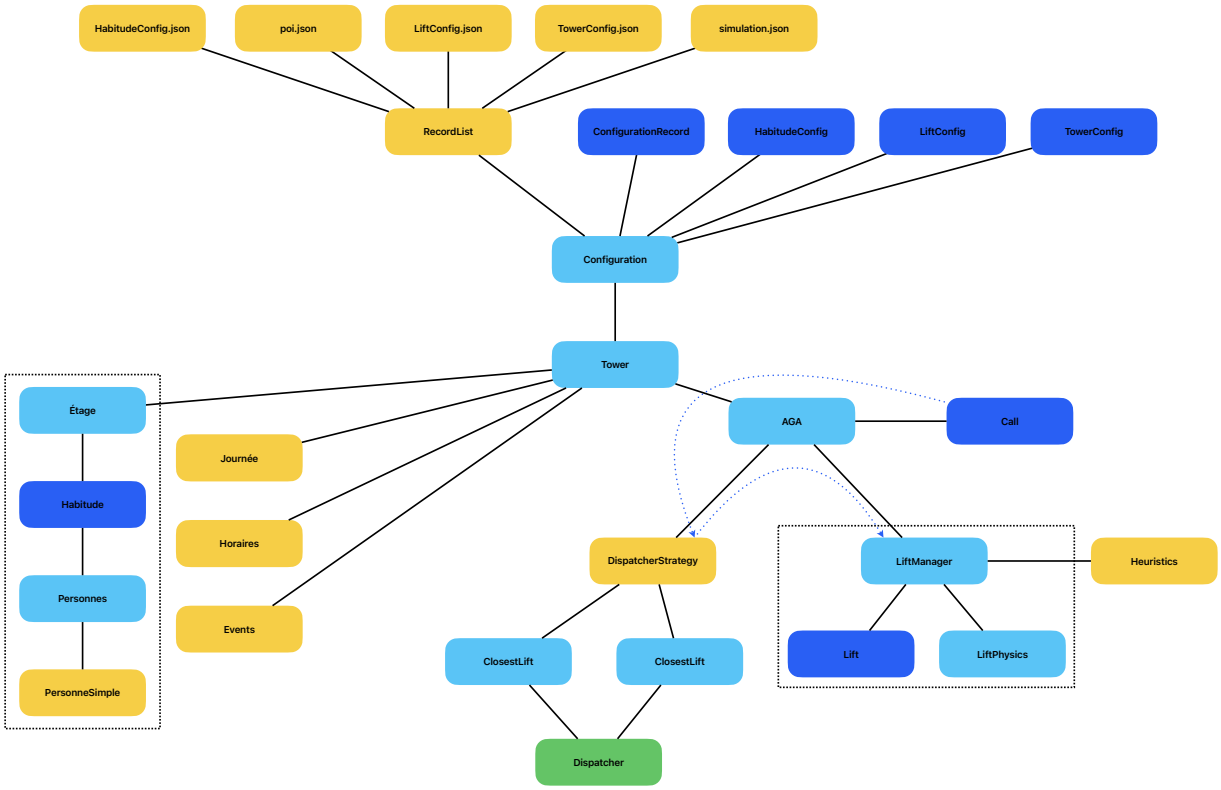
JavaLift-ESIEE
Documentation développeur

1. Structure du projet

Ce projet est structuré en trois grands blocs. Ceux-ci permettent de rendre le développement modulaire et de rajouter des algorithmes qui gèrent le comportement.



1.1. Représentation des ascenseurs

Il a été choisi de représenter les ascenseurs et leur algorithme complètement indépendamment du reste du programme. L'objectif était de pouvoir rendre extensible le nombre d'ascenseurs dans la tour (hors limitation d'une IHM) et de faire évoluer les attributs des différents ascenseurs à l'aide de méthodes principalement portées par la classe **AGA**.

Les méthodes accessibles dans l'**AGA** et **LiftManager** permettent déjà de réaliser la plupart des opérations (déplacer un ascenseur, rajouter une étape, faire descendre/monter des personnes…).

1.2. Simulation et visualisation

C'est le bloc dans lequel la simulation est gérée. Elle fait appel aux autres blocs du projet et est composée de deux sous-parties :

- **Configuration :** charge les fichiers de configuration JSON et place tous les éléments dans des classes objets dédiées.
- **Interface graphique :** génère une simulation visuelle à partir des données du reste du projet qui indique les positions (étage de l'ascenseur désigné, personne à qui souhaite monter dans l'ascenseur…).

C'est également dans ce bloc fonctionnel que le programme est compilé et lancé.

1.3. Étages et personnes

Ce bloc fonctionnel a pour objectif de représenter les personnes et les étages (qui « contiennent » les personnes). Les personnes se voient attribuer un certain nombre d'habitudes (des moments auxquels ils vont choisir de se déplacer depuis l'étage où ils sont vers un nouvel étage).

Chaque personne peut avoir plus d'une seule habitude. Lesdites habitudes sont données sous la forme d'une énumération **Habitudes** pour laquelle chaque élément est associé avec un horaire de début et un horaire de fin.

2. Extensivité – heuristiques, algorithmes de répartition et habitudes

Le déplacement des ascenseurs (du moins, la décision de la destination) est décidé par deux algorithmes distincts comme vu plus tôt.

La partie des heuristiques décide de la prochaine cible qu'un ascenseur doit atteindre en fonction d'une liste de requêtes (les appels fait plus tôt). Le second algorithme réalise la répartition des réponses aux appels d'ascenseurs, indépendamment des heuristiques. Il est appelé par l'algorithme de gestion des ascenseurs pour décider, en fonction du modèle algorithmique choisi et des données sur les ascenseurs, du meilleur ascenseur à envoyer pour répondre à l'appel.

2.1. Ajout d'heuristiques

Il est possible d'ajouter des heuristiques relativement facilement grâce au fait qu'il s'agisse d'une classe énumérative. On utilise une fonction définie en *public abstract* pour réimplémenter chaque fonction en fonction de l'heuristique choisie.

Pour ajouter l'heuristique, on rajoute un attribut dans l'énumération **Heuristics** de la manière suivante :

```
/* New heuristic */

NEW_HEURISTIC {
    @Override
    public int chooseTargetFloor(LiftManager lm) {
        int nextDest = 0;
        return nextDest;
    }
};
```

L'objectif est de renvoyer une valeur ***nextDest*** qui indiquera à l'ascenseur sa prochaine cible qu'il doit atteindre.

Cette valeur nextDest pourra être déterminer selon la stratégie souhaitée mais

2.2.Ajout d'algorithme de répartition des ascenseurs

Si l'idée est la même qu'avec les heuristiques, l'implémentation d'un nouvel algorithme de répartition, étant donné qu'il prend en compte plus de variables que les heuristiques, n'est pas fait avec le rajout d'une valeur d'énumération.

Dans ce cas, il faut rajouter un nouveau fichier (une nouvelle classe en l'occurrence) qui implémente l'interface **Dispatcher**. Dans cette nouvelle classe, il faut ajouter une méthode *selectLift()* qui prend en paramètres le **Call** et la liste des **LiftManager** et renvoie une classe **LiftResponse** qui contient notamment l'identifiant de l'ascenseur appelé (pour l'interface graphique), ainsi que les informations sur le déplacement effectué (grâce à la classe **LiftPhysics**).

La fonction doit pouvoir déterminer et renvoyer l'identifiant du meilleur ascenseur à missionner pour répondre à l'appel. L'ascenseur sélectionné verra sa destination actuelle supplanté par l'origine de l'appel.

2.3.Ajout d'habitudes

L'ajout d'habitudes est relativement plus simple que les ajouts précédents car il ne nécessite pas de fonction ou d'énumération. La simplicité de sa structure et des informations que dois contenir une habitude rend la tâche plus facile.

Pour rajouter une habitude, il faut rajouter une ligne au fichier ressource ***HabitudeConfig.json*** qui contient la liste de toutes les habitudes et sera chargé dans un objet record **HabitudeDefinitionConfig**.

Une habitude se caractérise par un type (i.e. un moyen d'identifier l'habitude), une description, une heure de début et une heure de fin (au format hh:mm), ainsi qu'une destination.