Félix-Olivier Latulippe et Charles-Olivier Lefebvre
CONCEPTION ET DÉVELOPPEMENT ORIENTÉS OBJET
420-C46-JO
Groupe : 01

Rapport Final TP-02

Travail présenté à :
Charles Jacob
Département d'informatique
CÉGEP Régional de Lanaudière à Joliette
26/05/2023

Table des matières

Générateur

Dictionnaire:

Lien vers doxygen

Lien vers executable

Commentaire sur GRASP

Dans notre générateur de scénarios, nous avons respecté les normes de programmation GRASPS (General Responsibility Assignment Software Patterns). Cela nous a permis de concevoir un code clair et modulaire.

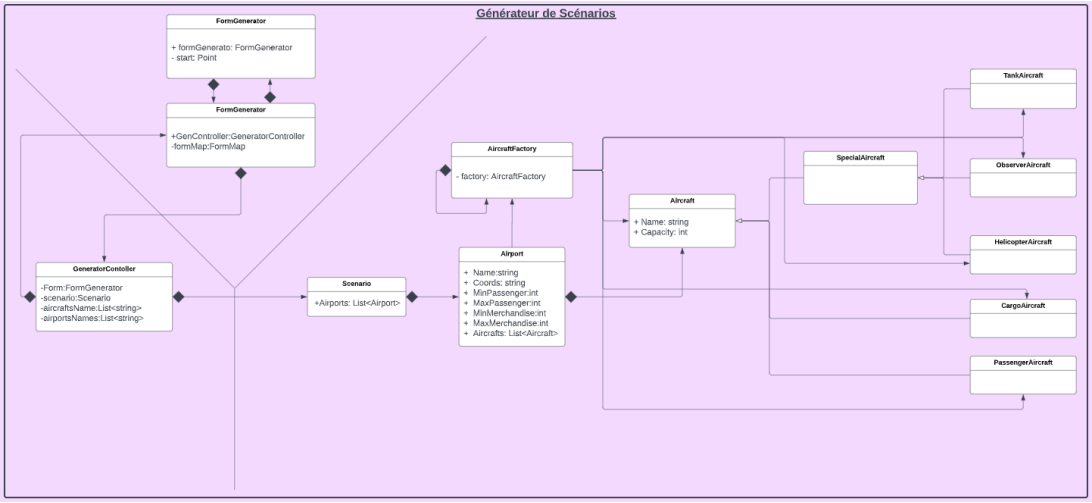
Nous avons utilisé la séparation des préoccupations pour répartir les responsabilités entre les modules du générateur. Le pattern du créateur a été appliqué pour gérer la création des éléments du scénario. Des classes de contrôle ont coordonné la génération et maintenu la cohérence. Les classes expertes ont été chargées des connaissances spécifiques, tandis que la haute cohésion a favorisé des classes fonctionnelles bien définies.

Ces efforts ont abouti à un générateur de scénarios structuré et extensible. Le code est facile à maintenir et à personnaliser, et les différentes parties du générateur sont clairement délimitées.

Patrons GOF

- *MVC* : GeneratorController, FormGenerator, Scenario. L'application de ce concept améliore considérablement la portabilité du code.
- *Singleton*: AircraftFactory. Nous avons implémenté le singleton sur cette fabrique pour éviter d'avoir plus d'une fabrique.
- *Fabrique* : Nous avons mis en place une fabrique avec les clients afin d'éviter l'instanciation de clients dans différentes classes.

Classe UML



Simulateur

Dictionnaire:

Lien vers doxygen

Lien vers executable

Commentaire sur GRASP

Dans le cadre de notre projet de simulateur de vols, nous avons déployé d'importants efforts pour respecter les normes de programmation GRASPS (General Responsibility Assignment Software Patterns). Ces normes ont été conçues pour promouvoir une conception logicielle claire, modulaire et maintenable.

Tout d'abord, nous avons veillé à appliquer le principe de la séparation des préoccupations (Separation of Concerns) en répartissant les responsabilités de manière appropriée entre les différents composants du simulateur. Cela nous a permis de garantir que chaque module est responsable d'une tâche spécifique et n'a pas de dépendances inutiles. Par exemple, nous avons créé des classes distinctes pour la gestion des entrées utilisateur, la simulation physique du vol, l'affichage graphique, etc.

Ensuite, nous avons utilisé le pattern GRASP du créateur (Creator) pour gérer la création des objets. Nous avons veillé à ce que les classes qui créent des instances d'autres classes soient responsables de cette tâche. Cela a favorisé un découplage des objets et une meilleure flexibilité lors de l'instanciation de nouvelles classes.

Un autre pattern GRASP que nous avons appliqué est le pattern du contrôleur (Controller). Nous avons créé des classes de contrôleur qui sont responsables de l'orchestration des actions et de la coordination entre les différents composants du simulateur. Cela a permis de réduire la complexité et d'améliorer la lisibilité du code.

Par ailleurs, nous avons utilisé le pattern GRASP de l'expert (Expert) pour assigner les responsabilités aux classes qui possèdent les informations nécessaires pour les accomplir. Par exemple, nous avons confié à une classe spécifique la responsabilité de calculer les performances aérodynamiques des avions, car elle avait toutes les connaissances requises dans ce domaine.

Enfin, nous avons appliqué le pattern GRASP de la haute cohésion (High Cohesion) en regroupant les fonctionnalités similaires dans des classes spécifiques. Cela a permis d'améliorer la modularité et la réutilisabilité du code, car chaque classe était concentrée sur une seule responsabilité bien définie.

Grâce à ces efforts déployés pour respecter les normes de programmation GRASPS, notre projet de simulateur de vols bénéficie d'une structure solide, d'un code maintenable et évolutif. Nous avons pu réduire la complexité, améliorer la flexibilité et faciliter la collaboration au sein de notre équipe de développement.

Patrons GOF

- *MVC* : COTAI, FormSimulator, Scenario. L'application de ce concept améliore considérablement la portabilité du code.
- *Fabrique* : Client. Nous avons mis en place une fabrique avec les clients afin d'éviter l'instanciation de clients dans différentes classes.
- *États* : Scenario et Avion. Nous avons introduit la gestion des états dans ces classes afin de détecter les changements d'état et d'automatiser le déroulement du simulateur.
- *Façade* : Scenario. Nous avons mis en œuvre une façade sur le scénario afin d'assurer une cohésion plus forte et de rendre le code plus modulaire.

Classe UML