TP2: Programmation orientée objet

C++

Objectifs pédagogiques :

- Savoir écrire du code objet d'après la spécification (Ex1).
- Comprendre l'organisation d'un code objet écrit (Ex2).
- Concevoir un code en suivant l'approche objet (Ex3).
- Gestion mémoire en C++ : objets automatiques, dynamiques, temporaires et statiques (Ex2).
- Utiliser les outils de développement C++ : tests avec Catch2, debugger, éditeur (Ex1, Ex2, Ex3).

À faire avant la séance :

— Essayer d'implémenter toutes les questions de l'exercice 1.
— Implémenter la première question de l'exercice 2.
— Installer la version developpeur de SFML (`apt install libsfml-dev`).

Le code associé à ce TP est téléchargeable depuis `https://mquinson.frama.io/prog2-cpp/`

★ **Exercice 1:** nombres complexes (à préparer).

On se propose dans ce premier exercice d'implémenter pas à pas une première classe simple pour manipuler des nombres complexes. Il s'agira d'une simplification du type standard `std::complex`, que l'on n'utilisera pas.

Avant tout, ouvrez `complexes/CMakeLists.txt` dans votre éditeur (`QtCreator` ou `Codium`), puis observez le code fourni. Il contient un fichier d'entête `Complex.hpp` donnant la définition de la classe `Complex` tandis que le fichier `ComplexTest.cpp` teste l'implémentation que vous devez écrire, en utilisant la bibliothèque Catch2 fournie dans le fichier `catch2.hpp`.

▷ **Question 1:** Créez un fichier `Complex.cpp` pour que le projet puisse compiler. Vous recopierez le prototype de toutes les méthodes définies dans le fichier d'entête, en laissant leur corps vide pour l'instant. N'oubliez pas d'ajouter `Complex::` devant les noms de méthodes dans le fichier d'implémentation.

Vous pouvez passer à la question suivante dès que le programme de test compile et s'exécute en indiquant une erreur après les lignes suivantes, indiquant que le problème correspond à la question 2.

Exemple d'exécution du test fourni

```
$ make && ./ComplexTest
[...]
-------------------------------------------------------------------------------
Question 2: Getters
-------------------------------------------------------------------------------
```

▷ **Question 2:** Implémentez les constructeurs, ainsi que les accesseurs `real()` et `imag()`. Comme les champs de la classe Complex sont marqués constants, il est impossible de modifier leur valeur après création. Il faut donc que votre constructeur *initialise* les champs avant son corps de fonction.

Après recompilation, le programme de test ne devrait plus indiquer d'erreur avant la `Question 3: Addition`.

▷ **Question 3:** Implémentez les opérateurs d'addition et de soustraction, qui doivent renvoyer un nouvel objet de type `Complex` sans modifier le receveur (comme indiqué par le `const` après le prototype de la méthode) ni l'argument (lui aussi marqué `const`).

▷ **Question 4:** Implémentez les opérateurs de multiplication et de division sur le même modèle.

▷ **Question 5:** Implémentez les opérateurs d'égalité et de différence (`operator==()` et `operator!=()`). Tester l'égalité entre deux nombres réels avec le signe `==` du langage étant toujours une mauvaise idée, vous devriez utiliser `std::abs` pour tester que la différence entre deux nombres est inférieure à `Complex::EPSILON` [1].

▷ **Question 6:** Implémentez l'opérateur de sérialisation dans un flux `operator<<()`. Comme d'habitude, cet opérateur n'est pas implémenté comme une fonction membre mais grâce une fonction séparée. L'inverse demanderait en effet de modifier la classe `std::ostream` puisque c'est le type du receveur de l'appel lorsque l'on écrit par exemple `out << cplx;`

★ **Exercice 2:** Schéma mémoire.

L'objectif de cet exercice est de comprendre un code mystère fourni dans le répertoire `memory/`, en insistant sur la gestion mémoire des objets en C++. Le fichier CMake à ouvrir est `memory/CMakeLists.txt`. Le projet comporte deux classes `A` et `B` dont seules l'implémentation est fournie. Le fichier `memory.cpp` teste différentes opérations des classes.

1. La représentation informatique des nombres réels est très particulière, menant à divers problèmes. Référez-vous à l'article de David Goldberg sur l'arithmétique sur ordinateur des nombres à virgule flotante (sur la page du cours) pour plus de détails.

▷ **Question 1:** (échauffement). Écrivez les fichiers d'interface `a.hpp` et `b.hpp` avec les prototypes des méthodes définies dans l'implémentation (toutes les méthodes sont publiques). Vous devrez également définir les champs (privés) utilisés par l'implémentation. Le projet doit ensuite compiler et s'exécuter sans erreur.

▷ **Question 2:** Conjecturez combien d'instances de la classe `A` sont créées pendant l'exécution de la fonction `f1()`, et dessinez un schéma mémoire de l'état du processus à la ligne 19. Vérifiez ensuite votre hypothèse en implémentant un compteur d'instances `static` dans la classe `A`. Il doit être incrémenté et affiché à chaque construction d'un nouvel objet `A`.

▷ **Question 3:** Parmi ces instances, combien sont encore vivantes à la fin de `f1()` ? Définissez le destructeur de `A` pour confirmer cette hypothèse. Vous pouvez vérifier que ce destructeur est bien appelé pour chaque objet temporaire en plaçant un point d'arrêt dans le debugger. Vérifiez de même que les variables locales ne sont en revanche détruite qu'au `return` de la fonction.

▷ **Question 4:** Faites un schéma mémoire aussi complet que possible de l'état du programme à la fin de `f2()`. Vous représenterez l'espace des globales, le tas et la pile du processus, ainsi que les différents objets qui peuplent chaque zone. Vous indiquerez pour chacun s'il s'agit d'un objet automatique, temporaire, statique ou dynamique.

▷ **Question 5:** Décommentez l'appel à la fonction `f3()` et observez le résultat. Comment expliquez-vous ce comportement ? Faites un schéma mémoire et aidez-vous du débugger.

▷ **Question 6:** (optionnelle) Définissez le constructeur de copie de la classe `B::operator=(B const& other)` pour corriger le problème.

★ Exercice 3: LogOOP

L'objectif de cet exercice est de réarchitecturer un code procédural pour suivre les principes de la programmation orientée objet. Le code fourni est un interpréteur LOGO rudimentaire.

✠ *L'interpréteur LOGO et code fourni.*

Ce langage a été crée dans les années 1960 par Wally Feurzeig et Seymour Papert pour initier les enfants à la programmation. On commande une tortue graphique qui se déplace sur une feuille en laissant une trace sur son passage. Voici les différentes commandes utilisables dans notre interpréteur :

- `FD x` pour faire avancer la tortue de x pixels
- `BD x` pour faire reculer la tortue de x pixels
- `LT d` pour faire tourner à gauche la tortue de d degrés
- `RT d` pour faire tourner à droite la tortue de d degrés
- `PENUP` pour lever le crayon
- `PENDOWN` pour poser le crayon
- `CLEAR` pour effacer l'écran
- `BC c` pour choisir la couleur numéro c du crayon (0 : blanc, 1 : noir, 2 : bleu, 3 : rouge, 4 : vert)
- `EXIT` pour quitter l'application

Le code fourni dans `logo/logo.cpp` est parfaitement fonctionnel. Compilez-le et lancez-le. Une fenêtre blanche s'ouvre. On peut écrire des instructions sur l'entrée standard du programme pour déplacer la tortue. Ce programme est certes écrit en C++, mais en suivant rigoureusement la philosophie procédurale du C. On trouve beaucoup de globales, quelques fonctions d'aide, des macros et une grosse fonction `main()`.

✠ *Classes, responsabilités et collaborateurs.*

Il est difficile de définir ce qui constitue un bon design logiciel pour un projet donné, surtout quand on manque d'expérience. Pour nous aider, nous allons réaliser des *cartes CRC* (classes, responsabilités et collaborteurs) [2]. Il s'agit de décrire chaque classe dans un format simple faisant abstraction de leur implémentation, mais plus simple que le formalisme UML utilisé habituellement pour cela. Une carte CRC comporte trois parties :

Un nom de la classe : Un bon nom est important pour créer un vocabulaire entre les différents humains impliqués dans le projet. Ce nom doit être évocateur pour la conception du projet : on s'attachera plutôt à ce que fait l'objet qu'à comment il est construit.

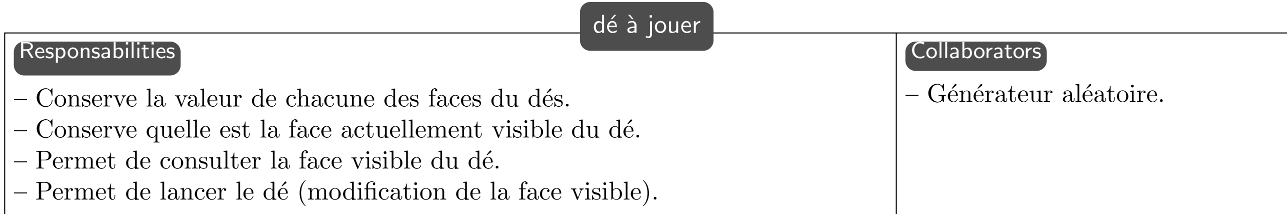
Des responsabilités : Celles-ci identifient les problèmes qui doivent être résolus par la classe. On distingue deux types de responsabilités : ce que l'objet *doit pouvoir faire* (effectuer un calcul, modifier son état interne, créer d'autres objets, coordonner d'autres objets, etc) pour contribuer à la résolution du problème, et ce qu'il *doit savoir* (valeurs à conserver) pour pouvoir implémenter les services proposés.

On s'attachera à décrire l'interface publique des services offerts sans trop s'attarder à comment ces services sont implémentés. Il est cependant important que chaque service reste possible à implémenter.

2. Voir l'article de K. Beck and W. Cunningham décrivant cette approche, disponible sur le site du cours.

Des collaborateurs : Ce sont les noms des classes avec lesquelles les objets de la classe décrite interagieront directement. Il peut s'agir de classes qui serviront d'outils pour la réalisation des responsabilités de la classe décrite, ou au contraire de classes qui utiliseront l'un des services offerts par la classe décrite.

La figure ci-dessous donne une illustration d'une carte CRC représentant un dé à jouer.



✠ LogOOP : micro-interpréteur LOGO architecturé à la mode OOP.

Il est proposé de découper le problème en trois entités : Un écran, qui sait dessiner des lignes à l'écran, une tortue, qui sait se déplacer en laissant des lignes sur son passage, et un interpréteur analysant les commandes écrites sur l'entrée standard pour les convertir en messages envoyés à la tortue.

▷ **Question 1:** Écrivez les cartes CRC des classes `Screen`, `Turtle` et `Interpreter`.

▷ **Question 2:** (optionnelle) Implémentez ces trois classes en vous aidant du code fourni pour les parties métier, c'est à dire les parties implémentant le coeur du projet.

▷ **Question 3:** On souhaite maintenant ne montrer que la dernière position de la tortue, là où la version fournie laisse affichées toutes les positions intermédiaires. Comme on ne peut pas effacer la tortue en préservant le reste, il faut tout effacer puis redessiner toutes les lignes à chaque étape. Cela demande bien entendu de stocker toutes les lignes dessinées à l'écran. Mais quelle classe doit stocker ces lignes selon vous ? Sous quelle forme ?

▷ **Question 4:** (optionnelle) Implémentez cette extension.

▷ **Question 5:** (optionnelle) Améliorez l'implémentation de l'interpréteur pour le rendre plus idiomatique du C++ utiliserant une fonction dotée par exemple du prototype suivant.

```
bool parse_line(std::vector<std::string>& args, std::string& cmd, double& param);
```