

**TP de Patrons De Conception**  
**Livrable 01**

**Réalisé par :**

- TADJER Amina
- FEKIR Nadjat

**Groupe : SIL2**

**Projet TP3\_3 : Football Game.**

**Proposé par : Mme BOUSBIA**

**Promotion : 2019/2020**

## I. Introduction :

Les problèmes de conception des logiciels sont très souvent de nature standard, répétitifs et récurrents, auxquels plusieurs experts en conception ont appliqué des procédés de résolution, tout en améliorant la lisibilité du code et anticipant les problématiques qui peuvent apparaître plus tard dans la mise en œuvre.

Au lieu de réinventer la roue à chaque fois, et afin d'accélérer le processus de développement, ces expériences ont été capitalisées en un ensemble de patrons de conception et paradigme éprouvé et reconnu comme bonne pratique, en réponse à un problème qui peut être difficilement résolu.

Dans ce premier livrable, nous aurons à faire à un travail de réingénierie et d'analyse d'un système existant (Football Game), des problèmes de conception ont été donc détectés et résolus, tout en respectant et explicitant les patrons de base de la POO et les patrons GRASP.

## II. Analyse du système :

### 1. Analyse du code :

Après plusieurs lectures du code, et après avoir effectué une analyse approfondie du système, nous avons déduit ceci :

Le système est composé de 55 classes, dont des classes de bas niveau (modèles), et des classes graphiques, sachant que dans ce code, il n'y a aucune séparation entre ces deux types de classes.

Nous avons aussi remarqué qu'il y'a des classes, des attributs et des méthodes non utilisés.

Nous allons essayer d'expliquer le fonctionnement de ce système en détaillant certaines classes et leurs méthodes, tous en expliquant leurs rôles.

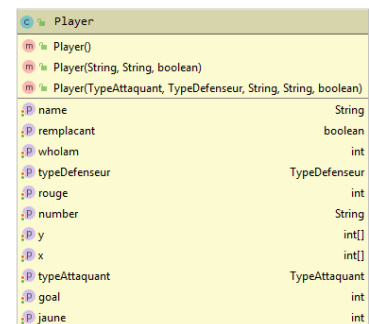
#### a. Player :

C'est la classe qui modélise un joueur de football elle a 9 sous classes :

Guardian, Guardian1, Guard, Defenseur, Defenseur1, Defnsr, Attaquant, Attaquant1, Attaq.

Les différences entre ces classes :

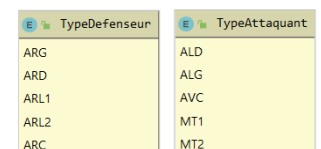
- La différence entre Defenseur et Defenseur1( respectivement entre Attaquant et Attaquant1 aussi entre Guardian et Guardian1) est que la première est utilisée pour une équipe, et l'autre pour son équipe adverse (différence dans les données d'initialisation -positionnement sur le terrain qui dépend aussi du type de Défenseur ou d'attaquant). Mais cette petite différence ne nécessite pas la duplication de la classe.
- Les classes respectivement Defnsr, Attaq et Guard n'ont pas initialisé les coordonnées contrairement à Attaquant et Defenseur et Guardian ( Une mauvaise conception, et non exploitation de l'un des principes de bases de POO qui est le polymorphisme).



Player	
Player()	
Player(String, String, boolean)	
Player(TypeAttaquant, TypeDefenseur, String, String, boolean)	
name	String
remplaçant	boolean
wholam	int
typeDefenseur	TypeDefenseur
rouge	int
number	String
y	int[]
x	int[]
typeAttaquant	TypeAttaquant
goal	int
jaune	int

- Les méthodes de la classe Player sont des setters et des getters, en plus de la méthode wholam qui indique si le joueur est un défenseur ou attaquant ou gardien.

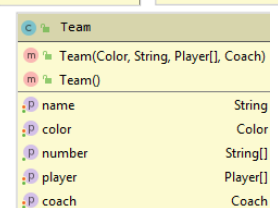
#### b. Les classes de types Enumération TypeDefenseur et TypeAttaquant pour préciser le type du défenseur ( respect de l'attaquant).



TypeDefenseur	TypeAttaquant
ARG	ALD
ARD	ALG
ARL1	AVC
ARL2	MT1
ARC	MT2

#### c. Team :

Elle modélise une équipe ayant un coach et un ensemble de joueur, un nom et une couleur.



Team	
Team(Color, String, Player[], Coach)	
Team()	
name	String
color	Color
number	String[]
player	Player[]
coach	Coach

**d. Coach :**

Elle modélise le coach d'une équipe, elle a un attribut unique name, et des getter et setters.

**e. Ball :**

Elle modélise un ballon, elle a comme attribut ses coordonnées ainsi que la couleur du ballon.

**f. Game :**

Elle modélise un match (entre 2 équipes) ainsi que le résultat du match.

Cette classe est très mal conçue, car elle a beaucoup de responsabilités, qui ne doivent pas y figurer ( des méthodes pour l'affichage et des méthodes pour le calcul, pour gérer la balle, les joueurs.....).

D'après ce qu'il existe, nous pouvons expliquer son rôle ainsi :

- Elle a comme attributs :

2 équipes (Team A et Team B)

La liste des Player.

Le nombre de buts de chaque équipe (Score).

Un ballon et ses coordonnées.

- Elle a comme méthode :

Passer(Player) et Passer1 : mettre à jour les coordonnées de la balle on la passant à un joueur.

Tirer(String ) et Tirer1 : mettre à jour les coordonnées de la balle on la passant à une distance donnée.

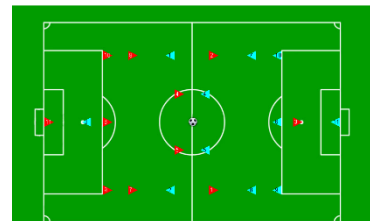
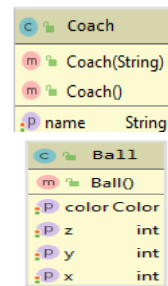
Change(String str,String str1) et Change1 : faire un remplacement d'un joueur par un autre, dans la 1<sup>ère</sup> et la 2<sup>ème</sup> équipe change et change1 respectivement.

- Pour les méthodes précédentes, il y'a une duplication car l'une est utilisée pour l'équipe A et l'autre pour l'équipe B.

Jaune(String str,String str1) : et Rouge affecter une carte jaune ou une carte rouge pour les joueurs des 2 équipes( Le joueur qui a deux cartons jaunes aura une rouge).

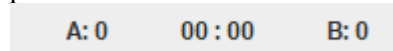
PaintComponent : est une méthode pour l'affichage graphique du terrain ( le terrain vert avec les joueurs qui ne sont pas exclus du jeu -n'ont pas de cartons rouges- des deux équipes et la balle), voir la figure à côté.

North,Sud,East,West : Des méthodes pour détecter si les joueurs sont hors zone( gardiens et défenseurs, selon leur positionnement sur le terrain).



**g. Clock :**

Elle modélise la durée d'un match, mais, dans cette conception elle permet aussi d'afficher les scores et les équipes (barre ci-dessous) :



Elle a comme attribut :minute, second, time, label, game.

Ses méthodes :

DisplayHour : pour afficher dans un label le temps ainsi que le nom de chaque équipe et le score.



- h. Les classes East et West :** Ce sont des classes graphiques, l'une pour l'affichage de la partie gauche de la fenêtre du jeu et l'autre pour l'affichage de la partie droite (pour TeamB respect TeamA)

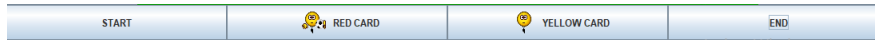


### i. Panell :

C'est une classe utilisée pour l'affichage, elle est de type JPanel.

Elle se compose de deux classes internes :

South : Utilisée pour l'affichage une partie de la fenêtre (la barre ci-dessous).



North : Utilisée aussi pour l'affichage d'une partie de la fenêtre (la barre ci-dessous ), elle utilise la classe Clock, car elle affiche l'heure et le score.



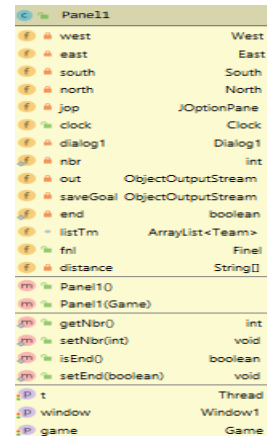
Panell utilise aussi West et East pour l'affichage des parties à gauche et à droites (classes citées précédemment), et Window1 pour la fenêtre, et Dialog( on va l'expliquer juste après).

Panell utilise aussi Game et une liste d'équipe, elle essaye de combiner ces dernières classes avec les éléments graphiques pour afficher à l'utilisateur le jeu de football, et elle utilise d'autres méthodes pour simuler le jeu.

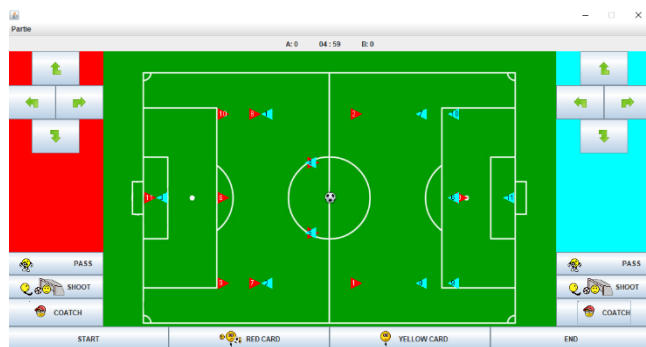
**j. Dialog et Dialog1** : Ces deux classes sont de type JDialog, elles utilisent la classe Game et une liste de joueurs, et des composant JPanel, pour l'affichage des boites de dialogues avec l'utilisateur.

**k. Window1** : Cette classe a pour rôle d'englober Panell dans une fenêtre pour l'afficher.

**l. Test** : La classe qui contient la méthode main qui crée les joueurs, les équipes, les coachs, le jeu Game, Panell et Window1 pour exécuter le code.



Les classes citées précédemment, contribuent pour la simulation d'un jeu de football simple entre deux équipes A et B, tout en suivant une certaine logique et certaines règles ( que les méthodes définissent). On aura l'affichage suivant en exécutant la méthode main :



Dans le code source, il y'a un package nommé tournoi, qui contient 34 classes( dont 3 citées précédemment-Guard et Defnsr et Attaq-), son rôle est de contenir les classes responsables pour simuler un tournoi. Dans ce qui suit nous allons expliquer le fonctionnement de ce Package.

### m. EtatTornei :

Elle donne l'ensemble des équipes du tournoi ainsi que le niveau atteint.

### n. FenetreEntrainement :

Ainsi que FentreEntrainement\_1 ce sont des classes graphiques, elles implémentent une méthode actionPerformed qui associe des actions aux boutons de la fenêtre :

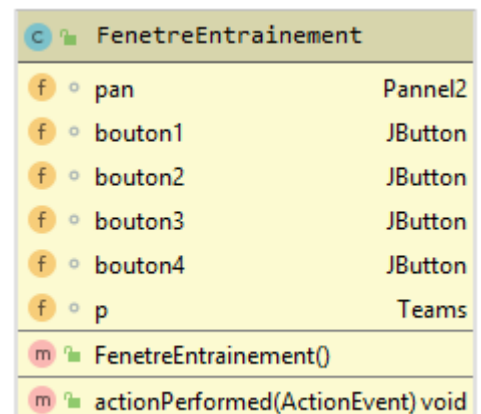
bouton1 : pour jouer directement sans passer par le choix des équipes.

Bouton2 :pour choisir les équipes.

Bouton3 : pour retourner à la fenêtre principale.

Bouton4 : pour quitter le jeu.

La seule différence entre les 2 classes c'est que la classe fenetreEntrainement\_1 ne donne pas la possibilité de ne pas choisir les équipes.



**o. FenetrePrincipale :**

Ainsi que fenetrePrincipale\_1 ce sont des classes graphiques, elles implémentent la méthode actionPerformed qui associe des actions aux boutons de la fenêtre :

Bouton1 :un nouveau tournoi.

Bouton2 :pour charger le tournoi précédent et continuer le jeu.

Bouton3 : pour charger la fenêtre d'entraînement.

Bouton4 : pour accéder aux options du jeu .

La différence entre les 2 classes, c'est que la classe fenetrePrincipale\_1 implémente la méthode play(String musique) pour activer la musique.

FenetrePrincipale		
f	loadGame	ObjectInputStream
f	pan	Pannel
f	bouton1	JButton
f	bouton2	JButton
f	bouton3	JButton
f	bouton4	JButton
f	bouton5	JButton
m	FenetrePrincipale()	
m	actionPerformed(ActionEvent)	void

**p. ChoixEquipe :**

Une classe graphique qui permet de choisir à partir d'un menu un ensemble d'option.

**q. InitialTm :**

elle contient 2 équipes comme attribut, et affiche les informations relatives à chaque équipe.

**r. Levels :** pour le choix de nombre d'équipes du tournoi, 4 ou 8.

**s. Teams :** pour l'écriture des équipes dans des fichiers.

**t. Buts :** sauvegarder les buts dans des fichiers.

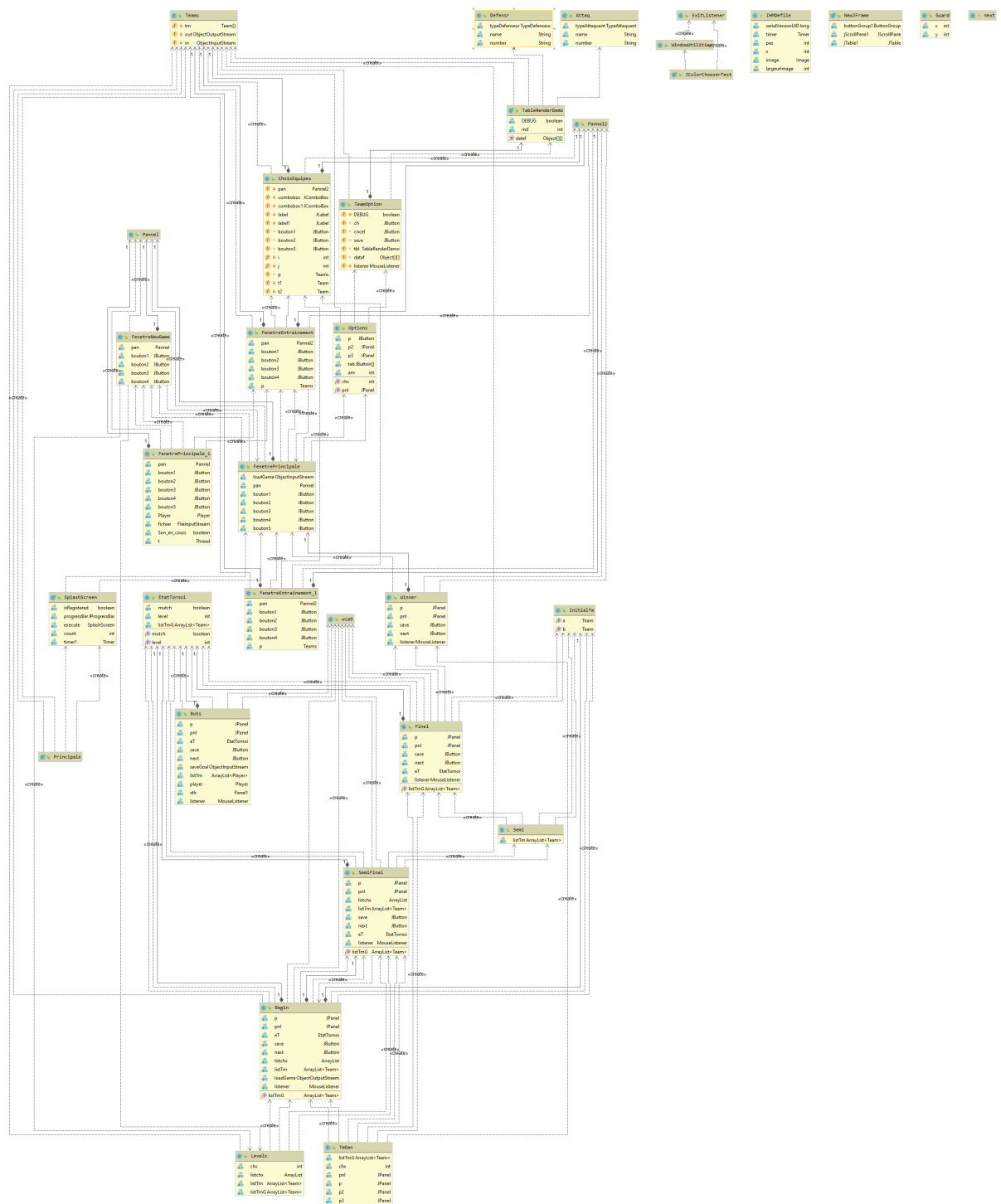
**u. Begin, SemiFinal, Semi, Finel :** représentent les tours du tournoi (en ordre), les équipes gagnées dans chaque tour passent dans une liste à l'autre Tour.

**v. TeamOption :** pour choisir les options des équipes (couleurs...)

**w. Les autres méthodes du package**(Winner, wzaq, IHMGefile,JColorChooserTest, NewJFrame, Options ,Panne2, windows, Principale ,SplashScreen, , TmGan..) sont des classes d'affichages pour le choix d'options ou autres...

(Il suffit de zoomer l'image est de bonne qualité)

Le diagramme du Package tournoi : (Zoommez, l'image est de bonne qualité)



Juste en voyant le diagramme, on peut conclure qu'il y'a beaucoup de dépendances entre des classes, et même beaucoup de créations ( une classe peut être créé par plusieurs classes). Ce qui nous mène à détecter sans voir le code, le problème du fort couplage et de Creator non respecté. On remarque aussi qu'il y'a des dépendances entre des classes de bas niveau ( modélisations) avec des classes purement graphiques, ce qui nous mène à déduire que l'indirection et le contrôleur n'ont pas été respectés.

### 3. Identification des problèmes :

#### Problème 01 :

##### a. Explication du problème et captures :

Tout d'abord, de première vue, on remarque qu'il y'a des classes dupliquées et qui font exactement la même chose, avec une petite différence dans l'initialisation des paramètres ( dans le constructeur) comme dans : Attaquant et Attaquant1, Guardian et Guardian1, Défenseur et Défenseur1...

Attaquant		
f	typeAttaquant	TypeAttaquant
f	x	int
f	y	int
m	Attaquant(TypeAttaquant, String, String, boolean)	

Guardian		
f	x	int
f	y	int
m	Guardian(String, String, boolean)	

Attaquant		
f	typeAttaquant	TypeAttaquant
f	x	int
f	y	int
m	Attaquant(TypeAttaquant, String, String, boolean)	

Attaquant1		
f	typeAttaquant	TypeAttaquant
f	x	int
f	y	int
m	Attaquant1(TypeAttaquant, String, String, boolean)	

Guardian1		
f	x	int
f	y	int
m	Guardian1(String, String, boolean)	

Attaquant1		
f	typeAttaquant	TypeAttaquant
f	x	int
f	y	int
m	Attaquant1(TypeAttaquant, String, String, boolean)	

Il y'a aussi les classes **Dialog** et **Dialog1** qui sont presque les mêmes.

Voyons un exemple du code de Guardian et Guardian1 : (zoomer pour voir)

```
package Game;
public class Guardian extends Player{
    private int x = 70;
    private int y = 240;
    public Guardian(String name, String number, boolean rempalcant) {
        X[0]=0+x;
        X[1]=0+x;
        X[2]=0+x;
        Y[0]=0+y;
        Y[1]=0+y;
        Y[2]=0+y;
        super.name = name;
        super.number = number;
        super.rempalcant = rempalcant;
    }
}
```

Mêmes signatures, même fonctionnement, une légère différence( dans l'initialisation de quelques attributs) et de même pour Attaquant et Attaquant1, Défenseur et Défenseur1.

```
package Game;
public class Guardian1 extends Player{
    private int x = 685;
    private int y = 240;
    public Guardian1(String name, String number, boolean rempalcant) {
        X[0]=20+x;
        X[1]=20+x;
        X[2]=0+x;
        Y[0]=0+y;
        Y[1]=20+y;
        Y[2]=10+y;
        super.name = name;
        super.number = number;
        super.rempalcant = rempalcant;
    }
}
```

Nous avons aussi remarqué qu'il y'a beaucoup de blocs et de parties dupliqués, parfois dans la même classe, et parfois dans des classes différentes ( nous allons faire une seule capture, en raison d'espace (plus de **50 blocs dupliqués**), voici une capture de [src/Game/tournoi/Begin.java](#) :



```

JTable table = new JTable(rowData, columnNames);
table.setShowGrid(true);
    table.setAlignmentX(30);
    table.setAlignmentY(100);
table.setRowHeight(80);
table.setMaximumSize(new Dimension( width: 100, height: 200));
// table.setSize(new Dimension(300,400));
// table.setForeground(Color.green);
table.setFont(titleFont);
table.setEnabled(false);
JScrollPane scrollPane = new JScrollPane(table);

save.setMaximumSize(new Dimension( width: 150, height: 30));
next.setMaximumSize(new Dimension( width: 150, height: 30));
next.addMouseListener(listener);

save.addMouseListener(listener);
p.add(save);
p.add(Box.createRigidArea(new Dimension( width: 100, height: 100)));
p.add(next);
scrollPane.setMaximumSize(new Dimension( width: 500, height: 100));
pnl.add(hh);
pnl.add(scrollPane);
pnl.add(p);
this.add(pnl);
this.pack();
this.setContentPane(pnl);

```

Dupliqué  
dans :

[src/Game/tournoi/Buts.java](#)  
Lines: 76 – 106

[src/Game/tournoi/SemiFinal.java](#)  
Lines: 69 – 99

[src/Game/tournoi/Finel.java](#)  
Lines: 63 – 92

De même pour les lignes (46-80) de FenetreEntrainement.java et (48-83) de FenetreEntrainementI.java, aussi pour (106-150) de FenetreEntrainement.java et (11-55) de Test.java, pour les classes Est.java et West.java aussi il y'a une grande partie du code dupliqué, .....

#### b. Patron ou Principe non respectés :

**DRY** : chaque ligne de code peut être source de bugs, la duplication de cette ligne augmente le risque.

#### c. Répercussions sur la qualité du code (conséquences) :

Donc n'importe quel changement dans une partie du code dupliqué (par exemple la signature d'une méthode appelée dans cette partie dupliquée), ou n'importe quel bug ou vulnérabilité ou code smell, va affecter les autres parties aussi, ce qui va réduire la **maintenabilité( grand cout)** et la **robustesse** du code.

### Problème 02 :

#### a. Explication du problème et captures :

La classe Clock utilise la classe Game ( ce qui est inutile, on en parlera plus tard), et dans cette classe elle fait appel à getName de la classe Team, en l'appelant à travers getGame :

```
game.getB().getName()      game.getA().getName()
```

De même pour la classe Dialog1 :

```
game.getA().getPlayer()[i].isRemplacant()    game.getB().getPlayer()[i].getRouge()
game.getA().getPlayer()[i].getNumber()        game.getB().getPlayer()[i].getNumber()
```

Et dans la classe East et West :

```
!B.getPlayer()[i].isRemplacant()
```

Et dans la classe Game :

```
A.getPlayer()[i].number: A.getPlayer()[i].getX()[0]
```

Et dans la classe Panel1 :

```
game.getA().getPlayer()[i].getGoal()
```

(Nous avons capturé quelques exemples seulement)

Donc le problème dans toutes ces méthodes c'est qu'elles appellent des méthodes des classes récupérées lors d'un appel d'une autre méthode.

#### b. Patrons ou Principes non respectés :

**Variation protégée (GRASP) : Ne pas parler aux inconnus (Loi de Demeter)** car les classes interrogent des classes qu'elles ne connaissent pas.

**Fort couplage (GRASP) :** Car la classe ne sera pas dépendante à une seule classe seulement mais à 2 ou à 3, comme dans le premier exemple, aux lieux que Clock soit dépendante seulement de Game elle est dépendante même de Team.

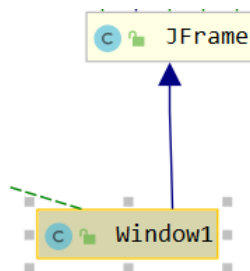
#### c. Répercussions sur la qualité du code (conséquences) :

Le fait de ne pas cacher les détails de l'implémentation du client, et le fait qu'il interroge des inconnus, va rendre difficile la flexibilité de la maintenabilité lors des changements futurs, et ça rend les classes très dépendantes entre elles ( fort couplage), ce qui va affecter la maintenabilité et la réutilisabilité.

### Problème 03 :

#### a. Explication du problème et captures :

Nous avons plusieurs héritages dans le code ( classes mères et classes filles) mais quand un client utilise une classe fille, il doit la déclarer avec le type de la classe mère, ce qui n'est pas respecté dans les captures ci-dessous ;



Mais lors des utilisations de Window1 on la déclare avec son type concret :

Dans Test.java: Window1 window = **new** Window1(panel);

Dans Game.java: Window1 **window**;

Dans Panel.java: Window1 **window** = **new** Window1();

Au lieu de la déclarer comme JFrame, ou bien aux lieux de définir une classe abstraite, et Window1 va hériter d'elle, elle est toujours appelée avec son type concret ( et c'est le cas dans toutes les classes où Window1 a été appelée).

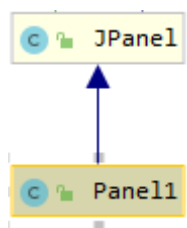
La même chose pour ;

```
ArrayList<Team> listTm = new ArrayList();
```

Qui devrait être déclarée aussi comme List et non pas ArrayList, car si on décide de la rendre une LinkedList ( chose qui est probable), on doit donc tout changer.

Ce problème apparait dans **Game.java, Begin.java, Finel.java, Semi.java, SemiFinal.java, TmGan.java, Levels.java.**

De même pour Panel1 :



Lors de l'utilisation de Panel1, on la déclare aussi avec son type concret, au lieu de la déclarer comme un JPanel.

Exemple dans Test.java : ( Et c'est le cas pour tous les classes qui l'utilisent)

```
Panel1 panel = new Panel1(game);
```

De même pour les classes : West et Est ( filles de JFrame aussi), de Dialog1/Dialog héritent de JDialog, et qui sont appelées dans ( Game.java, Panel1.java ...).

Et c'est le cas pour toutes les autres classes ( surtout celles qui héritent de JPanel, JFrame, JDialog).

### b. Patrons ou Principes non respectés :

**DIP** : les classes clientes ou les objets sont indépendants des classes qui implémentent l'interface.

**Programmer une interface et non une implémentation.**

### c. Répercussions sur la qualité du code (conséquences) :

Si après l'évolution des fonctionnalités on veut rajouter d'autres classes filles, on doit modifier dans le code de la classe cliente, chaque changement va impacter le client ( ça ne respecte pas aussi **OCP**, car nous serons menés à modifier le code au lieu de rajouter seulement, par exemple dans le cas de Window1, si on veut l'afficher d'une autre manière ou modifier les composants West et East), donc on préfère que les deux soient dépendantes d'une abstraction. De plus comme ce sont des classes d'affichages, il y'a une très grande probabilité de changer.

### Problème 04 :

#### a. Explication du problème et captures :

- Il y'a plusieurs chaines de caractères qui sont écrites en dur, et réutilisées plusieurs fois :

" " utilisée 8 fois (Clock.java) :  
`label.setText(String.valueOf(game.getA().getName())+" "+game.getGoal())+" "`

De même dans East.java ( 5fois) :

`"Please, choose a player :"`

Dans Game.java aussi (4 fois) :

`"Gardien hors zone"`

Dans West.java : (5fois)

`"Please, choose a player :"`

Dans ChoixEquipe.java (3fois)

`"Team A"` et `"Team B"`

Dans Options.java:

`"-----\n:MOUSE_PRESSED_EVENT:"` 8fois

`"Astigma"` 16 fois

`":MOUSE_EXITED_EVENT:"` 8fois

`":MOUSE_CLICK_EVENT:"` 8fois

- Il y'a aussi les attributs qui sont probables à changer comme dans attaquant :

`X[0]=0+x;`

`X[1]=0+x;`

`X[2]=20+x;`

`Y[0]=0+y;`

`Y[1]=20+y;`

`Y[2]=10+y;`

Nous avons initialisé sa position sur le terrain, or que si on change seulement les dimensions du terrain, tout va changer, donc on touchera à une classe stable qui est attaquant , à cause d'une variation non protégée.

**Il y'a beaucoup de problèmes de ce genre, dans les initialisations des X,Y,Z.**

### b. Patrons ou Principes non respectés :

**Isoler les parties changeantes :** Le changement n'est pas bien encapsulé.

### Variation protégée (GRASP)

#### c. Répercussions sur la qualité du code (conséquences) :

Un seul changement va entraîner jusqu'à 16 changements (car les affichages sont très changeants), et si on veut changer les fenêtres d'affichage, tout va changer pour les attributs de positionnement.

### Problème 05 :

#### a. Explication du problème et captures :

Les développeurs de ce système, n'ont pas très bien respecté la question : Qui doit avoir la responsabilité de créer : comme dans ces deux exemples (Déjà cité le problème précédemment).

#### Dans Test.java :

On a créé les Player, le coach, et on les a fait passer en paramètre à Team (or que c'est lui a de très grandes raisons pour les créer lui-même).

De même pour Team, on l'a créé dans test, or que c'est Game qui doit la créer (pour chaque Game 2 équipes différentes).

#### Dans FenetreEntrainement.java :

On crée les Player, le coach, Team, Game.

#### b. Patrons ou Principes non respectés :

**Créateur (GRASP)** ce qui induit à un Couplage Fort ( **Couplage faible** non respecté)

#### c. Répercussions sur la qualité du code (conséquences) :

Moins d'opportunités de réutilisabilité, et plus de dépendances de maintenance.

### Problème 06 :

#### a. Explication du problème et captures :

```
if (order.equals(orders[0])){
    for (int i = 0; i < B.getPlayer().length; i++)
        if (!B.getPlayer()[i].isRemplacant() && B.getPlayer()[i] instanceof Attaquant1)
            game.West(B.getPlayer()[i]);
};
if (order.equals(orders[1])){
    for (int i = 0; i < B.getPlayer().length; i++)
        if (!B.getPlayer()[i].isRemplacant() && B.getPlayer()[i] instanceof Defenseur1)
            game.East(B.getPlayer()[i]);
};
```

```
if ((player instanceof Defenseur) && X[0] > 390)
    player instanceof Guardian || player instanceof Guardian1)
    ((player instanceof Guardian1) && Z[0] < 665;
```

Dans la classe East.java et même dans West.java, nous avons appliqué des comportements en connaissant le type concret de la classe utilisée (instance of).

Nous avons aussi la classe Attaq et Attaquant, Defensr et Defenseur, qui sont les mêmes, avec des constructeurs différents, donc les développeurs ont créé de nouvelles classes justes pour utiliser le bon constructeur, or que on pouvait le déclarer dans la même classe en respectant le **Polymorphisme de méthode** (on change les signatures par exemple).

```

public class Defensr extends Player {

    protected TypeDefenseur typeDefenseur ;
    protected String name;
    protected String number;

    public Defensr() {
    }

    public Defensr(TypeDefenseur typeDefenseur , String name, String number, boolean remplaçant) {
        super.name = name;
        super.number = number;
        this.typeDefenseur = typeDefenseur ;
    }
}

```

**b. Patrons ou Principes non respectés :**

**LSP(pour le premier exemple seulement) :** L'appelant n'a pas à connaître le type exact de la classe qu'il manipule, ce qui n'est pas le cas ici.

**Polymorphisme(GRASP) .**

**c. Répercussions sur la qualité du code (conséquences) :**

Il faut vérifier à chaque fois qu'on utilise une classe, ce qui va encombrer le code pour rien.

## Problème 07 :

**a. Explication du problème et captures :**

```

public TypeAttaquant getTypeAttaquant() { return typeAttaquant; }

public void setTypeAttaquant(TypeAttaquant typeAttaquant) { this.typeAttaquant = typeAttaquant; }

public TypeDefenseur getTypeDefenseur() { return typeDefenseur; }

public void setTypeDefenseur(TypeDefenseur typeDefenseur) { this.typeDefenseur = typeDefenseur; }

```

Les classes Attaquant, Defenseur et Gardien sont forcés de dépendre des méthodes qu'elles n'utilisent pas, implémentées dans leurs classes mère Player. Par exemple la classe Gardien implémente les 2 méthodes getTypeAttaquant et getTypeDefenseur de la classe Player sans s'en servir.

**b. Patron ou Principe non respectés :**

**ISP :** les sous classes de la classe Player implémentent des méthodes qui ne les concernent pas.

**c. Répercussions sur la qualité du code (conséquences) :**

Sémantiquement c'est incohérent, les classes filles ont accès et peuvent appeler des méthodes qui ne les concernent, pas ce qui donne un code non compréhensible et même erroné.

## Problème 08 :

**a. Explication du problème et captures**

```

players1[0] = new Attaquant1(TypeAttaquant.ALD, name: "1", number: "1", remplaçant: false);
players1[1] = new Attaquant1(TypeAttaquant.ALG, name: "1", number: "2", remplaçant: false);
players1[2] = new Attaquant1(TypeAttaquant.AVC, name: "1", number: "3", remplaçant: false);
players1[3] = new Attaquant1(TypeAttaquant.MT1, name: "1", number: "4", remplaçant: false);
players1[4] = new Attaquant1(TypeAttaquant.MT2, name: "1", number: "5", remplaçant: false);
players1[5] = new Defenseur1(TypeDefenseur.ARC, name: "1", number: "6", remplaçant: false);
players1[6] = new Defenseur1(TypeDefenseur.ARD, name: "1", number: "7", remplaçant: false);
players1[7] = new Defenseur1(TypeDefenseur.ARG, name: "1", number: "8", remplaçant: false);
players1[8] = new Defenseur1(TypeDefenseur.ARL2, name: "1", number: "9", remplaçant: false);
players1[9] = new Defenseur1(TypeDefenseur.ARL1, name: "1", number: "10", remplaçant: false);
players1[10] = new Guardian1( name: "1", number: "11", remplaçant: false);

Team A;
Team B;
Game game = null;
Coach coach = new Coach();

A = new Team(Color.red, name: "A", players, coach);
B = new Team(Color.cyan, name: "B", players1, coach);

```

La classe FenetreEntrainement est une classe graphique de haut niveau, mais elle a d'autres rôles comme la création des instances de Defenseur Attaquant et Guardian, Game, Team ..des classe concrètes de bas niveau, on remarque aussi que ce code est le même que celui dans la classe test.

#### b. Patron ou Principe non respectés :

**DIP:** la classe FenetreEntrainement est une classe haut niveau elle dépend des classes bas niveau(Attaquant, Gardien...).

**Creator :** la classe FenetreEntrainement n'est pas la responsable de la création des instances des classes bas niveau.

**Fort Couplage :** la classe fenetreEntrainement n'est pas experte en création de plusieurs classes ce qui la rend fortement couplée à ces dernières.

**Faible cohésion :** la classe fenetreEntrainement implémente des méthodes ayant des rôle différents non cohésives entres elles.

#### c. Répercutions sur la qualité du code (conséquences) :

Les classes sont fortement couplées, elles ont une faible cohésion ce qui rend le code difficile à réutiliser et à maintenir.

## Problème 09 :

#### a. Explication du problème et captures :

```

public ArrayList<Team> getListTm() { return listTm; }
public void setListTm(ArrayList<Team> listTm) { this.listTm = listTm; }

public void paintComponent(Graphics g) {...}

public void North(Player player) {...}
public void Sud(Player player) {...}
public void East(Player player) {...}
public void West(Player player) {...}

public void Passer(Player player) {...}
public void Passer1(Player player) {...}
public void Tirer(String number) {...}
public void Tirer1(String number) {...}

```

```

public void change(String str, String str1) {...}
public void changel(String str, String str1) {...}

public void rouge(String str, String str1) {...}
public void jaune(String str, String str1) {...}

public void saveGoal(String number, int i) {...}

public Team getA() { return A; }
public void setA(Team a) { A = a; }
public Team getB() { return B; }
public void setB(Team b) { B = b; }

public boolean posseder(Player player) {...}

public int getGoal1() { return goal1; }
public void setGoal1(int goal1) { this.goal1 = goal1; }
public int getGoal() { return goal; }
public void setGoal(int goal) { this.goal = goal; }

```

La classe Game implémente un très grand nombre de méthodes non cohésives entres elles ayant des rôles carrément différents( déjà expliqué précédemment), donc elle a de multiples responsabilités, dont sémantiquement n'est pas experte ( Et c'est le cas de plusieurs classes, comme JPanel, Begin...).

-passer, passer1, tirer et tirer1.

-north, sud, east et west.

-rouge, jaune.

-getA, setA, getB et setB.

-getGoal1, setGoal1, getGoal et setGoal.

-paintComponent une méthode graphique.

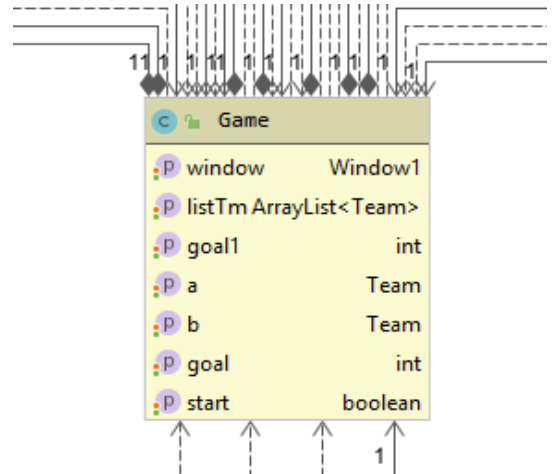
**b. Patron ou Principe non respectés :**

**SRP :** la classe Game n'a pas une responsabilité unique, elle n'a pas un seul objectif fonctionnel.

**Faible Cohésion :** les responsabilités de la classe ne sont pas reliées les unes aux autres, la cohésion de la classe est donc très faible.

**Coupage Fort :** la classe Game est fortement liée à plusieurs classes : window1, Team, Player, Ball, Guardian...

**Expert :** la classe Game ne possède pas toutes les informations nécessaires pour s'acquitter aux responsabilités passer et tirer.



**c. Répercutions sur la qualité du code (conséquences) :**

La classe Game est fortement liée aux autres et cela est dû au fait qu'elle n'a pas un seul objectif fonctionnel, ce qui la rend non réutilisable et difficile à maintenir, de plus cela va rendre la modification très pénible.

## Problème 10 :

**a. Explication du problème et captures (Voir le diagramme UML) :**

A partir du diagramme de classe on peut remarquer que les classes sont fortement couplées entre elles, par exemple les classes Game et Player, et cela est dû au non-respect du principe de l'unicité du rôle de chaque classe, on a des classes qui ont trop de responsabilités non cohésives et dont on n'a pas les informations nécessaires pour les accomplir ce qui crée de plus en plus de liens avec les autres classes ayant ces dernières.

**b. Patron ou Principe non respectés :**

**Fort couplage :** les classes sont fortement liées entre elles.

**Expert :** les tâches ne sont pas assignées aux classes convenables.

**Pure Fabrication :** car on doit créer d'autres classes pour effectuer des tâches du même type.

**c. Répercutions sur la qualité du code (conséquences) :**

Le principe de l'abstraction déjà n'est pas respecté (même responsabilité, même classes), le code est très difficile à comprendre, très difficile à modifier.

## Problème 11:

**a. Explication du problème et captures :**

La classe fenêtre Principale n'est pas ouverte à l'extension la lecture de la musique se fait directement à partir d'un fichier. Si de nouvelles extensions de lectures s'ajoutent, on touche à la conception du code

```

}
public void Play(String musique) {
    try {

        fichier = new FileInputStream(musique);
        Player = new Player(fichier);
    }
}

```

**b. Patron ou Principe non respectés :**

**OCP :** la classe fenetrePrincipale n'est pas fermée aux changements et ouverte à l'extension.

**c. Répercutions sur la qualité du code (conséquences) :**

On ne peut pas faire des extensions sans toucher au code.

## Problème 12:

### a. Explication du problème et captures:

```
public ArrayList<Team> getListTmG() { return listTmG; }

public void setListTmG(ArrayList<Team> listTmG) { this.listTmG = listTmG; }

private final MouseListener listener = new MouseListener() {

    Window1 window = new Window1(panel);
    if (gm.getGoal() > gm.getGoal()) {
        listTmG.add(tm1);
    } else {
        listTmG.add(tm2);
    }
}
```

### b. Explication du problème :

Comme cité précédemment, les classes graphiques de haut niveau dépendent directement des classes concrètes de bas niveau.

La classe Begin dans cet exemple manipule des objets des classes (modèles) et aussi des affichages. Et c'est le cas pour toutes les autres classes, Game, JPanel, East, West...

**Patron ou Principe non respectés :**

**Indirection (GRASP) et Pure Fabrication :** les calculs et la logique sont utilisées dans la même classe qui affiche, sans utiliser une indirection pour effectuer le travail qui ne concerne pas la même classe.

**Contrôleur (GRASP) :** un modèle ne doit pas interagir avec une interface directement, et c'est ce qui n'est pas le cas ici (Presque tout le code).

### c. Répercussions sur la qualité du code (conséquences) :

Fort couplage et dépendances.

## 4. Etude de l'évolution de la réutilisation du système :

Après avoir étudié le système, étudié et compris le code, reconstruit le diagramme des classes et identifié les problèmes, on peut dire que la conception de ce système n'a pas du tout respecté ni les bases de la POO, ni les bonnes pratiques ni les patrons GRASP.

Tout d'abord ce code est très difficile à comprendre, car il n'y a pas de commentaires, en plus les noms des classes ne reflètent pas toujours le rôle de la classe, et de plus quelquefois une seule classe fait le travail de 4 classes. Ensuite pour rajouter n'importe quelle fonctionnalité, il faut changer partout dans le code ( car il y'a un fort couplage d'une part, et car OCP n'est pas respecté), ce qui rend la maintenabilité de ce code très couteuse.

De plus ce système n'est pas du tout évolutif, ni réutilisable, car les développeurs n'ont pas prévu d'éventuelles fonctionnalités en plus, ce qui est très probable.




Sans oublier de citer qu'il y'a des bugs dans le package Tournoi.

## III. Proposition des solutions :

### 1. Solutions aux problèmes identifiés :

#### Solution en général :

Notre solution consiste à créer 3 catégories de classes :

-  Une pour les classes graphiques qui permettent d'afficher les différents composants (niv3)
-  Une pour les classes de modélisation (bas niveau). (niv0)
-  Une qui correspond à un niveau d'indirection et contrôleurs. (niv1)



Notre solution consiste aussi à supprimer les classes inutiles et dupliquées, et ajouter des classes artificielles (Pure fabrication), tout en essayant de résoudre les problèmes cités précédemment, et en respectant les principes de conception et patrons GRASP.

Tout d'abord on peut décomposer notre système à deux sous-systèmes (jeu normal et tournoi).

➤ **Jeu simple :**

Il se compose des classes de :

**Niv0:** Player, Attaquant, Defenseur, Guardian, Coach, Team, TypeDefenseur, TypeAttaquant, Ball et Game ( les classes existent déjà, mais nous allons apporter des modifications dedans).

**Niv3 :** Nous allons créer une classe abstraite Window (extends JFrame), et Window1 qui est la fenêtre du jeu va hériter d'elle, pour donner la possibilité d'ajouter d'autres fenêtres personnalisées ( qui peuvent avoir des attributs en commun). Nous allons aussi créer une classe abstraite pour les panels qui sera nommé APanel, et des sous Classes Panel qui hérites d'elles, cette sous classes Panel va se composer elle-même de plusieurs Panels ( ceci va remplacer les classes Panel1, East, West et les classes internes North et South et Clock car il seront des composants d'un panel principal qui va les créer et les gérer), nous aurons aussi la possibilité de créer d'autres panels qui hérites de Panel si on veut changer la façade. Et de même pour Dialog, on crée une classe abstraite ADialog qui hérite de JDialog, et ensuite toutes les boîtes à dialogues qu'on veut créer seront des classes filles de ADialog.

**Niv2 :** Créer un contrôleur de façade et un gestionnaire de jeu.

➤ **Tournoi :**

**Niv0 :** On aura seulement la classe Tournoi qui est composé d'un ensemble de tours, Tour aussi est une classe qui va remplacer ( Begin, SemiFinal...), pour que on pourra rajouter le nombre qu'on veut de tour sans toucher au code. Chaque Tour se compose d'un ensemble de Game.

**Niv3 :** On garde FentrePricipale et FenetreTournoi qui seront des filles de Window, nous aurons aussi besoins d'un ensemble de Panels et de Dialogs.

**Niv2 :** On rajoute ici des classes d'indirection telle que : gestionTour qui sera responsable de générer les jeux entre les équipes, d'élire les équipes gagnantes, de garder les points et buts... On rajoute aussi generationTournoi (abstraite) qui aura des sous classes generationAleatoire ou lictureFichier.

Y'aura aussi une classe gestionEquipe( qui va ressembler à la classer TeamOption) pour la gestion des équipes, mais aussi va contenir toutes les équipes dès le début, pour ne pas communiquer avec la classe Team après.

Nous rajoutons aussi des contrôleurs de Façades.

**Solution1 :**

Duplication des classes :

Pour Régler le problème1, on a fusionné les classes dupliquées en une seule, vu que la plupart du temps la seule différence entre ces classes sont les données d'initialisation. Donc au lieu de dupliquer les classes, on a opté pour une nouvelle classe au niveau de son constructeur, on affecte les données d'initialisation convenables (DRY respecté).

Exemple :

- les classes Attaquant, Attaquant1 et Attaq, on garde juste la classe Attaquant et on a implémenté 2 constructeurs, un qui initialise les coordonnées et un autre qui ne le fait.

Duplication de code :

Les parties du code dupliquées ont été factorisé en une méthode, qu'on appelle à chaque fois au lieu de réécrire le code.

Exemple :

- le code des 2 méthode `changer` et `changer1` sont pratiquement identiques la seule différence c'est que `changer` c'est de permuter 2 joueurs de l'équipe1 en revanche `changer1` de l'équipe2, on a donc factorisé ces 2 méthode en une méthode `changer`, qui fait appel à la méthode `whichTeam` de la classe `game` qui nous donne l'équipe à laquelle il appartient un joueur.

#### Solution2 :

Afin d'éviter qu'un client utilise des informations des objets indirects à partir des objets directs, des méthodes ont été ajoutées aux classes directes, ce qu'évite de connaître la structure d'autres objets indirectement (patron ne pas parler aux inconnus respecté).

Exemple :

`game.getA().getPlayer()[i].getGoal()` pour éviter cette situation une méthode `getPlayerGoal` est implémenté au niveau de la classe `team`, ce qu'évite de connaître la structure de la classe `Player`.

#### Solution3 :

Afin de régler le problème trois, nous avons ajouté des classes abstraites graphiques (`Window`, `APanel...`), et lors des déclarations de ses classes filles, on les déclare avec ces types abstraits.

De même pour les autres classes comme :

```
ArrayList<Team> listTm = new ArrayList(); → List<Team> listTm = new ArrayList();
```

Donc nous allons programmer des interfaces et non pas des implémentations.

#### Solution4 :

Encapsuler tous les messages d'affichage dupliqués dans des constantes, et pour les positions des joueurs et du ballon, c'est au contrôleur de façade de les initialiser tout dépend de la fenêtre utilisée ( pour permettre de les afficher correctement sur d'autres fenêtres).

#### Solution5 :

On affecte la responsabilité de créer des instances d'objets aux classes qui conviennent (Creator respecté) pour augmenter la cohésion des classes et respecter le patron Créateur (problème 5 et 8).

Exemple :

- On affecte la responsabilité de créer la liste des joueurs d'une équipe et coach à `Team`(la méthode `construirePlayers` dans la classe `Team`).
- On affecte la responsabilité de créer les 2 équipe d'un jeu et un ballon à `Game`(la méthode `construireJeu` dans `Game`).
- On affecte la responsabilité de créer la liste des affrontassions à tournoi(la méthode `construireGames` au niveau de la classe `Tournoi` ).

#### Solution6 :

Pour respecter LSP, on utilise le polymorphisme, on redéfinit les méthodes (s'il est nécessaire) dans les classes filles, pour les appeler directement et chacune a un comportement différent de l'autre, et ceci pour éviter de connaître son type concret.

#### Solution7 :

Attribuer les responsabilités aux classes abstraites et interfaces d'une telle sorte que les sous classes ne seront pas forcées d'implémenter des méthodes qu'elles n'utilisent pas (ISP respecté).

Exemple :

Les méthodes `getTypeAttaquant` et `setTypeAttaquant` sont implémentées au niveau de la classe `Attaquant` et non pas dans `Player` (les autres sous classes comme `Defenseur` et `Gardien` n'implémenteront donc pas ces méthodes) de même pour `getTypeDefenseur` et `setTypeDefenseur` qui sont implémentées au niveau de la classe `Defenseur`.

Comme conséquence la classe `Player` est plus cohésive.

### Solution8 :

Créer des contrôleurs de façades (problèmes 8 et 12) et des classes artificielles (Pure Fabrication) (Problème 8 et 10 et 9) si nécessaire. L'interface interroge le contrôleur et ce dernier interroge la classe voulue :

Exemple : Nous avons rajouté un contrôleur de façade pour le sous-système jeu simple, pour créer le jeu lorsque on clique par commencer, déplacer les joueurs lorsqu'on clique sur les flèches, passer et tirer....

Exemple Pure Fabrication : créer une classe `gestionTournoi` pour enlever la charge sur les autres classes et pour contribuer à les rendre plus cohésives.

### Solution9 :

Attribuer les responsabilités aux classes qui contiennent les informations nécessaires et d'une telle sorte qu'elles auront une seule responsabilité, et que les méthodes qu'elles implémentent soient cohésives entre elles ce qui va réduire le couplage entre les classes (SRP, Expert Fort Couplage sont respectées).

Exemples :

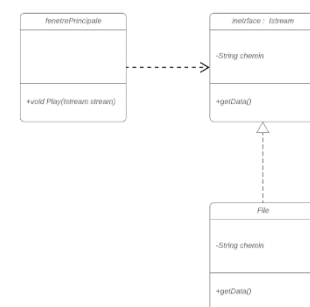
- Les méthodes `passer` et `tirer` sont implémentées au niveau de la classe `Player` au lieu de la classe `Game`, c'est le joueur qui est responsable de passer un ballon à un autre joueur ou de le tirer à une distance donnée et non pas `Game`.

### Solution10 :

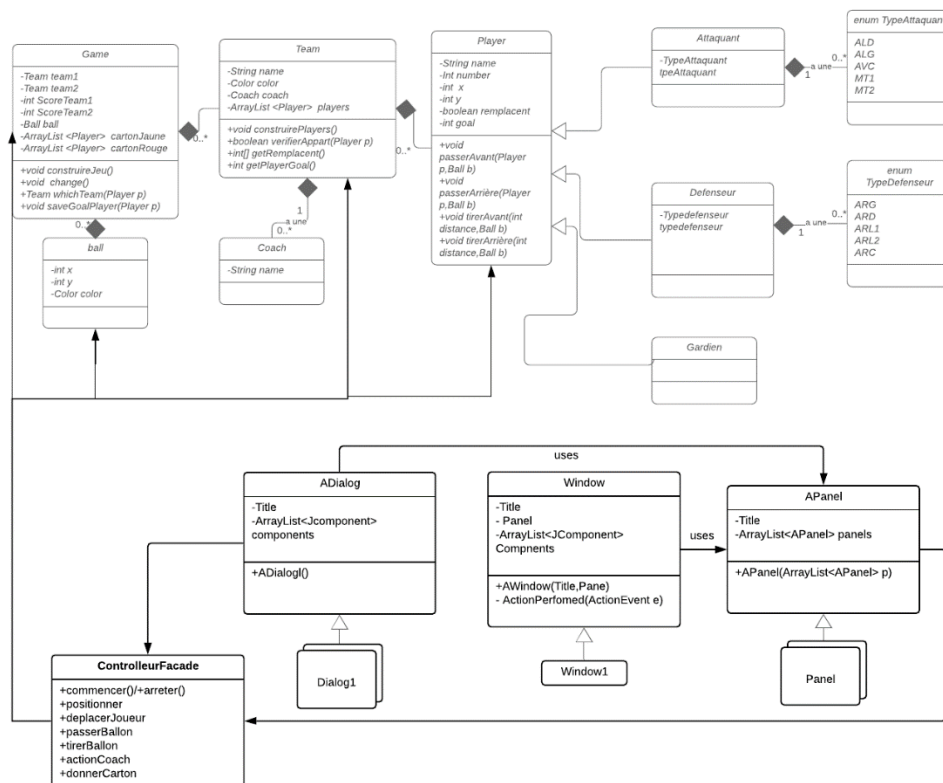
Les comportements appelés à évoluer sont transformés en abstraction, afin de pouvoir étendre le système sans toucher à son code (le principe OCP et variation protégée ont été respecté) (problème 11)

Exemple :

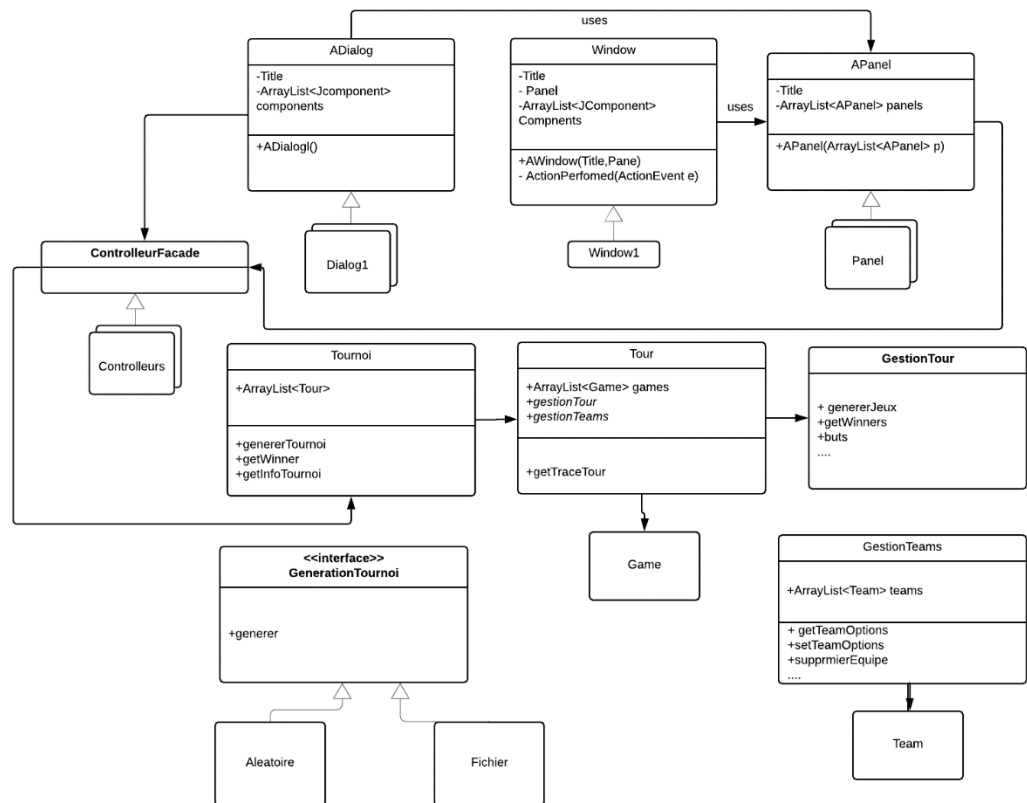
La méthode `Play` de la classe `fenetrePrincipale` utilise une interface `IStream` au lieu de se restreindre aux données provenant d'un fichier. Le code est donc ouvert à l'extension et fermé aux modifications.



## 2. Nouveau diagramme des classes :



## Package Tournoi :



## IV. Conclusion :

Au-delà de l'analyse du système existant et reconstitution du diagramme de classe, un travail de réingénierie a été effectué, afin d'identifier les problèmes de conception, ce qui a abouti à un ensemble de solutions, proposées en s'appuyant sur les principes de base de la POO ainsi que les patrons GRASP.

En revanche l'étape qui nous a été la plus lourde c'était celle de l'analyse et compréhension du système existant, et cela revient au fait que le développeur de l'ancien système n'a pas bien documenté son code et n'a pas respecté les Concepts de base de la POO, son code était donc difficile à comprendre, avec des classes généralement fortement liées entre elles et ayant trop de responsabilités.

Le respect des bonnes pratiques, nous a donc permis d'améliorer le système existant en un système de qualité ayant une implémentation facile à comprendre, réutilisable, évolutive et facile à intégrer dans un autre système.