Daix Théo

Dubrulle Allan

Verhoye Victor

Rapport d'implémentation

Génie Logiciel

Projet de modélisation et d'implémentation

Simulateur pour un distributeur de billets de train

Faculté des Sciences

UMONS Université de Mons

Enseignants : Mr Mens et Mr Hauweele
Année Scolaire 2017-2018

Table des matières

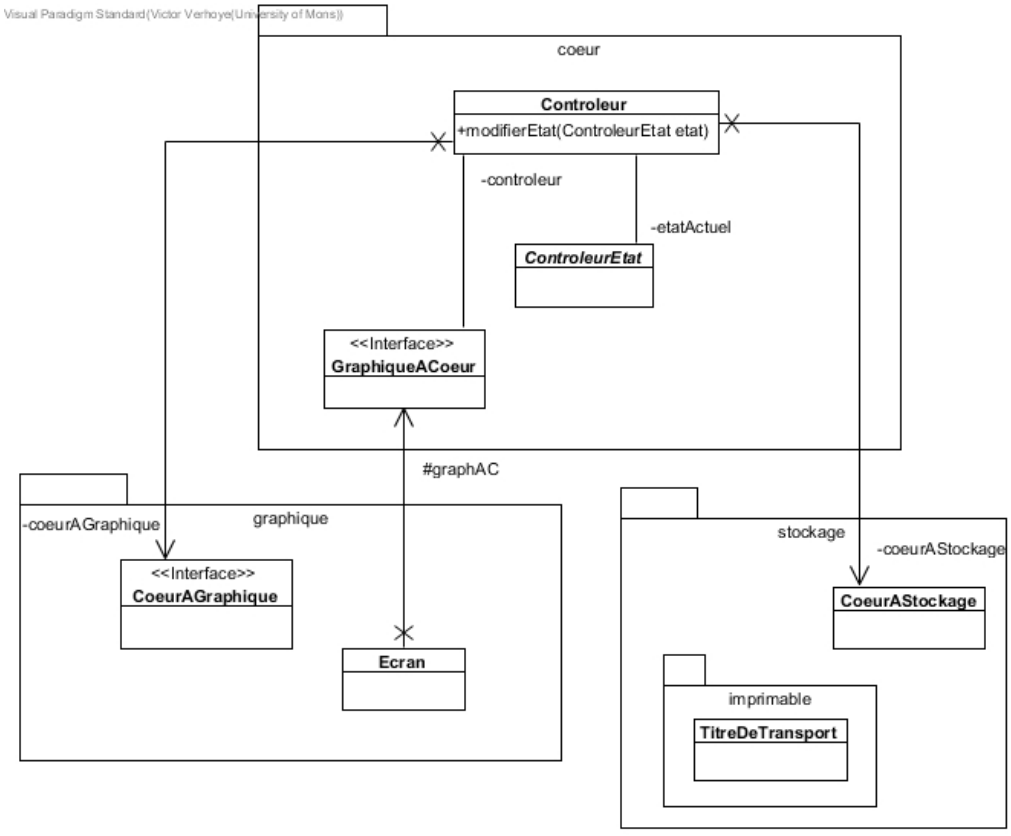
1 Introduction

Dans ce qui suit, nous allons parler des modifications apportées à notre application par rapport à ce qui avait été pensé lors de la modélisation, des design patterns utilisés et des problèmes connus de l'application. De plus, nous parlerons d'un choix de conception concernant le mode non tactile de l'écran.

2 Modifications apportées

Comme nous avons pu avoir l'occasion d'en parler avec Mr Hauweele, deux problèmes principaux ont mené à une quantité non-négligeable de modifications par rapport à notre modélisation : l'absence de design patterns et d'encapsulation entre la partie « cœur », la partie « stockage » et la partie « graphique » de notre application. Nous n'avions pas développé toute cette partie dans la modélisation de notre application car à l'époque où nous avons travaillé sur la modélisation, nous ne savions pas encore ce qu'étaient les design pattern. Concernant l'encapsulation, étant donné que c'était le premier projet d'informatique de notre parcours scolaire, nous manquions d'expérience et n'avions donc ni conscience de l'importance d'une bonne encapsulation, ni la bonne manière de s'y prendre.

Nous avons donc dû restructurer nos packages, et de ce fait la majorité des messages sont modifiés et ne correspondent plus à ceux indiqués dans les diagrammes de séquence. Nous avons tout de même essayé, tant bien que mal, d'être cohérent avec les méthodes qui existaient de base. Pour mieux comprendre la restructuration des packages, voici un condensé du nouveau diagramme de classe (celui-ci est bien sûr loin d'être complet, que ça soit au niveau des associations ou du nombre de classes) :



Voici une liste des modifications :

1 La méthode calculerPrix() prenait un titre de transport comme paramètre auparavant. Elle prend maintenant tous les paramètres d'un futur titre car, étant donné que le titre de transport est créé après que le distributeur ait connaissance de tous ses paramètres (afin de respecter l'encapsulation), on ne pourrait le passer en paramètre vu qu'il n'existe pas encore. Vu que le prix des différents titres est stocké dans des tables différentes en fonction du titre et que ces prix n'ont aucun lien, nous avons aussi du séparer calculerPrix() en calculerPrixAbo(), calculerPrixBillet() et calculerPrixPass().

2 Nous avons ajouté une classe monnayeur qui se charge de tout ce qui est rendu d'argent. Nous avons fait cela car, dû à notre problème d'encapsulation, une classe propre au graphique (FenetreSimulation) stockait des informations (nombre de pièces restantes, …), ce qui est le boulot du package stockage.

3 Pour les cartes bancaires, nous ne faisons plus une recherche de toutes les cartes que nous affichons, mais nous laissons plutôt à l'utilisateur la possibilité de taper le numéro de carte qu'il désire. Ensuite, nous vérifions dans la base de données si ce numéro existe bien. Si c'est le cas, on crée un objet Carte qui a pour paramètres les informations présentes dans la base de données.

4 Finalement, la classe PaiementLiquide n'avait pas suffisamment d'utilité et a donc été supprimée. Son seul argument montantRecu a été déplacé dans CoeurAStockageImpl (une des interfaces qui permet la communication du package cœur vers le package stockage).

5 Concernant les bases de données, nous avons décidé de séparer le travail en la gestion concernant les horaires de train (HoraireTrains), les titres de transport (BDDTitre) et la banque (BDDBanque). Vu qu'une partie de leur comportement est commun, la classe GestionBaseDeDonnées est devenue abstraite et contient tout ce qu'il leur est commun. De ce fait, toutes les méthodes qu'on pouvait trouver auparavant dans celle-ci sont réparties dans ses sous-classes. Certaines méthodes et arguments de GestionBaseDeDonnees (et ses sous classes) ne portent plus le même ou ont disparu :
   - infoHoraire() n'avait pas de sens car des méthodes équivalentes et plus détaillées se trouvaient dans HoraireTrains (et y sont toujours) ;
   - les méthodes infoCarte(), rechercheCartes(),… ont été remplacées en une seule méthode : infoCarte() ;
   - Dans HoraireTrains, nous avons supprimé tous les attributs. Ceux-ci n'avaient plus de sens car il était plus simple de faire passer ces attributs plutôt comme des paramètres des trois méthodes de la classe.

6 Afin de respecter l'idée d'un titre de transport unique, le nombre de titres désiré par l'utilisateur n'est plus un paramètre propre au titre lui-même.

7 Etant donné que nous n'avions pas bien structurer la modélisation, nous avons dû séparer et s'aider d'interfaces pour permettre aux packages de s'envoyer des messages. Dans notre modélisation, nous avions mis dans le Controleur des méthodes de choix telles que choixPass( ), choixAbonnement(),… qui permettaient à l'interface graphique de prévenir le Controleur des choix de l'utilisateur. Ces méthodes ont été déplacées dans l'interface GraphiqueACoeur (une des interfaces dont je parle en début de ce paragraphe), car c'est exactement le but de cette interface. Certaines méthodes de Controleur ont été modifiées (ou supprimées) en étant déplacées dans GraphiqueACoeur car elles n'étaient pas correctes (ou ne correspondaient plus aux besoins). Par exemple, la méthode choixAbonnement() a été séparée en choixRenouvAbo() et choixAchatAbo() (car ce sont deux boutons différents dans l'interface graphique).

8 Dans notre modélisation, certaines classes avaient une méthode qui s'appelait preparation(). A l'époque, nous en avions besoin car les instances de ces classes étaient créées, et ensuite nous utilisions preparation() afin de passer en paramètres les arguments de l'instance. Dans

l'implémentation, chaque objet créé a directement tous ses arguments en paramètre dans le constructeur. Nous n'en avons donc plus l'utilité.

9 Dans l'implémentation, la classe abstraite Composant est devenue une énumération de certain des composants de l'application. Notre idée de faire descendre les composants de la classe Composant n'était plus réalisable vu la restructuration en package que nous avons fait (par exemple, le composant Ecran, qui est purement graphique, n'aurait pas pu descendre de Composant, vu qu'elle se trouve dans un autre package). Cette énumération nous sert à présent à savoir si un composant optionnel est actif ou pas, à savoir si un composant est en panne,… (pour un exemple, voir en ligne 32 de EtatImpressionRecu ou en ligne 27 de EtatChoixTitre).

10 Dans la classe Imprimante, les méthodes imprimerTitre() et imprimerRecu() ont été rassemblées en une seule méthode imprimer(). La raison est que nous avons remarqué que le comportement était le même pour les deux (décrémenter le nombre d'impressions).

11 La classe Recu a été supprimée car elle ne correspond qu'à l'affichage des données déjà passées en paramètre d'un certain titre. Elle n'avait donc pas d'utilité.

12 Concernant le package « InterfaceGraphique » dans la modélisation, chaque classe à l'intérieur de celui-ci qui stockait des valeurs (principalement booléennes) se les sont vu retirées car le package graphique n'est pas censé stocker quoi que ce soit. La plupart de ces variables peuvent être à présent retrouvées dans le package stockage (par exemple, je parle du booléen tactile dans Ecran).

13 La méthode lancerSimulation() dans FenetreConfiguration a été supprimé car elle ne servait à rien. En effet, l'action de lancer la simulation est en fait juste l'action d'un bouton (le bouton « Valider »).

14 Dans FenetreSimulation, la méthode calculerRenduArgent() n'avait pas sa place vu que cette classe ne stocke rien (aucun calcul ne pourrait donc être fait). Elle a donc été déplacée dans la classe Monnayeur.

15 Dans Reception, les méthodes ouvertureTrappe() et rendreArgent() n'existent plus car elles correspondent juste à l'affichage d'un titre ou l'affichage d'un rendu, ce qui est géré par différentes classes de l'interface graphique directement.

16 Dans Ecran, nous avions mis des méthodes d'affichage afin de donner une idée d'où nous voulions aller. En réalité, vu notre encapsulation et vu que ces méthodes sont en fait des messages allant du cœur à l'interface graphique afin de le notifier des fenêtres à afficher, elles se trouvent maintenant dans la classe CoeurAGraphiqueImpl.

17 Nous avions mis pour variable d'instance des dates de titre de transport une instance de type Date, or cette classe comporte des méthodes dites « deprecated » (déconseillées). Vu que nous utilisions ces méthodes, nous avons décidé de plutôt nous tourner vers la classe LocalDate.

3 Justifications concernant les design patterns utilisés

1 State Design Pattern

Afin de représenter le comportement du distributeur, nous avons choisi d'utiliser des états. Le contrôleur (classe Controleur) du distributeur aura ainsi son état (attribut etatActuel) qui changera en fonction des actions de l'utilisateur. En fonction de l'état dans lequel le contrôleur se trouve, son comportement sera différent. Les changements d'état nous permettent aussi de demander à l'interface graphique (package interfaceGraphique) d'afficher ce qu'il faut quand il faut.

2 Singleton Design Pattern

Nous avons adopté ce Design Pattern pour plusieurs classes :

1 Controleur
2 Tous les états (les classes qui descendent de ControleurEtat)
3 GraphiqueACoeurImpl (permet à l'interface graphique de communiquer avec le cœur)
4 CoeurAGraphiqueImpl (permet au cœur de communiquer avec l'interface graphique)
5 CoeurAStockageImpl (permet au cœur de communiquer avec le package stockage)

6 FenetreConfiguration
7 FenetreSimulation

Nous avons adopté ce Design Pattern car :

- Pour le Controleur (1), vu que celui-ci comprend la machine à état, afin d'éviter les comportements étranges dus à plusieurs machines à état fonctionnant en même temps, nous avons créé le singleton afin de limiter ce nombre de machine à état à une.
- Concernant les états (2), ils sont munis d'un Singleton Design Pattern car ceux-ci servent à un State Design Pattern, or on sait que les états d'une machine à état requièrent l'utilisation singleton.
- La justification pour les points (3), (4) et (5) est que nous ne voulions pas risquer que la machine à état soit liée à plusieurs interfaces graphiques ou à plusieurs stockages.
- Pour les points (6) et (7), la raison du Singleton est tout simplement car nous voulions restreindre la classe à une seule instance.

4 Choix de conception concernant l'écran non-tactile

Concernant le mode non tactile de l'écran, on peut tout de même cliquer sur des zones de texte et bouger dans les fenêtres à l'aide de la souris. Nous avons laissé cette partie « tactile » de l'écran car c'était la seule solution pour avoir une idée de la position du pointeur (permettant d'indiquer où on se trouve dans la fenêtre). De même, afin de garder la fonctionnalité des menus déroulants, il est possible à l'aide de la souris de cliquer dessus pour voir le contenu de celui-ci. Mais, étant donné que le mode sélectionné est non-tactile, nous supposons que l'utilisateur n'est pas censé cliquer dessus.

5 Problèmes connus

1 Problème d'affichage en fonction de l'OS

Etant donné que nous avons adapté l'application à Windows (nous codions tous les trois avec ce système d'exploitation), certaines fenêtres (principalement l'affichage des titres de transport) ne s'affichent pas parfaitement sous Linux ou Mac.

2 Problème d'actualisation de la base de données

Lors de certaines actions telles que l'achat d'un abonnement ou le paiement par carte, certaines mises à jour sont effectuées sur la base de données. Malheureusement, ces mises à jour ne sont effectives que le temps d'une simulation du distributeur. Nous entendons par cela que si vous fermez et relancez l'application, les modifications apportées à la base de données auront disparues.