POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

LOG2400 - Analyse et conception de logiciels

Discussion TP4 et TP5

Par Caouette, Olivier (2291747) et Vidal Herrera, Alonso ()

remise le:
03 décembre 2024

TP4:

Pour commencer l'application, nous avons décidé de séparer le pilote en plusieurs documents .h et .cpp pour diviser les classes et avoir une conception plus claire et facile à manipuler une classe à la fois. Comme nous avons besoin d'imprimer de façon itérative tous les éléments d'un voyage et que celui-ci est composé de segments, composés de plusieurs journées, celles-ci ayant plusieurs réservations, il a fallu créer un héritage. Cependant, un simple héritage aurait mis une grande responsabilité sur la classe Réservation, dans laquelle seraient stockés tous les attributs et méthodes des classes dérivées.

C'est pourquoi nous avons premièrement ajouté la classe AbstractReservation comme classe abstraite pour avoir une interface commune à toutes les classes dérivées de Réservation et pouvoir utiliser le polymorphisme sans affecter la structure de base. Nous avons ensuite créé une classe ReservationComposite qui, bien sûr, utilise le patron composite pour gérer de manière plus concise les classes arborescentes de Reservation. Elle permet d'utiliser une seule fonction pour que chaque classe puisse calculer son prix, ainsi qu'imprimer ses informations avec tous ces enfants, ce qui est l'objectif final. Pour ce faire, le composite définit les méthodes calculerPrix() à l'aide de getPrix() de la classe abstraite ainsi que la logique du accept() pour l'impression par le visiteur. Cette classe contient également l'instanciation d'une méthode usine pour créer des deep copies d'un objet à l'aide d'un autre objet et pouvoir les manipuler indépendamment. Les classes Journee, Segment et Voyage héritent donc du composite et non directement de AbstractReservation. Cela crée un couplage similaire, mais une meilleure cohésion.

Comme mentionné ci-haut, nous utiliserons les méthodes accept, associées à une classe visiteur comme logique d'impression des éléments. La classe VisiteurImprimeur nous permet d'avoir accès à la classe Reservation et ses dérivés sans pouvoir les modifier, ce qui garantit l'intégrité du code. Cette classe virtuelle pure est redéfinie dans la classe ImprimeurReservation, qui crée à son tour une implémentation spécifique au type d'objet mis en paramètre, que ce soit une seule réservation ou un voyage complet. Un seul appel sur un voyage pourra imprimer de façon récursive chaque composant de celui-ci et leurs composants, ainsi de suite, afin d'avoir les informations complètes du voyage d'un coup.

Le système MiniVoyage doit se servir d'une base de données de voyages offerts, leurs prix et autres informations pertinentes. La première étape à ce processus est une classe LecteurFichier qui englobe les méthodes servant à lire un document CSV et y extraire les informations pour les stocker dans une map de vecteurs. La méthode ObternirDonnees permet de distinguer les informations parmi les documents de vols, excursions ou hébergements et les obtenir complètement.

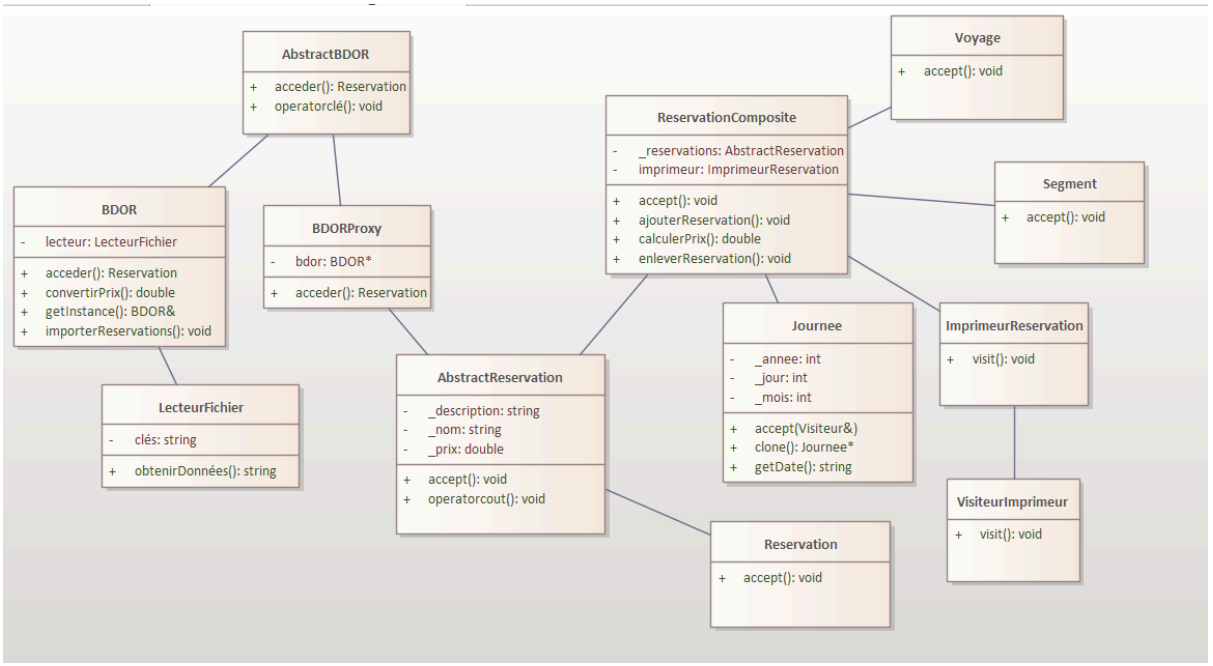
L'étape suivante à celle-ci est la base de données des offres de réservation, d'où la classe BDOR. Celle-ci est créée à partir des éléments lus par LecteurFichier et les garde en mémoire pour permettre l'accès à cette information par les autres classes. Nous avons décidé d'utiliser le patron singleton pour cette classe afin de limiter une utilisation superflue de

données en créant plus d'une base de données à la fois. Cela permet aussi un accès plus facile, car toutes les données se retrouvent dans une seule et unique instance. La fonction importerReservations se sert d'un LecteurFichier pour recueillir toutes les informations de tous les documents fournis pour les classer dans une map de vecteurs, convertis en réservations à l'aide du constructeur d'une réservation. Nous y avons aussi intégré la logique de changement de prix selon la devise associée pour tout avoir en CAD.

Pour accéder à cette base de données nous avons implémenté la classe BDORproxy, qui suit le patron de proxy. Cela permet d'améliorer la flexibilité, la sécurité, et l'efficacité du système tout en respectant l'interface de AbstractBDOR, qui, elle, définit la structure du BDOR dans une classe abstraite. Le proxy est celui qui sera instancié dans le main pour être utilisé comme base de données.

Les classes de Reservation, Journee, Segment et Voyage sont relativement simples, avec leurs constructeurs, getters et autres méthodes qui servent à certaines fonctionnalités durant l'exécution du code. La plupart de leurs fonctionnalités sont déjà établies dans le ReservationComposite.

Voici le diagramme de classe pour la partie du TP4:



Les complexités ne sont pas incluses par souci de clarté, mais les relations sont généralement 1 pour 1 à l'exception de l'arborescence des réservations, où il peut y avoir plusieurs réservations par journée, mais pas l'inverse, plusieurs journées par segments, mais pas l'inverse, etc. Les méthodes et attributs les plus importants sont inclus, mais plusieurs ne sont pas décrits aussi par souci de clarté et parce qu'ils n'agissent pas sur d'autres classes.

TP5:

Afin d'atteindre les objectifs de la partie 5, nous avons ajouté certaines classes et méthodes simples qui ne devraient pas affecter l'utilisation générale de l'application tout en bonifiant les manipulations possibles. La première classe ajoutée est l'ImprimeurLoggeur, qui permet un fonctionnement similaire à celui de l'imprimeurReservation, mais celui-ci permettra une sortie vers un nouveau fichier texte propre au voyage créé, soit un par personne. On utilise donc le même patron visiteur afin d'imprimer les informations dans un fichier externe sans accéder directement aux objets, gardant ainsi l'intégrité du code.

Une autre classe ajoutée est ReservationDecoree, un décorateur qui permet aux objets de type réservation de se comporter différemment de ce qu'a défini la classe abstraite des réservations. Notamment ici pour créer ou modifier des commentaires par rapport à une réservation existante. Le comportement de cette réservation sera la même, à l'exception des manipulations possibles sur le commentaire. Cela permet de garder la logique de base du code intacte, pour le moins possible affecter le code présent avant la mise à niveau. Une heure est prise en paramètre pour être affichée dans les logs et ainsi voir à quel moment chaque manipulation a été faite. Une classe abstraite AbstractReservationDecoree a aussi été créée afin d'instancier les fonctions qui seront utilisées dans la nouvelle classe de réservations.

Un loggeur est créé dans la classe BDOR afin de permettre l'ajout de commentaires sur une offre en particulier. Elle peut aussi être enlevée. La méthode changerPrix s'occupera de changer les prix des réservations stockées dans le BDOR de façon permanente, en changeant directement les valeurs de prix. Une nouvelle classe Rabais servira à créer des objets de rabais, qui changent le prix de l'offre et peuvent être retirés de la base de données au besoin. Cela permet un changement de prix temporaire et une gestion facile des rabais qui peuvent être mis sur les offres de réservation. Nous avons aussi implémenté un compteur pour connaître le nombre d'offres dans le BDOR.

Nous avons aussi changé la classe Reservation pour qu'elle hérite aussi de la classe composite. De cette façon, elle obtient les méthodes utiles aux manipulations de réservations au courant d'un voyage ou autre. Ainsi, nous pouvons ajouter et contenir une réservation dans une réservation et prendre en compte le prix total qui change en conséquence.