Projet

André Abramé - andre.abrame@univ-amu.fr
Sophie Nabitz - sophie.nabitz@univ-avignon.fr
Petru Valicov - petru.valicov@univ-amu.fr

Date de rendu : Lundi 18 Janvier 2016 à minuit. Aucun retard ne sera accepté.

Le non-respect d'une des consignes ci-dessous impliquera une pénalité de 3 points minimum sur la note du projet.

Généralités

- Les modalités de déroulement des soutenances seront précisées ultérieurement.
- Le rendu du projet se fait par mail aux trois enseignants dont les adresses e-mail figurent en haut de ce document. Vous enverrez une archive `.tar.gz` ou `.zip` qui contiendra l'intégralité du code ainsi qu'un fichier INSTALL/README contenant les indications pour compiler et exécuter votre programme... Il ne faut pas inclure les fichiers `.class` dans votre archive !
- L'archive `tar.gz` ou `.zip` portera les noms de famille de chaque membre accolés. Par exemple, si les noms sont Dupont, Martin, Durand et Cornet, cette dernière s'appellera :
  `CornetDupontDurantMartin.tar.gz` OU `CornetDupontDurantMartin.zip`

Remarques

La note du projet prendra en compte :
- Le respect des consignes sur cette page et dans le sujet du projet.
- La modélisation objet de votre application. Également, nous serons particulièrement attentifs à l'utilisation appropriée des patrons de conception que vous aurez essayé de mettre en œuvre
- La propreté et la lisibilité du code ainsi que tout ce qui facilitera sa compréhension par les correcteurs (noms des variables, commentaires, modularité, etc).
- La facilité d'utilisation du code (le fichier INSTALL/README est-il lisible et suffit-il pour utiliser le code ?). La compilation de votre code devrait marcher sur tout type de machine - pensez à tester avant le rendu final.
- La correction et la performance du code.
- La présentation de votre application durant la soutenance.

Instructions

- Il est **strictement interdit de copier du code** des projets de vos collègues. Attention : l'obfuscation du code n'est pas une solution, c'est très facile à détecter !
- Des **tests unitaires** doivent être réalisés pour valider chaque fonctionnalité programmée. Il est conseillé donc de les réaliser tout au long du développement de votre projet. Vous utiliserez l'API `JUnit` intégrée dans votre IDE.
- Vous devez respecter les **conventions de nommage** `Java`
- Votre programme devra gérer proprement les **exceptions**.
- Vous êtes fortement incités à utiliser un outil de gestion de versions que vous préférez (Git, Svn, Mercurial). Vous pouvez en discuter avec votre chargé de TD pour des éventuels problèmes.
- Vous pouvez aussi implémenter les bonus et/ou améliorations que vous aurez imaginés. Il est cependant conseillé de venir en parler avec votre enseignant de projet afin d'en discuter. Dans tous les cas, il faut d'abord implémenter les fonctionnalités demandées dans le sujet.
- Il est également conseillé de fournir une **documentation complète** du code source (`JavaDoc`).

Gestion du fonctionnement d'un ascenseur

Au cours de votre travail de conception et de réalisation, pensez à respecter les principes vus dans vos cours de conception objet durant votre formation.

Les modalités de réalisation de certaines fonctionnalités ne sont pas forcément spécifiées afin de vous donner une certaine liberté dans la conception. En revanche, vous devez respecter les contraintes imposées dans le sujet et justifier tout choix qui les contredit.

1 Descriptif

L'entreprise **Ôtiste** dans laquelle vous travaillez (!), fournit des ascenseurs et le système de gestion permettant leur bon fonctionnement. L'objectif de ce projet est de réaliser un tel système. Pour anticiper tous les aléas d'un système temps réel, votre programme devra permettre de simuler "à la main" le fonctionnement du logiciel.

On se place dans le cas de figure général où un bâtiment est équipé d'un ou plusieurs ascenseurs. L'utilisateur définit le nombre d'étages à desservir ainsi que le nombre d'ascenseurs qu'il voudra installer. Les ascenseurs sont contrôlés via un ensemble de boutons. La cabine de chaque ascenseur comprend un bouton par étage, plus un bouton d'arrêt d'urgence permettant de bloquer/débloquer l'ascenseur. Sur le palier de chaque étage de l'immeuble, deux boutons sont présents : un pour demander un déplacement vers le haut de l'immeuble et un pour demander un déplacement vers le bas. Ainsi, la pression d'un bouton de palier par un utilisateur provoque le déplacement d'un des ascenseurs (l'utilisateur ne sait pas nécessairement lequel) de l'immeuble vers l'étage en question.

Une des classes principales de l'application est la classe `Ascenseur`. Elle regroupe les informations relatives aux ascenseurs du système : leur état, l'étage auquel ils se trouvent, etc.

Les demandes de déplacement d'un ascenseur sont gérées via un système de requêtes (classe `Requete`). On distingue deux types de requêtes : `RequeteInterne` et `RequeteExterne`. Les premières correspondent aux pressions des boutons de la cabine d'un ascenseur. Elles sont propres à cet ascenseur et ne peuvent donc être satisfaites que par lui. Les secondes, les requêtes externes, correspondent aux pressions des boutons situés à chaque étage de l'immeuble. Elles peuvent être satisfaites par tout ascenseur du système. Dans un premier temps, pour simplifier et situer le sujet, vous pouvez supposer que toutes les requêtes doivent être traitées par ordre de création avec une seule exception : les requêtes internes sont prioritaires par rapport aux requêtes externes.

Les mouvements des ascenseurs sont gérés par un `Controleur`. C'est lui qui se chargera de répartir les demandes des utilisateurs (les requêtes) entre les ascenseurs du système.

Fonctionnement général de l'application

La simulation du passage du temps dans votre système se fera par des itérations actionnées par l'utilisateur de l'application. À chaque itération, toutes les actions suivantes doivent être réalisées :

1. Proposer à l'utilisateur de créer de nouvelles requêtes (simuler la pression sur les boutons).
2. Affecter les nouvelles requêtes externes du contrôleur aux ascenseurs.
3. Actionner tous les ascenseurs selon les principes suivants :
   - si la destination de l'ascenseur n'est pas atteinte, il se déplace d'un étage dans la direction correspondante.
   - si l'étage courant de l'ascenseur est la destination, il ouvre ses portes pour que les utilisateurs entrent/sortent.
   - si l'ascenseur était immobile ouvert au début de l'itération, alors ses portes se ferment
   - s'il n'y a plus de requêtes, l'ascenseur reste immobile fermé
4. Afficher l'état du système.

Ainsi, un utilisateur pourra simuler le fonctionnement de votre logiciel et vérifier son bon fonctionnement.

Vous pourrez ajouter une classe **`Constantes`** permettant de stocker toutes les variables statiques (les constantes) dont vous aurez besoin. Pour rendre le code plus compréhensible et simplifier la maintenance du projet, les parties «*traitement des données*» et la partie «*affichage*» seront séparées.

— La couche *traitements* (package «**`ascenseur.traitement`**») : toutes les données et les traitements spécifiques (ex : ajouter des requêtes, gérer les actions de l'ascenseur, etc.) sont regroupés dans ce package. Les classes de cette couche ne concernent pas la partie graphique et se contentent seulement d'effectuer des traitements et de renvoyer des résultats.

— La couche *graphique* (package «**`ascenseur.affichage`**») : cette couche gère l'affichage (interface utilisateur) et les actions de l'utilisateur (clics ou saisies au clavier).

2 Mise en œuvre

La mise en œuvre du projet passera par la réalisation des classes et méthodes décrites ci-dessous.

2.1 Requêtes

Comme dit précédemment, on distinguera deux types de requêtes. Les requêtes internes peuvent être décrites par le numéro de l'étage demandé. Les requêtes externes peuvent être décrites par l'étage auquel a été fait l'appel et par la direction demandée (vers le haut ou vers le bas).

1. Écrire la ou les classes permettant de représenter les requêtes.

2.2 Ascenseurs

Un ascenseur peut être décrit par l'étage auquel il se trouve et par son état (*immobile ouvert*, *immobile fermé*, *mouvement vers le haut* ou *mouvement vers le bas*). Un automate décrivant les états et les transitions d'un ascenseur est présenté dans la Figure 1. De plus, chaque ascenseur doit mémoriser les requêtes qu'il doit traiter, c'est-à-dire ses requêtes internes et les requêtes externes qui lui ont été attribuées par le contrôleur.

1. Écrire les classes permettant de gérer les données relatives aux ascenseurs. En fonction de la demande de l'utilisateur (et de son portefeuille), l'entreprise peut fournir des ascenseurs équipés d'options diverses :
   — vitesse de déplacement augmentée,
   — musique d'ambiance lors des déplacements de l'ascenseur,
   — etc.
   Pour mettre en évidence ces options, vous pourrez vous contenter d'un simple message d'affichage correspondant.
2. Écrire une méthode **`bloquer()`** qui empêche tout mouvement de l'ascenseur (mais pas l'ajout de nouvelles requêtes).
3. Écrire une méthode **`débloquer()`** qui rétablit le fonctionnement normal de l'ascenseur. L'état de l'ascenseur après le blocage doit correspondre à son état précédent avec toutes ses requêtes (celles d'avant le blocage et celles qui ont été ajoutées après).
4. Écrire une méthode **`ajouterRequete()`** qui prend en paramètre une requête existante et l'ajoute à la collection des requêtes de l'ascenseur.
5. Ajouter une méthode **`creerRequeteInterne()`** qui prend en paramètre un étage et qui créer une nouvelle requête interne avec l'étage en attribut puis qui ajoute la requête à la collection des requêtes de l'ascenseur.
6. Écrire une méthode **`action()`** qui modifie l'état de l'ascenseur en fonction de son état présent et de la collection de ses requêtes. Le comportement de cette méthode est décrit dans l'automate de la Figure 1.
   Cette fonction ne doit pas modifier l'état de l'ascenseur si celui-ci est bloqué. De plus, lors de l'ouverture des portes, toutes les requêtes concernant l'étage courant seront considérées comme réalisées.

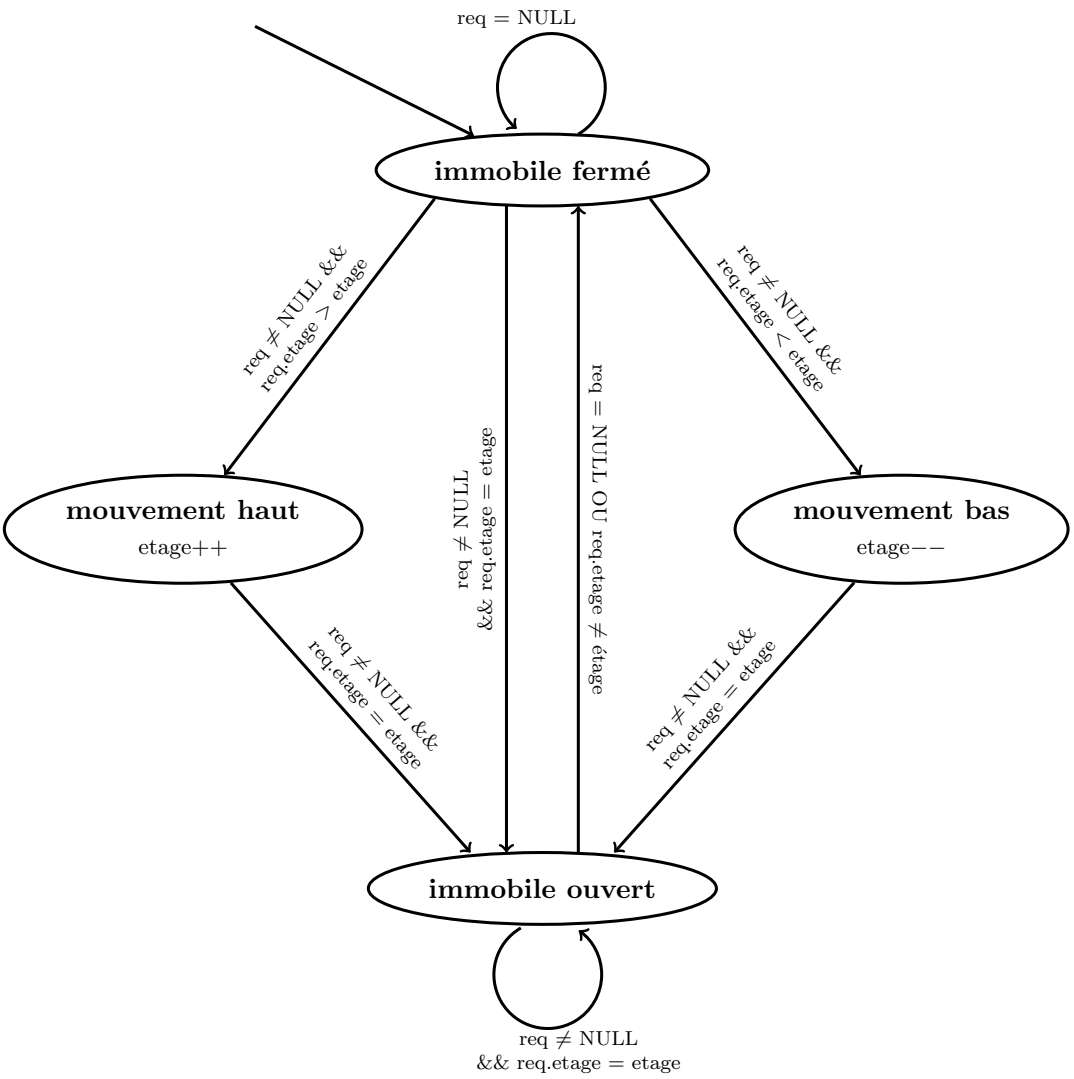

FIGURE 1 – Automate décrivant les transitions entre les différents états de l'ascenseur. Ici `req` est une requête attribuée à l'ascenseur et `etage` est l'étage actuel de l'ascenseur.

2.3 Contrôleur

Le contrôleur fait le lien entre les requêtes externes faites par les utilisateurs et les ascenseurs. Il se caractérise par une collection mémorisant les requêtes externes existantes et une collection incluant les ascenseurs dont il assure le fonctionnement.

1. Écrire la classe **`Controleur`**. Naturellement, le contrôleur est unique pour tous les ascenseurs. Les composantes qui doivent pouvoir y accéder, ont un moyen global et unique de le faire.
2. Écrire une méthode **`creerRequeteExterne()`** qui prend en paramètre un étage et une direction et qui ajoute une nouvelle requête externe à la collection des requêtes du contrôleur.
3. Proposer une solution permettant d'affecter toutes les requêtes externes existantes à des ascenseurs. Pour cela, il faudra écrire une méthode **`choisirAscenseur()`** qui permet, pour une requête donnée, de choisir l'ascenseur auquel elle sera affectée. Dans un premier temps, on supposera que l'algorithme d'affection est très naïf – l'ascenseur est choisi arbitrairement.
4. Comme indiqué, la méthode **`choisirAscenseur()`** écrite précédemment n'est pas optimale en terme d'affectation des requêtes aux ascenseurs. Par exemple, un ascenseur qui est en mouvement vers un étage peut récupérer des utilisateurs sur les étages intermédiaires. Donc parfois il serait peut-être mieux de pouvoir imposer à un ascenseur le traitement d'une

requête externe avant que l'ascenseur ait fini le traitement de ses requêtes internes. De multiples façons/critères d'optimiser le fonctionnement du système peuvent être envisagés. Proposez un (ou plusieurs) algorithmes alternatifs, qui permettront soit de mieux répartir les tâches entre les ascenseurs, soit de minimiser les déplacements qu'ils effectuent ... ou une version hybride de ces deux critères. Le choix d'un de ces algorithmes sera fait par l'utilisateur.

2.4 Visualisation du système – mode simplifié

Vous implémenterez un ensemble de vues permettant de représenter et modifier l'état interne du système. Votre programme devra en particulier intégrer les vues suivantes :

Vue 1 Une vue en coupe de l'immeuble, permettant de visualiser en temps réel l'état des ascenseurs (leur étage courant, porte ouverte/fermée, boutons allumés/éteints) et leur déplacements. Cette vue doit également intégrer les états des boutons (allumé ou éteint) de chaque palier.

Vue 2 Une vue interactive avec l'utilisateur. Elle regroupera l'ensemble de boutons du système *i.e.* tous les panneaux de contrôle des ascenseurs (leurs boutons internes) et les boutons de chaque palier.

Vue 3 Une vue montrant en temps réel les requêtes à satisfaire (internes et externes) associées à chaque étage.

Afin de ne pas retarder le travail, dans un premier temps, vos différentes vues seront représentées par des messages appropriés affichés dans le terminal.

2.5 Visualisation du système – IHM

(CETTE PARTIE EST À RÉALISER EN DERNIER)

Réalisez une interface graphique incluant les éléments suivants :

— Les trois vues présentées précédemment sous forme de fenêtre indépendantes.

— La vue 2 sera une vue interactive, c'est sur cette vue que les boutons seront "cliquables"

— Un menu d'application incluant : paramètres, documentation, à propos de, quitter l'application, etc.

— La gestion des erreurs d'affichage.