ISIMA 3ème année - MODL/C++
TP 6 : Design patterns

On souhaite concevoir un jeu de réussite. L'objectif est de produire un code qui soit modulaire et extensible. Pour cela certains patrons de conception devront être utilisés, notamment pour faciliter le changement des règles du jeu, mais également pour permettre de proposer différentes interfaces. Une base de code vous est fournie : une interface graphique, une interface texte et le départ du code métier sont disponibles sur le dépôt *Git* suivant : https://forge.clermont-universite.fr/git/zz3-cpp-tp5

Récupérez votre copie du dépôt à l'aide de la commande suivante :

```
git clone https://forge.clermont-universite.fr/git/zz3-cpp-tp5
```

L'interface graphique a été conçue avec Qt, il faut donc l'installer sur votre machine. En cas d'impossibilité, vous pouvez compiler en activant uniquement l'interface texte. Pour cela, il suffit de lancer CMake avec l'option suivante :

```
cmake .. -DWITH_QT:bool=false
```

Rappel des règles

Tout d'abord, un petit rappel des règles. Le jeu de réussite utilise les 52 cartes classiques. Trois zones sont définies dans le jeu : la zone de pioche, la zone de jeu et la zone finale.

- La zone de jeu contient sept colonnes contenant respectivement 1, 2 ... 7 cartes au départ. À l'initialisation, seule la dernière est visible et accessible. On peut déplacer une carte visible, ou un ensemble de cartes visibles, vers une autre colonne. Il suffit de respecter l'ordre d'empilement : alternance des couleurs (rouge/noir), seule une carte de valeur immédiatement inférieure peut être placée sur une autre carte.

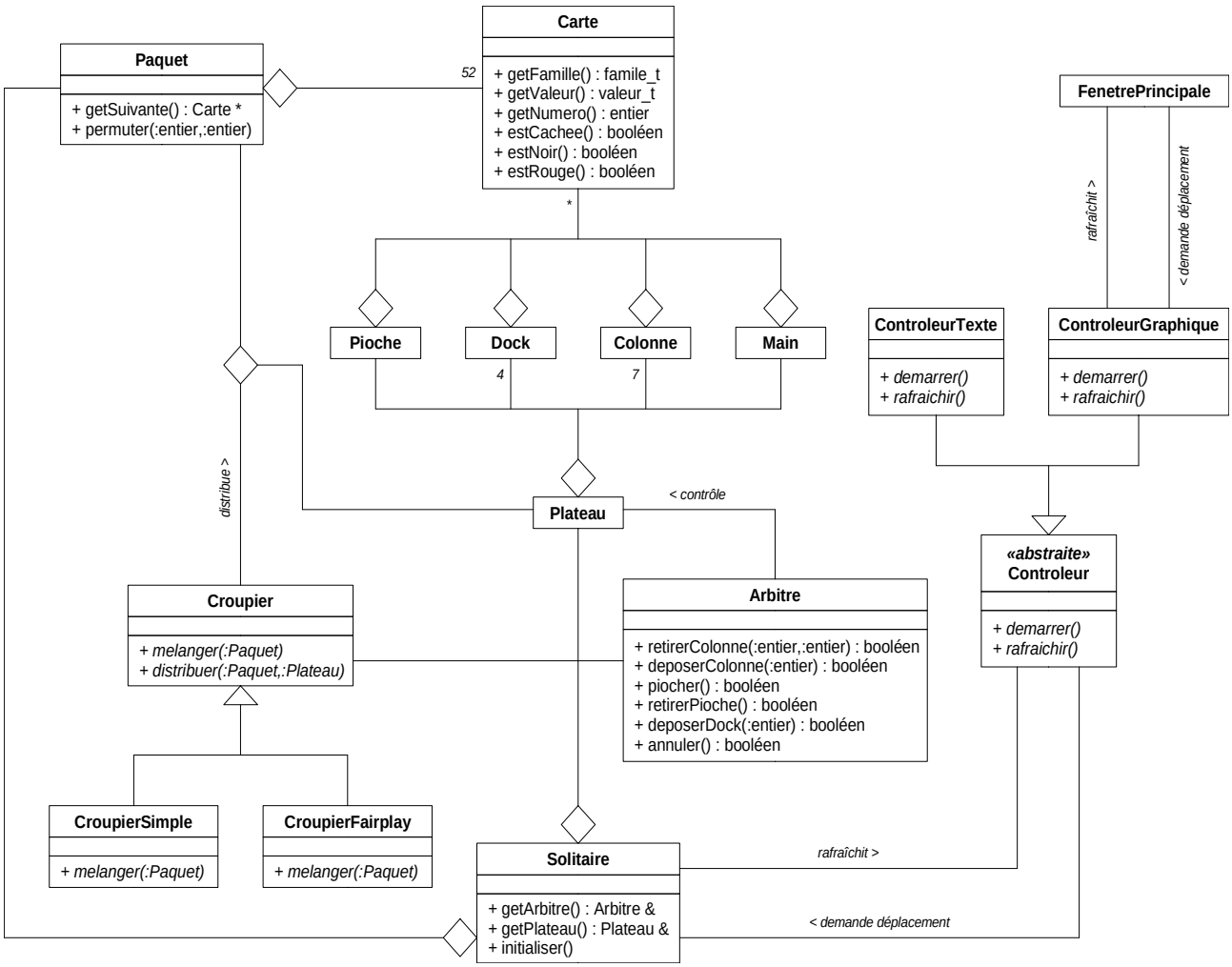
- La zone finale contient quatre emplacements (les docks), un par couleur. Les cartes y sont placées dans l'ordre croissant des valeurs, en commençant par l'as. Les cartes peuvent venir de la pioche ou de l'une des colonnes.

- La zone de pioche est constituée par l'ensemble des cartes restantes. Elle est formée d'un tas de cartes cachées, et d'un tas de cartes retournées où seule la dernière est visible. Au départ, toutes les cartes de la pioche sont cachées. À tout moment, on peut retourner la dernière carte cachée, et la déposer sur le tas de cartes visibles (la défausse). La seule carte visible peut à tout moment être placée dans une autre zone du jeu.

La partie se termine lorsqu'on ne peut plus jouer ou lorsque toutes les cartes ont été transférées vers la zone finale.

Structure logicielle

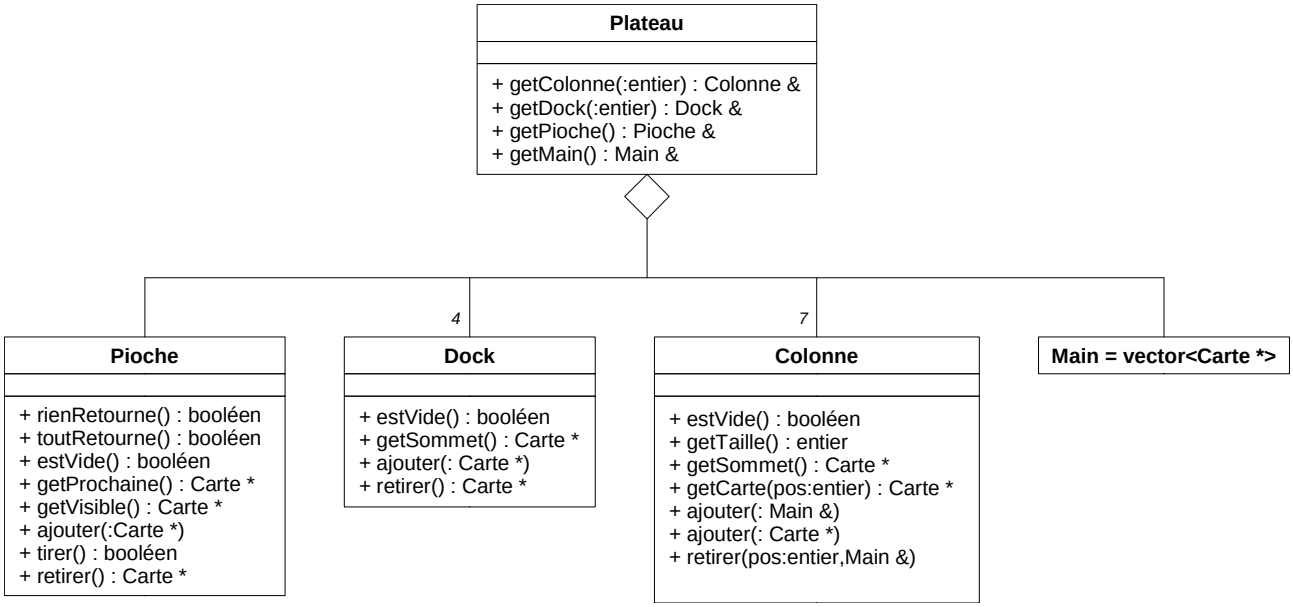
Voici un diagramme de classes qui décrit l'organisation globale du logiciel.



Un code de départ est fourni, incluant une vue texte (dont une partie du code du contrôleur est commentée, vous devrez la réactiver progressivement, à mesure que les composants nécessaires sont disponibles), une vue graphique (que vous utiliserez un peu plus tard, les détails viendront), ainsi qu'une ébauche du code métier (les classes **Carte**, **Paquet** et **Controleur** sont complètes, les classes **Solitaire** et **Plateau** devront être complétées progressivement).

Ce code est compilable en l'état (CMake). Le passage de la version texte à la version graphique se fait dynamiquement à l'aide du premier argument passé en ligne de commande à l'exécution. Si cet argument est « txt » (paramètre par défaut si aucun n'est spécifié), l'affichage se fait en mode texte. Si cet argument est « gui », l'affichage est alors réalisé à l'aide de la version graphique.

Représentation du plateau de jeu



a. Concevoir les classes **Dock**, **Colonne** et **Pioche** qui représentent les trois zones du jeu. La classe **Plateau** devra être complétée. Pour tester vos structures, remplissez les zones avec les cartes d'un paquet de manière arbitraire (code à placer dans la méthode initialiser de l'objet **Solitaire),** et vérifiez l'affichage en mode texte (activer le code de la méthode rafraichir du contrôleur texte). Pour obtenir les cartes, vous utiliserez un objet **Paquet** qui est une forme de « Fabrique » qui produit les cartes une par une avec la méthode getSuivante. La classe **Main** est simplement un vecteur de cartes qui représentera les cartes en transit entre deux tas (cf. question *d*).

b. Concevoir la classe **Croupier** qui représente l'objet chargé de mélanger les cartes d'un paquet (utiliser la méthode permuter de **Paquet** pour faire le mélange). On applique ici le patron « Stratégie » : **Croupier** est une classe abstraite (la méthode melanger est abstraite), et des sous-classes **CroupierSimple** (la méthode melanger mélange totalement au hasard les cartes) et **CroupierFairplay** (subsidiaire, les cartes sont mélangées au hasard, mais de manière à garantir qu'il existe une solution au jeu). La méthode distribuer se charge de prendre les cartes mélangées pour les répartir sur les trois zones de jeu. Le mélange et la distribution des cartes se feront dans la méthode initialiser de l'objet **Solitaire.**

Affichage graphique (subsidiaire)

c. Compléter la classe **ControleurGraphique** (la méthode rafraichir) pour afficher le contenu du plateau sur la vue graphique. Les méthodes/slots miseAJourPioche (partie cachée de la pioche), miseAJourDefausse (partie visible de la pioche), miseAJourDock et miseAJourColonne de la classe FenetrePrincipale seront utilisées pour mettre à jour les zones de la vue. Par souci d'homogénéité, et au cas où l'on souhaiterait améliorer l'interface graphique pour représenter l'épaisseur de ces tas, les slots concernant la pioche et les docks attendent une **QList** de cartes (des paires *(valeur;famille)*), même si pour le moment, seule la dernière carte de la liste est affichée. Voici la déclaration des slots :

```
void miseAJourPioche(
 const QList< std::pair<Valeur::Enum,Couleur::Enum> > & listeCartes);

void miseAJourDefausse(
 const QList< std::pair<Valeur::Enum,Couleur::Enum> > & listeCartes);
```

```
void miseAJourDock(
 const int & numeroDock,
 const QList< std::pair<Valeur::Enum,Couleur::Enum> > & listeCartes);

void miseAJourColonne(
 const int & numeroColonne,
 const Qlist< std::pair<Valeur::Enum,Couleur::Enum> > & listeCartes);
```

Consulter le fichier idCarte.hpp pour plus de renseignements sur les énumérations utilisées pour décrire une carte.

Règles du jeu

d. Concevoir la classe **Arbitre** qui va implémenter les règles du jeu. Cette classe a un rôle de « Médiateur » entre les différentes classes de l'application : elle sert d'intermédiaire et vérifie que les règles sont respectées. Pour faciliter les manipulations de cartes, on vous conseille d'opérer en 2 temps : (i) prendre des cartes dans une zone, ces cartes deviennent la « main » ; (ii) déposer les cartes de la main dans une autre zone. Les méthodes suivantes doivent être implémentées :

- retirerColonne(i,j), retire les cartes de la colonne *i* à partir de la position *j* pour les mettre dans la main.
- deposerColonne(i), dépose les cartes de la main dans la colonne *i*.
- piocher(), dévoile une carte de la pioche (si toutes les cartes ont été dévoilées, elles sont toutes retournées à nouveau).
- retirerPioche(), retire la carte visible de la pioche pour la mettre dans la main.
- deposerDock(i), dépose la carte de la main sur le dock *i*.
- annuler(), replace les cartes de la main d'où elles viennent, cela permet d'annuler un mouvement en cours.

Toutes ces méthodes retournent un booléen indiquant la réussite de l'opération. Testez les mouvements avec l'interface texte (activer le code de la méthode demarrer).

Interaction avec l'interface graphique (subsidiaire)

e. Compléter la classe **ControleurGraphique** pour connecter les mouvements sélectionnés par l'interface graphique avec le code métier. Utiliser le signal demandeDeplacement de la classe FenetrePrincipale, émis par l'interface chaque fois qu'on valide un déplacement. Les zones de départ et d'arrivée, ainsi que le nombre de cartes à déplacer sont transmises par le signal :

```
void demandeDeplacement(const Zone::Enum & depart,
                        const Zone::Enum & arrivee,
                        const int & nbCarte);
```

Consulter le fichier idZone.hpp pour plus de renseignements sur les énumérations utilisées pour décrire une zone.