TP2: Programmation orientée objet
C++

Objectifs pédagogiques :
- Savoir écrire du code objet d'après la spécification (Ex4).
- Comprendre l'organisation d'un code objet écrit (Ex5).
- Concevoir un code en suivant l'approche objet (Ex1, Ex2, Ex3).
- Gestion mémoire en C++ : objets automatiques, dynamiques, temporaires et statiques (Ex2).
- Utiliser les outils de développement C++ : tests avec Catch2, debugger, éditeur (Ex3, Ex4, Ex5).

Séance théorique (sans machine) :
— Exercices 1, 2 et 3 à faire en séance (sauf l'implémentation d'Ex3, optionnelle).

À faire avant la séance sur machine :
— Essayer d'implémenter toutes les questions de l'exercice 4.
— Implémenter la première question de l'exercice 5.
— Installer la version developpeur de SFML (`apt install libsfml-dev` / `dnf install SFML-devel`).

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

★ **Exercice 1:** Modélisation CRC d'un gestionnaire de bibliothèque (sans machine).

▷ **Question 1:** En utilisant la méthodologie CRC vue en cours, identifiez les classes, leurs responsabilités et leurs collaborations impliquées dans la modélisation d'un système de gestion pour une bibliothèque où des membres peuvent emprunter des livres qui sont à rendre au bout d'un certain temps.

★ **Exercice 2:** Modélisation CRC d'un système de vente de billets de spectacles (sans machine).

On souhaite maintenant modéliser un système assurant le suivi des ventes de billets pour un auditorium comportant 32 rangées de sièges avec un nombre varié de sièges dans chaque rangée. Chaque billet est associé à un siège (rangée lettre et numéro de siège tel que A12), un prix (bas, moyen ou élevé en fonction de la localisation), l'information si le siège est à vendre ou à distribuer à titre gracieux (vendu au public ou offert à amis d'un artiste), le nom de l'occupant (acheteur ou bénéficiaire), son adresse e-mail, et la date, l'heure et le nom du spectacle afin que les billets car chaque siège peut être vendu à différentes personnes à des moments différents.

Il devrait être possible d'obtenir une liste des noms de toutes les personnes qui ont acheté des billets pour une date particulière. Pour une date donnée, il devrait également être possible d'imprimer une grille des places avec 'x' pour les sièges occupés et 'o' pour les sièges encore disponibles.

Quiconque achète un billet à prix élevé rejoint automatiquement le club Gold. Il devrait être possible de générer une liste de diffusion de toutes les personnes appartenant à ce club, et de supprimer quelqu'un de cette liste de diffusion sur demande. Un détenteur de billet doit pouvoir créer et accéder à un compte qui stocke les sièges, les dates, et les noms des spectacles attachés à tous les billets qu'ils ont achetés.

On trouve page suivante une première de modélisation CRC pour ce problème, qui est loin d'être la meilleure possible.

▷ **Question 1:** Dans la solution proposée, où est stocké le nom des utilisateurs ?

▷ **Question 2:** Comment le système génère-t-il la liste des noms des spectateurs à un spectacle donné ?

▷ **Question 3:** Comment est générée la grille des places avec 'x' ou 'o' pour les sièges occupés ou disponibles ?

▷ **Question 4:** Que faut-il faire quand on crée un nouveau spectacle ? Quelle classe doit s'en charger ?

▷ **Question 5:** Comment un utilisateur peut-il acheter (créer un ticket) ? Où sont stockés les Tickets une fois créés ? Quelles classes ont un accès direct ou indirect aux informations du ticket ?

▷ **Question 6:** Faut-il faire collaborer UserAccount et Show ? Pourquoi, ou pourquoi pas ?

▷ **Question 7:** Comment pourrait-on fusionner la classe Show et ShowManager pour ne plus avoir d'instances de Show ? Est-ce une bonne idée ?

▷ **Question 8:** Comment faire pour pouvoir gérer plusieurs salles de spectacles dans le même système ?

TicketVendorSystem

Responsabilities	Collaborators
– Conserve la liste des utilisateurs. – Conserve un ShowManager. – Conserve la liste des places existantes. – Génère la liste des spectacles avec leur nom et la liste des places disponibles. – Génère la liste des emails des membres du club. – Génère et vend des tickets.	– UserAccount – ShowManager

UserAccount

Responsabilities	Collaborators
– Conserve, donne et stocke le nom et l'email d'un individu – Conserve, donne et stocke le statut de membre du club – Donne la liste des tickets achetés – Donne la liste des spectacles, celle associant dates aux sièges – Supprime l'utilisateur de la liste de diffusion – Achète un ticket	– TicketVendorSystem – ShowManager

ShowManager

Responsabilities	Collaborators
– Conserve la liste des spectacles – Donne la liste des places invendues pour un spectacle donné par son nom – Crée un ticket – Cherche un ticket (par place+date, par utilisateur+date) – Donne la liste des tickets d'un utilisateur	– TicketVendorSystem – Show – Ticket – UserAccount

Show

Responsabilities	Collaborators
– Conserve le TicketManager associé – Passe les demandes d'achat de ticket au TicketManager – Génère la liste de toutes les emails à un prix donné – Affiche la liste des places vendues, et celle des invendues – Donne la liste de tous les Tickets – Donne la liste des tickets d'un utilisateur	– Ticket – ShowManager – TicketManager

TicketManager

Responsabilities	Collaborators
– Gère les tickets d'un spectacle donné – Conserve et donne la liste des places et les tickets associés – Crée un ticket pour une place donnée – Conserve la liste des tickets – Donne la liste des emails et noms de la collection de tickets	– Show – Ticket

Ticket

Responsabilities	Collaborators
– Conserve le nom, la date et l'heure du spectacle – Conserve la position du siège et le prix – Conserve le nom et l'email de l'acheteur	– Show – ShowManager – TicketManager

★ **Exercice 3:** LogOOP (conception sans machine, puis implémentation sur machine)

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.

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()`.

✠ 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);
```

★ **Exercice 4:** 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 n'indiquant pas d'erreur avant les lignes suivantes, ce qui signifie que le problème suivant est à la question 2.

Exemple d'exécution du test fourni

```
$ make && ./ComplexTest
[... affichage sans erreur indiquée ...]
-------------------------------------------------------------------------------
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 5:** Schéma mémoire.

L'objectif de cet exercice est de comprendre un code 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 `Point` et `Triangle` dont seule l'implémentation est fournie. Le fichier `memory.cpp` teste différentes opérations des classes.

▷ **Question 1:** (échauffement). Écrivez les fichiers d'interface `point.hpp` et `triangle.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 `Point` 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

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.

implémentant un compteur d'instances `static` dans la classe `Point`. Il doit être incrémenté et affiché à chaque construction d'un nouvel objet `Point`.

▷ **Question 3:** Parmi ces instances, combien sont encore vivantes à la fin de `f1()` ? Définissez le destructeur de `Point` 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 ou avec un affichage adapté. 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 pour comprendre le problème.

▷ **Question 6:** Définissez le constructeur de copie `Triangle::Triangle(Triangle const& other)` pour corriger le problème en réalisant une copie profonde de l'objet.