Labo 2 - Squadron

POA

Walid Slimani et Timothée Van Hove

12 avril 2024

Table des matières

Introduction

Le but de ce laboratoire est de définir une classe `Squadron` représentant une escadrille de plusieurs vaisseaux. Une escadrille possède un nom et éventuellement un vaisseau chef d'escadrille. Chaque vaisseau de l'escadrille est d'un modèle donné et peut avoir un surnom spécifique.

Depuis que nous avons découvert que les vaisseaux pouvaient avoir des problèmes d'approvisionnement en carburant, il nous faut un outil pour déterminer le rayon d'actions de nos escadrilles ou la quantité de carburant nécessaire pour qu'elles puissent effectuer des opérations. Nous supposerons qu'il est possible de déterminer la consommation d'un vaisseau dans une situation normale selon la formule:

$$consommation = \frac{\sqrt[3]{poids}}{2} \times log_{10}(poids \times vitesse) \times log_{10}(distance + 1)$$

Compilateur et options de compilation

Compilateur : `g++ version 12.2.0`

Version c++ : `17`

Build systems : `cmake 3.23.2` et `ninja 1.10.1`

Options de compilation : `-Wall -Wextra -Wpedantic -Wconversion -Wsign-conversion -Wvla -Werror`

Compiler et lancer le programme

note: ninja et cmake doivent être installés sur votre machine

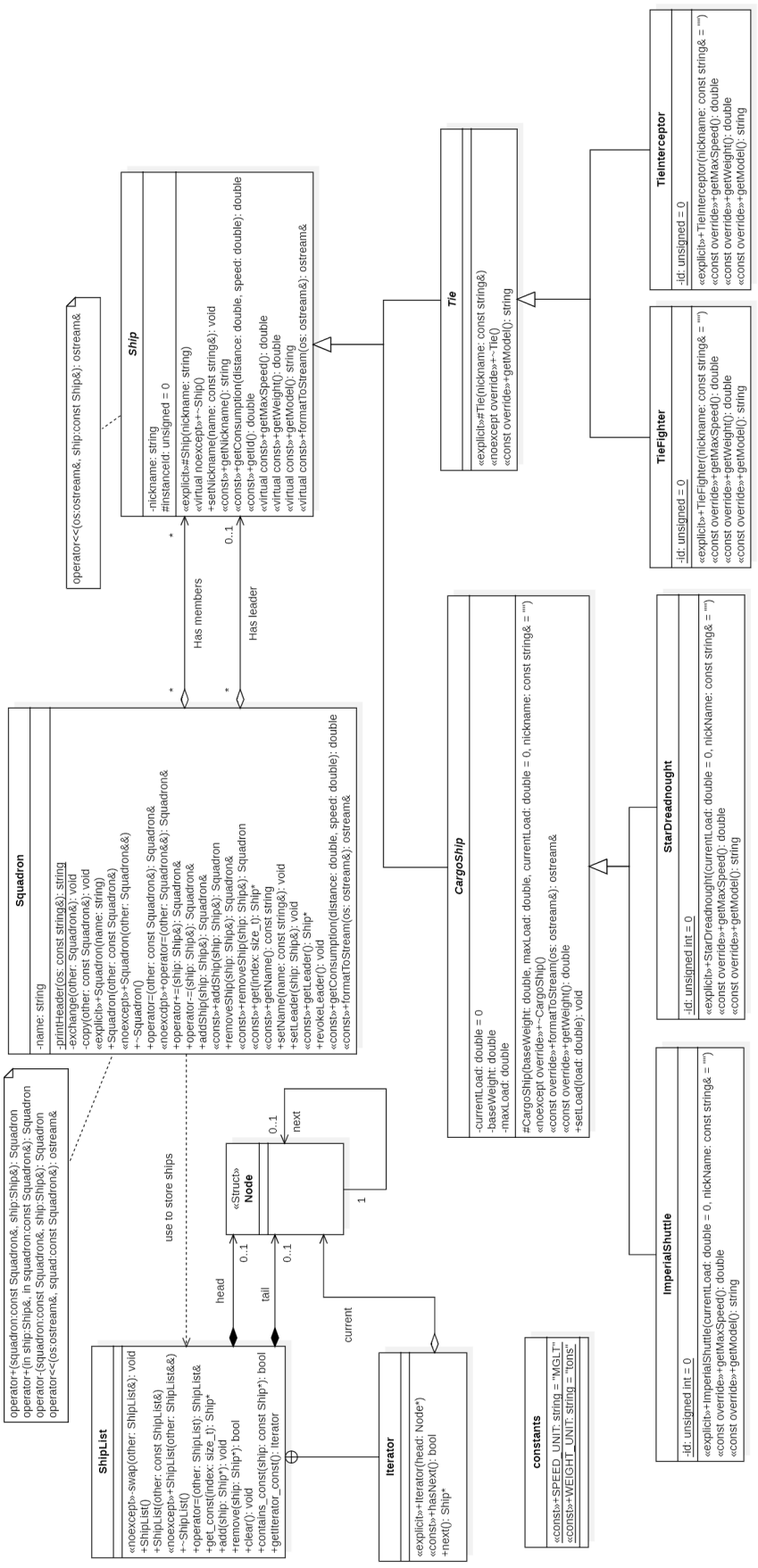
Windows

```
# Depuis la racine du projet
New-Item -Path build -ItemType Directory -Force; cd build; cmake .. -G "Ninja"; ninja;
↪  .\squadron.exe; cd ..
```

Linux

```
# Depuis la racine du projet
mkdir -p build && cd build && cmake .. -G "Ninja" && ninja && ./squadron && cd ..
```

Diagramme de classe



Choix d'implémentation

Ship et classes dérivées

La classe `Ship` sert de classe de base abstraite, fournissant les attributs et méthodes communs à tous les vaisseaux. Elle implémente aussi l'opérateur de flux `<<`.

La classe `Tie` encapsule les propriétés communes aux vaisseaux de type TIE, offrant une méthode pour obtenir une partie du modèle de vaisseau. Les classes concrètes `TieFighter` et `TieInterceptor` héritent de `Tie` et définissent les attributs spécifiques à chaque modèle.

La classe `CargoShip` est une spécialisation de `Ship` pour les vaisseaux cargo. Elle ajoute des méthodes pour définir et obtenir la charge actuelle tout en s'assurant que celle-ci reste dans les limites spécifiées. Les classes `ImperialShuttle` et `StarDreadnought` héritent de `CargoShip` et définissent les attributs spécifiques à chaque modèle.

ShipList

Pour stocker les vaisseaux dans `Squadron`, nous avons implémenté une liste simplement chaînée custom qui permet une gestion simple des vaisseaux et un ajout / suppression sans besoin de redimensionner ou déplacer en mémoire les éléments. L'utilisation d'une liste dédiée permet de découpler la collection de la classe, rendant `Squadron` flexible à la modification si la classe venait à évoluer (par exemple si nous voulions lui ajouter d'autres collections de vaisseaux).

Nous avons choisis d'implémenter un itérateur dans `ShipList` comme moyen standard de parcourir les éléments de la liste sans exposer la structure interne. ça permet d'encapsuler la logique de navigation et améliore la sécurité du code en empêchant l'accès direct aux nœuds de la liste.

Nous avons choisi le `copy-and-swap idiom` pour l'opérateur d'affectation car il simplifie la gestion de la mémoire. ça réduit le code dupliqué (un seul opérateur d'affectation), élimine le besoin de gérer explicitement les auto-affectations et garantit que notre objet est dans un état consistant même en cas d'exception. On passe l'objet `par valeur` dans l'opérateur d'assignation, ce qui appelle le constructeur de copie ou de déplacement selon le contexte (voir ce post).

Squadron

La classe `Squadron` utilise une composition avec la classe `ShipList` pour stocker les vaisseaux, et un pointeur sur le chef d'escadrille. En plus des méthodes demandées dans les instructions du labo, nous avons choisi de mettre à disposition les méthodes `getLeader` et `getName` pour nous aider dans les tests unitaires.

Note : `Squadron` ne prend pas en charge la gestion du cycle de vie de ses vaisseaux. Si un vaisseau est créé dynamiquement et ajouté à l'escadron, sa destruction externe entraînera un dangling pointer au sein de l'escadron (de la liste). Pour pallier ce problème, l'utilisation de `std::shared_ptr` pour gérer les vaisseaux dans la liste aurait été une solution, mais nous ne les avons pas encore vus en cours.

Tests

Pour assurer la fiabilité de notre implémentation, nous avons développé une suite de tests unitaires en utilisant le framework Google Test. Ces tests sont situés dans les fichiers `src/test/ShipListTest.cpp`, `src/test/ShipTest.cpp`, `src/test/SquadronTest.cpp`. Nous avons manuellement testé la sortie sur la console de l'opérateur de flux `<<` pour vérifier que la sortie console corresponde aux instructions fournies.

Compiler et lancer les tests

Windows

```
# Depuis la racine du projet
New-Item -Path build -ItemType Directory -Force; cd build; cmake .. -G "Ninja"; ninja; .\tests.exe;
↪  cd ..
```

Linux

```
# Depuis la racine du projet
mkdir -p build && cd build && cmake .. -G "Ninja" && ninja && ./tests && cd ..
```

Récapitulatif des tests unitaires

ShipList tests

#	Description	ok/nok
1	Vérifie que le constructeur par défaut d'une ShipList fonctionne correctement	ok
2	Vérifie que le constructeur de copie crée effectivement une copie d'un autre objet	ok
3	Vérifie que l'opérateur d'affectation copie effectivement l'objet quand nécessaire	ok
4	Vérifie que le constructeur de déplacement transfère la propriété à la nouvelle instance	ok
5	Vérifie que l'opérateur d'affectation de déplacement fonctionne correctement	ok
6	Vérifie que l'auto-assignation d'une ShipList à elle-même est une opération sans effet	ok
7	Vérifie qu'après un déplacement, une ShipList est laissée dans un état valide mais non spécifié	ok
8	Vérifie que les vaisseaux peuvent être correctement retirés de la ShipList	ok
9	Vérifie que la méthode get récupère correctement les vaisseaux de la ShipList	ok
10	Vérifie que la ShipList peut être vidée correctement, laissant la liste vide	ok
11	Vérifie qu'un itérateur peut être obtenu d'une ShipList et qu'il pointe initialement sur le premier élément si la liste n'est pas vide	ok
12	Vérifie que l'itérateur itère correctement sur tous les éléments	ok
13	Vérifie que les méthodes constantes peuvent être utilisées sur une ShipList constante, assurant des opérations en lecture seule	ok
14	Vérifie qu'une ShipList constante peut être itérée sans être modifiée, testant la correction constante du processus d'itération	ok
15	Vérifie qu'ajouter un vaisseau nul à la liste lance l'exception attendue	ok
16	Vérifie le comportement lors de la tentative de retrait d'un vaisseau qui n'est pas dans la liste	ok
17	Vérifie que l'accès à un élément dans une liste vide lance une exception	ok
18	Vérifie le comportement d'un itérateur obtenu à partir d'une liste vide	ok

Ship tests

#	Description	ok/nok
1	Vérifie que chaque instance de classe dérivée de Ship reçoit un ID séquentiel unique lors de la construction.	ok
2	Vérifie que le constructeur de Ship attribue correctement un surnom.	ok
3	Vérifie que la méthode setNickname change correctement le surnom.	ok
4	Vérifie que getConsumption avec des paramètres d'entrée valides retourne la consommation de carburant attendue.	ok
5	Vérifie que getConsumption lance une exception pour des distances négatives ou si la vitesse dépasse la vitesse maximale.	ok
6	Vérifie que la méthode getModel retourne la valeur correcte.	ok
7	Vérifie que getWeight retourne une valeur valide.	ok
8	Vérifie que getMaxSpeed retourne une valeur valide.	ok
9	Vérifie que setLoad met à jour correctement currentLoad.	ok
10	Vérifie que setLoad lance une exception pour des charges négatives ou lorsque la charge dépasse maxLoad.	ok
11	La construction de classes dérivées de CargoShip lance une exception avec des valeurs de charge invalides.	ok

Squadron tests

#	Description	ok/nok
1	Vérifie que le constructeur de valeur fonctionne avec des escadrons statiques, dynamiques et constants	ok
2	Vérifie que le constructeur de copie fonctionne avec des escadrons statiques, dynamiques et constants	ok
3	Vérifie que le constructeur de déplacement fonctionne avec des escadrons statiques, dynamiques et constants	ok
4	Vérifie que l'opération d'assignation par copie fonctionne avec des escadrons statiques et dynamiques	ok
5	Vérifie que l'opération d'assignation par déplacement fonctionne avec des escadrons statiques et dynamiques	ok
6	Vérifie que l'auto-assignation n'a aucun effet sur les escadrons alloués statiquement et dynamiquement	ok
7	Vérifie que addShip sur place ajoute réellement un vaisseau à l'escadron	ok
8	Vérifie que nous pouvons enchaîner les opérations avec addShip sur place	ok
9	Vérifie que addShip qui retourne une copie, ne modifie pas l'escadron const original	ok
10	Vérifie que removeShip sur place retire réellement un vaisseau de l'escadron	ok
11	Vérifie que nous pouvons enchaîner les suppressions de vaisseaux	ok
12	Vérifie que removeShip qui retourne une copie, ne modifie pas l'escadron const original	ok
13	Vérifie que nous pouvons ajouter et enchaîner l'ajout de vaisseaux à un escadron dans les deux sens	ok
14	Vérifie que nous pouvons soustraire et enchaîner la soustraction de vaisseaux à un escadron	ok
15	Vérifie que l'opérateur d'addition affecte fonctionne comme prévu	ok
16	Vérifie que l'opérateur de soustraction affecte fonctionne comme prévu	ok
17	Vérifie que le nom de l'escadron est le même que celui donné dans le constructeur	ok
18	Vérifie que setName change effectivement le nom de l'escadron	ok
19	Vérifie que getLeader récupère effectivement le vaisseau leader de l'escadron	ok
20	Vérifie que setLeader définit effectivement le vaisseau leader de l'escadron	ok
21	Vérifie que révoquer le leader ne le retire pas des membres	ok
22	Tests que getShip retourne un vaisseau s'il est présent, ou lance une exception si ce n'est pas le cas	ok
23	Vérifie que la consommation de l'escadron est la même que la somme des consommations des membres	ok
24	Vérifie qu'un vaisseau peut appartenir à plusieurs escadrons et que sa suppression d'un n'affecte pas l'autre	ok

Conclusion

Ce laboratoire nous a donné l'opportunité de concevoir et d'implémenter une hiérarchie de classe représentant des vaisseaux, une liste simplement chainée et une classe `Squadron` englobant le tout.

En prenant des décisions de conception réfléchies, nous pensons avoir réussi à répondre aux contraintes de l'énoncé tout en explorant différentes approches d'implémentation. Cela nous a notamment permis de faire un refresh sur le fonctionnement des listes, et la manière de gérer la mémoire en son sein.

Les tests unitaires ont joué un rôle important dans la validation de notre code en garantissant son bon fonctionnement malgré les refactorisations successives.