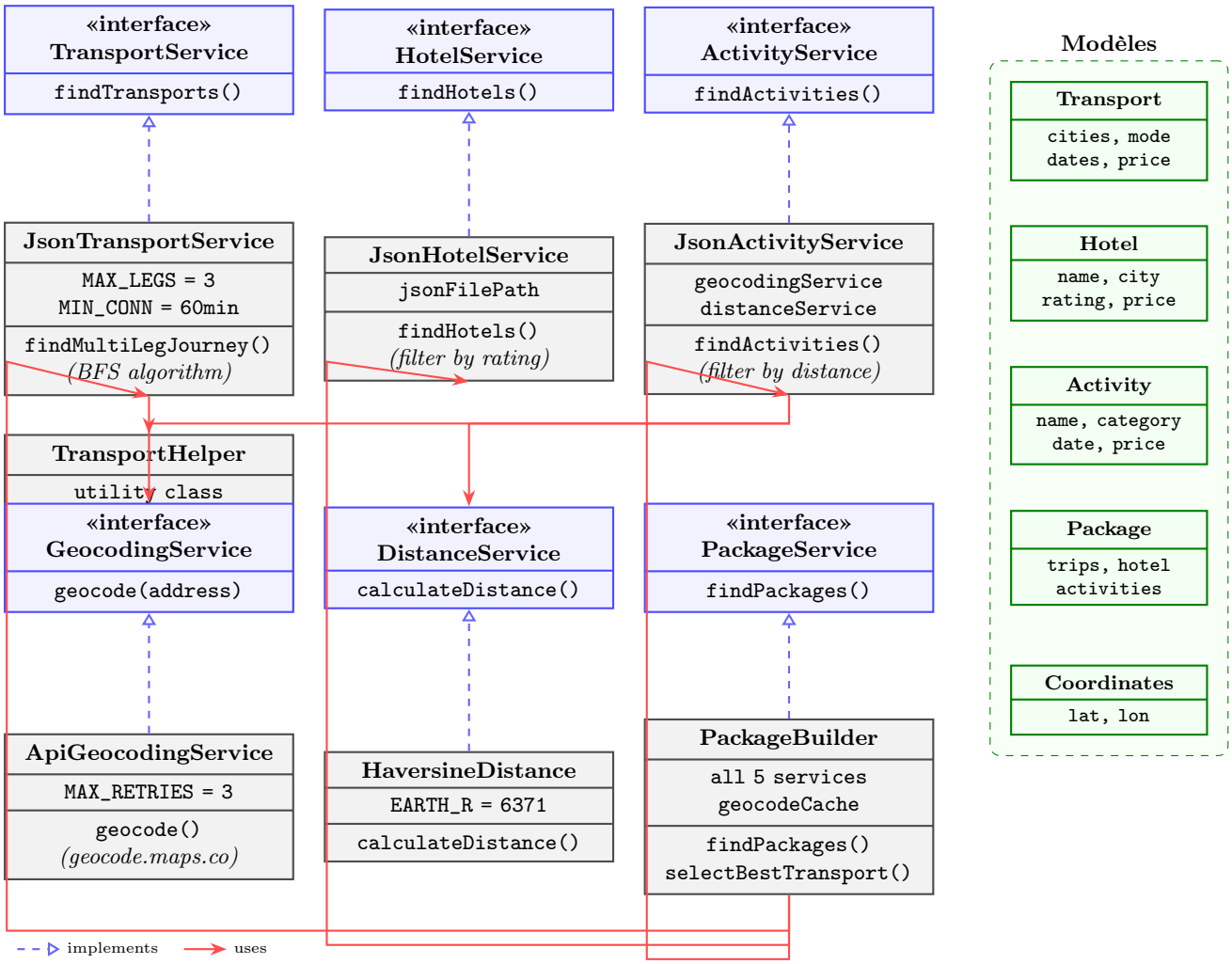

FIGURE 1 – Diagramme de classes UML de l'application Holitrip

1.5 Diagramme de séquence

Le diagramme suivant illustre le flux d'exécution de la méthode `findPackages()` qui orchestre la recherche d'un forfait complet.

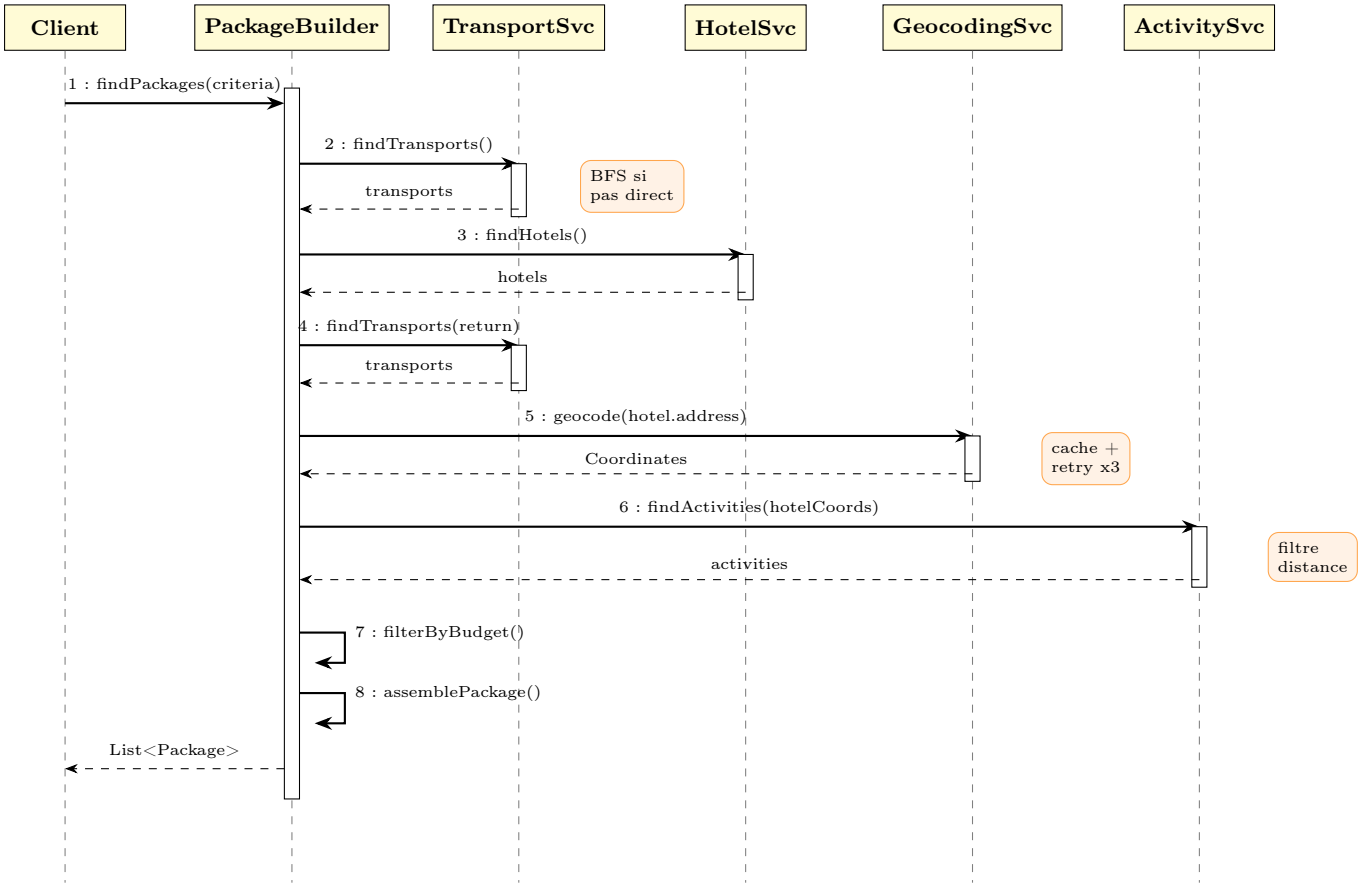

FIGURE 2 – Diagramme de séquence de `findPackages()`

1.6 Choix de conception

1. **Injection de dépendances** : toutes les implémentations reçoivent leurs dépendances via constructeur, permettant de les remplacer par des mocks/stubs lors des tests.
2. **Algorithme BFS** : recherche en largeur pour les trajets multi-correspondances, garantissant l'homogénéité du mode de transport et un temps de correspondance minimum de 60 minutes.
3. **Données JSON** : stockage dans des fichiers JSON (`transports.json`, `hotels.json`, `activities.json`).
4. **Gestion d'erreurs** : mécanisme de retry (3 tentatives) pour l'API de géocodage externe, exception dédiée `GeocodingException`.

2 Manuel d'utilisation

2.1 Prérequis

— Java 17 ou supérieur
— Maven 3.8+
— Connexion Internet (pour le géocodage)

Note : Toutes les commandes doivent être exécutées depuis le dossier `serveur/`.

2.2 Commandes

```
# Compilation
mvn clean compile

# Tests unitaires uniquement (107 tests)
mvn clean test -DskipITs

# Tests d'integration uniquement (17 tests) - necessite d'abord mvn test
mvn failsafe:integration-test

# Tous les tests (unitaires + integration)
mvn clean verify

# Rapport de couverture JaCoCo -> target/site/jacoco/index.html
mvn clean test jacoco:report

# Rapport de mutation PIT -> target/pit-reports/*/index.html
mvn org.pitest:pitest-maven:mutationCoverage

# Execution de l'application (scenario Bordeaux -> Paris)
mvn exec:java -Dexec.mainClass="fr.univ.holitrip.HolitripMain"

# Autres scenarios de demonstration
mvn exec:java -Dexec.mainClass="fr.univ.holitrip.HolitripMain2"
```

3 Rapport de couverture de code

Nous avons utilisé **JaCoCo** pour mesurer la couverture de notre suite de **107 tests unitaires** aux niveaux instruction et branche.

3.1 Scores globaux

Métrique	Score
Couverture des instructions	**93%** (1809/1936)
Couverture des branches	**78%** (253/322)
Couverture des lignes	**91%** (409/451)
Couverture des méthodes	**97%** (58/60)

TABLE 2 – Scores de couverture JaCoCo (13 classes analysées)

Configuration JaCoCo

Le `pom.xml` exclut de l'analyse les classes non métier : `HolitripMain*` (points d'entrée CLI) et les DTOs du package `model` (`Activity`, `Coordinates`, `Hotel`, `Transport`, `Trip`). Ces exclusions permettent de mesurer la couverture réelle du code métier.

3.2 Détail par classe

Classe	Instr.	Branch.	Commentaire
`TransportHelper` (util)	98%	85%	Utilitaires transport
`HaversineDistanceService`	100%	–	Formule mathématique
`JsonTransportService`	95%	83%	Algorithme BFS bien couvert
`PackageBuilder`	96%	73%	Logique d'assemblage
`JsonActivityService`	91%	57%	Filtrage par distance
`JsonHotelService`	89%	83%	Filtrage par critères
`ApiGeocodingService`	77%	64%	Retry et erreurs testés
`Package` (model)	93%	80%	Méthodes métier testées
`GeocodingException`	100%	–	Exception personnalisée

Table 3 – Détail de la couverture par classe

3.3 Justification des parties non couvertes

— **ApiGeocodingService (77%)** : Les branches non couvertes concernent le code de gestion d'erreurs HTTP rares (timeout réseau, codes HTTP inattendus). Ces cas sont difficiles à reproduire en test unitaire sans mocking intensif du client HTTP.

— **JsonActivityService (57% branches)** : Les branches manquantes concernent les cas où aucune activité ne correspond aux critères (catégorie inexistante, distance maximale trop restrictive).

4 Rapport d'analyse par mutation

Nous avons utilisé **PIT** (PITest) pour évaluer la qualité de nos tests via l'analyse par mutation. PIT est configuré pour cibler les mêmes classes que JaCoCo (exclusion des `HolitripMain*` et DTOs).

4.1 Score de mutation

Métrique	Valeur
Couverture des lignes (classes mutées)	398/442 (**90%**)
Mutations générées	281
Mutations tuées	215
Score de mutation	**77%**
Mutations sans couverture	15
Force des tests	**81%**
Tests exécutés	1179 (4.2 tests/mutation)

Table 4 – Résumé de l'analyse PIT

Interprétation

Le score de mutation de **77%** indique que nos tests détectent 77% des bugs potentiels introduits par PIT. La **force des tests** de **81%** représente le taux de détection sur le code effectivement couvert, montrant l'excellente efficacité des assertions.

4.2 Détail par mutateur

Mutateur	Générés	Tués	Taux
NegateConditionalsMutator	155	125	**81%**
MathMutator	29	26	**90%**
BooleanTrueReturnValsMutator	26	20	**77%**
ConditionalsBoundaryMutator	18	7	39%
EmptyObjectReturnValsMutator	18	10	56%
VoidMethodCallMutator	14	8	57%
NullReturnValsMutator	9	8	**89%**
BooleanFalseReturnValsMutator	7	7	**100%**
PrimitiveReturnsMutator	5	4	80%
Total	**281**	**215**	**77%**

TABLE 5 – Détail des mutations par type

4.3 Justification des mutants survivants

— **NegateConditionalsMutator (30 survivants)** : Conditions défensives (null checks) et branches d'erreur. Certains sont des **mutants équivalents** où inverser la condition ne change pas le comportement (ex : liste toujours non-null).

— **ConditionalsBoundaryMutator (11 survivants)** : Changements de bornes (< → <=) non détectés car nos tests n'utilisent pas systématiquement des valeurs exactement aux limites (ex : 60 minutes exactement de correspondance).

— **EmptyObjectReturnValsMutator (8 survivants)** : Mutations sur les retours de collections vides dans des cas où le test vérifie uniquement la présence d'erreurs, pas la valeur de retour spécifique.

— **BooleanTrueReturnValsMutator (6 survivants)** : Mutations sur des méthodes booléennes dont le résultat n'est pas directement vérifié dans tous les tests (ex : méthodes de validation interne).

— **VoidMethodCallMutator (6 survivants)** : Appels de méthodes void dont les effets de bord ne sont pas toujours vérifiés (ex : logging).

— **MathMutator (3 survivants)** : Mutations dans la formule d'Haversine où les variations sont trop faibles pour être détectées par les assertions avec delta.

4.4 Mutants équivalents

Certains mutants produisent un comportement identique au code original :

— Retour de liste vide vs `Collections.emptyList()`
— Conditions redondantes (double validation)
— Opérations sur des valeurs limites (0, chaîne vide)

5 Difficultés rencontrées et retour d'expérience

5.1 Difficultés techniques

— **API de géocodage externe** : L'API `geocode.maps.co` impose des limites de requêtes (rate limiting). Nous avons implémenté un mécanisme de retry avec backoff et créé un stub (`TestGeocodingService`) pour les tests d'intégration.

— **Algorithme BFS** : L'implémentation des trajets multi-correspondances avec toutes les contraintes (homogénéité du mode, temps minimum, pas de cycles) a nécessité plusieurs itérations. La logique complexe de validation des correspondances rendait difficile l'atteinte d'un bon score de mutation.

— **Amélioration du score de mutation** : Le score initial de 68% révélait des mutants survivants dans la logique de validation des transports. La refactorisation avec création de `TransportHelper` a permis d'isoler cette logique et d'écrire des tests unitaires ciblés, augmentant le score à 77%.

— **Couverture des branches conditionnelles** : Certaines branches (`ConditionalsBoundaryMutator` à 39%) nécessiteraient des tests avec valeurs exactement aux limites (ex : 60 minutes de correspondance), ce qui n'était pas toujours pertinent du point de vue métier.

5.2 Retour d'expérience

— L'injection de dépendances a grandement facilité l'écriture des tests unitaires avec mocks et le travail en parallèle sur différents services

— Les tests d'intégration ont permis de détecter des bugs de composition non couverts par les tests unitaires (interactions entre services réels)

— L'analyse par mutation a révélé des faiblesses dans nos assertions (trop permissives) et guidé la refactorisation du code pour améliorer la testabilité

— La **convention AAA** (Arrange-Act-Assert) a amélioré la lisibilité et la maintenabilité des tests, facilitant l'identification des scénarios testés

— Le respect des exigences du projet (correspondances, homogénéité, critères utilisateurs) a guidé la conception des tests fonctionnels

Conclusion

Ce projet nous a permis de mettre en pratique les notions de test logiciel : tests unitaires avec doublures (Mockito), tests d'intégration, couverture de code (JaCoCo) et analyse par mutation (PIT).

Nous avons développé **107 tests unitaires** et **17 tests d'intégration**, atteignant une couverture de **93%** en instructions et **78%** en branches sur le code métier. L'analyse par mutation révèle une force de test de **81%**, indiquant que nos assertions détectent efficacement les bugs potentiels.

L'architecture modulaire avec injection de dépendances s'est révélée essentielle pour permettre le test unitaire efficace et le travail parallèle de l'équipe.