Projet *Tournament*

Gestion et organisation d'un tournoi

En bref....1
Sens de lecture....2
Échéances....2
Préalables....4
Plan de tests....6
Le grand tournoi v1....7
Le grand tournoi v2....18
Modalités d'évaluation....27

Crédit photo France3 aquitaine http://goo.gl/CNgpb

1 En bref

Ce projet vous permettra de mettre en œuvre la plupart des concepts vus pendant cette première année. Lisez bien complètement[1] ce document. Il ne contient pas que des données relatives à l'exercice. Il vous renseigne également sur les modalités de remises et vous donne une progression dans la résolution de l'exercice.

Nous allons coder une application permettant de gérer un **tournoi** de l'inscription jusqu'à ce qu'il n'en reste qu'un !

1 Cette lecture pourra vous paraitre fastidieuse mais n'oubliez pas que ce travail vous tiendra en haleine durant plusieurs semaines.

2 Sens de lecture

Ce document est relativement long, pour ne pas s'y perdre, vous pouvez suivre ces indications,

Je voudrais ...	**... alors il faut**
Connaitre[2] les diverses échéances	Lire la section 3 *Échéances*, p 2
Savoir à quelle sauce je vais être mangé ou comment je vais être évalué	Lire la section 8 *Modalités d'évaluation* p 27
Commencer la **première partie** du travail	Lire les sections 4 *Préalables* p 4 et 5 *Plan de tests* p 6
Commencer la **seconde partie** du travail	Lire la section 6 *Le grand tournoi v1* p 7
Commencer la **troisième partie** du travail	Lire la section 7 *Le grand tournoi v2* p 18

3 Échéances

Échéances....2
Différentes phases....3
Phase 1, le plan de tests....3
Phase 2, Tournament v1....3
Phase 3, Tournament v2....3
Remises intermédiaires obligatoires....3
Pondération....3

Le projet sera évalué en plusieurs phases. Vous veillerez à remettre chacun des travaux en temps et en heure sous peine de ne pas être évalué (voir paragraphe Error: Reference source not found, **Modalités d'évaluation** p 27).

Le tableau qui suit reprend les différentes semaines pendant lesquelles seront fixées, par votre professeur, les dates butoires[3] de remises.

Partie	Plan de tests	Tournament, v1	Tournament, v2	Défense orale
Semaine du	11 mars 2013	25 mars 2013	22 avril 2013	29 avril 2013

2 Nouvelle orthographe de *connaître*

3 Une **date** doit, dans le cadre de l'école, plutôt être vue comme un *moment*. C'est-à-dire qu'elle renseigne un jour et une heure (par exemple le 21 octobre 2015 à 1h21). La date est **butoire** lorsqu'elle ne peut en **aucun cas** être dépassée.
Ce qui veut dire que dans l'exemple précédent, remettre son travail le 21 octobre 2015 à 2h15', est inutile. C'est trop tard ! Les esprits chagrins peuvent trouver cette règle exigeante mais vous voilà prévenu. Rien ne vous empêche de remettre votre travail **avant** (la veille, la semaine qui précède, ...) la date butoire.

Différentes phases

Le projet se décompose en plusieurs phases. La première phase consiste à réfléchir aux **tests** à mettre en œuvre, tandis que les deux autres sont la réalisation des différentes versions de l'application. Tout ceci est décrit dans la suite du document.

Phase 1, le plan de tests

Dans un **premier temps** nous vous demandons de réfléchir à un plan de tests pour une méthode d'une classe particulière (voir Que tester ? p 6).

Un **plan de tests** est une description de tous les cas de test que vous jugez nécessaires pour prouver le bon fonctionnement de votre code. Cette description fera l'objet d'une première évaluation sur base du document que vous nous remettrez.

Phase 2, *Tournament* v1

Dans un **second temps**, vous codez la première version de votre application que vous nous remettrez. Votre professeur fera des commentaires sur cette première version dont vous tiendrez compte pour coder la suite.

Phase 3, *Tournament* v2

La deuxième version de votre application contiendra des fonctionnalités supplémentaires. Il est inutile de commencer la version 2 si la première version n'est ni fonctionnelle, ni évaluée.

Remises intermédiaires obligatoires

En plus des échéances fixées, une **remise hebdomadaire** de votre travail est **exigée.** Vous déposerez votre projet dans le casier de votre professeur (*acr*) via la commande
`casier <acr>`.

Attention Cette remise ne sera pas évaluée, mais elle est **obligatoire** sous peine de ne pas voir votre projet corrigé.

Pondération

À chaque partie évaluée est associée une pondération de la cote finale, soit

Partie	Pond.
plan de tests	1/8
Projet version 1	3/8
Projet version 2	1/2

4 Préalables

Préalables....4
Présentation de l'application....4
Présentation des classes....4
Packages....5

Présentation de l'application

L'application va permettre de gérer un tournoi au sens le plus général possible[4] ; foot, jeux de café, tennis, whist, pêche, …

Un tournoi se décompose en plusieurs phases :

- ✗ l'inscription des participants ;
- ✗ la répartition en poules et la gestion des rencontres au sein des poules afin d'en faire ressortir les vainqueurs ;
- ✗ le tournoi à élimination directe avec les vainqueurs de chaque poule.

Un **participant** peut être une personne ou une équipe en fonction du type de tournoi. Pour nous, un participant est une entité ayant un nom et un total de points.

Les **rencontres de poules** sont telles que « tout le monde rencontre tout le monde ». Les 2 participants ayant le plus de points (par poules) passent à l'étape suivante. Les autres sont éliminés. Pas de repêchage possible.

Le **tournoi à élimination directe** fait se rencontrer les participants issus des poules. Ici, chaque rencontre élimine le perdant. Pas de match nul possible, seul le vainqueur passe à l'étape suivante.

Présentation des classes

Cette présentation est une présentation **sommaire** des classes qui vont intervenir dans le projet afin de pouvoir travailler sur le plan de tests. Une description détaillée suivra (voir Le grand tournoi v1 p7).

Nous allons distinguer la partie **métier** (*business*) de la partie **vue** (*view*) de l'application[5].

La partie **métier** *(business)* s'intéresse aux classes représentant les objets que l'on manipule ainsi que les classes responsables du « travail à faire » (à tout ce qui n'est pas interface avec l'utilisateur).

Dans notre application, la classe `MainTournament` (tournoi principal) est la classe principale permettant la gestion du tournoi, c'est elle qui permettra l'inscription des participants et qui fera appel à la classe `PoolTournament` (tournoi en « poule ») pour la gestion des rencontres

4 L'application ne tient pas compte de règles spécifiques à un sport ou à un type de tournoi en particulier. En ce sens, c'est un modèle théorique qui devrait être adapté … ou utilisé en l'état.

5 Cette distinction/répartition des classes en fonction de leur rôle est mise en œuvre dans le *design pattern* ***Model/View/Controller*** (en français; patron de conception Modèle/Vue/Controleur). Vous verrez cette notion tout à fait dans le détail en deuxième, ce que nous mettons en œuvre ici n'en est qu'une légère approche.

de poules et `SingleEliminationTournament` (tournoi à élimination directe) pour la gestion du tournoi par élimination directe ensuite.

La classe `PoolTournament` fera appel à une classe `Pool` pour la gestion de chaque poule.

Les données que l'on manipule sont des participants (classe `Player`) et des rencontres (classe `Match`) entre deux participants.

La partie **vue** (*view)* quant à elle ne s'occupera que de présenter le tournoi et de gérer l'interaction avec l'utilisateur.

Résumé de quelques classes

Classe	Rôle
`MainTournament`	Gère le tournoi
`Pool` et `SingleEliminationTournament`	Permettent de gérer un ensemble de rencontres soit « tout le monde rencontre tout de monde » soit par élimination directe
`PoolTournament`	Pour gérer toutes les poules
`Player`	Un participant qui peut être un individu ou une équipe
`Match`	Une rencontre (un match, un combat, …) entre deux participants
`MainTournamentView`	La classe contenant le *main* et toute l'interface utilisateur

Packages

Dans ce projet, nous travaillerons dans plusieurs *packages*; les classes métiers seront rassemblées ainsi que les classes caractéristiques de la vue.

Package
`g12345.tournament.business`
`g12345.tournament.view`

5 Plan de tests

Plan de tests.....6
Que tester ?.....6
Un modèle pour aider à démarrer ….....6

Avant de commencer à coder son application, c'est un bon usage de planifier et mettre en place des tests unitaires qui pourront être lancés régulièrement pendant le développement de l'application.

Afin de vous y habituer, cette partie du projet vous demande de réaliser un plan de tests. Ce plan de tests est un document écrit[6] qui reprend une liste de tests à faire pour une méthode donnée d'une classe donnée. Un test vérifie qu'une valeur retournée par une méthode est bien la valeur que l'on attendait.

Que tester ?

Nous vous invitons à écrire un plan de tests pour la classe **`MainTournament`**. Nous vous proposons de tester les méthodes : `addPlayer` et `removePlayer`, la méthode `setTurnResult` et les méthodes `getMatchsxxx`.

Un modèle pour aider à démarrer …

Un plan de tests pourrait commencer comme ça[7] …

```
|plan de tests pour le projet « Le grand Tournoi »
|
| 2012-2013
| @author Pierre Bettens (pbt)
|
| La méthode isInscriptionsOpen de la classe MainTournament.
| Les inscriptions sont ouvertes en début de tournoi et sont fermées
| dès l'appel à la méthode closeInscription.
|
|- isInscriptionsOpen()
/**
 * En début de tournoi, retourner vrai
 */

|- isInscriptionsOpen()
/**
 * Après l'appel à la clôture des inscriptions, retourner faux
 */
```

6 Nous vous demandons de nous remettre un document au format PDF, inutile de l'imprimer.

7 J'écris mon plan de tests en texte et pas avec un traitement de texte en insérant des tableaux car c'est plus facile pour moi ensuite d'écrire la classe de tests JUnit correspondante … vous faites évidemment comme vous voulez du moment que vous nous remettez un PDF.

6 Le grand tournoi v1

Le grand tournoi v1....7
Classe Config....7
Classe Player....8
Méthodes....8
Classement....8
Récapitulatif de la classe Player....9
Énumération ResultEnum....9
Récapitulatif de l'énumération ResultEnum....10
Classe Match....10
Récapitulatif de la classe Match....11
Classe TournamentException....11
Classe SingleEliminationTournament....11
Les attributs....12
Les méthodes....12
Récapitulatif de la classe SingleEliminationTournament....14
Classe MainTournament....14
Les attributs....14
Les méthodes....15
Récapitulatif de la classe MainTournament....16
Classe MainTournamentView....16
Conclusion....17

Dans cette première partie de l'application, nous gèrerons[8] l'inscription des participants et la gestion d'un tournoi à élimination directe. Nous ne nous occupons pas d'une répartition des participants en poules. Cette répartition en poules sera pour la deuxième partie.

Voici en détail les différentes classes de cette première version du projet.

Classe Config

Cette classe regroupe toutes les constantes (de classe) du projet. Afin de ne pas trainer[9] des valeurs codées en dur et de savoir où sont définies les constantes, nous les rassemblons dans une même classe … et puis, ça servira pour la suite.

À ce stade, elle contient les constantes suivantes :

- ✗ le nombre maximal de participants au tournoi, `PLAYER_MAX_NUMBER ;`
- ✗ les point gagnés ou perdus lorsqu'une rencontre a eu lieu, `POINT_WINNER`, `POINT_LOSER` et `POINT_DRAW`.

8 Nouvelle orthographe de *gérerons*
9 Nouvelle orthographe de *traîner*

Constante	Valeur
`PLAYER_MAX_NUMBER`	30
`POINT_WINNER`	2
`POINT_DRAW`	1
`POINT_LOSER`	0

Classe Player

Cette classe représente un participant *(player*). Un participant a comme attributs :

- ✗ `name : String`, un nom et ;
- ✗ `points : int`, un nombre de points.

Un participant peut être un individu ou une équipe, ça ne change rien.

Outre ces deux premiers attributs, la classe aura un identifiant unique (attribut **id** de type `int`) incrémenté automatiquement chaque fois que l'on crée un participant *(player).*

Pour permettre cette auto-incrémentation, il faut ajouter un attribut de classe, *nextId* par exemple, qui donnera sa valeur à l'attribut *id* et s'incrémentera.

Méthodes

Cette classe contient un constructeur à un paramètre (le nom du participant), des accesseurs (`String getName()` et `int getPoints()`) et un mutateur[10] (`void setPoints(int)`) pour ses attributs, elle récrit également la méthode `toString`.

Cette classe proposera également une méthode permettant d'ajouter des points à ceux déjà gagnés ;

```
public void addPoints(int)
```

Classement

Si l'on veut donner un classement[11], il faut pouvoir ordonner une liste de participants par ordre de points. Java permet de signaler quels éléments peuvent être triés. Dit autrement, il est possible de savoir si une relation d'ordre existe pour un type d'élément[12].

10 Tous les mutateurs ne sont pas utiles, n'écrivez que les pertinents.

11 Cette fonctionnalité ne sera utilisée que dans la version 2 de l'application.
Nous proposons d'écrire la méthode `compareTo` maintenant afin de ne pas devoir revenir sur cette classe plus tard.
Si vous voulez vous convaincre que votre méthode est correctement écrite, rien ne vous empêche d'écrire une petite classe de test (que vous effacerez après) permettant de voir si vous pouvez effectivement trier des participants.

12 Vous savez déjà qu'une telle relation existe pour les entiers (int, long, …), pour les chaines, …

Pour qu'une classe soit triable, il faut que :

- ✗ elle implémente l'interface `Comparable<E>` ;
- ✗ cette implémentation impose l'écriture d'une méthode `int compareTo(E)`[13] qui devra être écrite.

Lorsque cela est fait, nous pouvons écrire quelque chose du style :

```
List<Player> players = new ArrayList<Player>() ;
players.add(…) ;  // Plusieurs fois
Collections.sort(players) ;
```

Récapitulatif de la classe Player

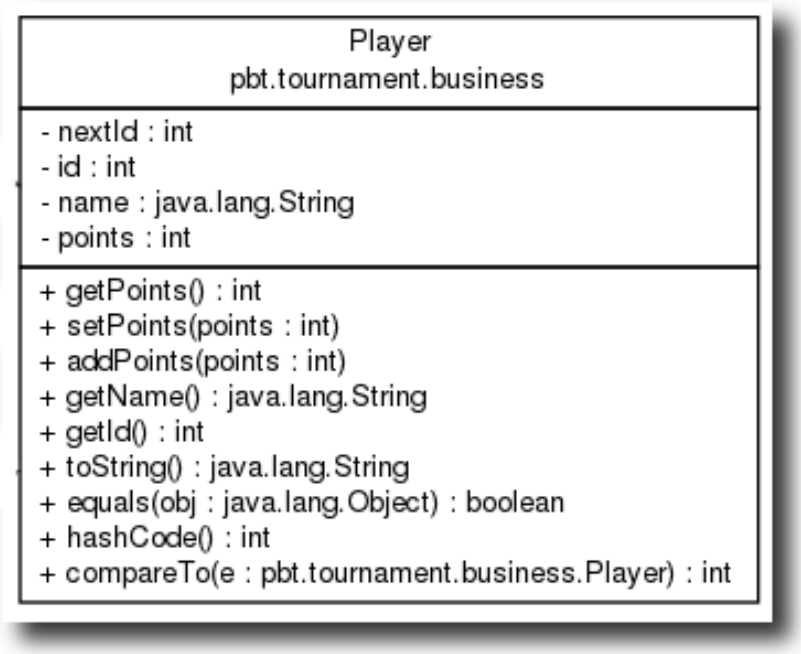

Illustration 1: Extrait de diagramme de classes pour la classe Player

Énumération ResultEnum

Écrivons une énumération qui représentera le résultat que peut avoir une rencontre. Cette énumération a les valeurs suivantes :

- ✗ `NOT_PLAYED`, lorsque le match n'est pas encore joué ;
- ✗ `PLAYER1`, lorsque le premier participant gagne ;
- ✗ `PLAYER2`, lorsque le second participant gagne ;
- ✗ `DRAW`, lorsque le match est nul, pas de gagnant ni de perdant.

13 Consulter la doc de l'API Java pour savoir qu'écrire dans cette méthode

Récapitulatif de l'énumération ResultEnum

Illustration 2: Extrait du diagramme de classes pour l'énumération ResultEnum

Classe Match

Cette classe représente une rencontre (*match*), un match, un combat, … elle a comme attributs :

- ✗ deux **participants** (*player*)[14] ;
- ✗ `id : int`, un **identifiant unique** et ;
- ✗ `result : ResultEnum`, un **résultat** (*result*)**.**

Outre le constructeur à deux paramètres[15], les accesseurs (int `getId()` et ResultEnum `getResult()`), le mutateur (void `setResult(ResultEnum)`[16]) et la méthode `String toString()`, la classe aura les méthodes suivantes ;

```
boolean isDone()
Player getWinner()
Player getLoser()
```

La méthode `setResult` se chargera de mettre à jour les points du joueur en fonction du résultat obtenu.

La méthode `isDone` retourne vrai si la rencontre a eu lieu, faux sinon.

La méthode `getWinner` retourne le gagnant s'il existe, `null` sinon. Même chose pour la méthode `getLoser`.

14 Je n'en dis pas plus afin de vous laisser le choix du type pour cet attribut ; deux variables, un tableau, …

15 Les deux paramètres sont bien les deux participants, le résultat étant initialisé par défaut à « non joué ».
Vous remarquez que l'on vous laisse le choix pour stocker les deux participants ; deux attributs ou un tableau de participants ou … ? Faites votre propre choix, c'est libre (bien que le diagramme de classes soit représentatif de mon choix) …

16 Lorsque je présente le prototype d'une méthode, je précise le type de retour, le nom et le type des paramètres qui doivent être respectés. Vous avez par contre le choix du nom que vous donnez aux paramètres.

Récapitulatif de la classe Match

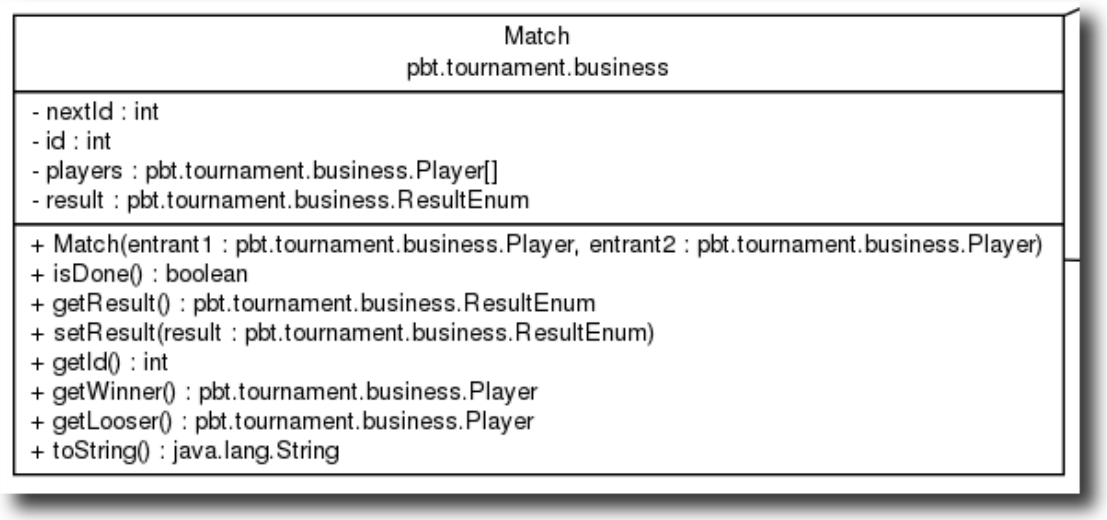

Illustration 3: Extrait du diagramme de classes pour la classe Match

Classe TournamentException

Cette classe est une exception contrôlée par le compilateur. Cette exception sera lancée dans la suite dès que l'on demande à la partie business quelque chose d'incohérent. Dans ce cas l'exception est lancée.

Pratiquement, cette classe hérite de la classe `Exception` et a un constructeur ; le constructeur à un paramètre de type *String* (le paramètre représentant l'erreur qui s'est produite).

Classe SingleEliminationTournament

Un **tournoi à élimination directe** est un tournoi où l'on rencontre, au premier tour, un participant au hasard (tous les couples sont formés) et ensuite, seuls les vainqueurs accèdent au tour suivant. Les rencontres ne peuvent pas engendrer un nul, il faut un vainqueur. Les participants sont éliminés au fur et à mesure, il ne peut en rester qu'un[17].

Il y a donc bien plusieurs tours dans ce tournoi :

- ✗ au premier tour, tous les (*n*) participants sont répartis par couples et les (*n/2*) rencontres se jouent. Chaque rencontre donne un gagnant qui accède au tour suivant ;
- ✗ au second tour, les (n/2) participants restants se rencontrent de la même manière. Les rencontres sont telles que le gagnant de la première rencontre au premier tour rencontre le gagnant de la deuxième rencontre au premier tour et ainsi de suite. À l'issue du second tour, il reste n/4 participants ;
- ✗ …
- ✗ au dernier tour les deux derniers participants se rencontrent, c'est la finale.

17 Dans cette dernière phrase se cache une référence cinématographique que les plus jeunes ne peuvent pas percevoir ;-)

Les attributs

Cette classe a comme attributs :

- ✗ `players : List<Player>`, la liste des participants au tournoi
- ✗ `matchs : List<Matchs>`, la liste des rencontres du tour courant
- ✗ `theLuckyGuy : Player`, un participant qui passe directement au tour suivant si le nombre de participants est impair[18] ;
- ✗ `playersCurrentTurn : List<Player>`, la liste des participants au tour « en cours ». Il faudra ôter le perdant de cette liste à chaque rencontre. Au départ, cette liste est reçue en paramètre, ce sont les participants inscrits au tournoi. Ensuite, à chaque tour, ce sont les gagnants de chaque rencontre ;
- ✗ `matchsHistory : List<List<Match>>`, la liste des listes de rencontres afin de pouvoir regénérer l'historique des rencontres[19] ;

Les méthodes

Cette classe a les méthodes suivantes :

- ✗ un constructeur à un paramètre, la liste des participants. Les rencontres seront calculées en mélangeant la liste et en prenant ensuite les participants deux par deux (ce sera fait dans une méthode privée, *cfr. infra*). Il faudra **copier**[20] dans son attribut `playersCurrentTurn` la liste des participants reçue en paramètre. Le constructeur instancie l'attribut `matchsHistory` qui est pour l'instant vide ;
- ✗ des *getters* ;

```
public List<Player> getPlayers()
public List<Match> getMatchs()
public List<List<Match>> getMatchsHistory()
```

- ✗ une méthode permettant de donner un résultat à une rencontre[21] et ayant comme paramètre l'identifiant de la rencontre et le résultat obtenu ;

```
public void setResult(int, ResultEnum)
     throws TournamentException
```

  Dans ce tournoi, on n'accepte pas que la rencontre soit nulle (pas de vainqueur) et il faut ôter le perdant de la liste des participants classés.

18 Il est bien clair que pour un tour, le premier de la liste rencontre le deuxième de la liste, le troisième rencontre le quatrième et ainsi de suite … il faut donc qu'il y ait un nombre pair de participants. Si le nombre est impair, je choisis un participant au hasard qui passera directement au tour suivant.

19 C'est bien une liste de listes un peu comme un tableau à deux dimensions. Cette liste sert à mémoriser les matchs d'un tour … ces matchs sont dans une liste.

20 Il est nécessaire de copier cette liste car nous allons supprimer des éléments … et l'application veut conserver sa liste complète de participants (même les éliminés) jusqu'à la fin.

21 Lorsque l'on donne un résultat à une rencontre, celle-ci reste dans la liste des rencontres, inutile de la supprimer de la liste car il sera intéressant de la conserver pour la suite

Le code de la méthode teste la valeur du résultat proposé :

- s'il vaut DRAW, je lance une exception ;
- sinon,
  - je récupère le match et positionne le résultat
  - je retire le perdant de la liste des participants du tour en cours

Si *l'id* de la rencontre ne correspond pas ou si le rétultat est NOT_PLAYED, une exception est levée.

- des méthodes permettant d'obtenir les rencontres déjà jouées et celles qui doivent encore avoir lieu ;

```
public List<Match> getMatchsToPlay()
public List<Match> getMatchsDone()
```

- une méthode précisant s'il reste des rencontres qui ne sont pas jouées

```
public boolean hasMatchToPlay()
```

- une méthode permettant d'obtenir une rencontre ayant cet identifiant dans la liste de ses rencontres. Pratique pour changer son résultat ensuite (cette méthode retourne simplement `null` si le match n'est pas trouvé) ;

```
public Match getMatch(int id)
```

- une méthode privée qui génère toutes les rencontres qui doivent avoir lieu[22]. Cette méthode sera appelée par le constructeur ;

```
private abstract void generateMatchs();
```

Pratiquement, cette méthode fera :

- gérer le « chanceux » :
  - s'il y avait un chanceux le tour précédent, l'ajouter à la liste des participants de ce tour ;
  - si le nombre de participants est impair, choisir un nouveau chanceux **au hasard** qui accédera directement au tour suivant et l'ôter de la liste ;
- ajouter les matchs[23] (mélanger la liste des participants et les regrouper deux par deux pour détermirer les matchs) à la liste des matchs après l'avoir instanciée ;
- ajouter cette liste de matchs à l'historique des tours.

- deux méthodes permettant de savoir si c'est le dernier tour et de passer au tour suivant.

```
public boolean hasNextTurn(){…}
public void nextTurn() throws TournamentException{…}
```

La méthode **`hasNextTurn`** précise s'il y a encore un tour après le tour en cours. Il n'y a plus de tour suivant lorsque l'on joue la finale. Je sais que ce dernier tour ne comporte qu'un seul match qui désignera (enfin) le vainqueur de ce grand tournoi.

Passer au tour suivant (méthode **`nextTurn`**), c'est regénérer une liste de matchs à jouer dès qu'il n'y a plus de matchs à jouer. Cette méthode consiste en un simple appel à la méthode `generateMatch`. On ne fera cet appel que s'il n'y a plus de match à jouer sinon, on lancera une exception.

22 Cette méthode a une certaine difficulté logique à laquelle il faudra réfléchir

23 Il suffit de prendre les participants dans l'ordre et deux par deux pour les faire se rencontrer.

Récapitulatif de la classe SingleEliminationTournament[24]

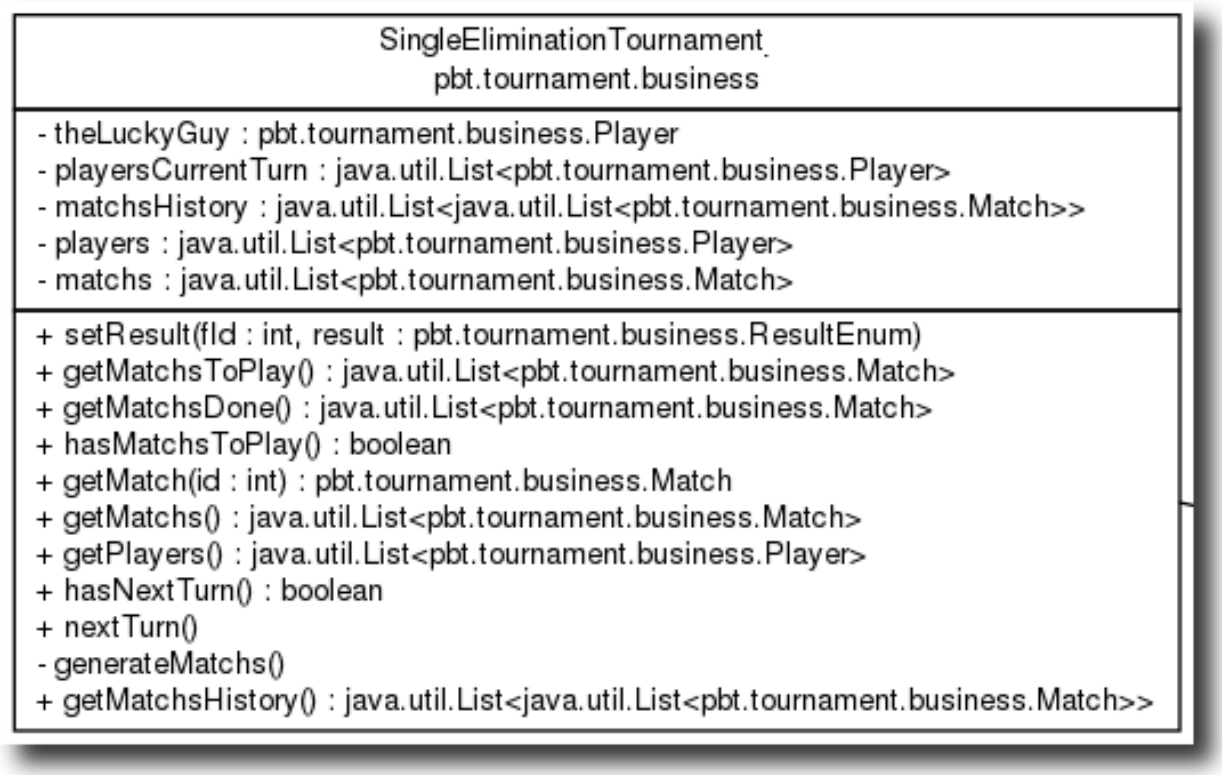

Illustration 4: Extrait du diagramme de classes pour la classe SingleEliminationTournament

Classe MainTournament

C'est cette classe qui est la classe principale de la partie métier de l'application. C'est elle qui sera l'interface avec la partie « vue » (cette vue est l'interface[25] avec l'utilisateur) de l'application.

Les attributs

Cette classe aura au minimum les attributs suivants :

- `players : List<Player>`, la liste des participants au tournoi ;
- `singleEliminationTournament : SingleEliminationTournament`, ce sont les participants fraichement[26] inscrits qui participent au tournoi à élimination directe ;
- et quelques attributs de type booléen qui permettront de se simplifier la tâche ; *inscriptionsOpen* et *turnPlaying*[27].

24 Cet extrait du diagramme de classe provient de la v2. Pour cette première version de l'application, il suffit de rassembler toutes les méthodes dans la même classe, `SingleEliminationTournament`.

25 J'utilise deux fois le même terme. La première fois pour parler de l'interface entre la partie métier et la partie vue et la seconde fois pour parler de l'interface entre l'application et l'utilisateur. Attention de ne pas confondre.

26 Nouvelle orthographe de *fraîchement*

27 Ce sont ceux que j'utilise même s'ils sont un peu redondants.

Le booléen *inscriptionOpen* précise si l'on est dans la phase d'inscription des participants au tournoi. À ce stade on ne peut faire que des ajouts de participants à la liste.

Le booléen *turnPlaying* précise si l'on est dans la phase tournoi à élimination directe.

Les méthodes

La classe proposera les méthodes suivantes :

```
public void openInscription()
public boolean addPlayer(Player)
public boolean removePlayer(Player)
public int getFreePlacesNumber()
public boolean isInscriptionsOpen()
public void closeInscription()

public void setTurnResult(int, ResultEnum)

public List<Player> getPlayers()
public List<Match> getMatchs()
public List<Match> getMatchsDone()
public List<Match> getMatchsToPlay()
public List<Player> getRanking()
public boolean hasMatchsToPlay()
public Player getWinner()
```

La méthode `openInscription` positionne le booléen correspondant à vrai.

La méthode `addPlayer` ajoute le participant s'il reste de la place. Elle retourne vrai si l'ajout a pu se faire.

La méthode `removePlayer` permet de supprimer un participant tant que les inscriptions sont encore ouvertes. Elle retourne vrai si le participant a pu être supprimé.

La méthode `getFreePlacesNumber` permet de savoir s'il reste des places libres pour le tournoi.

La méthode `isInscriptionsOpen` retourne vrai si les inscriptions sont ouvertes et faux sinon … c'est simplement la valeur du booléen correspondant qui est retournée.

La méthode `closeInscription` positionne le booléen correspondant à *false* et initialise `singleEliminationTournament` avec la liste des participants.

La méthode `setTurnResult` permet de donner le résultat d'une rencontre. Cette méthode prend en arguments *l'id* de la rencontre et le résultat.

La méthode `getPlayers` retourne la liste complète des participants au tournoi.

Les méthodes `getMatchs`, `getMatchsToPlay` et `getMatchsDone` retournent une liste de matchs. Cette liste dépend de l'état d'avancement du tournoi.

La méthode `getRanking` donne le classement des participants[28].

La méthode `hasMatchsToPlay` nous précise s'il y a déjà lieu de passer au tour suivant.

La méthode `getWinner` retourne le gagnant lorsque tous les matchs sont joués.

Certaines de ces méthodes lanceront une exception `TournamentException`.

28 Comme la classe Player est Comparable le contenu de cette méthode prend 2 lignes, elle consiste à classer la liste des participants par ordre de points et à la retourner.

Récapitulatif de la classe MainTournament

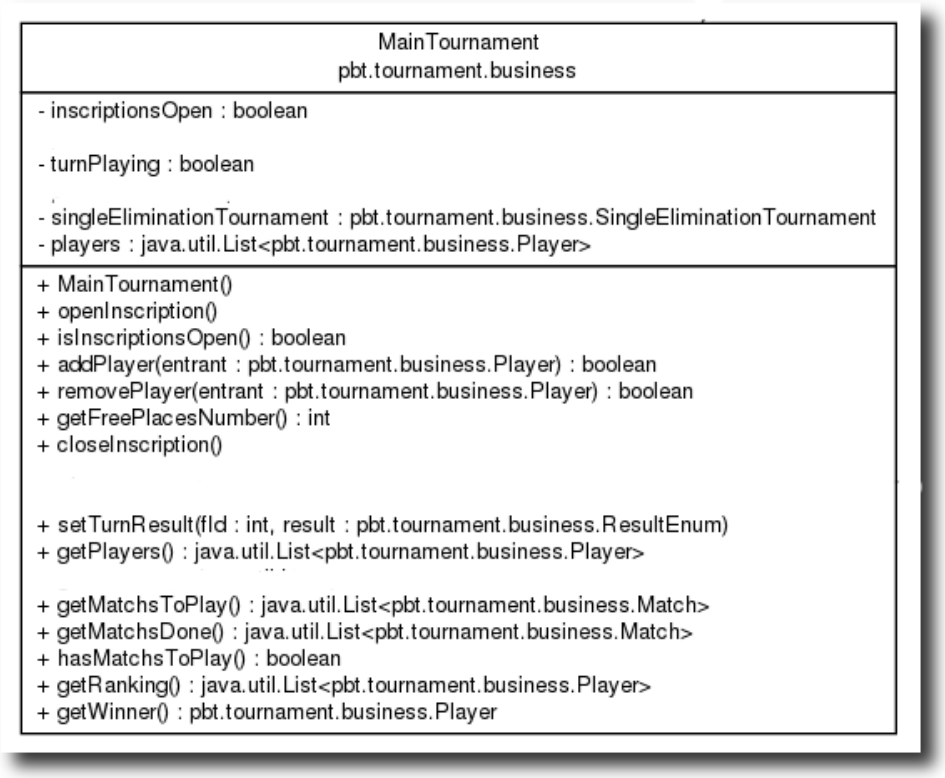

Illustration 5: Extrait du diagramme de classes pour la classe MainTournament

Classe MainTournamentView

Cette classe fait partie du package « view » et est destinée à l'interface utilisateur qui sera dans notre cas **en console**[29]. C'est cette classe qui contient la méthode ***main***.

Nous ne détaillons pas trop cette classe, c'est elle qui devra faire tous les affichages[30] pour l'utilisateur ainsi que les lectures au clavier et faire appel à la logique métier pour faire évoluer le tournoi.

Le fait par exemple qu'une rencontre ait un id unique va vous permettre de l'afficher et de facilement y faire référence pour que l'utilisateur puisse donner un résultat.

Bien qu'écrire une interface texte ne soit pas le plus gai en soi, nous tiendrons compte de son utilisabilité. Il faut que l'interface soit la plus conviviale possible … et sur ce point, nous vous laissons le loisir d'y réfléchir.

29 Si vous vouliez écrire une interface graphique, vous ne devriez modifier que les classes de ce package, rien dans la partie business

30 Pour ma part (ce n'est pas une obligation et si vous le faites ce ne sera pas considéré comme un écart par rapport à l'analyse), j'ajoute une classe Display qui contiendra toutes des méthodes statiques se chargeant des affichages.

Conclusion

À ce stade votre application est fonctionnelle ... dans sa version 1. Vous pouvez préparer sa remise[31], en conserver précieusement une (ou plusieurs) copie(s) et prendre une pause …

… ♬ ♪♬♪

Après cette courte pause, la version 2 verra apparaitre[32] quelques améliorations et modifications de votre application.

31 Avant de remettre un code, on vérifie qu'il est correctement mis en forme; les lignes ne sont pas trop longues, elles sont correctement indentées, il n'y a pas de passage à la ligne intempestif, la javadoc est écrite, ...
Une (re)lecture des conventions d'écriture en Java **CodeConventions** (poÉSI / Aide / CodeConventions.pdf) peut être bénéfique

32 Orthographe nouvelle de *apparaître*

7 Le grand tournoi v2

Le grand tournoi v2....18
Gestion des poules....19
Héritage et abstraction....19
Classe AbstractTournement....20
Les attributs....20
Les méthodes....21
Classe SingleEliminationTournament....22
Classe Pool....22
Classe PoolTournament....23
Classes MainTournament et MainTournamentView....24
Sauvegarder un tournoi....24
Où faire la sauvegarde ?....24
Comment faire la sauvegarde ?....25
Générer des fichiers PDF....26
Conclusion....26

Nous allons ajouter diverses fonctionnalités à notre application, l'occasion d'aborder d'autres notions que les concepts du noyau de Java.

Pour commencer, nous allons ajouter la possibilité de permettre aux participants de jouer un minimum de rencontres en introduisant le concept de **poules**.

Une **poule** est un tournoi où « tout le monde rencontre tout de monde » et les rencontres peuvent être nulles[33]. Tous les participants restent dans la course, ils sont classés en ordre utile, c'est-à-dire par ordre croissant de points. Ce type de tournoi diffère du tournoi à élimination directe :

- ✗ par la manière de choisir la liste des rencontres qui seront jouées ;
- ✗ par le fait qu'aucun joueur n'est éliminé, ils sont simplement classés en ordre utile[34]

Le tournoi se déroulera alors en trois parties ; les inscriptions, les rencontres de poules où personne n'est éliminé et le tournoi à élimination directe **avec les participants issus des poules**.

Deuxièmement, nous ajouterons la possibilité de sauvegarder le tournoi en cours pour le reprendre plus tard mais également pour se prémunir d'une défaillance matérielle.

Et troisièmement, nous donnerons la possibilité de générer des documents au format PDF afin qu'ils puissent être imprimés. On pourra par exemple avoir la liste des participants, la liste des matchs, un récapitulatif des rencontres d'une poule, …

Pour cette **troisième phase**, nous ne donnons pas de diagramme de classes et nous vous dirigerons moins dans la réalisation des diverses fonctionnalités ... à vous d'être plus autonome[35].

33 Au sens qu'il n'y a pas de gagnant, pas au sens « sans intérêt » !
34 Celui qui a remporté le plus de points se trouve en tête, suivi par les autres.

Fonctionnalités de la version 2
Gestion des poules
Sauvegarde du tournoi
Documents PDF

Gestion des poules

Gérer un tournoi à élimination directe ou gérer les rencontres d'une poule sont deux choses différentes et cependant fort proches ; tous deux ont une liste de rencontres qui doivent être jouées et une liste de participants à gérer.

Afin d'éviter de la réécriture de code et comme ces concepts sont logiquement proches nous allons rassembler les similitudes dans une classe abstraite et les deux types de tournoi en hériterons.

Classe abstraite et héritage sont deux concepts qui n'ont été que survolés au cours. Nous n'en ferons qu'un usage très limité et nous vous en (re)donnons les explications de base ici.

Nous aurons une classe abstraite **`AbstractTournement`** et deux classes qui en héritent ; la classe **`Pool`** et la classe **`SingleEliminationTournament`**[36].

Héritage et abstraction

Très brièvement, que recouvrent ces deux concepts[37].

Une classe **hérite** d'une autre grâce au mot clé ***extends.***

Hériter signifie posséder les membres (attributs et méthodes) d'une classe parent. L'héritage sera mis en œuvre dans diverses situations. Celle qui nous occupe est un vil problème de redondance de code qui survient car deux classes sont différentes … mais ont cependant pas mal de similitudes. Nous sommes obligés d'avoir deux classes différentes[38] mais elles sont quand même tellement semblables que ce serait dommage de le nier.

Ce qui les rend proches, c'est qu'elles représentent toutes deux une sorte de tournoi. En tant que tournoi, elles ont toute une série de similitudes (une liste de matchs à jouer par exemple) que nous voudrions rassembler.

Héritage
Mot clé `extends`
Écrire une classe plus spécifique

35 Mais que ceci ne vous empêche pas de poser vos questions. Nous sommes à votre disposition; physiquement 4h par semaine et électroniquement « tout le temps ».
En fait nous vous conseillons de poster vos questions sur *fora* (http://fora.namok.be) où quelqu'un vous répondra sans doute. Nous n'avons **aucune obligation** de vous répondre et n'avons **aucune** contrainte de **délais** ... mais comme « on aime ça » ... Par contre ne posez pas de questions sur le projet par mail. Les profs qui désirent aider électroniquement le font via *fora*.

36 Celle-là est déjà écrite, il faudra cependant la refactoriser.

37 Pour rappel, ces concepts seront expliqués et détaillés l'an prochain … mais ça ne nous empêche pas d'y toucher.

38 Nous parlons bien des classes Pool et SingleEliminationTournament.

Une **classe abstraite** est une classe contenant au moins une méthode abstraite.

Une méthode abstraite est une méthode que l'on **déclare** mais que l'on ne **définit** pas. Ceci implique que la classe abstraite ne pourra pas être instanciée puisqu'elle est incomplète. La classe qui héritera de la classe abstraite devra définir les méthodes abstraites afin d'être complète et de pouvoir être instanciée.

Le mot clé ***abstract*** permet de dire qu'une classe est abstraite.

La classe abstraite sera la classe parent des deux types de tournoi et définira du code qui sera commun aux deux classes enfants.

Abstraction
Mot clé `abstract`
Classe incomplète, certaines méthodes ne sont pas définies

Pour aller plus loin, on pourrait lire

- ✗ http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html et
- ✗ http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

Classe AbstractTournement

Cette classe va nous permettre de récupérer toute une partie du code écrit dans la classe `SingleEliminationTournament` et qui sera utile également dans la classe `Pool`. La classe `Pool` est la classe qui permet de gérer une **poule.**

Que les choses soient bien claires : **tout le code dont on parle dans la suite pour cette classe est récupéré de la classe `SingleEliminationTournament`** (et y est supprimé).

Les attributs

Cette classe a deux attributs dont la visibilité sera *protected* afin que ses enfants puissent accéder aux attributs sans passer par un accesseur :

- ✗ `players : List<Player>`, la liste des participants au tournoi
- ✗ `matchs : List<Matchs>`, la liste des rencontres qui doivent avoir lieu

Les méthodes

Cette classe a les méthodes suivantes (qu'il suffit de déplacer de la classe `SingleEliminationTournament`) :

- ✗ un constructeur à un paramètre, la liste des participants.
  Les rencontres devront être calculées en fonction du type de tournoi, l'attribut *matchs* sera donc initialisé dans chaque classe enfant ;
- ✗ des *getters* et des *setters* ;

```
public List<Player> getPlayers()
public void setPlayers(List<Player>)
public List<Match> getMatchs()
```

- ✗ une méthode permettant de donner un résultat à une rencontre[39] et ayant comme paramètre l'identifiant de la rencontre et le résultat obtenu ;

```
public void setResult(int, ResultEnum)
```

- ✗ des méthodes permettant d'obtenir les rencontres déjà jouées et celles qui doivent encore avoir lieu ;

```
public List<Match> getMatchsToPlay()
public List<Match> getMatchsDone()
```

- ✗ une méthode précisant s'il reste des rencontres qui ne sont pas jouées ;

```
public boolean hasMatchToPlay()
```

- ✗ une méthode permettant d'obtenir une rencontre ayant cet identifiant dans la liste de ses rencontres.
  Pratique pour changer son résultat ensuite (cette méthode retourne simplement `null` si le match n'est pas trouvé) ;

```
public Match getMatch(int id)
```

- ✗ et la méthode abstraite, que devront définir les enfants, qui génère toutes les rencontres qui doivent avoir lieu[40].
  Cette méthode sera appelée par le constructeur ;

```
protected abstract void generateMatchs();
```

  Il faut écrire cette signature dans la classe abstraite et laisser la méthode et son contenu dans la classe `SingleEliminationTournament`. La classe `Pool` récrira également cette méthode (un peu différemment).

39 Lorsque l'on donne un résultat à une rencontre, celle-ci reste dans la liste des rencontres, inutile de la supprimer de la liste car il sera intéressant de la conserver pour la suite.

40 Cette méthode a une certaine difficulté logique à laquelle il faudra réfléchir et ce, dans chaque classe enfant.

Classe SingleEliminationTournament

Cette classe hérite de la classe *AbstractTournament* par le biais de l'instruction suivante. Toute une partie du code qu'elle contenait a été « déménagé ».

```
public class SinqleEliminationTournament
     extends AbstractTournament {…}
```

Pour rappel, cette classe doit écrire la méthode `generateMatchs`. Java aime bien, lorsque l'on récrit une méthode, que l'on ajoute une **annotation**[41] *@Override*.

La signature de la méthode aura donc l'allure suivante[42] :

```
@Override
protected void generateMatchs() {…}
```

Classe Pool

Cette classe représente et permet de gérer les rencontres et les participants d'une seule poule. Dans la suite nous écrirons une classe qui gère toutes les poules (voir Classe PoolTournament p 23).

Dans une poule :

- ✗ chacun joue contre tout le monde ;
- ✗ une rencontre peut-être « nul » (pas de gagnant ni de perdant)

Cette classe hérite de la classe *AbstractTournament*

```
public class Pool extends AbstractTournament
```

En plus des attributs dont elle hérite, elle aura un attribut supplémentaire, un identifiant unique[43] **id** pour lequel vous fournirez l'accesseur.

Le **constructeur** de la classe permet l'attribution de l'id unique, fait appel au constructeur parent[44] et à la méthode `generateMatchs`.

Cette classe a une méthode permettant d'obtenir la liste des participants en ordre utile, c'est-à-dire par ordre croissant de points :

```
public List<Player> getRanking()
```

N'oubliez pas de définir la méthode `generateMatchs`. Cette méthode demande un petit effort de logique[45].

41 Plus d'infos sur ce concept, http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html. Dans le cas de l'annotation @Override, celle-ci permet de se prémunir d'erreurs de typos et d'effectivement récrire la méthode héritée. Cette annotation peut être également ajoutée aux méthodes `toString` et `equals` déjà récrites.

42 Le code est , quant à lui, déjà écrit.

43 Vous avez l'habitude maintenant …

44 Attention, java impose que l'appel au constructeur de la classe parent soit la **première instruction** du constructeur de la classe enfant (via le mot clé ***super***).

45 Dans une poule tous les participants rencontrent tous les autres participants et un participant ne se rencontre pas lui-même ! Deux boucles imbriquées « correctement bornées » font l'affaire.

Remarques.

- ✗ Vous constatez que cette classe est plus simple que la classe `SingleEliminationTournament`.
- ✗ Il faudra faire un détour par la classe `Config` afin d'y ajouter quelques constantes ; POOL_MINIMAL_SIZE et POOL_MAXIMAL_SIZE[46].

Classe PoolTournament

Représente un tournoi en « poules »; plusieurs équipes sont composées de participants et, dans une poule, tout le monde joue contre tout le monde. Cette classe va gérer un ensemble d'objets de type `Pool`.

Cette classe aura les attributs :

- ✗ `pools : List<Pool>`, représentant toutes les poules,
- ✗ `poolSize : int`, un entier représentant la taille des poules.
  Cette taille devra être optimale et calculée sur base de la taille minimale d'une poule et de sa taille maximale. Il faut que les poules soient remplies au mieux[47] !

Le **constructeur** devra créer toutes les poules après avoir calculé la taille optimale de celles-ci sur base de la liste de participants qu'il reçoit en paramètre.

Cette classe proposera les méthodes suivantes :

```
public List<Pool> getPools()
public boolean hasMatchs()
public List<Match> getMatchsToPlay()
public List<Match> getMatchsDone()
public List<Match> getMatchs()
public void setResult(int, ResultEnum)
```

La première est un accesseur, les quatre suivantes pourront être codées par délégation[48] puisque les poules qui constituent la classe offrent ces méthodes et la dernière se fera également par délégation après avoir trouvé la poule contenant la bonne rencontre[49].

46 Vous pouvez par exemple les initialiser à 3 et 6 par exemple.

47 Par exemple avec 20 participants, c'est mieux d'avoir 4 poules de 5 participants plutôt que 3 poules de 6 et 1 poule de 2 ou encore 2 poules de 7 et une de 6.
Je fais un calcul du style « C'est la taille telle que le reste de la division entière du nombre de participants par la taille de la poule est le plus proche du diviseur (la taille de la poule) qui remplit le mieux les poules ».

48 On parle de délégation lorsque la méthode que l'on propose est directement proposée par un attribut, c'est donc cet attribut qui « fera le boulot ». Dans notre cas, ce sera à l'aide d'une boucle qui parcourra toutes les poules en invoquant simplement la méthode qui va bien.

49 Tout se passera bien car les identifiants de rencontres sont uniques.

Classes MainTournament et MainTournamentView

Il faudra revoir la classe `MainTournament` afin de lui ajouter un attribut de type PoolTournament et un attribut de type booléen (`poolPlaying` par exemple) pour gérer la phase de poules.

Le tournoi se déroule en trois phases :

- ✗ la phase d'inscription des participants ;
- ✗ la phase de poules ;
- ✗ la phase d'élimination directe avec les gagnants issus des poules.

Après la phase de poules **les deux gagnants de chaque poule** sont sélectionnés pour participer à la phase suivante.

On ajoutera les deux méthodes suivantes à la classe :

```
public void setPoolResult(int, ResultEnum )
     throws TournamentException;
public void closePoolTournament() throws TournamentException;
```

La première permettant de donner un résultat à une rencontre de poule.

La méthode closePoolTournament permet de clôturer la phase de poules. Elle jette une exception s'il restait des rencontres à jouer. La méthode positionne les booléens correspondants et instancie *singleEliminationTournament.* Pour ce faire, il faut créer une liste de joueurs que l'on passera au constructeur **en prenant les deux vainqueurs[50] de chaque poule**.

Il faudra également adapter la classe `MainTournamentView` afin qu'elle gère cette nouvelle phase de poules.

Sauvegarder un tournoi

Nous allons maintenant donner la possibilité de sauvegarder le tournoi afin de pouvoir le reprendre plus tard ou simplement en faire une sauvegarde par prudence pendant le tournoi.

Nous ne proposons qu'une seule sauvegarde[51] et nous choisissons le nom du fichier.

Où faire la sauvegarde ?

Le mieux est de sauvegarder le tournoi dans le répertoire *home* de l'utilisateur … mais comment le trouver ? Java permet de consulter des propriétés de l'utilisateur et de son environnement.

50 Les deux vainqueurs d'une poule sont bien les deux participants ayant obtenu le plus de points (plus précisément, les deux premiers de la liste … que faire en cas d'égalité du 2e et 3e ? Cette décision doit être prise dans la méthode *compareTo* de la classe *Player).*

51 L'utilisateur peut sauvegarder autant de fois qu'il le veut le tournoi en cours. Deux tournois différents ne peuvent pas être sauvegardés en parallèle.

On pourra donc écrire[52] :

```
System.getProperty("user.home") ;
```

et si l'on choisit comme nom de fichier *tournament.sav*, on pourra écrire :

```
String filename = System.getProperty("user.home")
     + System.getProperty("path.separator")
     + "tournament.sav" ;
```

Comment faire la sauvegarde ?

Java offre la possibilité de sauvegarder directement des objets grâce à un mécanisme qui s'appelle : la **sérialisation** *(serialization)*.

Un bon point d'entrée pour se documenter est la javadoc de l'interface **`Serializable`**.

Il faudra à nouveau adapter les classes `MainTournament` et `MainTournamentView` afin d'ajouter cette fonctionnalité.

En bref
Rendre les classes qui doivent l'être *serializable*
Ajouter deux méthodes[53] *read* et *write*
Adapter l'interface utilisateur

Générer des fichiers PDF

Pour générer un fichier au format PDF, nous allons utiliser une bibliothèque (*library*) externe qui fait le boulot. C'est la bibliothèque **iText**[54] (Site *web*, téléchargement).

Il faudra utiliser des classes comme `com.lowagie.text.pdf.PdfWriter` ou `com.lowagie.text.Document`, ... qui ne font pas partie du JDK standard. Des exemples sont disponibles sur le site de l'auteur, par exemple l'habituel Hello World.

Vous devrez faire deux trois petites choses pour que tout fonctionne bien :

- ✗ télécharger l'archive *itext-5.3.5.zip*, dézipper et considérer le fichier *itextpdf-5.3.5.jar*[55] ;
- ✗ déplacer ce fichier jar dans le répertoire qui va bien (par exemple /usr/local/java/) et créer un lien soft de *itextpdf-5.3.5.jar* vers *itextpdf.jar* [56];
- ✗ ajouter à la variable d'environnement CLASSPATH le fichier (son nom complètement qualifié) jar de la bibliothèque ;
- ✗ (automatiser le tout dans votre .bashrc)

52 Java permet également de récupérer la valeur d'une variable d'environnement. On pourrait donc écrire System.getEnv("HOME") … qui fonctionnerait très bien sous GNU Linux mais pas sous MS Windows. Java demande d'utiliser une « propriété » plutôt qu'une variable d'environnement lorsque les deux existent. Voir http://docs.oracle.com/javase/tutorial/essential/environment/env.html (tout en bas).

53 Quel est le meilleur endroit pour écrire ces méthodes ?

54 Site web http://itextpdf.com/

55 Un fichier jar est généralement une arborescence de classes (.class) archivées dans un seul fichier.

56 Sur *linux1*, cette étape sera faites pour vous … chez vous non !

Vous créerez au minimum deux PDF différents au choix et avec l'accord de votre professeur. Choisissez ceux qui vous paraissent utiles[57].

Vous en avez maintenant l'habitude, il faudra adapter les classes `MainTournament` et `MainTournamentView` afin d'ajouter cette fonctionnalité.

En bref
Adapter son environnement de travail à la nouvelle bibliothèque
Ajouter des méthodes créant les pdf choisis
Adapter l'interface utilisateur à cette nouvelle fonctionnalité

Conclusion

Voilà qui clôture la troisième et dernière partie de votre projet. Vous êtes arrivé au bout !

Il reste maintenant à bien veiller à respecter les consignes de remises (ce serait dommage de rendre son travail trop tard) … et à préparer la défense du projet (voir Défense orale du projet p 30).

57 Je trouve que la liste des rencontres (par poule) avec de la place pour y indiquer les résultats est un bon exemple. Ou bien le classement des participants. Ou encore l'historique des différents tours dans un tournoi à élimination directe …

8 Modalités d'évaluation

Modalités d'évaluation....27
Attribution de la cote de base....27
Critères modifiant la cote....27
La Javadoc....28
Lisibilité du code....28
Utilisation des éléments adaptés du langage....28
Écarts par rapport à l'analyse....28
Interface utilisateur....29
Orthographe et grammaire....29
Modalités de remises....29
Date butoire....29
Contenu et support du rapport....29
Remise des versions intermédiaires....30
Défense orale du projet....30

Ce paragraphe vous explique les modalités d'évaluation du projet.

À lire attentivement, régulièrement des étudiants sont surpris de notre manière d'évaluer ou de nos contraintes de remises.

Attribution de la cote de base

Nous allons d'abord vous attribuer une cote en fonction de ce qui a été réalisé et testé (le projet est accompagné d'une liste des fonctionnalités attendues avec leur pondération).

Vous ne serez évalué à l'étape *i+1* que si vous avez remis en temps et en heure l'étape *i*[58].

Une fonctionnalité dont on demande les tests, ne sera considérée comme réalisée que **si vous fournissez un programme de test qui montre que cela fonctionne effectivement**. C'est donc à vous de prouver que quelque chose a été fait et pas à votre professeur à le découvrir à partir de votre code. En pratique, vous devez montrer que votre code passe une série de tests JUnit. Des classes de test vous sont parfois fournies par les enseignants, d'autres devront être écrites par vos soins.

Critères modifiant la cote

Un programme qui fait ce qu'on lui demande, c'est bien ... mais loin d'être suffisant. Nous sommes également attentifs au style, à la documentation, ... Votre cote sera donc **réduite** si les différents critères décrits ci-dessous ne sont pas rencontrés.

58 En cas de soucis / problème / litige / désaccord, venez présenter l'examen de laboratoire.

La Javadoc

Javadoc	jusqu'à **8 points** de pénalité

Une javadoc complète est essentielle. Est-ce que tout est décrit ? Est-ce que les cas particuliers sont documentés ? Est-ce que les balises sont utilisées correctement (`return`, `param`, `throws`, ...) ?

Lisibilité du code

Lisibilité	jusqu'à **8 points** de pénalité

Un code illisible mais qui fonctionne, c'est mal. Nous évaluons ici

- ✗ l'indentation,
- ✗ les noms judicieux de variables et de méthodes,
- ✗ la non redondance de code,
- ✗ l'utilisation de constantes littérales,
- ✗ l'utilisation judicieuse de commentaires,
- ✗ ...

Lire et relire les recommandations[59] de Oracle / Sun

Utilisation des éléments adaptés du langage

Connaissance du langage	jusqu'à 5 **points** de pénalité

Nous évaluons ici si vous utilisez à bon escient les différents éléments du langage.

Exemple : Vous serez sanctionnés si, pour initialiser un petit tableau avec des constantes, vous le faites en assignant case par case plutôt que d'utiliser la possibilité offerte par le langage de spécifier toutes les valeurs via une liste entre accolades lors de la déclaration du tableau.

Écarts par rapport à l'analyse

Analyse	jusqu'à **8 points** de pénalité

L'énoncé reprend déjà une analyse plus ou moins développée du problème (classes et méthodes à fournir). Il est impératif de suivre cet énoncé. Tout écart sera sanctionné. (sauf accord particulier avec votre professeur)

59 Document *CodeConventions* disponible sur poÉSI entre autre .

Interface utilisateur

Interface	jusqu'à **2 points** de pénalité

Nous accordons peu d'importance à l'interface utilisateur du programme. Il faut néanmoins que l'on puisse comprendre ce qu'il écrit et qu'on puisse interagir avec lui. Cette interface doit être **fonctionnelle** et **utilisable.**

Orthographe et grammaire

Orthographe	jusqu'à **5 points** de pénalité

Vous devez soigner le style de votre javadoc. Nous sanctionnerons les **fautes d'orthographe**, les problèmes de **grammaire**, les phrases incompréhensibles.

Modalités de remises

Date butoire

Nous vous avons indiqué une date et votre professeur a précisé cette date en indiquant un **jour et une heure** de remise de chacunes des phases du projet.

Celui-ci doit-être remis en main propre[60] **au plus tard à cette date**[61]. Tout projet remis après la date ultime ne sera pas corrigé. Aucune excuse ne sera tolérée. Et si vous êtes **malade** ? Arrangez-vous pour le faire remettre par quelqu'un d'autre avant la date ultime.

Les projets remis en retard ne seront pas évalués[62]**.**

Contenu et support du rapport

Voyez avec votre professeur sur quel support informatique vous devez rendre le rapport[63]. Il doit contenir :

- ✗ toutes les sources,
- ✗ les versions compilées,
- ✗ la javadoc

60 C'est-à-dire que la remise doit se faire pendant un laboratore et prend un peu de temps. Votre professeur devra faire certaines vérifications lors de la remise.
61 Date *butoire* veut dire que l'on peut remettre avant ... mais pas après.
62 Oui je sais, on répète;-)
63 Normalement ce sera sur *linux1* mais certains d'entre nous demandons une copie dans une archive.

Tout doit être **directement utilisable.** Nous ne procéderons à aucune compilation ou génération de javadoc pour vous. **Aucun document ne doit être imprimé** sauf si demandé explicitement.

Remise des versions intermédiaires

Vous avez l'obligation de nous remettre une version intermédiaire de votre travail **une fois par semaine** (dans le casier électronique sur *linux1* via la commande *casier*). Cela nous permet de suivre votre évolution.

- ✗ Cette remise est obligatoire sous peine de non correction de votre projet.
- ✗ Ces versions intermédiaires ne seront pas évaluées. Elles n'interviendront donc pas dans l'établissement de votre cote.
- ✗ Votre professeur ne doit pas vous rappeler de faire ce dépôt[64], vous le savez.
  Si vous n'avez pas beaucoup avancé ? Remettez le peu que vous avez déjà fait.

Défense orale du projet

La défense du projet n'a qu'un seul but; nous **convaincre que vous êtes l'auteur du travail**.

Pour ce faire nous vous poserons des questions sur ce que vous avez fait, pourquoi vous l'avez fait comme ça et nous vous demanderons des petites modifications / améliorations de votre code.

Bien sûr nous vous encourageons à vous entraider. Mais cela doit rester un échange[65] : discussion sur l'énoncé, comparaison de logiques, entraide pour corriger du code, ...

Aucune copie de code ne sera tolérée.

Si nous trouvons des projets qui sont trop proches (changer le nom des variables ou modifier les commentaires ne suffit pas !) nous mettrons 0 à tous ces projets. Aussi bien celui qui a laissé copier que celui qui a copié. Soyez donc vigilants.

Sur Linux, **adaptez les permissions de votre dossier** contenant le projet et ne laissez rien trainer[66].

Un projet **non défendu** oralement est considéré comme **non remis**[67].

64 ...même s'il le fera plus que probablement
65 Un échange va dans **deux** sens et pas un seul.
66 Nouvelle orthographe de *traîner*
67 La cote pour tout le projet sera nulle. Si nous vous avons déjà remis des cotes intermédiaires, elles sont perdues.