Discussion Design

Observateur : Ici, le patron de l'observateur est un moyen efficace d'assurer la cohérence entre les différentes vues du logiciel. Les notions d'observateur (Client ou Admin) et d'observable (classe de voyage) permettent de limiter le couplage entre les modules. Le patron permet aussi une gestion simplifiée d'observateurs multiples sur un même objet observable. Dans le cas de notre design logiciel, nous voyons l'avantage du patron de l'observateur lorsque nous voulons maintenir une cohérence entre la vue Client et la vue Admin. (ex: ajout de voyages par admin et immédiatement reflété dans la vue Client)

Visiteur : Le patron du visiteur effectue une opération sur les éléments d'une structure de données, sans changer les classes des éléments sur lesquels l'opération agit. Suivant notre design, il est possible de voir comment ce patron interagit avec nos classes de voyages. Nous priorisons l'utilisation d'un tel patron pour augmenter la cohésion de notre logiciel, puisque nous rassemblons les opérations de parcours de notre modèle interne avec ce dernier. De plus, le patron facilite l'ajout de nouvelles opérations à effectuer sur les éléments de notre structure. Il est cependant difficile d'ajouter de nouveaux éléments concrets à notre design,
car nous serions forcés de modifier tous les visiteurs.

Singleton : On se sert du patron singleton de concert avec le patron de Fabrique. Nos singleton correspondent aux différentes classes *Factory.* Premièrement, on se sert de ce patron afin de réduire le nombre d'objets en mémoire, ce qui augmente la performance de notre application. De plus, en le combinant avec le patron de fabrique, on fait en sorte que les utilisateurs n'aient pas accès directement aux méthodes des objets créés par les singleton, ce qui augmente la sécurité de notre application.

État : On se sert du patron d'État afin de faire changer le comportement des sièges en fonction de s'ils sont réservés ou non. De cette façon, on permet de rendre plus facile de *upscale* notre application en ajoutant des états. Par exemple, il serait facile de rajouter une classe de place, si on voulait un état de plus que libre, confirmée ou réservée. De cette façon, on applique le OCP, puisqu'il est facile de rajouter des états, sans avoir à modifier les classes de ceux existant.

Fabrique : Ici, on se sert du patron de fabrique dans plusieurs parties de notre application, soit pour la création de voyages, d'utilisateurs, de destinations et de compagnies. Pour toutes ces situations, le but est tout d'abord de faire respecter le DIP, en faisant en sorte que les classes de création concrètes dépendent des classes de type *abstractFactory* qui ne sont pas sujettes à changer. Du même coup, on applique aussi le OCP, puisqu'il est possible d'ajouter des fonctionnalités aux différentes *concreteFactory*, tout en s'assurant que les méthodes de bases du *abstractFactory* sont fermées aux modifications. Alors, on augmente la cohésion de notre application en plus de réduire le couplage.

Commande : Afin que la maintenance des actions effectuée par l'administrateur ne deviennent pas trop difficile à maintenir, nous avons implémenté le patron de commande. Ce dernier permet de séparer complètement le code initiateur de l'action, du code de l'action elle-même (découplage). De plus, nous soulignons l'avantage de l'utilisation de ce patron puisqu'une telle implémentation satisfait le principe OCP. Il est maintenant facile d'ajouter des nouvelles commandes sans complexifier les interactions dans notre code. De plus, ce patron rend l'application plus facile à utiliser en donnant les options de *undo* et *redo* aux utilisateurs.

Stratégie : Ici, on se sert du patron de stratégie afin de gérer le calcul du prix des cabines et des sièges. L'utilisation de ce patron rend facile d'upscale l'application dans le futur dans le cas où l'on voudrait ajouter de nouveaux types de sièges ou cabines, étant donné qu'il n'y aurait qu'à rajouter une classe de calcul de prix et de la passer comme calcul de prix au nouveau type. Ce faisant, on applique aussi le SRP, puisque les classes de calcul de prix ne s'en tiennent qu'à ça, au lieu d'avoir le calcul du prix dans les classes de siège et cabine, on améliore donc la cohésion de notre application. En utilisant ce patron on favorise également le CRP, puisqu'on favorise la composition en passant les classes de calcul aux sièges / cabines.

Itérateur : On se sert du patron de l'itérateur afin d'itérer sur les voyages disponibles. Ce faisant, on applique plusieurs principes SOLID. Premièrement, on applique le LSP, puisque l'itérateur fonctionne peu importe de quel type de voyage il s'agit. On applique aussi le OCP, puisqu'il est facile d'ajouter de nouvelles façons d'itérer à travers les listes de voyages tel que parcourir en ordre de prix, par colonne, rangés, etc. De plus, l'itérateur n'a pas accès aux méthodes du conteneur, soit la liste de voyage, étant donné qu'il ne connait qu'un voyage à la fois, on favorise donc la dissimulation de l'information, ce qui augmente la sécurité de notre application.