TD/TP 9

Abstraction & design patterns

Objectifs, méthodes

L'utilisation de l'héritage comme façon de mettre en place des mécanismes polymorphiques conduit naturellement à des classes abstraites (avec au moins une méthode virtuelle pure) ou des classes interfaces (avec que des méthodes virtuelles pures), cas extrême de classes abstraites. Ces classes abstraites en servant de « guichet unique » entre groupes de classes concrètes vont permettre d'éviter le fort couplage qui peut exister entre éléments logiciels qui s'utilisent les uns les autres (appels, compositions...).

Une classe représente un ensemble d'objets qui partagent les mêmes caractéristiques (mêmes méthodes, même attributs). Une **classe abstraite** est en quelque sort une « classe de classes » : elle **représente un ensemble de classes concrètes** qui partagent des mêmes méthodes, c'est à dire **la même façon d'être utilisées**, une même « télécommande ». Le mécanisme orienté objet qui permet de relier un ensemble de classes concrètes à une classe abstraite est l'héritage : la classe abstraire sert de classe de base. Voir cours 9 slide 27.

En désignant la classe de base abstraite pour désigner/utiliser (composer ou appeler des méthodes de) groupes de classes concrètes, non seulement on diminue le couplage entre code appelant et code appelé, mais en plus le code appelant peut être écrit avant le code appelé : c'est la fameuse « inversion de contrôle » qui permet d'offrir aux développeurs non plus de simples bibliothèques traditionnelles (*libraries*) càd. des briques de bas niveau qu'il faut assembler mais des **frameworks** qui sont déjà des applications et qu'il ne reste plus qu'à customiser.

Ce TP vous propose de découvrir un « modèle réduit » de framework. Étant donné le niveau technique et conceptuel de la mise en place en C++ d'un tel modèle, une base de code vous est fournie. Mais contrairement aux frameworks professionnels réels celui-ci est assez compact (~400 lignes) et relativement simple (uniquement des techniques vues en cours) pour être en principe compréhensible à votre niveau. L'objectif est de comprendre la **structure générale** de ce framework, les *design patterns* utilisés, et l'utilisation qui peut en être faite.
Autre objectif (préliminaire) étudier l'héritage multiple
Autre objectif (complément) étudier le *pattern* composite, dans le cadre du projet.

Pour aborder confortablement les 3H de ce TD/TP il est indispensable d'avoir bien suivi et probablement relu le cours 9 sur l'abstraction et les design patterns en C++.

Les concepts du cours couverts par ce TP seront (pas nécessairement dans cet ordre)

- **`polymorphisme`** en paramètre, en attribut, en conteneur
- **`méthodes virtuelles pures`**
- **`classes abstraites`** non instanciables
- **`classes interfaces`** comme éléments d'articulation d'une application
- **`inversion du contrôle`** code appelant (framework) écrit avant le code appelé (client)
- **`delegation pattern`** une classe déléguée reçoit et modifie un objet d'une autre classe
- **`strategy pattern`** est un usage de la délégation avec des hiérarchies de classes délégués
- **`composite pattern`** pour gérer des arbres et des hiérarchies contenant/contenu (projet...)

1. Échauffement : héritage multiple

Le projet suivant vous propose un code simple à tester/lire/comprendre sur l'héritage multiple.
https://fercoq.bitbucket.io/cpp/tdtp/tdtp9/multiple_exo.zip

Il correspond au diagramme du slide 38 du cours 9, mais le ColoredTalkingZebra n'est pas codé... Définissez la classe ColoredTalkingZebra et intégrez une instance de la couleur de votre choix (paramètre du constructeur) aux conteneurs polymorphes qui peuvent l'accueillir, et tester.

2. Télécharger et tester le framework « RoboticPlayground »

https://fercoq.bitbucket.io/cpp/tdtp/tdtp9/robotic_playground_exo.zip

L'affichage dynamique est optimisé pour une console Windows mais en principe le code est portable. Merci de me prévenir si ça ne tourne pas correctement en macOS ou Linux ou Visual... Sur les plates-formes autre que Windows ça clignote un peu, c'est « normal ». Pour info le code qui gère la fonction non portable d'effacer la console est dans robotic_playground.cpp à partir de la ligne 113. Ce fichier d'implémentation comporte beaucoup de code de dessin et de formatage console, pas très intéressant, il n'est sans doute pas utile de le décortiquer en détail.

A partir de ce framework l'objectif (exo 4) sera d'obtenir le résultat animé visible sur ce lien :
https://fercoq.bitbucket.io/cpp/tdtp/tdtp9/robotic_playground.gif

Mais avant de se lancer dans le codage, essayons de comprendre les abstractions que nous propose le frameworks : le résultat en soi ne nous intéresse pas, on pourrait coder tout ça en C avec des switch et des if. Mais ici le *dispatching* vers une diversité de comportements va se faire grâce au mécanisme de polymorphisme.

3. Analyse, compréhension, diagramme

Faites un diagramme de classes simplifié, sans attributs, avec les méthodes virtuelles pures, du framework au complet. Repérez sur ce diagramme les niveaux de « stratification » : la couche de plus bas niveau, la couche de plus haut niveau. Tracez une frontière en pointillé pour les séparer. Repérez les classes interfaces qui servent à coupler les modules au niveau de la frontière. Repérez une ou deux situation correspondant à de l'inversion de contrôle. Repérez un ou deux patterns de délégation (une classe déléguée qui se comporte « au nom » d'une autre classe). Repérez le pattern strategy qui utilise ces délégations. Et enfin (la finalité !) où est-ce que vous pensez que l'utilisateur du frameworks va pouvoir intervenir : imaginez des classes qui vont customiser/étendre le système, avec une autre couleur indiquer ces classes et leur relation avec les classes existantes. Essayez de retrouver l'allure générale de ce qui est présenté slide 28 à 30 du cours 9.

Attention spoiler : 2 diagrammes corrigés sont disponibles ici et là. **Résistez à la tentation de les regarder avant d'avoir fait l'effort de tracer vous même !** L'objectif est d'apprendre à naviguer dans une conception objet tierce et d'arriver à en extraire son architecture....

Note : dans un « vrai framework » comme pour une bibliothèque classique il n'est en général pas indispensable de connaître son architecture interne pour pouvoir l'utiliser, heureusement car cette architecture peut être très complexes et volumineuse ! Cependant ceci nécessite la mise à disposition des utilisateurs d'une documentation très complète et à jour, avec de nombreux

exemples de code utilisateur. Souvent cette documentation ne suffit pas et il faut un « service après vente » sous forme de forums, de formations, de livres débutants/avancés/experts... Les développeurs de frameworks les plus spécifiques n'ont pas forcément les moyens d'assurer une telle logistique ! Prenons par exemple un domaine comme la bio-informatique et un framework expérimental de recherche en génomique. Il est fréquent pour ces frameworks spécifiques de ne pas fournir beaucoup plus qu'une documentation « auto-générée » à partir du code source comme avec *Doxygen*, et l'utilisateur doit se débrouiller à partir de quelques exemples. On n'est pas très loin du travail que je vous demande de fournir : un travail de lecture de modèle.

4. Utilisation du framework

On va maintenant essayer d'utiliser ce framework pour obtenir le résultat animé proposé : https://fercoq.bitbucket.io/cpp/tdtp/tdtp9/robotic_playground.gif

Ceci ne correspond qu'à une partie des combinaisons possibles ! Le pattern stratégie est utilisé 2 fois : pour les stratégie d'animation (l'aspect et le changement d'aspect des robots stratèges) et pour les stratégie de mouvement (déplacement en position). Il y a en tout 7 stratégies de mouvement (2 déjà implémentées) et 6 stratégies d'animation (2 déjà implémentées). *Si vous n'avez pas le temps sur les 2 séance de toutes les compléter ce n'est pas grave : essayez de réaliser au moins les deux 1ères : strategy_move_wiper et strategy_animate_turn, et testez avec les objets correspondants.*

Voici la matrice croisée des stratégies. Toutes les cases ne sont pas forcément intéressantes, par exemple la stratégie d'animation strategy_animate_turn qui change d'aspect quand la direction du mouvement change ne présente aucun intérêt avec la stratégie d'animation strategy_move_null qui justement n'a pas de mouvement ! Mais sans être toutes intéressantes en tout cas toutes ces combinaisons sont possibles. Et ceci en implémentant 6+7 = 13 codes et non pas 6*7 = 42 codes !

Les nombres dans les cases de la matrice renvoient aux indices des robots indiqués en haut à gauche de chaque « track » du terrain animé (voir image animée en lien ci-dessus). Si vous suivez dans l'ordre ces nombres dans votre séquence de développement vous aurez une progression cohérente.

	null	***cyclic***	***turn***	***ghost***	***framed_text***	***cyclic_turn***
null		1				
once	2					
wiper	3	4	5	7	9	
pacman		6				
wiper_slow		8				11
accelerator			10			
scripted						12

Pour vous éviter d'avoir à retaper les créations des objets et voir à quoi peuvent ressembler les constructeurs, le code appelant qui a servi à enregistrer l'animation est en commentaire après le main : vous pouvez piocher dedans. Par contre **il est en principe inutile de modifier ni même d'ajouter du code aux classes fournies**. Le seul fichier fourni qui est à modifier pour intégrer une nouvelle classe stratégie concrète au framework est strategies.h qui facilite l'import des bibliothèques de stratégie vers le code utilisateur càd le main.

5. Classes spécialisées

Grâce au pattern strategy on a une façon extrêmement souple de fabriquer des instances « sur mesure » au moment de leur instanciation. Mais cette souplesse se paye par une syntaxe de création de objets qui est plutôt lourde puisqu'il faut spécifier toutes les stratégies et les paramètres des stratégies ! Supposons qu'on souhaite créer beaucoup d'objets comme l'objet 4 des animations de test qui est une espèce de Tank. Plutôt que de devoir à chaque fois écrire :

```
rp.addRobot(  new StrategicRobot{
    new StrategyMoveWiper{},
    new StrategyAnimateCyclic{ {"[_  _  _ ]=", "[  _  _  ]=", "[ _  _  _]="} } } );
```

on préférerait pouvoir écrire dans le code client juste :

```
rp.addRobot( new Tank );
```

Spécialisez une classe fille de StrategicRobot qui fait le job dans son constructeur ! Testez...

6. Complément : pattern Composite à partir d'un exemple & projet

Voici un exemple simple d'implémentation du pattern composite. Le code client du main n'est pas représentatif de la façon typique d'utiliser une telle architecture des données, concentrez vous sur les classes et leurs relations en correspondance avec le diagramme du pattern composite indiqué slide 67 du cours 9

https://fercoq.bitbucket.io/cpp/tdtp/tdtp9/composite_simple.zip

Essayez de voir le rapport de ce pattern avec le projet. Il est peut-être nécessaire de compléter ou d'étendre le pattern pour le rendre intéressant/opérationnel dans le cadre du projet. Proposer un ou des diagrammes de classes + ou – détaillés. A quel(s) condition(s) ce pattern peut s'appliquer ? Quels seraient les bénéfices à en retirer ? Quels pourraient être les inconvénients ou les risques ? Comment feriez vous pour tester assez rapidement la faisabilité/l'intérêt de ce pattern dans le cadre du projet selon les aspects que vous venez d'identifier ?

```
Good guys
{
    Marvel Comics
    {
        XMen
        {
            Wolverine
            Jean Grey (ex)
        }
        Avengers
        {
            Black Widow
            Iron Man
        }
    }
    DC Comics
    {
        Justice League
        {
            Wonder Woman
            Batman
        }
    }
}
```