C++ : modélisation et programmation
CHESSQUITO

Durée : 4 séances de 4h en ½ groupe

Les TP qui vont suivre s'intéresseront au même sujet : il s'agit de la modélisation du jeu *Chessquito (de Sentosphère)*, dont les règles sont disponibles sous Spiral. Évidemment, au départ, il ne s'agit pas ici d'écrire un programme qui joue contre vous, mais plutôt une interface qui permette à deux joueurs de jouer ensemble.
Ce processus se décompose en plusieurs étapes, certaines pouvant être faites dans un même TP.

Chaque étape est à valider avec l'encadrant, avant de passer à la suite.

Étape 0

Nous commençons donc dans ce premier TP avec la gestion de la partie proprement dite : une partie, c'est avant tout deux joueurs, un échiquier, des pièces. Nous allons donc appliquer les principes de base de l'approche objet vus en cours que sont la définition de classes ainsi que leur utilisation.

La première chose à faire est donc de lire l'ensemble du projet, de l'analyser et de modéliser le problème pour toutes les étapes, en suivant les concepts vus dans le module de « Modélisation Objet ». Cette modélisation est à rendre à la fin de la première heure de TP.

Étape1

Spécifications de l'architecture

Commençons par les deux classes : *Joueur* et *Partie*. Dans l'approche objet comme dans les autres méthodologies, les classes ne suffisent pas, il faut également décrire les relations qui les relient entre elles. Ici, le lien qui unit ces deux classes est du type association. La cardinalité de cette association est simple : une *Partie* n'est jouée que par deux *Joueurs*, un *Joueur* peut participer à autant de *Parties* qu'il le souhaite mais pour des raisons de simplicité, nous supposons pour l'instant qu'il ne peut pas jouer à plus d'une partie à la fois. La traduction de l'association en C++ est « simple », il s'agira généralement de pointeur. En effet, les deux classes sont indépendantes : un *Joueur* existera après qu'on ait détruit sa *Partie*. Par contre, attention à la relation inverse ! Pour les *Joueurs*, on choisira deux pointeurs distincts plutôt qu'un tableau de 2 pointeurs.

Spécifications des fonctions

Vous devez bien établir la séparation entre les classes et l'interface du programme. L'interface est simple, vous devez pouvoir créer, supprimer des Joueurs, idem pour les Parties. Et enfin, vous devez pouvoir ajouter un *Joueur* à une *Partie* (retirer un joueur d'une partie reviendrait à terminer la partie).

Ensuite, vous pourrez écrire les classes *Joueur* et *Partie*, ce qui revient à écrire des éléments qui ne changeront pas beaucoup :

- Les constructeurs (qui nous le rappelons sont multiples pour chaque classe) ;
- Le destructeur (un par classe) ;
- Les accesseurs et les mutateurs ;
- Enfin les fonctions internes qui sont généralement liées à celles de l'interface.

Classe Joueur

Le *Joueur* de nos *Parties* de CHESSQUITO ne possède pas beaucoup d'attributs : un nom (string) et évidemment la *Partie* en cours (Partie*).
Déterminez l'ensemble des constructeurs possibles pour cette classe. Un *Joueur* doit pouvoir être créé et affecté à une partie. L'association entre les deux classes étant bidirectionnelle vous devez veiller à mettre à jour les deux instances. Vérifiez également qu'un *Joueur* ne peut pas en remplacer un autre.
Tester votre classe en essayant les instructions suivantes :

```
Joueur B("Bobby") ;
Joueur* G=new Joueur("Garry");
Joueur X;
```

Attention : à ce niveau, vous ne devez pas écrire les fonctions pour la gestion de la *Partie* à laquelle participe le *Joueur*.

Classe Partie

La classe Partie ne pose pas plus de difficultés. Il vous faudra être très attentif à l'initialisation des instances, notamment en utilisant des valeurs par défaut (*NULL*, par exemple). En plus des deux *Joueurs*, une *Partie* comportera aussi une date (string) qui définira le début de la *Partie*. Modifiez la classe *Joueur* pour qu'on puisse créer un *Joueur* qui s'insère directement dans une Partie.

```
Partie P1("01-03-2015");
Joueur H("Hou", P1);
```

Partie en Cours

Modifiez le code des classes *Joueur* et *Partie* pour pouvoir gérer correctement la relation entre ces deux classes. L'ajout d'un *Joueur* à une *Partie* doit donc mettre à jour le lien réciproque de l'association, à savoir que le *Joueur* à une nouvelle *Partie* en cours. Bien penser à vérifier qu'il n'est pas déjà en train de jouer (=NULL). De même, lorsqu'une partie s'arrête, vous devez libérer cette *Partie* pour chacun des Joueurs qui y participait.

Interface

Écrire une interface propre qui corresponde à la réalité du jeu. On doit pouvoir créer un nouveau *Joueur*, une nouvelle *Partie*, affecter un *Joueur* à une *Partie*, afficher les informations relatives à une *Partie* (date et qui joue), à un *Joueur* (nom et *Partie* en cours). Enfin, une *Partie* peut s'arrêter, bien penser à mettre à jour le lien pour chaque *Joueur.* (Un joueur lui peut s'arrêter mais dans ce cas-là, c'est la *Partie* qui s'arrête pour les deux *Joueurs*).

Dans le main() …

```
Partie* p1=new Partie("03-03-2015");
Joueur* j1 = new Joueur("Magnus");
```

```
Joueur* j2 = new Joueur("Judith");
p1->AddJoueur(j1);
p2->AddJoueur(j2) ;
delete j1 ; // vérifier que p1 est bien détruit aussi et que j2 est mis à jour
etc.
```

Étape 2

L'étape n°2 va nous permettre d'introduire la notion d'héritage au sein de notre jeu de Chessquito... Dans le jeu de Chessquito, il y a des pièces différentes qui possèdent chacune un comportement particulier. Nous nous limiterons ici à la résolution de leur déplacement. Nous prendrons comme exemple deux cas simples : la *Tour* et le *Fou*. La *Tour* ne peut se déplacer que sur une même ligne horizontale ou verticale. Le *Fou* ne peut se déplacer que sur une diagonale. En dehors de leur déplacement, la *Tour* et le *Fou* possèdent quand même des points communs : 1) cette capacité à être placées sur la plateau, et de s'y déplacer et 2) des attributs tels qu'une *couleur*. Ces points communs et ces spécificités sont l'essence même d'un héritage entre les classes *Fou* et *Tour* et une classe *Pièce*.

Gestion de l'héritage : attributs

Dans un premier temps, nous allons nous intéresser seulement à l'héritage du point de vue de ces attributs. Il va vous falloir écrire l'héritage qui existe entre les classes *Fou* et *Tour* et la classe *Pièce*. Les classes dérivées n'ont pas d'attributs spécifiques (ce sont les méthodes qui les rendent spécifiques). La classe *Pièce* présente un ou deux attributs, qui sont :

- une couleur : Blanc ou Noir.
- Un état (optionnel) : Disponible, Placé, Pris

Note : il n'est pas utile de connaître spécifiquement la position d'une *Pièce*. Celle-ci sera obtenue à partir du plateau de jeu, que l'on construira plus tard dans la *Partie*.
Par contre, les *Pièces* peuvent partager un *état*, qui simplifie leur gestion

Traduisez l'héritage et tester la création de chaque type d'objet. Pour cela il faudra écrire les constructeurs de la classe mère et des classes dérivées (les constructeurs ne sont pas hérités en C++). Pour les classes dérivées, vous pouvez ajouter un message qui tracera bien la création d'un objet de type *Tour* ou *Fou*. **Écrivez les tests.**
Vous ajouterez également des messages au niveau des destructeurs pour constater la destruction "progressive" des classes dérivées. (en C++, contrairement au constructeur, l'appel au destructeur de la classe mère est automatique lors de la destructeur d'un objet hérité).

Par exemple, dans le main()…

```
Tour Tblanc(Blanc);
Tour Tnoir= Tblanc;
Tnoir.setCouleur(Noir) ;
Fou *Fnoir= new Fou(Noir,Dispo);
delete Fnoir;
// et vérifier que Tblanc et Tnoir sont détruits automatiquement
```

Initialisation de l'échiquier

Lorsque l'introduction de l'héritage sera terminée pour la gestion des différentes *Pièces* de notre jeu de Chessquito, nous verrons comment elle nous facilite la tâche pour gérer la partie d'une manière plus homogène. En effet, l'interaction entre le *Joueur* et les *Pièces* se fera naturellement entre la

classe *Partie* et la classe *Pièce* (sans se préoccuper s'il s'agit d'une *Tour* ou d'un *Fou*). Pour cela, nous allons créer l'échiquier qui est la passerelle entre la *Partie* et les *Pièces*. Nous n'allons faire dans cette étape que l'initialisation de cet échiquier : la gestion des méthodes ne pouvant être abordée que dans la suite.

a4	b4	c4	d4
a3	b3	c3	d3
a2	b2	c2	d2
a1	b1	c1	d1

Il vous faut donc créer un tableau de Pièces. Au départ, l'échiquier est vide, puisque contrairement aux échecs, ici les *Pièces* seront placées à tour de rôle par les joueurs.

La première question que l'on doit se poser concerne le choix entre un tableau de pièces ou un tableau de pointeurs sur *Pièces*. Sémantiquement, la première solution est la meilleure car lorsqu'une partie est détruite, il semble logique que les pièces qui lui sont associées, le soient aussi. Mais comme l'échiquier ne comprend pas 16 mais 8 pièces, à placer sur 16 cases vides, il semble plus judicieux de gérer des pointeurs (sur NULL) pour les cases vides. De plus, vous prendrez garde au fait que les positions désignées sous la forme "f1" sont stockées dans le tableau sous la forme d'indices entiers.

Par exemple :

const int taille=4;

typedef Piece Plateau**[taille][taille]**; // tableau 4x4 à 2 dimensions de pointeurs*

et créer une instance de type Plateau dans la classe *Partie*.

Il est utile à l'usage d'avoir des méthodes qui passent de la coordonnée (i,j) de la case à la notation "a1" ; ainsi que des méthodes qui vérifient si une case (i,j) ou "a1" est dans l'échiquier.

Remarque : *Un tableau à une seule dimension de 16 cases, bien que moins logique par rapport à un échiquier peut quelques fois simplifier la gestion. Dans certains, cas, on ajoute aussi une bordure d'une ou 2 cases à l'extérieur du plateau, pour simplifier les tests lors des déplacements si une Pièce sort du plateau (la position n'est pas valide, mais la case existe).*

Algorithmique et codage des méthodes spécifiques

Pour déterminer le comportement de nos instances, nous allons devoir écrire les méthodes de nos classes dérivées. Ainsi, le placement d'une *Pièce*, est commun, mais le déplacement et l'affichage sont spécifiques pour chacune des classes dérivées. Une *Tour* ne se déplace pas comme un *Fou*. Il faut donc écrire une méthode spécifique pour la *Tour*. D'un autre côté, toutes les pièces se déplacent même si elles ne le font pas de la même manière. L'idée est similaire aux attributs à savoir que nous allons écrire une méthode *verifierDéplacement()* pour la classe Pièce, et qui sera surchargée pour chacune des classes filles. Ainsi, chacune aura son comportement propre.

Déterminez, dans un premier temps, quel est l'algorithme qui permet de valider un déplacement entre deux positions. Un exemple de test pourra prendre la forme suivante :

```
Tour Tblanc(Blanc);
Tblanc.verifierDeplacement("a3","a8");
```

Vous ferez la même chose avec la fonction *Affichage* qui doit afficher la couleur ainsi que la nature de la pièce. Vous utiliserez le principe de « réutilisabilité » afin de ne pas réécrire la partie concernant la pièce pour chacune des sous-classes.

Remarque *: comme on l'a dit précédemment, la Pièce ne connaît pas sa position. Donc ici, on ne vérifie que si le coup est conforme : une Tour se déplace sur une ligne ou une colonne, pas si le déplacement est possible sur l'échiquier !*

Initialisation des Pièces dans la Partie

Normalement, la création des pièces ainsi que leur affectation à l'échiquier se fera dans le programme principal (il faut pouvoir détruire l'objet créé dynamiquement). Par contre, dans le jeu de Chessquito, il peut être utile de stocker une liste des *Pièces* dans la *Partie.* Cela va obliger à maintenir une structure additionnelle (de type Tableau), mais cela peut simplifier leur création (dans le constructueur de *Partie*, par exemple), leur placement (Dispo ou pas ?), et leur destruction automatique lorsque l'on détruit la Partie.

Écrire une fonction de placement des *Pièces* sur l'échiquier. Cette méthode a-t-elle besoin d'être surchargée pour chaque type de *Pièce* ?

```
Partie *p1=new Partie() ;
Tour *Tblanc=new Tour(Blanc) ;
partie->placePiece(Tblanc,"a2") ;
```

Liste d'initialisation

Indiquez à quel endroit, il est possible d'utiliser la liste d'initialisation sur *Partie*, *Pièces* et *Joueur*. Une des utilisations les plus communes de cette liste d'initialisation concerne l'initialisation des attributs, et surtout des pointeurs présents dans les instances afin de les positionner à *NULL* pour d'éventuels tests.

Surcharge de ()

Écrire deux surcharges de l'opérateur () qui permettent d'accéder à la Pièce sur une case précise de l'échiquier à partir de la Partie, par exemple en faisant :

partie(i) ou partie(i,j) ou partie("a4")
… au lieu de partie->echiquier[i] ou partie->echiquier[i][j]

Bien réfléchir au prototype de telles méthodes !

***Attention** : pour chaque méthode, on aura 2 prototypes différents selon que l'on travaille sur une instance constante ou pas :*

```
cout << partie("a4") ;              // constant
partie("a4") = new Tour(Blanc) ;    // référence, partie("a4") est modifiée !
```

Étape 3

La partie précédente nous a permis de mettre en place un héritage basé sur les différentes *Pièces* de l'échiquier (*Fou*, *Tour*, etc.). Cet héritage permet d'avoir un comportement similaire au niveau de la *Partie* pour toutes les *Pièces* sans se préoccuper de la nature de la *Pièce*. Nous n'avons vu jusqu'à maintenant la partie statique de cette implémentation : le stockage de l'échiquier sous forme d'un tableau de pointeurs sur *Pièce*. Il nous reste à voir le comportement dynamique et notamment l'apport de la liaison dynamique pour gérer les déplacements des Pièces ou leur affichage.

Liaison dynamique : méthodes virtuelles

La première application de la liaison dynamique concerne l'affichage de la partie de manière transparente, c'est-à-dire sans se préoccuper de la nature des pièces. Il s'agit ici d'utiliser le stockage des pièces sous forme de pointeurs pour afficher les pièces. L'emploi d'une méthode virtuelle d'affichage répond clairement à cet objectif. Les fonctions amies ne pouvant pas être héritées, une surcharge de l'opérateur << n'est pas pertinente. Il faut une fonction spécifique au niveau de la classe *Pièce* pour que la liaison dynamique soit applicable (méthode *toString()*, par exemple). Au contraire,

la fonction d'affichage (globale) de la partie n'appartient pas à la hiérarchie et donc peut se faire sous forme d'une surcharge de <<. Écrivez les méthodes nécessaires pour permettre l'affichage généralisé de la partie.

Prototype classique :

ostream& operator<<(ostream &flux, Partie const &p)

Un exemple d'affichage peut être le suivant :

*cout << *partie ; // avec « appel » à la méthode virtuelle polymorphe Piece::toString() !*

```
|  a  |  b  |  c  |  d  |
-------------------------
|  TN |     |     |     | 3
-------------------------
|     |     |  FB |     | 2
…
```

Classe abstraite

La classe Pièce de notre jeu de Chessquito est de manière évidente une classe abstraite. Elle ne doit pas être instanciée. En effet, si nous ne connaissons pas la nature de la pièce, il est impossible de vérifier son déplacement ou quelles informations afficher *("Vous venez de déplacer un fou blanc…"*). Par rapport à ce que vous venez de faire pour permettre l'affichage, la transformation ne doit pas apporter de modifications majeures. Une fois que vous aurez rendu votre classe *Pièce* abstraite (en définissant une méthode virtuelle pure) : essayez de définir une instance de la classe Pièce. Maintenant, toutes les fonctions que vous déclarerez comme virtuelle pure devront forcément posséder une définition dans les classes dérivées. La classe mère définit alors un patron que devront suivre les classes héritées.

Étape 4

Nous arrivons au terme de la définition de notre jeu de Chessquito. La fin du travail va concerner les finitions de notre application. En particulier, le fichier principal (main) de notre application joue le rôle de point d'entrée de notre application. C'est lui qui fait la passerelle entre le joueur et les données. Parmi les modèles d'IHM, le paradigme *Modèle/Vue/Contrôle* est un des plus fréquents. Vous avez un **modèle** de données (ici, le diagramme des classes), les **vues** celles qui permettent de gérer l'interaction avec le joueur et celles générées à partir du modèle (*Partie,* par exemple). Enfin, le **contrôle** permet de s'assurer que les opérations effectuées sont autorisées (le déplacement des pièces).

Interface Homme/Machine

Nous allons écrire une interface qui corresponde à ce modèle. L'interface se compose de deux parties : la première correspond à la gestion du jeu (gérer les joueurs, les parties etc.), la seconde concerne le déroulement d'une partie. Nous nous focaliserons dans cet énoncé sur cette partie. Comment se déroule une partie ? Un algorithme simplifié est le suivant :

- La partie initialisée est affichée à l'écran ;
- Chaque joueur dispose des 4 pièces de sa couleur ;
- Les blancs commencent : l'interface demande au joueur Blanc de choisir une pièce et de la placer sur l'échiquier.
- Le système vérifie que la Pièce est disponible, et l'effectue le placement.
- Ainsi, à tour de rôle jusqu'à ce que toutes les Pièces soient placées de manière stratégique, connaissant leur valeur et leur déplacement afin d'éviter de les perdre trop rapidement ;

- Ensuite, les blancs déplacent une Pièce sur l´échiquier, afin de capturer au plus vite celles de l'adversaire ;
- Le système vérifie le déplacement et l'effectue. S'il est facile de vérifier la prise d'une pièce, il est, par contre, plus difficile de s'assurer qu'une pièce s'est déplacée en passant sur une pièce alors que son déplacement l'interdit.
- Affichez et passez au joueur Noir, et ainsi de suite (n'oubliez pas de prévoir une sortie pour interrompre une partie).

La vérification du déplacement (qui se passe au niveau de la *Partie*, puis de la *Pièce*) passe donc par une autre utilisation des méthodes virtuelles, voir plus loin.

Autre type de Pièces

Écrivez maintenant le code pour les autres Pièces (Reine, Cavalier, Pion, Roi).
Remarque *: pour le Pion, celui-ci ne prend pas comme il se déplace. Il faudra en tenir compte lors de la surcharge de verifierDeplacement(), par exemple en rajoutant un paramètre (bool) optionnel pour les autres Pièces !*

Initialiser une partie, par exemple en suivant la règle 1 (Reine-Tour-Fou-Cavalier).
Écrire la fonction (bool) qui permet de décider si une *Partie* est terminée. Pour la règle 1 :

> *« Garder la dernière pièce sur le plateau. La partie est interrompue au bout de 5 déplacements sans prise. Le gagnant est, dans ce cas, celui qui conserve la plus forte valeur de pièce(s). »*

Ce qui implique d'avoir deux informations supplémentaires dans la partie : numéro du coup joué, et numéro de la dernière prise (renseigné dans lors du déplacement effectif de la Pièce dans Partie)…

En profiter pour écrire une fonction qui calcule la valeur de la position (échiquier) : on additionne les valeurs des Pièces Blanches restantes sur l'échiquier, et on soustrait les valeurs des Pièces Noires. Cette méthode permet de dire qui a gagné (et sera bien utile lorsque l'on cherchera le meilleur coup, mais ce n'est pas pour tout de suite…). Quelle est la meilleure façon de connaître la valeur d'une Pièce : attribut ou méthode retournant une valeur ?

Déplacement valide sur l'échiquier ?

Cet algorithme nécessite une attention particulière, car il est finalement assez complexe, et il est facile de se tromper. Voici les opérations qu'il est nécessaire de réaliser :

- Vérifier que la case d'arrivée (et dans une moindre mesure celle de départ) est sur le Plateau ;
- Vérifier si la case d'arrivée est occupée, et si elle est de la Couleur adverse (ce test doit être fait avant, surtout à cause du Pion) ;
- Vérifier que le déplacement est conforme à la Pièce que l'on veut déplacer (polymorphisme) ;
- Si il s'agit d'une Pièce qui se déplace de plusieurs cases, et que ce n'est pas un Cavalier, vérifier que le chemin est libre. Il peut être utile, mais ce n'est pas une obligation, d'écrire une méthode spécifique qui détermine la direction du déplacement, qui définit des vecteurs unitaires horizontaux, verticaux et diagonaux selon les Pièces pour passer à la case suivante, et qui simule le déplacement de la Pièce case par case jusqu'à sa position d'arrivée ;
- Enfin, prendre la Pièce adverse si échéant (et mettre à jour les informations), libérer la case de départ et occuper celle d'arrivée.
- Retourner vrai ou faux !

Étape 5

À ce stade, l'objectif est clairement de revenir et de compléter la définition de la relation Joueur/Partie que nous avons initiée dans la première étape. Après avoir construit les deux classes, nous avons défini le lien qui unissait Partie et Joueurs. Nous allons modifier cette association : le *Joueur* ne peut participer qu'à un nombre restreint de parties (5). Le pointeur *This* sera requis pour traiter la mise à jour des liens. Nous avons vu, en cours, la nécessité de spécifier proprement une classe par un ensemble de constructeurs, destructeurs ou autres opérateurs. Il manque à nos deux classes leur constructeur par copie, leur opérateur d'affectation. Nous ajouterons également des surcharges de l'opérateur d'écriture. Ces fonctions, pas toujours nécessaires, doivent être testées par des instructions adaptées. Voyons cela en détail.

Association bidirectionnelle ou agrégation ?

Comment définissez-vous ce lien au niveau de la classe Joueur ? Vous devez gérer l'initialisation de ce lien et enfin, sa mise à jour chaque fois que le joueur engage (s'inscrit) à une nouvelle partie. De même, vous devez penser aux mises à jour lorsqu'un joueur décide d'arrêter ou lorsqu'une partie se termine. **Écrivez les tests pour valider cette modification**.

Copies et assignation

Pour compléter l'écriture de notre classe, il faut s'assurer que les différentes opérations liées à la manipulation d'instances d'une classe pourront fonctionner. Il s'agit d'écrire et de **tester** l'écriture du constructeur par copie ainsi que l'opérateur d'assignation. Il est conseillé de mettre des traces à l'écran pour s'assurer de la méthode employée (copie ou assignation).

Exemple de traitement :

```
Partie P1 ;       // constructeur sans paramètres
Partie P2=P1      // constructeur par copie
Partie P3 ;
P3=P1 ;           // opérateur d'affectation
```

***Attention**, si la copie (ou affectation) d'une Pièce (sa Couleur et son État si échéant) ou d'un Joueur (son nom, on ne copie pas ses partie en cours) est assez simple, il n'en est pas de même pour la Partie… On se contentera ici de copier l'échiquier et les différentes Pièces (position et état). On peut copier les Joueurs, mais cela implique de vérifier que le nombre de Parties qu'ils jouent est inférieur à 5 !*

Pour copier les Pièces, on a un petit souci : on ne connaît pas le type de Pièce qui est sur l'échiquier, et le C++ ne permet pas d'hériter des constructeurs, donc il n'est pas possible d'utiliser le constructeur par copie qui serait pourtant bien pratique ici. La solution classique est de définir une fonction virtuelle *Piece::clone()* qui sera spécifiée dans chaque classe héritée (en attendant de savoir manipuler les *template*s, ce qui ne serait tarder).

Opérateur << : surcharge pour l'écriture du Joueur

Pour afficher les informations relatives au *Joueur*, nous allons écrire une méthode spécifique qui surcharge l'opérateur <<. Ainsi, il sera possible de disposer d'un moyen homogène pour afficher sur la sortie standard *(cout << Garry;).* Affichez aussi les Parties en cours pour ce Joueur.
Une fois que vous aurez écrit et testé cette méthode faites en sorte qu'il ne soit pas nécessaire de passer par les accesseurs pour afficher les informations de l'instance (*friend*).

Étape 6

Maintenant que notre jeu de Chessquito est opérationnel, il va falloir mettre en place des mécanismes de sauvegarde qui permettent d'arrêter ou de redémarrer une partie sans avoir à ressaisir toutes les informations des Joueurs et des Parties.

Étapes

Sauvegarde et lecture dans un fichier

1. Écrire dans un fichier : dans un premier temps, nous allons sauvegarder les informations relatives à une partie. Vous devez stocker dans un fichier, dont le nom résulte de la concaténation de la chaîne "partie" et de la date (ex : "partie-04-03-2015.txt"), toutes les informations relatives à une partie. Le point difficile de ce genre de mécanisme de sauvegarde réside dans la nécessité de conserver le lien entre la partie et les joueurs qui lui sont associés. Pour cet exercice, nous conserverons l'approche choisie pour la sortie écran, à savoir le nom du joueur. Vous ferez de même avec la sauvegarde de vos joueurs. On rappelle qu'il est important de noter que, normalement, l'écriture d'une surcharge d'affichage (à l'écran par exemple) suffit pour toutes les autres surcharges (écriture dans un fichier par exemple).

2. Lire dans un fichier : La lecture dans un fichier repose sur le même principe que l'écriture à savoir utiliser la surcharge de saisie à l'écran. Ce mécanisme est souvent plus difficile car si l'écriture et la lecture des champs de l'objet sont évidents, on peut oublier toutes les chaînes de caractères que l'on a ajouté pour faciliter la lecture des informations à l'écran (par exemple on écrira "le joueur 1 est Bobby"). Quand vous allez lire dans votre fichier, il ne faut pas oublier de décomposer la chaîne pour extraire le nom à savoir "Bobby". Il faut faire un choix : lisibilité dans le fichier (pour savoir à quoi correspond chaque information) contre facilité pour la lecture d'une partie lors de son chargement.

3. Rajouter la sauvegarde de l'ensemble des coups d'une partie selon la notation Ta2xb2 (Tour en a2 prend en b2) ou Fa1-c3 (Fou en a1 va en c3), suivi de 1-0, 1/2-1/2 ou 0-1 selon le gagnant.

Étape 7, options optionnelles :

- Implémenter les différentes règles, et ajouter les notions de « *mat* » et « pat » pour la terminaison d'une partir pour la règle 3, permettre au joueur de choisir la règle pour chaque partie.
- Ajouter une interface graphique à l'aide de la SDL

Compte-Rendu final : ….

À rendre selon les indications de l'encadrant, à une date qui vous sera communiquée en temps voulu.