HEIG-VD — POA

Laboratoire 2 – Rapport

[redacted]

17 avril 2024

1 Introduction

Dans ce laboratoire, nous allons modéliser un squadron, qui représente une escadrille de plusieurs vaisseaux. Un squadron doit fournir plusieurs fonctionnalités, telles que l'ajout et la suppression de vaisseaux, la détermination de la consommation en carburant et la gestion du commandement. Notre but est d'avoir un outil pour déterminer le rayon d'actions de nos escadrilles ou la quantité de carburant nécessaire pour qu'elles puissent effectuer des opérations.

2 Modélisation

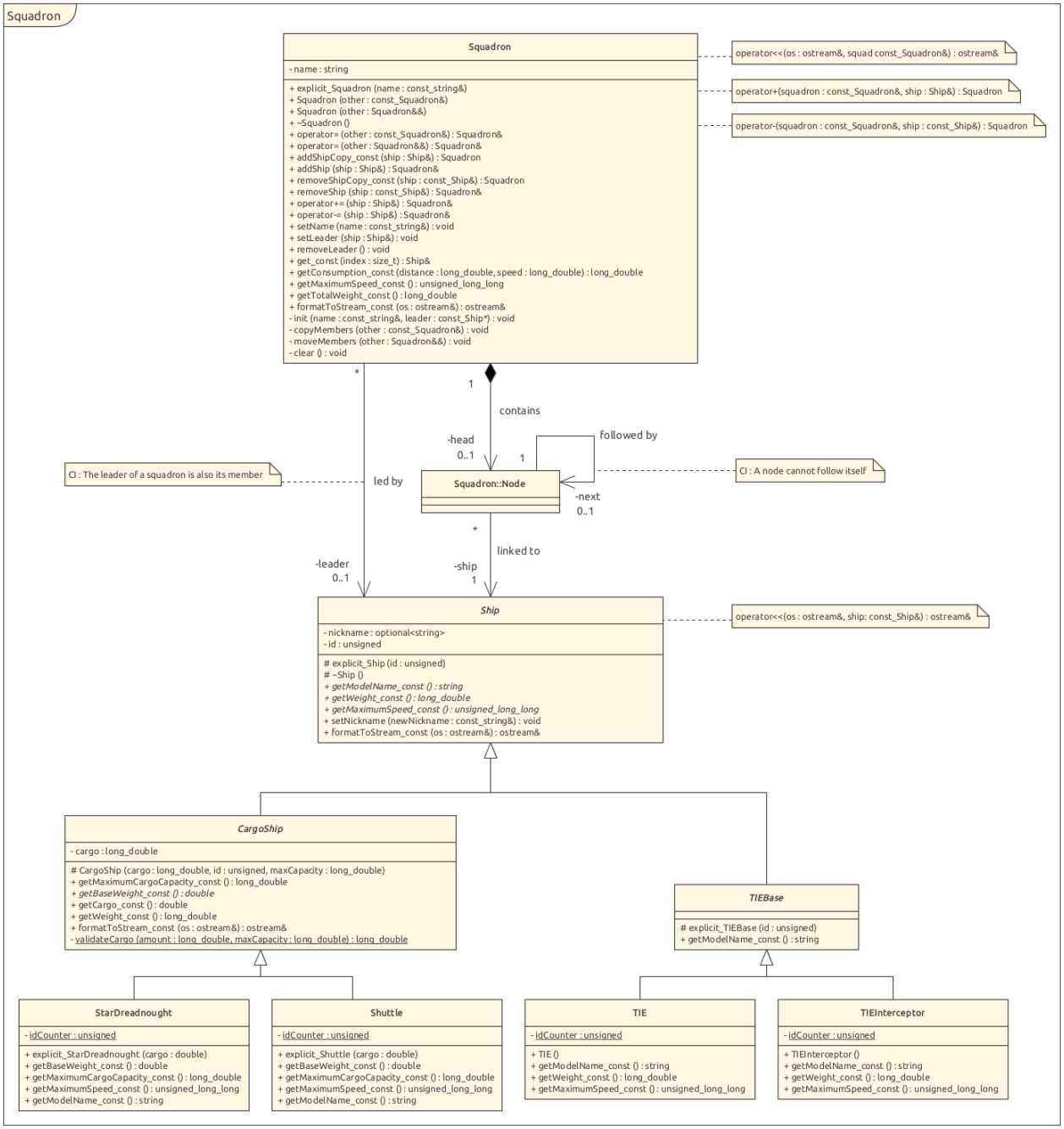

Figure 1 – Diagramme de classes de notre implémentation

3 Instructions de compilation

Nous avons conservé la structure avec *cmake* pour notre implémentation.

Un exécutable additionnel a été défini pour nos tests et qui nécessite la librairie de tests unitaires googletest. Nous avons utilisé la version « live at head », c'est-à-dire la révision n° 110 de la `1.14.0` au commit `eff443c6` pour compiler et exécuter nos tests. Il faut toutefois noter que ce système est optionnel et que si la librairie n'est pas trouvée par *cmake* l'exécutable des tests ne sera pas généré.

Nos versions de la suite de compilation utilisée par *cmake* sont les suivantes :

- **cmake** : 3.29.2
- **ninja** : 1.11.1
- **gcc** : 13.2.1 20230801
- Compilé sur Linux avec le kernel 6.7.12

4 Choix et hypothèses de travail

4.1 Ships

4.1.1 Héritage

Les différents types de vaisseaux sont représentés par un héritage. La classe `Ship` est la classe abstraite de base, qui définit les attributs et méthodes communs à tous les vaisseaux. Les classes `TIEBase` et `CargoShip` sont des classes abstraites intermédiaires qui héritent de `Ship` et qui ajoutent des attributs et méthodes sp écifiques à chaque type de vaisseau. Finalement, les classes `Shuttle` et `StarDreadnought` qui héritent de `CargoShip` et les classes `TIE` et `TIEInterceptor` qui héritent de `TIEBase` sont les classes concrètes qui implémentent les méthodes abstraites de leurs classes parentes.

4.1.2 Identifiants

Chaque vaisseau doit avoir un identifiant qui dépend de son type. Pour cela, nous avons créé un attribut statique `nextId` dans chaque classe de vaisseau. Lorsqu'un vaisseau est créé, son identifiant est initialisé à la valeur de `nextId` et `nextId` est incrémenté. Nous avons décidé de gérer cette initialisation à travers les constructeurs des classes de vaisseau. Ainsi, nous avons créé un constructeur protégé dans chaque classe de vaisseau abstraite qui prend en paramètre un `int` qui est l'identifiant du vaisseau. Ces constructeurs sont ensuite appel és par les constructeurs des classes concrètes avec la valeur de `nextId`.

4.1.3 Propriétés

Chaque vaisseau a un certain nombre de propriétés dont la valeur dépend de son type. Le propriétés concernées sont la vitesse maximale, le poids, le nom du modèle et le chargement maximal pour les vaisseaux de type `CargoShip`. Afin d'éviter de stocker ces valeurs dans chaque instance de vaisseau, nous avons décidé de créer des getters virtuels dans la classe `Ship` pour les trois premières propriétés et dans la classe `CargoShip` pour la dernière. Ces getters sont ensuite implémentés dans les classes concrètes et retournent simplement la valeur de l'attribut qui correspond à leur type.

4.1.4 Charge Maximale

Les vaisseaux de type cargo peuvent transporter une cargaison. Il est important de vérifier que la cargaison ne dépasse pas la charge maximale du vaisseau. Pour cela, nous avons ajouté un attribut `maxCargoCapacity` dans le constructeur de la classe abstraite `CargoShip` qui est ensuite utilisé pour vérifier que la cargaison ne dépasse pas la charge maximale du vaisseau. Les sous-classes de `CargoShip` appellent ce constructeur en passant leur charge maximale, ainsi, la vérification est faite automatiquement.

4.1.5 Nickname

Chaque vaisseau peut avoir un surnom mais ce n'est pas obligatoire. Pour représenter cette propriété, nous avons décidé d'utiliser un attribut `std::optional<std::string>`. Ainsi, un vaisseau peut avoir un surnom ou pas. Nous avons décidé de ne pas utiliser un pointeur pour stocker le surnom car cela aurait pu amener à des problèmes de gestion de la mémoire et l'implémentation de la "règle des trois", ce qui ne nous semblait pas nécessaire pour un simple surnom.

4.2 Squadron

4.2.1 LinkedList

Pour stocker les vaisseaux d'un squadron, nous avons décidé d'utiliser une liste chaînée. C'est la classe `squadron` elle-même qui implémente la liste chaînée. Nous avons créé une structure interne privée `Node` qui contient un pointeur vers le vaisseau et un pointeur vers le noeud suivant. La classe `squadron` contient un pointeur vers le premier noeud de la liste, appelé `head`. Nous avons décidé de stocker directement la taille de la liste comme attribut de la classe `squadron`, afin d'éviter de devoir itérer sur l'ensemble de la liste pour la calculer à chaque fois. Nous avons également décidé de ne pas stocker de pointeur vers le dernier noeud de la liste. En effet, cela aurait pu être utile pour ajouter des éléments en fin de liste efficacement, mais comme nous n'avons pas d'itérateur à disposition, l'insertion en fin de liste nécessite de toute manière de parcourir la liste. Ainsi, avoir un pointeur vers le dernier noeud n'aurait pas apporté de gain de performance, en plus de demander une gestion supplémentaire.

4.2.2 Opérateurs

Afin de garantir une bonne encapsulation et d'éviter l'utilisation du mot-clé `friend`, nous avons décidé de surcharger les opérateurs `+`, `-` et l'opérateur de flux à l'extérieur de la classe `squadron`. Pour cela, nous avons créé des méthodes publiques dans la classe `squadron` qui définissent le comportement attendu pour chaque opérateur. Ces méthodes sont ensuite appelées directement dans les surcharges des opérateurs. De manière générale, nous avons toujours créé une méthode auxiliaire pour chaque opérateur qui effectue le travail et qui est ensuite appelée par l'opérateur. Cela permet de garantir une bonne encapsulation et de faciliter la lecture du code.

4.2.3 Ajout et Suppression de Vaisseaux

Les méthodes d'ajout et de suppression de vaisseaux sont implémentées en suivant la manière typique de manipuler une liste chaînée. Pour ajouter un vaisseau, nous créons un nouveau noeud, nous le plaçons en tête de liste et nous mettons à jour le pointeur `head`. Pour supprimer un vaisseau, nous parcourons la liste jusqu'à trouver le vaisseau à supprimer, nous mettons à jour les pointeurs du noeud précédent et du noeud suivant et nous supprimons le noeud.

4.2.4 Copie de Vaisseaux

Pour copier les vaisseaux d'un squadron, nous avons décidé de créer une méthode `copyMembers` qui prend en paramètre un squadron et qui copie les vaisseaux de ce squadron dans le squadron courant en les ajoutant un à un avec la méthode `addShip`. Comme `addShip` crée un nouveau noeud pour chaque vaisseau, cela implique qu'une des allocations de mémoire peut échouer. Dans ce cas, il ne faut pas que le vaisseau reste dans un état incohérent, i.e. avec seulement une partie des vaisseaux copiés. Pour éviter cela, un try catch est utilisé pour attraper les exceptions et supprimer les vaisseaux déjà copiés. Ainsi, si une exception est levée, le squadron reste dans l'état initial. L'exception est ensuite relancée pour que l'appelant puisse la traiter.

5 Tests effectués

Certains tests ont été effectués à la main avec le programme d'exemple et quelques modifications pour vérifier l'état de la mémoire avant et après les traitements. Nous avons en outre implémenté des tests unitaires au moyen de la librairie googletest, dont le détail sommaire est le suivant :

Cas de test	**Cas évalués**	**Résultat attendu**
Calcul de consommation	Calcul de la consommation totale	Calcul précis de la consommation
	Paramètres invalides	Erreurs générées pour entrées invalides
Ajout de vaisseaux	Ajouter différents vaisseaux	Description correcte après ajout
	Ajout de doublons	Pas de doublons dans la description
Suppression de vaisseaux	Suppression de différents vaisseaux	Description correcte après suppressions
	Suppression d'un vaisseau inexistant	Aucun changement dans la description
Accès aux vaisseaux	Accès par position	Description correcte des vaisseaux
	Accès hors limites	Erreur pour indice hors limite
Opérations de copie	Copie et modification de l'escadron	Indépendance et exactitude de la copie
Définition du leader	Assignation et retrait du leader	Gestion correcte du leader
Définition du nom	Changement du nom du squadron	Le nouveau nom est affiché
Validations de cargo	Validation des capacités de cargo	Erreurs pour valeurs hors limites, aucun problème pour valeurs valides

Table 1 – Cas de tests et résultats attendus

6 Conclusion

En conclusion, ce laboratoire nous a permis de créer une solution robuste pour la modélisation et la gestion des escadrilles de vaisseaux spatiaux. En développant la classe Squadron et les classes associées pour différents types de vaisseaux, nous avons pu mettre en œuvre des fonctionnalités essentielles telles que l'ajout et la suppression de vaisseaux, le calcul de la consommation en carburant en fonction de divers paramètres, et la gestion du commandement.

L'utilisation de concepts avancés de programmation orientée objet nous a permis de structurer notre code de manière modulaire, facilitant ainsi sa maintenance et son extension. De plus, en respectant les principes de la règle des cinq pour la classe Squadron, nous avons assuré une gestion appropriée des ressources et une prévention des fuites de mémoire.