Expérience ***Flavio BOTTARLINI***

Programmation Java :
Modélisation de l'ascenseur

Avant-propos

Il est essentiel de savoir qu'avec mes coéquipiers nous avions définit un modèle à suivre, une structure, pour représenter l'ensemble du projet. Nous avons notamment tenté de définir quelles classes choisir pour quels éléments, combien de classes nous devrions créer pour chaque partie et ce qu'elles devraient contenir. Avec ce modèle fait, nous avons pu attribuer les tâches par blocs.

Mon rôle fut de **modéliser la cage d'ascenseur** et toute la partie **algorithmique qui permet de gérer les décisions de déplacement** ainsi que le déplacement lui-même.

Ce document rapporte essentiellement les problèmes rencontrés au cours du développement et les solutions apportées.

Pour la sécurité des fichiers de mon ordinateur ainsi que la définition claire de l'environnement de programmation, le développement de ma partie a été réalisé dans un conteneur Debian 11 (Dev Container Java : mcr.microsoft.com/devcontainers/java:1-21-bullseye) d'architecture ARM.

A savoir que pour accélérer la programmation, j'utilise régulièrement GitHub Copilot pour la complétion de lignes mais aussi la vérification et l'amélioration structurelle et syntaxique de code, ainsi que la recherche de précédents (projets similaires sur des forums). Pour ce projet, j'ai utilisé les modèles GPT-4.1 (complétion et corrections) et Claude Haiku/Sonnet 4.5 (vérification et amélioration) sur Visual Studio Code.

1. Mise en œuvre

Tel que nous l'avions défini, la partie, concernant elle-seule les ascenseurs, devait être constituée de 3 classes distinctes :

- **AGA** (Algorithme de Gestion des Ascenseurs) – *class* : seule classe qui aura un nom en français, son rôle est de gérer les différents ascenseurs avec notamment une méthode pour en créer ou en détruire un et une méthode pour prendre en compte un appel d'ascenseur et l'attribuer à un ascenseur spécifique.

- **Lift** – *record* : c'est la classe qui représente un ascenseur en tant que tel. Elle contient notamment les paramètres d'un ascenseur (capacité maximale, étage maximal / minimal, vitesse, accélération, décélération…).
- **Heuristics** – *enum* : contient les supplantations de méthode permettant de réaliser le choix du prochain étage à visiter.

Mais ce modèle à rapidement posé un problème : il était simple sur le papier (trop simple) mais ne rendait aucunement compte des potentiels problèmes de mutabilité, d'implémentation ou encore de prise en compte de la lisibilité.

La structure a donc grandement évolué par la suite, et cette évolution est celle présentée ci-après, pour faire place à une structure subdivisée en éléments et méthodes plus indépendants les uns des autres.

2. Mutabilité et structure

2.1. Problème : comment représenter correctement un objet aux propriétés fixes mais pouvant évoluer dans un environnement ?

L'un des premiers problèmes sur lequel je suis tombé, faute d'avoir considéré correctement ce qu'est une classe *record*, est l'utilisation de variables immutables pour des concepts qui nécessitent la mutabilité, notamment :

- ***currentFloor*** : suit la position de l'ascenseur
- ***currentCapacity*** : suit l'évolution du nombre de personnes dans l'ascenseur
- ***targetFloor*** : l'objectif de position suivant de l'ascenseur
- ***requestedFloors*** : la liste (type de liste non précisé) qui permet d'enregistrer toutes les demandes de desserte
- ***heuristics*** : l'heuristique choisie, qui pourrait être changée

Une classe *record* n'était donc pas adaptée pour enregistrer toutes les variables.

2.2. Solution : ajout de classes pour séparer les fonctionnalités

Pour pallier le problème de la mutabilité, j'ai donc préféré diviser la classe devant représenter un ascenseur en deux classes distinctes mais fortement liées :

- **Lift** – *record* : contient les informations immutables
- **LiftManager** – *class* : contient les méthodes d'évolution avec l'heuristique et les valeurs mutables

D'autre part, j'ai rajouté les classes indépendantes, portant chacune des méthodes propres à leur utilité, suivantes :

- **LiftPhysics** – *class* : contient uniquement les méthodes permettant de donner une dimension physique à la simulation.
- **LiftEvolution** – *record* : utilisée pour renvoyer un objet unique qui contient les informations sur le déplacement d'un ascenseur (méthode contenue dans **LiftManager**).

Une autre de mes erreurs, et je m'en rends compte une fois l'examen de Java passé, est de ne pas correctement avoir utilisé des éléments de programmation du langage comme *abstract*, *extends* ou *super()*. Ceux-ci auraient certainement été plus adaptés pour la manière dont j'ai représenté les ascenseurs.

3. Deux niveaux d'algorithme

3.1. Problème : heuristique, global ou local ?

Une considération qui a rapidement fait surface est la question de la gestion des ascenseurs d'un point de vue extérieur, l'AGA. Dans notre modèle d'origine c'était bien l'AGA qui définissait l'entièreté des chemins à suivre, que ce soit parce qu'une personne demande un étage spécifique ou qu'une autre personne appelle l'ascenseur depuis un autre étage.

Mais rapidement je me suis posé la question de savoir s'il s'agissait vraiment de la meilleure manière de voir les choses. La décision du chemin à prendre en fonction des passagers à l'intérieur semble plutôt être un problème de grandeur locale par rapport à l'ascenseur. D'autre part, la décision de missionner un ascenseur spécifique semble de grandeur globale, c'est un élément extérieur à l'ascenseur qui va envoyer une demande pertinente à un des ascenseurs (et non pas à tous).

3.2. Solution : mise en place de deux niveaux distincts

Pour faire cette représentation qui a un aspect « global » et « local », j'ai choisi d'implémenter des classes supplémentaires qui permettent d'interagir avec l'**AGA** :

- **Call** – *record* : c'est ce à quoi une personne, par le biais de la simulation, fait appel. Un **Call** sera utilisé pour stocker les informations sur l'origine et la destination pour une personne.
- **Dispatcher** – *interface* : impose l'implémentation d'une méthode *selectLift* qui permet de définir comment l'**AGA** doit se comporter, quel ascenseur il doit appeler.

- **DispatcherStrategy** – *enum* : permet de faire passer une stratégie spécifique à l'**AGA** sans avoir à importer ladite classe à chaque fois que l'on souhaite une autre stratégie (vision qui a plus tendance à faciliter la programmation que la structure du code).

Les demandes provenant d'un **Call** seront considérées comme prioritaires sur les choix faits par les heuristiques. L'étage d'origine d'un appel viendra placera alors en prochaine cible d'un ascenseur choisi.

La méthode *selectLift* ne définit pas le comportement de l'ascenseur mais bien le choix d'un ascenseur qui reçoit un ordre : celui de répondre à un appel. Je l'ai modélisé comme une « interruption » du cycle de choix de l'ascenseur car c'est ainsi que je le visualise.

4. Décider de la bonne structure à adopter pour représenter une liste de destination

4.1. Problème : Set, Map ou List ?

Si ce problème peut être banal en apparence, il fut loin de l'être pour moi. Au départ, je fus intimement persuadé que les ascenseurs (au travers de gestionnaire **LiftManager**) devaient connaître le nombre de passager et leurs destinations. Cela aurait impliquée une structure comme une **HashMap** ou une **LinkedHashMap** (pour l'implémentation d'heuristiques comme FIFO, LIFO, etc…). À minima, savoir combien de passagers souhaitent aller à un étage en particulier.

La structure de cette information est donc passée par plusieurs étapes, en commençant par ce qui me paraissait le plus logique : une structure Set, pour ne pas avoir de répétitions sachant que les appels étaient individuels (1 par personne, même si plusieurs personnes prennent l'ascenseur au même étage).

4.2. Solution : un ascenseur réel n'a jamais cette information

Après un long combat interne et beaucoup de simulations réalisées avec des codes générés par IA (Claude Haiku 4.5), je suis finalement revenu à l'idée de départ. Si la flatterie de l'IA saluait l'idée que je lui soumettais de vouloir utiliser une structure contenant le maximum d'informations, je me suis rendu compte que c'était parfaitement inutile en faisant une simple analogie avec la réalité.

Finalement, j'ai adopté définitivement ma première idée : celle d'une structure **Set** avec un **HashSet**. Elle est simple, efficace et ne demande pas de refaire des vérifications supplémentaires. Le plus important pour l'ascenseur est bien de savoir où il doit aller et dans la plupart des cas ne

tient pas compte de combien de personnes veulent y aller pour une simple raison : si quelqu'un appuie sur le bouton, ce n'est pas systématique que quelqu'un appuie de nouveau sur celui-ci tant que la demande est en attente (bouton allumé généralement).

Bien sûr, pour le bien d'une simulation (donc d'une abstraction), il aurait été intéressant d'implémenter une telle possibilité.