Transports En Commun

Ce document introduit l'application à développer au cours des prochaines séances. Vous partez de l'expression des besoins et d'une première spécification des classes à réaliser.

Comme dans tout projet, vous avez un travail pas toujours facile d'appropriation du sujet : beaucoup de questions et d'incompréhension et sans au début maitriser les aspects techniques. Ne négligez pas l'importance de ce travail (il faut fouiller).

Table des matières

1 Expression des Besoins

Le domaine de l'application est la simulation de la montée et la sortie des usagers d'un transport sur une ligne d'arrêts. Le caractère d'un usager définit deux sortes d'action :

- A la montée, il peut choisir dans le transport une place assise ou une place debout. Il peut aussi ne pas monter dans le transport (rester dehors).
- A chaque arrêt, il peut choisir de changer de place (assis en debout ou debout en assis) ou sortir du transport (même avant sa destination).

Lorsqu'un usager arrive à son arrêt de destination, il sort du bus.

Les arrêt sont ordonnés. Pour simplifier, un arrêt est représenté par un nombre entier positif.

Un usager est caractérisé par un nom et une destination. Un transport est caractérisé par un nombre maximal de places assises et un nombre maximal de places debout.

L'objectif n'est pas de développer l'application de simulation compléte mais un cadriciel ou "framework") qui permet :

- à des usagers de monter dans un transport à un arrêt,
- à un transport de déclencher le déplacement vers l'arrêt suivant,
- de définir des caractères différents d'usager,

— de fournir en cas d'erreur des informations sur le transport et l'usager à l'origine du problème.

Le développement se fait en deux étapes avec production d'un délivrable à chaque étape.

Le premier délivrable (environ 5 séances de 2 heures) est centré sur la réalisation des fonctionnalités en prenant un exemple particulier **PassagerStandard** et **Autobus** décrit par le client.

Le deuxième délivrable (environ 5 séances de 2 heures) est centré sur le paramétrage du "framework". Il s'agit de remanier le code obtenu au premier délivrable pour permettre la prise en compte des caractéres des usagers.

2 Première Spécification

L'ensemble des classes du "framework" appartient à un paquetage nommé `tec`. L'utilisation du "framework" est fourni par le client à travers un (ou plusieurs) programme principal qui n'appartient pas au paquetage `tec`.

Cette spécification est décrite par :

1. la documentation de départ du paquetage tec,
2. la classe Simple.java : un exemple d'utilisation du "framework" par le client. Cette classe correspond à un programme principale (voir la méthode **main**())
3. une documentation UML avec un diagramme de classe et trois diagrammes de séquence précisant l'enchaênement des envois de message du client, et des messages internes au paquetage à la montée dans le bus, puis à chaque arrêt.

Suivans le conseil : "Programmer avec des abstractions pas avec des réalisations".

Deux interfaces publiques au paquetage, **Usager** et **Transport** déclarent les méthodes utilisable par le programme principal. La construction interface correspond à la déclaration d'un type sans réalisation (assimilable à un type abstrait de données). Elles définissent les abstractions manipulées par le client.

Pour réaliser les fonctionnalités demandées, nous devons spécifier les interactions entre les usagers et le transport. Ces interactions vont être masquées au client.

Les interactions à la montée d'un usager et avant chaque arret sont définis par deux interfaces privées au paquetage : **Passager** et **Bus**.

Elles déclarent l'ensemble des méthodes utilisables à l'intérieur du paquetage. Comme les deux interfaces publiques , elles représentent un type abstrait (les abstractions manipulées à l'intérieur du paquetage).

La gestion d'erreur au niveau du client utilise l'exception contrôlée **UsagerInvalideException** définie dans le paquetage. La gestion d'erreur interne au paquetage utilise des exceptions non contrôlées.

Le développement doit suivre le processus suivant :

— adopter une démarche itérative dans la production du code,
— spécifier les tests du développeur pour chaque classe (ces tests font partie du délivrable),
— effectuer le développement en parallèle : la réalisation d'une classe se fait sans le code de l'autre.

2.1 PassagerStandard

La classe **PassagerStandard** correspond au caractère défini par les deux actions suivantes :

— à la montée dans un bus, chercher d'abord une place assise, sinon une place debout,
— à chaque arrêt vérifier pour sortir à son arrêt de destination.

Selon le diagramme de classe et l'exemple du client, **PassagerStandard** est une classe concrête publique au paquetage sous-type de **Usager** et **Passager**.

Son instanciation nécessite le nom de l'usager et sa destination.

Pour une instance de **PassagerStandard** :

- Pour représenter son état (assis, debout, dehors), elle utilise les instances de la classe **EtatPassager**.
- Les quatre méthodes **estAssis**(), **estDebout**(), **estDehors**() et **non**() donnent son état (accesseur).
- Les trois méthodes **accepterPlaceAssis**() **accepterPlaceDebout**() et **accepterSortie**() changent son état (modificateur).
- Les deux méthodes **monterDans**() et **nouvelArret**() codent les deux actions définissant le caractère de **PassagerStandard**. Elles interagissent avec une instance de la classe **Autobus**.

2.2 Autobus

La classe **Autobus** calcule le nombre de places occupées du transport. Les instances de **PassagerStandard** sont stockées dans un tableau.

Selon le diagramme de classe et l'exemple du client (Simple.java), c'est une classe concrête publique au paquetage sous-type de **Transport** et **Bus**.

L'instanciation nécessite le nombre maximal de places assises et le nombre maximal de places debout.

Pour une instance d'**Autobus** :

- Pour représenter le nombre de places assises ou debout, elle utilise les instances de la classe **JaugeNaturel**.
- Les deux méthodes **aPlaceDebout**() et **aPlaceAssise**() donne son état.
- Les cinq méthodes **demanderPlaceAssise**(), **demanderPlaceDebout**() **demanderSortie**(), **demanderChangerEnAssis**(), **demanderChangerEnDebout**() changent son état et interagissent avec une instance de **PassagerStandard**.
- La méthode **allerArretSuivant**() interagit avec tous les instances de **PassagerStandard** stockée.

2.3 Dépendances entre classes

Etudier le diaqramme de séquence de la méthode de la classe **PassagerStandard**.

⇒ *Quelles méthodes de la classe **Autobus** sont utilisées dans le code ce cette méthode ?*

⇒ *La méthode **demanderPlaceAssice**() de la classe **Autobus** utilise quelle méthode de la classe **Autobus** ?*

Noter la même dépendance dans le diagramme de séquence de la méthode *allerArretSuivant()*.

L'analyse impose un envoi de message demander/accepter ou changer/accepter entre les deux abstractions Bus et Passager.

Cette dépendance symétrique permet à l'abstraction Bus de contrôler les entrées-sorties (même si notre réalisation ne le fait pas ici).

Le code de la classe **Autobus** a besoin d'une instance de la classe **PassagerStandard** et inversement.

Les deux classes vont être développées en parallèle dans des répertoires différents. Du coup, il faut s'assurer que le code de chaque classe suit bien la spécification établie.

Par contre, la dépendance entre les classes posent un problème au niveau des tests. Comment tester le code de **PassagerStandard** sans avoir le code de **Autobus** (puisqu'il est en cours de développement). Et inversement.

Avoir des tests indépendants est obligatoire dans un développement en parallèle.

Nous verrons plus loin comment la technique des objets faussaires ("mock object") et le polymorphisme peuvent nous venir en aide.

3 Première itération : Tout doit compiler

Objectifs

Voici, les objectifs de cette itération :
— inspecter (compiler) les fichiers sources distribués ;
— ajouter les classes **EtatPassager**, **JaugeNaturel** et leurs tests dans le paquetage `tec` ;
— créer une version minimale des classes **PassagerStandard** et **Autobus**.
— produire la documentation "à la javadoc" de ces deux classes ainsi que de leurs méthodes de test
— produire une nouvelle version du modèle d'architecture du système (diagramme de classe).

Méthode de travail

— Approche TDD : L'objectif des tests du développeur est de montrer l'adéquation entre le code écrit par le développeur et la spécification fournie par le client. Le principe est d'écrire et d'exécuter ces tests en parallèle avec l'écriture du code.
— Répartir le travail : Un tandem prend en charge les réalisation autours de **PassagerStandard** et l'autre les réalisations autours de **Autobus**, en respectant la contrainte suivante : l'auteur de `jauge` s'occupe de **PassagerStandard** et l'auteur de `etatPassager` s'occupe de **Autobus**.

3.1 Le paquetage `tec`

Après extraction du fichier archive tec-source.zip dans votre répertoire de travail, vous trouverez :
— l'exemple du programme principale fourni par le client `Simple.java` ;
— et dans le répertoire tec, les fichiers sources de départ du paquetage `tec` :
  — deux interfaces publiques,
  — deux interfaces privées au paquetage,
  — la classe définissant l'exception contrôlée,
  — une classe permettant de lancer la suite des tests du développeur.

Remarque : La classe du programme client n'appartient pas au paquetage `tec`.

3.2 Compilons l'existant

Vérifier que tous les fichiers sources fournis sont compilés.

3.3 Ajouter les classes au paquetage

Dans la description du sujet, il est demandé d'utiliser :
— des instances de la classe **EtatPassager** dans la réalisation de la classe **PassagerStandard** ;
— des instances de la classe **JaugeNaturel** dans la réalisation de la classe **Autobus**.

La documentation indique que ces classes et leur classe de test sont des classes privées au paquetage `tec`.

Pour `PassagerStandard`, recopier dans son répertoire `tec` les fichiers pour la classe **EtatPassager**. Prenez la version qui passe tous les tests.

Pour `Autobus`, recopier dans son répertoire `tec` les fichiers pour la classe **JaugeNatuel**. Prenez la verion qui passe tous les tests. Modifiez les fichiers sources de ces classes, si besoins. Compilez et corrigez les erreurs.

3.4 Compilons les nouvelles classes

Version minimale de la classe PassagerStandard et Autobus.

Elle est créée de la manière suivante :
- La classe fait partie du paquetage `tec`.
- Elle est déclarée publique, pour être instanciée au dehors du paquetage (voir **Simple**).
- Elle réalise les bonnes relations de type/sous-type suivant le diagramme de classe.
- Son constructeur a un corps vide mais avec la bonne liste de paramètres.
- Ses méthodes ont un corps vide ou bien seulement une instruction de retour (`return false` pour une valeur booléenne et `return null` pour une valeur référence)
- Ajouter les annoations JavaDoc pour la production de la documentation

Corriger les erreurs jusqu'à compilation compléte de votre classe.

Version minimales de la classe PassagerStandardTest ou AutobusTest.

- Proposer quelques méthodes de test. Ses méthodes ont un corps vide.
- Ajouter les annoations JavaDoc pour la production de la documentation

3.4.1 Test d'intégration

Utiliser la classe **Simple** pour simuler un test d'intégration. Corriger les erreurs jusqu'à la compilation compléte des classes.

3.5 Fin de l'itération

Maintenant nous sommes assurés que tout compile et s'exécute. Pour terminer cette itération, vérifiez que les tests des classes **EtatPassager** et **JaugeNaturel** sont toujours OK.