Elevator Simulator

Documentation développeur

Projet Java — ESIEP

19 novembre 2025

1 Introduction

Ce document présente l'architecture interne du simulateur d'ascenseurs et indique comment un développeur peut le faire évoluer : ajout de stratégies, de scénarios de simulation, ou de nouvelles métriques.

Le projet est écrit en Java 17, construit avec Maven, et n'utilise qu'une seule bibliothèque externe pour la sérialisation JSON (Jackson).

2 Architecture globale

Le code est organisé en plusieurs packages :

- `fr.esipe.elevatorsim` : point d'entrée (`App`), parsing des arguments.
- `fr.esipe.elevatorsim.config` :
  - `SimulationConfig` : mapping du JSON de configuration ;
  - `ConfigLoader` : lecture JSON → `SimulationConfig` ;
  - `ModelFactory` : `SimulationConfig` → `Building` + résidents + ascenseurs.
- `fr.esipe.elevatorsim.model` :
  - `Building`, `Floor`, `Resident`, `ResidentTripPlan` ;
  - `Elevator` : état physique simplifié ;
  - `ElevatorRequest` : une demande de transport.
- `fr.esipe.elevatorsim.simulation` :
  - `SimulationClock` : temps discret ;
  - `Simulation` : boucle principale, calcul des métriques, collecte d'événements.
- `fr.esipe.elevatorsim.strategy` :
  - `ElevatorStrategy` : interface ;
  - `FcfsElevatorStrategy`, `NearestRequestStrategy` ;
  - `StrategyFactory` : sélection de stratégie par nom.
- `fr.esipe.elevatorsim.stats` :
  - `SimulationStats`, `SimulationReport` ;
  - `JsonReportWriter`, `ElevatorStopsJsonWriter`, `ResidentsReportJsonWriter`.
- `fr.esipe.elevatorsim.ui` :
  - `ConsoleIO` : E/S console robustes ;
  - `ConsoleUI` : menus interactifs, historique des actions.

L'application peut fonctionner en mode "batch" (ligne de commande) ou en mode interactif (menus console) mais utilise toujours le même coeur de simulation.

3 Boucle de simulation

La classe `Simulation` orchestre l'exécution :

1. Initialisation :
   — récupération du `Building` et des résidents via `ModelFactory` ;
   — génération des `ElevatorRequest` en fonction des `ResidentTripPlan` et de la graine aléatoire ;
   — création d'un `SimulationClock`.
2. Boucle `run()` (temps discret) :
   (a) récupération de l'heure courante ;
   (b) activation des requêtes dont `requestTime` est atteint ;
   (c) pour chaque ascenseur :
      — appel de `ElevatorStrategy.step(...)` ;
      — appel de `Elevator.step(tickSeconds)` ;
      — gestion des montées / descentes (`handleStopsAndRequests`) ;
   (d) mise à jour continue des métriques : temps d'attente, temps de trajet, énergie, occupation ;
   (e) `clock.tick()`.
3. Fin de simulation :
   — calcul des statistiques (moyenne, médiane, max, etc.) ;
   — écriture des rapports JSON via les classes du package `stats`.

Les stratégies ne modifient pas directement la position des ascenseurs : elles définissent le comportement souhaité (cible, ouverture des portes, etc.), la physique reste centralisée dans `Elevator`.

4 Ajout d'une nouvelle stratégie d'ascenseur

Pour ajouter une heuristique, la marche à suivre est la suivante :

1. Créer une classe dans `fr.esipe.elevatorsim.strategy` implémentant `ElevatorStrategy`, par exemple :

```
public class MySmartStrategy implements ElevatorStrategy {
    @Override
    public void step(Building building,
                     Elevator elevator,
                     List<ElevatorRequest> pending,
                     int currentTime,
                     int tickSeconds) {
        // logique de sélection de requêtes et de déplacement
    }
}
```

2. Enregistrer la stratégie dans `StrategyFactory`, par exemple

```
public static ElevatorStrategy fromName(String name) {
    return switch (name.toLowerCase()) {
        case "fcfs"    -> new FcfsElevatorStrategy();
        case "nearest" -> new NearestRequestStrategy();
        case "smart"   -> new MySmartStrategy();
        default        -> new NearestRequestStrategy();
    };
}
```

3. Optionnel : exposer la stratégie dans l'UI (menu Stratégie) ou via un argument de ligne de commande supplémentaire.

Aucune modification de la classe `Simulation` n'est nécessaire tant que l'on respecte la signature de `ElevatorStrategy.step`.

5 Ajout d'un nouveau scénario de configuration

Les scénarios (tour, profils de résidents, ascenseurs) sont décrits dans des fichiers JSON sous `src/main/resources/config/`.

Pour ajouter un scénario :

1. Créer un nouveau fichier, par exemple `config/weekday.json`, en suivant la structure de `demo-config.json`.
2. Ajuster :
   — le nombre d'étages et de résidents ;
   — les fenêtres horaires des déplacements ;
   — la liste des ascenseurs.
3. Lancer l'application avec :

```
java -jar target/...jar-with-dependencies.jar \
  --config=config/weekday.json \
  --strategy=nearest \
  --report=target/reports/weekday-report.json
```

La sérialisation est gérée par `SimulationConfig` et `ConfigLoader`. Aucune modification de code n'est nécessaire tant que le JSON reste compatible avec `SimulationConfig`.

6 Extension des métriques et des rapports

Les métriques globales sont regroupées dans `SimulationStats`.

Pour ajouter une nouvelle métrique :

1. Ajouter un champ dans `SimulationStats` (par exemple un nouveau compteur ou une moyenne).
2. Incrémenter cette métrique dans `Simulation` pendant la boucle `run()` (par exemple à chaque tick ou à chaque requête complétée).
3. Étendre `SimulationReport` si la nouvelle métrique doit apparaître dans le JSON global.
4. Adapter `JsonReportWriter` (et éventuellement les autres writers) pour sérialiser cette métrique.

Les rapports sont produits sous forme de fichiers JSON facilement exploitable par des scripts ou outils externes (Python, R, tableur, etc.).

7 Build, tests et exécution

Le projet utilise Maven. Les principales commandes sont :
— `mvn clean package` : compile et construit le JAR exécutable ;
— `mvn test` : exécute les tests (par exemple `SimulationSmokeTest`).

L'exécution se fait via :

```
java -jar target/elevator-simulator-1.0-SNAPSHOT-jar-with-dependencies.jar \
  [--config=...] [--strategy=...] [--report=...] [--interactive]
```