Rapport d'Expérience Personnelle : Moteur de Simulation d'Ascenseurs

Yanis Chennoufi

19 novembre 2025

1 Rôle dans le projet

Mon rôle principal était de développer la partie dynamique du simulateur, incluant le moteur de simulation, le comportement des ascenseurs et les stratégies de contrôle.

Mes responsabilités spécifiques comprenaient :

- l'implémentation de la boucle de simulation (`Simulation`, `SimulationClock`) ;
- la gestion de la physique simplifiée des ascenseurs (mouvement, portes, arrêts) ;
- lécriture des stratégies dascenseur de base (`fcfs`, `nearest`) ;
- le calcul et l'agrégation des métriques de performance (temps dattente, énergie, etc.).

Ces éléments sappuient sur le modèle et sont exploités par l'interface/les rapports JSON .

2 Travail réalisé

2.1 Conception du moteur de simulation

J'ai conçu le moteur de simulation temps-discret (`fr.esipe.elevatorsim.simulation`).

Classe `Simulation` : Centralise l'exécution. Elle reçoit le `Building`, le `SimulationClock` et l'`ElevatorStrategy`, et gère la liste des requêtes générées et en attente.

Classe `SimulationClock` : Gère le temps discret (courant, tick, durée totale).

Boucle d'exécution (`run()`) :

1. Initialisation des requêtes programmées.
2. Boucle journalière (`isFinished()`) :
   - Activation des nouvelles requêtes.
   - Appel à la stratégie pour obtenir les actions des ascenseurs.
   - Avancement physique des ascenseurs (`step`).
   - Gestion des montées/descentes (`pickups`, `dropoffs`).
   - Mise à jour des métriques.
   - Avancement du temps (`tickSeconds`).

2.2 Modélisation du comportement des ascenseurs

J'ai implémenté la dynamique de la classe `Elevator` dans `fr.esipe.elevatorsim.model`.

L'ascenseur est une modélisation physique simplifiée (position continue, vitesse, accélération) avec des contraintes (capacité, vitesse max, accélération max, temps de portes) issues de la configuration.

La méthode `step(int tickSeconds)` :

- met à jour la vitesse et la position en fonction de la direction et de l'accélération ;
- gère l'ouverture/fermeture des portes et le temps passé à l'étage ;
- comptabilise l'énergie consommée lors des déplacements (modèle simplifié : montée seule).

L'architecture vise à séparer la décision (Stratégie) de l'exécution physique (Ascenseur).

2.3 Implémentation des stratégies dascenseur

L'interface `ElevatorStrategy` est mise en place pour définir l'action de l'ascenseur à chaque *tick* :

```
Action chooseAction(Building b, Elevator e, List<Request> pending)
```

J'ai implémenté deux stratégies pour validation :

- **FCFS (First-Come, First-Served)** : Affectation à la requête la plus ancienne, l'ascenseur étant bloqué sur cette mission jusqu'à son terme.
- **Nearest (Plus Proche)** : Une fois libre, l'ascenseur est assigné à la requête dont le point de départ est le plus proche de sa position actuelle.

2.4 Calcul des métriques et préparation des rapports

Les métriques sont calculées directement au sein de la boucle `Simulation` et agrégées pour les rapports JSON.

- **Temps d'attente (Wait Time)** : Temps entre l'appel (`timeCreated`) et le `pickup`.
- **Temps de trajet (Trip Time)** : Temps entre le `pickup` et le `dropoff`.
- **Énergie consommée (Energy)** : Cumul des incréments lors des phases d'accélération et de montée.
- **Occupation maximale (Max Occupancy)** : Nombre maximal de passagers simultanés.

3 Difficultés rencontrées et solutions

La principale difficulté a été la gestion du mouvement continu dans un temps discret.

- **Difficulté 1 : Précision de l'arrêt aux étages.** L'ascenseur évoluant par étapes discrètes, il risquait de dépasser l'étage cible.
- **Solution 1 :** Implémentation d'une logique de décélération prédictive à chaque *tick*. L'ascenseur utilise la distance d'arrêt (distance $= v^2/(2a)$) pour inverser l'accélération à temps, assurant un arrêt précis sur l'étage.
- **Difficulté 2 : Synchronisation des états.** La coordination des changements d'état entre la `Simulation` (requêtes), l'`Elevator` (physique) et la `ElevatorStrategy` (décision) était complexe.
- **Solution 2 :** Formalisation d'états et d'actions atomiques clairs. La `Strategy` ne renvoie qu'une `Action` simple (`MOVE_UP`, `STOP`, `DOORS_OPEN`), et la `Simulation` applique les conséquences logiques (ex : mise à jour des requêtes en `onBoard` lors du `pickup`).

4 Limites et idées damélioration

Le simulateur fournit une base fonctionnelle mais présente des limites.

4.1 Limites du modèle actuel

- **Physique simplifiée :** Le modèle ignore la charge utile variable, l'inertie et la friction.
- **Génération déterministe :** La génération des requêtes manque de la nature imprévisible et stochastique du trafic réel.
- **Stratégie minimaliste :** L'interface est synchrone (décision immédiate) et ne permet pas aux stratégies d'anticiper ou de planifier des objectifs futurs.

4.2 Idées d'amélioration

- **Stratégies avancées :** Intégration d'algorithmes sophistiqués (Group Control System, Apprentissage par Renforcement).
- **Modèle énergétique réaliste :** Affiner le calcul pour inclure la consommation en fonction de la charge et la récupération d'énergie.
- **Interface utilisateur graphique (IUG) :** Ajouter une IUG pour la visualisation en temps réel des mouvements et des files d'attente.

5 Bilan personnel

Ce projet a été très formateur sur la modélisation de systèmes continus en environnement discret. J'ai renforcé mes compétences en POO, notamment par l'application du patron de conception *Strategy* et le principe de séparation des préoccupations (physique vs. décision). Le défi de la synchronisation m'a obligé à structurer le code de manière très modulaire, ce qui a été essentiel pour valider l'architecture et garantir que le simulateur puisse supporter l'intégration de stratégies plus complexes à l'avenir.