Compte-Rendu PLD Agile

Membres de l’hexanome :

Alexis Bosio

Liam Bette

Romain Declercq

Taki Eddine Mekhalfa

Tianjan Ye

Mathieu Leral

Clément Florant

Sommaire

…

Choix architecturaux

L’application développée avait pour contrainte de suivre l’architecture Modèle Vue Contrôleur (MVC). Nous avons donc opté pour la structure qui nous semblait la plus modulaire possible en séparant les fonctionnalités dans des packages et des classes différents pour garantir une bonne indépendance et une bonne réutilisabilité du code.

Concernant le modèle, nous avons choisi de séparer le traitement des données des données elles-mêmes. Nous avons ainsi un package Model.Metier qui comprend le stockage des données liées aux objets métier comme les livraisons ou le plan. Les traitements de ces données seront effectués par le package Model.Calculs. Malgré une certaine interdépendance créée entre ces deux packages, le code des classes des objets métier s’en trouve très allégé et bien plus lisible. De la même manière, nous avons le package Model.XMLHelpers qui se charge de toutes les opérations liées aux fichiers XML et construit les objets métiers grâce aux informations prises dans ces fichiers. La dernière partie du package Model représente l’interface avec le contrôleur. Le contrôleur possède une instance de la classe Planification qui lui sert à obtenir les informations concernant le modèle et à le modifier si nécessaire. La classe Planification encapsule donc via des méthodes toutes les modifications que le contrôleur a besoin de faire sur le modèle pour que le contrôleur n’ait pas à se soucier de comment faire ces modifications.

Pour simplifier le travail du contrôleur, nous avons choisi d’utiliser le design pattern Etat. Cela nous permet de savoir exactement où nous sommes à n’importe quel moment de l’exécution du programme, les boutons qu’il convient d’activer ainsi que les messages qu’il faut envoyer à l’utilisateur. Le code est ainsi facilement maintenable et très évolutif. Concernant les fonctionnalités Undo/Redo, nous utilisons le design pattern Commande et nous stockons les informations nécessaires dans une liste de commandes à chaque modification pour pouvoir restaurer l’état précédent une modification. Notre liste de commande contient donc des sauvegardes des états précédents et non uniquement les actions effectuées à chaque modification (ce qui aurait nécessité de réaliser l’opération inverse pour restaurer l’état précédent). L’utilisation du design pattern Etat facilite également l’implémentation de cette fonctionnalité.

Dans le cadre de la réalisation de la vue liée à notre modèle MVC, nous avons choisi de l’implémenter en plusieurs éléments, tous contenus dans l’unique package vue.

Pour partir de l’utilisateur, nous avons tout d’abord la classe « InterfaceGUI » qui se charge de la création de la fenêtre et du menu avec ses boutons. L’import de la classe du contrôleur permet ensuite de réagir aux actions de l’utilisateur sur ces boutons et de lancer les appels de méthodes correspondantes. Cette classe se charge aussi de gérer la désactivation des boutons grâce à des méthodes que le contrôleur se chargera ensuite d’utiliser. La saisie des différentes informations comme le nombre de livreurs et la durée d’une livraison est aussi mise en place au sein de cette classe qui se chargera de transmettre ces informations au contrôleur.

Ensuite, le reste de l’affichage est divisé entre deux classes héritant toutes deux de « Vue », qui leur permet de manipuler les « Observer » avec une méthode « update » et organise leur structure commune pour contenir une liste de tournées, un objet planification pour l’appel des éléments du modèle, un plan et une demande de livraisons. De ces deux classes, la première, « VueTextuelle » se charge de remplir et de permettre à l’utilisateur d’interagir avec le tableau sur le côté droit de l’écran alors que la seconde, « VueGraphique », comporte l’affichage concret du plan avec la demande de livraisons, les tournées et les modifications de celles-ci. Le dessin des tournées à proprement parlé se fait via les structures de données que nous avons choisies en utilisant des équivalences pour chaque élément.

Les rues seront manipulées par une classe « LineModifiee » qui se chargera d’affecter à une « Line » classique une couleur et un nom. Cette classe permettra également à l’utilisateur de sélectionner une rue via une mécanisme d’évènements ainsi que de changer ou d’obtenir le nom de la rue voulue.

Pour la représentation des livraisons sur l’affichage, nous avons utilisé la classe « CercleLivraison » qui nous permet, avec des attributs tels que l’heure de passage, la livraison associée et l’ordre de passage, de gérer cet aspect de l’application. En effet, des méthodes nous permettent de déterminer l’ordre de passage de cette livraison dans sa tournée et d’obtenir la livraison associée tandis que d’autres nous permettent de fixer la couleur et l’heure de passage correspondant à ce point du plan.

Et enfin, pour les nœuds du graphe et donc plus concrètement les intersections entre les rues, nous avons utilisés la classe « CercleIntersection » qui permet de joindre l’identifiant du nœud au cercle en question. En plus d’une méthode pour obtenir cet identifiant, cette classe dispose aussi d’une méthode pour changer l’affichage d’un nœud quand il est sélectionné au niveau de sa taille et de sa couleur.