

Tyrus

Projet 2 de Java

Remarques :

↪ Date de remise de l'énoncé du projet : semaine du 11 avril 2005

↪ Date **buttoire** de remise du rapport : semaine du 9 mai 2005 (lors de la première séance de laboratoire)

Renseignez-vous auprès de votre professeur afin de connaître ses modalités de remise avant la date fatidique¹.

N'oubliez pas non plus de consulter le document "Méthode d'évaluation" (evaluation.pdf).

↪ Date de défense du projet : semaine du 9 mai 2005 (lors de la seconde semaine de laboratoire)
La défense de votre projet se fera pendant vos heures de laboratoires ... renseignez-vous auprès de votre professeur pour les détails.

POTS-DE-VIN, pressions et intimidations, ²la période électorale est mouvementée dans la cité antique ³. Deux puissances politiques se disputent le contrôle de la cité. Celle-ci étant (plus ou moins) civilisée, des élections ont lieu dans les lieux stratégiques ; la **citadelle**, le **marché** et le **temple**. Ces élections vont nommer les neuf représentants du **Grand Conseil**. Chaque partie veut obtenir un maximum de représentants au sein du **Grand Conseil**, un jeu d'influence se met en place.

Les élections se déroulent dans chacune des places de la cité et chaque partie doit, judicieusement, répartir ses **soldats**, **marchands** et **prêtres** dans les différents lieux d'élections afin d'y remporter les dites élections. En effet il est possible de faire pression sur les membres de la partie adverse afin d'annuler la voix d'un de ses représentants.

Pour gagner une élection, rien de plus simple ... elle se remporte à la majorité. Si vous voulez avoir un représentant commercial ⁴, un *Grand Marchand*, il vous suffit d'envoyer aux urnes plus de marchands que votre adversaire ... mais vous êtes quelqu'un de peu scrupuleux ...

Pour être plus tranquille, vous envoyez donc quelques prêtres faire du prosélytisme chez les marchands adverses...

Mais si vous avez l'impression que votre adversaire fait de même en infiltrant des prêtres à lui parmi vos marchands, rien ne vous empêche d'y placer quelques-uns de vos soldats pour leur faire peur... À moins que vous ne préfériez garder vos soldats pour les faire voter à l'élection du prochain *Grand Soldat* ... Ou pour les envoyer promener leurs glaives dans le Temple de votre adversaire, histoire de faire réfléchir les prêtres adverses sur leur motivation profonde à participer à l'élection du prochain *Grand Prêtre* ?

¹N'oubliez pas qu'une date se compose, dans notre cas, du jour et d'une *heure*

²Toutes ressemblances et envies cachées à l'encontre des professeurs doivent être proscrites ...

³... ouf, rien à voir avec l'école.

⁴Difficile de se représenter un tel représentant sans le costume, la cravatte et la belle voiture mais essayez



FIG. 1 – Le plateau de jeu

1 Le jeu

1.1 But du jeu

Le but du jeu est de remporter trois élections d'affilée **ou**, au terme des neuf élections, avoir plus de représentants que son adversaire.

1.2 Matériel

La boîte du jeu contient les éléments suivants (voir figure 1) :

- ↪ 1 plateau,
- ↪ 9 pions "Représentants",
- ↪ 9 cartes "Election",
- ↪ 60 tuiles "Personnages",
(30 de chaque couleur)
- ↪ 1 pièce 1er joueur,⁵
- ↪ 1 règle de jeu.



⁵Permet le tirage au sort du joueur qui commence la partie

1.3 Règles en détail

Ces règles sont reprises de [1] et [2], on les trouve en détail là, [3]⁶

Chaque joueur dispose de 30 tuiles en bois (imprimées sur une seule face) qui vont servir à remporter diverses élections dans 3 castes distinctes : les **marchands**, les **prêtres** et les **soldats**. Il y a 10 tuiles par domaine, numérotées à chaque fois de 1 à 10.

Chaque domaine exerce un pouvoir sur un autre, et ce, de manière cyclique (soldats > prêtres > marchands > soldats). En détail, on a :

- ↪ un **soldat** neutralise un **prêtre**,
- ↪ un **prêtre** neutralise un **marchand** et,
- ↪ un **marchand** neutralise un **soldat**

En début de partie, chaque joueur pioche 9 tuiles à sa couleur au hasard dans son sac de tuiles.



1.3.1 Déroulement

Une partie se déroule en **9 manches** (9 élections). Il y aura ainsi 3 élections par domaine (3 élections de marchand, 3 de prêtre et 3 de soldat) qui vont intervenir dans un ordre tiré au sort.

A chaque manche, les joueurs vont placer alternativement 3 de leur tuiles dans les bâtiments désirés, sachant qu'une election a lieu ensuite dans les bâtiments des **deux** joueurs au symbole correspondant. Si l'on élit un *Grand Prêtre*, il y a election à la fois chez les "CLAIR" et les "FONCÉS".

Le décompte se fait en dévoilant les tuiles de chacun dans les deux "endroits". On additionne les voix et l'on soustrait les "contres" eventuels.

Dans le cas d'une election dans la maison des soldats (election d'un *Grand Soldat*), on compte le nombre de soldats et l'on décompte sur base de la règle ; les soldats peuvent être contrés par des marchands adverses, qui peuvent eux mêmes être surcontrés par des prêtres.

Le joueur disposant du plus de voix gagne la manche et l'indique par un marqueur de sa couleur. Il peut y avoir égalité.

Exemples de décompte

Si l'on se base sur la figure ci-contre, qui contient dans le "nord", deux tuiles 'soldat foncé' de valeur 10 et 8 et une tuile marchand clair de valeur 7 et dans le "sud" deux tuiles 'soldat foncé' de valeurs 7 et 5, une tuile 'prêtre clair' de valeur 6 et une tuile 'marchand foncé' de valeur 8. on obtient

- ↪ pour le joueur FONCÉ ; $10 + 8 - 7 = 11$
car un marchand neutralise un soldat,
- ↪ pour le joueur CLAIR ; $7 + 5 - (8 - 6) = 10$
car les marchands neutralisent les soldats
mais ils sont neutralisés par les prêtres.

⁶En cas d'imprécision dans la suite référez-vous à ces règles, [3]

2.2 Détail des classes et tests

Nous allons passer en revue les différentes classes. Pour chacune d'elles, nous précisons les différents tests à mettre en oeuvre. Pour les tests nous ne vous demandons pas de tester les aspects **trop** élémentaires de vos classes ; on ne testera pas les *getters & setters* par exemple. Voici, classe par classe, l'ébauche des tests et les ⁸ explications.

2.2.1 classe Couleur

Cette classe est une **énumération**⁹. Elle sert à représenter la couleur de chacun des joueurs.

Vous devez ajouter des attributs et des méthodes qui ne se trouvent pas nécessairement dans le diagramme de classes ¹⁰. En reprenant paragraphe par paragraphe l'annexe A, on obtient ¹¹.

En plus des attributs `public`, `CLAIR` et `FONCÉ` vous définirez des attributs `private`
↪ **nom** de type `String` et
↪ **indice** de type `int`,

vous prévoyez une méthode retournant une liste, ... bref vous implémentez un *typesafe enum pattern*.

Pour les méthodes `public`, vous avez

↪ `Couleur nextCircular()`, retourne la couleur suivante ... de manière circulaire
`CLAIR - FONCÉ - CLAIR - FONCÉ ...`

Test : vérifier que `CLAIR` est le suivant de `FONCÉ` et vice-versa.

↪ `int getNuméro()`, retourne le numéro de l'attribut. Ce numéro est utilisé pour facilement ajouter une notion "d'ordre" dans son type énuméré.

↪ `Couleur getRandom()`, retourne une couleur au hasard¹².

↪ `List list()`, retourne une liste des attributs du type énuméré.

↪ `String toString()`, *no comment*

↪ ...¹³.

2.2.2 classe Caste

Cette classe est, également, une énumération (voir annexe A). Elle représente à la fois les lieux d'élections et le type de tuile. Vous l'implémenterez comme la précédente. Soulignons, ici, l'importance d'attribuer le "numéro" par ordre d'influence. Ceci afin de faciliter l'implémentation de la méthode `getSupérieur`.

Nous renseignons ci-dessous uniquement la méthode propre à la classe, les autres sont les mêmes que pour la section 2.2.1.

↪ `Caste getSupérieur()`, retourne la caste supérieure.

Test : vérifier la relation soldat gagne sur prêtre qui gagne sur marchand qui gagne sur soldat

⁸Je devrais plutôt écrire **des** explications parce que vous devrez quand même vous posez des questions.

⁹Nous vous conseillons vivement la lecture de l'annexe A

¹⁰Ce sera le cas pour toutes les classes ... finalement le diagramme de classe n'est pas très locale.

¹¹Dois-je tout détailler puisque sur nos conseils vous avez été lire l'annexe A ?

¹²Peut-être utile en début de jeu

¹³Les ... représentent les *getters & setters*, pour l'ajout de méthodes consultez votre professeur.

2.2.3 classe Tuile

Une tuile possède une certaine couleur (`Couleur.CLAIR`), appartient à une certaine caste (`Caste.PRETRE`) et a une certaine valeur ($v \in [1 - 10]$).

Les méthodes ...

- ↪ `String toString(Couleur)`, vérifier qu'une tuile apparaît bien comme cachée si nécessaire et visible sinon. Si la couleur passée en paramètre correspond à celle de la tuile, je l'affiche sinon je la cache (je montre qu'il y a une tuile mais je ne montre pas ses valeurs).
- ↪ `boolean equals()`

2.2.4 classe TasTuiles

Une **collection**¹⁴ de tuiles. Cette collection est utilisée pour représenter ; la main de chaque joueur, son sac de tuiles restantes, le contenu des 'pots'. Vous pouvez choisir la manière dont vous implémenterez votre collection.

Les constructeurs permettront de créer un tas de tuiles vide et de créer les tas de tuiles initiaux de chaque joueur.

Les méthodes ...

- ↪ `void ajouter()`, ajoute une tuile à la collection.
Test : créer un tas avec quelques tuiles. Vérifier qu'une tuile ne s'y trouve pas, l'ajouter et vérifier qu'elle s'y trouve.
- ↪ `boolean contient(Tuile t)`, teste si la collection contient la tuile.
Test : créer un Tas et y ajouter des tuiles. Vérifier `contient()` en posant la question pour une tuile qui s'y trouve et une qui ne s'y trouve pas,
- ↪ `Tuile enlever()`, enlève une tuile au hasard.
Test : créer un tas avec quelques tuiles. Enlever une tuile au hasard, vérifier qu'elle ne s'y trouve plus.
- ↪ `boolean enlever(Tuile t)`, enlève la tuile donnée du tas.
Test : créer un tas avec quelques tuiles. Vérifier qu'une tuile s'y trouve, l'enlever et vérifier qu'elle ne s'y trouve plus.
- ↪ `Iterator iterator()`, retourne un itérateur sur la collection de tuiles ... permettra de parcourir cet ensemble.
- ↪ `int taille()`, le nombre de tuiles.
Test : créer un tas avec quelques tuiles et vérifier que la taille est correcte.
- ↪ `int somme(Caste caste, Couleur couleur)`, somme les valeurs des tuiles de même caste et même couleur.
Test : créer un tas représentatif et vérifier la méthode
- ↪ `TasTuiles(Couleur)`, crée le sac de tuiles de départ.
Test : créer un sac de départ et :
 - ~ vérifier qu'il a une taille de 30,
 - ~ tester la présence de certaines tuiles (pas la peine de faire tout...)
- ↪ `TasTuiles()`, crée un tas de tuiles vide.

¹⁴Consultez l'API Java et le cours (leçon sur `java.util`) pour des infos complémentaires.

2.2.5 classe Pot

Un *pot* est un lieu d'élections, il y en a donc 6 ; *marché clair, temple clair, ...* chaque pot a, comme attribut, une **caste**, une **couleur** et un **tas de tuiles**.

↪ `Pot(Caste caste, Couleur couleur)`, crée un pot

↪ `int nbPoints()`, calcule les points d'un pot (en tenant compte des différences de couleurs, ...).

Test : créer un pot, y déposer des tuiles et vérifier que la méthode retourne la bonne valeur. Refaire le test avec plusieurs pots représentatifs. Tester notamment que la valeur n'est pas négative si la parade est plus grande que la mise principale, ...

↪ `TasTuiles vider()`, vide le pot **et** retourne le tas de tuiles qu'il contient.

2.2.6 classe Plateau

Représente le plateau de jeu, c'est-à-dire six pots.

↪ `Pot get(Caste caste, Couleur couleur)`, retourne un pot.

Test : vérifier que la plateau retourne le bon pot (en testant la couleur et la caste du pot retourné)

Remarque L'utilisation de cette classe peut être discutée. En effet, elle a pleinement son utilité si l'on se place du point de vue de l'analyse. Pour jouer une partie, il est nécessaire de posséder un plateau de jeu.

Par contre, il est fréquent, que le programmeur s'écarte un peu de l'analyse lorsque cet écart simplifie son implémentation. Dans ce cas, nous pourrions nous passer de la classe `Plateau` puisqu'un plateau n'est finalement qu'un ensemble de six pots. La partie peut donc simplement posséder ces six pots.

Peut-être quelques moments de discussions avec vos professeurs en perspective ...

2.2.7 classe Joueur

Un joueur possède un **nom**, un **cerveau**¹⁵, une **couleur**, une **main** et un **sac de tuiles**.

Le cerveau d'un joueur pourra être de type *humain* ou *machine*, voir 2.2.8. Pour ce qui est de sa *main*, elle est puisée dans son *sac* (voir règles).

↪ `Joueur(Couleur couleur, String nom, Cerveau cerveau)`, crée un joueur.

↪ `compléterMain()`, complète la main d'un joueur en début de partie et de manche.

Test : créer un joueur et tester que la main est vide. Demander de compléter la main et vérifier qu'elle est de taille 9.

Remarque Pour aller plus loin, il faut tester que la main est inférieure à 9 si le sac ne contient plus assez de tuiles. Pour que ce ne soit pas trop lourd à tester, nous vous conseillons d'ajouter une méthode `setSac()` qui permette de créer un sac de petite taille.

¹⁵ :-)

2.2.8 interface `Cerveau`, classe `Humain`, ...

Un cerveau permet de proposer un coup. Cet interface n'impose qu'une seule méthode. Deux types d'implémentation de cet interface vont exister. Le premier sera un `Humain`. Cette implémentation de la méthode `proposerCoup ...` en fera la demande au joueur. Le second type d'implémentation, la "machine" est plus complexe car la proposition d'un coup dépendra du choix d'une tactique de jeu. C'est ici que réside "l'intelligence artificielle" du jeu.

↪ `Coup proposerCoup(Couleur couleur, TasTuiles main, Caste election, Plateau plateau), propose un coup`¹⁶.

Vous ne devez pas implémenter votre propre "intelligence artificielle". Vous écrivez l'interface `Cerveau`, vous l'implémentez en `Humain`¹⁷ et vous utilisez une implémentation de 'machine' écrite par les professeurs¹⁸.

2.2.9 classe `Manche`

Retient le résultat d'une manche. Il est utilisé par la classe `Partie` lorsqu'une manche se termine et avant de passer à la suivante.

Cette classe contiendra les méthodes suivantes ainsi que ses *getters & setters*. Pour le reste, nous vous laissons y réfléchir.

↪ `Manche(Caste lieuElection, TasTuiles miseClair, int valeurClaire, TasTuiles miseFoncé, int valeurFoncé), crée une manche.`

↪ `Joueur getVainqueur()`, retourne le vainqueur de la manche.

Test : créer des instances de `Manche` avec différents cas de figures (CLAIR gagne, FONCÉ gagne et égalité) et vérifier que la méthode répond correctement.

2.2.10 classe `Resultat`

Cette classe va mémoriser les gagnants de chaque manche (en terme de couleur ou en terme de joueur ... à vous de voir). Elle prend en charge la fin de partie et la détermination du vainqueur.

↪ `Resultat()`, crée l'objet.

↪ `ajouter(Couleur c)`, ajoute le gagnant d'une manche¹⁹.

↪ `boolean estPartieFinie()` et `getVainqueur()`, détermine si la partie est finie et qui est le vainqueur.

Test : créer quelques situations représentatives (via la création d'un résultat et des appels successifs à `ajouter()`) et vérifier que les méthodes donnent le bon résultat.

On devra vérifier notamment que

- ~ un joueur gagne s'il obtient 3 victoires consécutives,
- ~ un joueur gagne s'il obtient plus de victoire que l'autre au bout de 9 manches,
- ~ il peut y avoir égalité au bout des 9 manches,

¹⁶Pour les distraits, cette méthode n'est pas implémentée dans l'interface.

¹⁷Un humain n'a pas d'intelligence '... la méthode `proposeCoup` demande le coup à jouer au joueur.

¹⁸voir la classe `RobotMCD`

¹⁹Dans l'hypothèse où l'on détermine le gagnant en terme de la couleur et pas en terme de joueur

- ˘ un joueur gagne si moins de 9 manches mais qu'il ne peut plus être rattrapé.

2.2.11 classe Partie

Implémente toutes les facettes d'une partie de "Tyrus". Il faut distinguer ici la classe `Partie` de la classe `Tyrus`.

- ↪ la classe `Partie` met tout à disposition afin de pouvoir jouer ... une partie,
- ↪ la classe `Tyrus` est une *implémentation de partie en mode console*.

C'est-à-dire que pour implémenter une version du jeu avec un "autre design", il suffit de réécrire la classe `Tyrus`.

Une partie contient toute une série d'attributs privés ; la **dernière manche**, ainsi que son **vainqueur**, un lieu d'**élection**, un tableau de deux **joueurs**, le joueur en cours, un **plateau de jeu**²⁰, un **résultat** et un nombre de **tours**.

Les méthodes ...

- ↪ `Partie(String nomPartie, Cerveau c1, Cerveau c2)`, crée une nouvelle partie.
- ↪ `boolean estMancheFinie()`, précise si le dernier coup a terminé une manche.
Test : vérifier qu'une manche est finie après 3 tours (6 coups) exactement
- ↪ `boolean estPartieFinie()`, précise si la partie est finie.
- ↪ `jouerCoup()`, joue un coup **valide**.
Test : vérifier qu'un coup invalide est rejeté
- ↪ ... par exemple ; `preparerCoupSuivant`, `preparerNouvelleManche`, `terminerManche`, ...

2.2.12 La classe Tyrus

(voir 2.2.11)

Cette classe contient la méthode `main` permettant de jouer. C'est elle qui gère toutes les entrées-sorties. On aura une structure du type :

```

creer_la_partie
while ( la_partie_n_est_pas_finie ) {
    demander_au_joueur_courant_un_coup
    jouer_le_coup
    if ( la_manche_est_finie ) {
        afficher_le_vainqueur_de_la_manche
    }
}
afficher_le_vainqueur_de_la_partie

```

Cette classe contient donc une méthode permettant d'afficher l'état de la partie (les tuiles posées, les gagants, des manches, ...). Nous vous laissons ici le choix de l'interface utilisateur du jeu. Nous savons que cette interface relève d'une certaine importance à vos yeux mais nous vous demandons de ne pas trop vous y attarder dans un premier temps ... la base du jeu doit fonctionner même si l'interface est des plus basique.

²⁰... ou un ensemble de six pots, voir 2.2.6

A Enumération

Un type énuméré est un type pour lequel l'ensemble des valeurs est **petit, connu à l'avance et chaque valeur porte un nom**. Par exemple : les couleurs d'une carte (Coeur, Carreau, Pique, Trèfle).

Dans le projet 2, vous allez rencontrer 2 énumérations : la **couleur** d'une tuile (Clair, Foncé) et la **caste** d'une tuile (Soldat, Marchand, Prêtre). Comme vous n'avez pas encore rencontré ce type ni au cours de Java ni au cours de Logique, nous vous expliquons ici comment le faire en Java. Nous allons commencer par une implémentation simple pour l'enrichir petit à petit.

Dans la littérature, on parlera de *typesafe enum pattern*.

A.1 Première approche

Au tout début de la programmation, on a pensé réaliser un type énuméré via un type entier. A chaque valeur de l'énumération, on utilise un entier précis. Pour garder une certaine lisibilité, on utilise des constantes. En Java, cela donnerait

```
public class CouleurCarte {
    public static final int COEUR = 1;
    public static final int PIQUE = 2;
    public static final int TREFLE = 3;
    public static final int CARREAU = 4;
}
```

Cette solution souffre de plusieurs problèmes, le principal étant l'absence d'un type. En effet, si vous voulez une variable représentant une couleur de carte, vous devez la déclarer comme un entier. Cela permet d'écrire n'importe quoi. Par exemple,

```
int couleurMaCarte = CouleurCarte.PIQUE; // OK
couleurMaCarte = 5; // Accepte mais incoherent !
int organeCorps = CouleurCarte.COEUR; // On melange
// pommes et poires!
```

L'idée est d'utiliser la classe comme type pour l'énumération. Cela donne

```
public class CouleurCarte {
    public static final CouleurCarte COEUR = new CouleurCarte();
    public static final CouleurCarte PIQUE = new CouleurCarte();
    ...
    private CouleurCarte() {};
}
```

Remarquons que le constructeur est rendu privé pour empêcher l'ajout d'autres valeurs. Voyons ce que cela donne au niveau de l'utilisation.

```
CouleurCarte couleurMaCarte = CouleurCarte.PIQUE; // OK
couleurMaCarte = 5; // Refuse a la compilation!
int organeCorps = CouleurCarte.COEUR;
// Refuse a la compilation!
```

A.2 Enrichir le type énuméré

On peut aller plus loin encore et enrichir notre type énuméré de nombreuses facilités.

↪ Un affichage correct

Pour le moment, si on écrit une variable de type énuméré, l'affichage de donne rien de bien. Pour arranger cela, on peut ajouter une méthode `toString()` en ajoutant un attribut `nom` qui sera passé au constructeur.

↪ Parcourir les valeurs

En l'état actuel, il est impossible d'effectuer une boucle sur toutes les valeurs de l'énumération. Pour faire cela, on peut ajouter un tableau contenant toutes les valeurs.

```
public class CouleurCarte {
    public static final CouleurCarte COEUR =
        new CouleurCarte("Coeur");
    public static final CouleurCarte PIQUE =
        new CouleurCarte("Pique");
    ...
    public static final CouleurCarte[] liste = {COEUR, PIQUE, ... };
    private String nom;

    private CouleurCarte(String nom) {
        this.nom = nom;
    }

    public String toString() {
        return nom;
    }
}
```

Exemple d'utilisation

```
// Afficher toutes les couleurs de carte
for (int i = 0 ; i<CouleurCarte.liste.length; i++ ) {
    System.out.println( CouleurCarte.liste[i] ) ;
}
```

A.3 Remarques

On pourrait faire mieux en proposant une collection (comme `ArrayList`) plutôt qu'un tableau et encore mieux en s'arrangeant pour que cette liste soit construite automatiquement dans le constructeur.

Nous ne disposons plus d'un entier associé à chaque valeur énumérée. C'est parfois utile, pour servir d'index dans un tableau par exemple. Il suffirait d'ajouter un attribut entier. Il est également possible d'ajouter des méthodes. C'est demandé dans le projet 2.

Bref, nous n'avons exposé que les principes de base du type énuméré. Pour aller plus

loin, renseignez-vous sur le *Typesafe enum pattern*²¹.

²¹Qui n'a, dans la version 1.5.0 "Tiger", plus rien à voir avec la construction du type énuméré "en 1.4.2"

Webographie

- [1] Bord Game Team. Bordgamegeek.com, 2005. <http://www.boardgamegeek.com/game/10520>.
- [2] Tric Trac. Tric trac, le magazine internet des jeux de société, 2005. <http://www.orleans.tv/index.php3?id=jeux&rub=detail&inf=detail&jeu=2539>.
- [3] Inconnu. Règles du jeu, 2005. http://jeuxstrategieter.free.fr/Tyrus_complet.php.
- [4] Marco CODUTTI. Site perso à destination des étudiants de l'esi, de l'ulb et loisirs, 2000. <http://www.users.skynet.be/Marco-Codutti>.
- [5] David FRANCK. Télécharger tyrus, 2005? <http://davidfranck.chez.tiscali.fr>.
- [6] Ludi Gaume. Ludigaume.net, 2005. http://www.ludigaume.net/Jeux/Tyrus/tyrus_i.htm.

Table des matières

1	Le jeu	2
1.1	But du jeu	2
1.2	Matériel	2
1.3	Règles en détail	3
1.3.1	Déroulement	3
1.3.2	Fin de partie	4
2	Le projet	4
2.1	Diagramme de classes	4
2.2	Détail des classes et tests	5
2.2.1	classe Couleur	5
2.2.2	classe Caste	5
2.2.3	classe Tuile	6
2.2.4	classe TasTuiles	6
2.2.5	classe Pot	7
2.2.6	classe Plateau	7
2.2.7	classe Joueur	7
2.2.8	interface Cerveau, classe Humain,	8
2.2.9	classe Manche	8
2.2.10	classe Resultat	8
2.2.11	classe Partie	9
2.2.12	La classe Tyrus	9
A	Enumération	10
A.1	Première approche	10
A.2	Enrichir le type énuméré	11
A.3	Remarques	11