TP C++

Bruno Bouzy

bouzy@math-info.univ-paris5.fr

24 octobre 2003

Ce document est un sujet de TP de C++ qui peut être associé ou bien à un cours de C++ ou à cours de Génie Logiciel. Il a été mis au point suite aux enseignements de C++ et Génie Logiciel donné en MIAIF2, MST2, IASV à l'Ufr de Mathématiques et Informatique de l'université Paris 5 entre 1997 et aujourd'hui.

Dans le cadre d'un module de Génie Logiciel l'objectif visé est :

- l'écriture d'un programme simple pour un utilisateur,
- la programmation d'un modèle UML très simple,
- la mise en œuvre d'associations entre instances.

et l'objectif visé n'est pas :

- l'utilisation d'outils de gestion de base de données.

Dans le cadre d'un cours de C++, l'objectif visé est la maîtrise de concepts C++ suivants :

- contenu d'une classe, constructeur, destructeur,
- pointeurs et références, passage de paramètre par référence,
- propriété de classe (mot-clé `static`) ou propriété d'instance,
- la classe `string` et la classe `vector` de la STL (Standard Template Library),
- chargement et sauvegarde sur fichiers.

et les points suivants ne sont pas abordés :

- visibilité des propriétés, encapsulation des données,
- généralisation, héritage, redéfinition,
- utilisation d'un environnement de programmation C++.

Prérequis :

- connaissance du C, de UML éventuellement,
- un compilateur C++, ici `g++`.

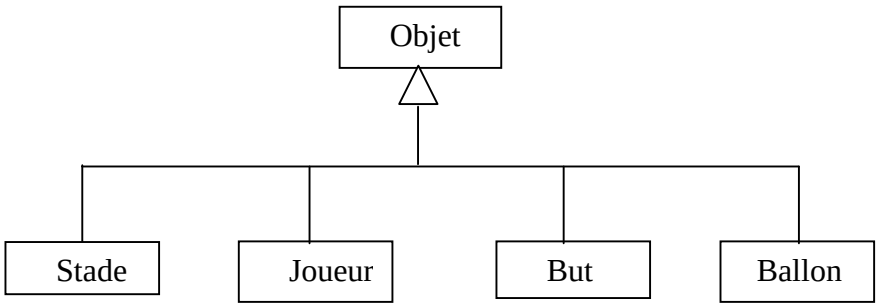
Ce document contient :

- un bref modèle UML du programme final `Stade`,
- un TP « premiers pas »,
- un TP « créer détruire des instances avec interaction utilisateur »,
- un TP « associer dissocier des instances avec interaction utilisateur »,
- un TP « charger sauver des instances sur fichier ».

SPECIFICATION UML

Cette page contient les diagrammes UML du logiciel STADE.

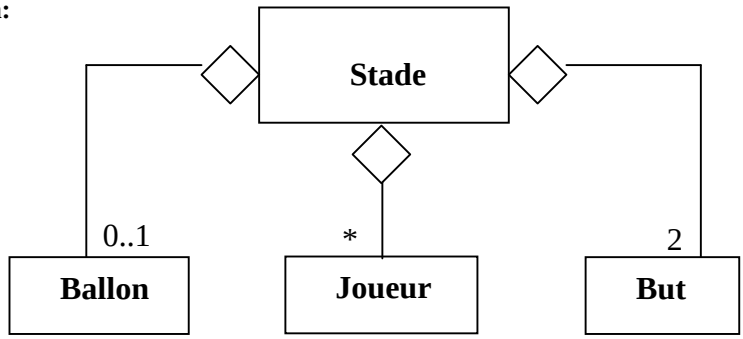
Diagramme de généralisation :



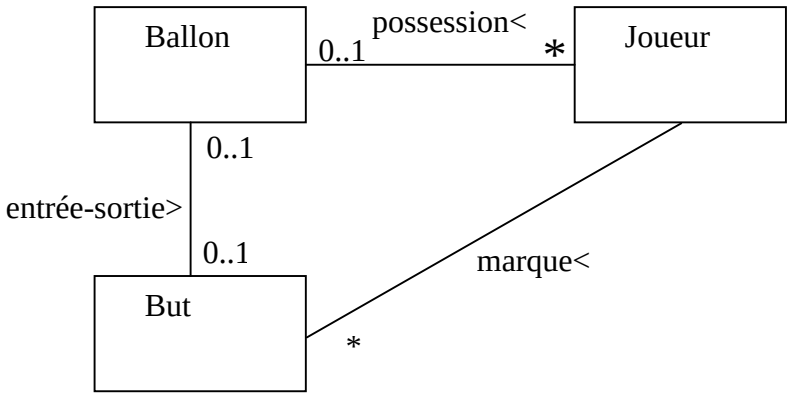
(nb : ce diagramme est donné à titre de complétude car la généralisation, ici en râteau, est simplissime)

Diagrammes classe-association

Agrégation:



Associations :



(nb : un joueur peut posséder le ballon qui peut être possédé par plusieurs joueurs (une mêlée ?), un ballon peut être dans un but qui contient éventuellement un ballon, un joueur peut marquer entre 0 et plusieurs buts, un but est marqué par un joueur exactement).

Créer une mini-application avec une seule classe

Le but de ce paragraphe est d'écrire un programme qui saisit des informations au clavier et les affiche à l'écran.

Créer une classe

Ecrire le contenu suivant dans le fichier `stade.cpp` :

```
// stade.cpp
#include <iostream>
void main ()
{
    cout << "Projet stade" << endl ;
}
```

NB : selon la version du compilateur, il peut être nécessaire de rajouter la ligne suivante en début de fichier:

```
using namespace std
```

Si cela est le cas, alors le faire pour chaque fichier *.cpp.

Saisir des informations sur l'entrée standard

Pour saisir des informations au clavier, C++ dispose de l'opérateur >>. Pour manipuler des chaînes de caractères, C++ dispose de la classe `string`. Modifier `stade.cpp` comme suit :

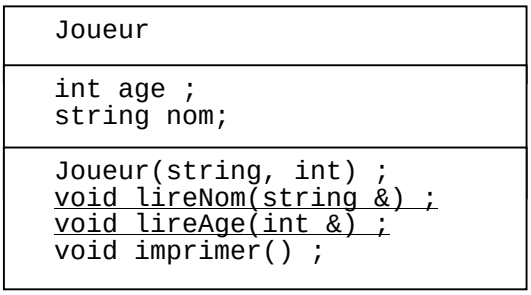
```
#include <iostream>
#include <string>
void main ()
{
    cout << "Projet stade" << endl;
    string s;
    int i ;
    cout << " Taper une chaine:" << endl;
    cin >> s ;
    cout << " Taper un entier:" << endl;
    cin >> i ;
    cout << "La chaine lue est: <" << s << ">" << endl;
    cout << "Le double de l'entier lu est:" << 2*i << endl;
}
```

Maintenant vous savez faire un programme qui lit une chaîne de caractères ou un entier sur l'entrée standard et qui affiche une chaîne de caractères sur la sortie standard.

Créer une mini-application avec 2 classes

Le but de ce paragraphe est d'écrire un programme qui comprend 2 classes : une classe utilisatrice, la classe `Stade`, et une classe utilisée, la classe `Joueur`. La classe utilisatrice lit des informations sur les propriétés d'une instance à créer ; elle crée l'instance avec ses propriétés ; elle imprime l'instance sur la sortie standard.

On suppose que la classe `Joueur` est définie comme suit :



Créer un fichier `joueur.cpp` et un fichier `joueur.h` qui correspondent à la spécification UML ci-dessus. Noter que les méthodes `lireNom` et `lireAge` sont des méthodes *de classe* (car elles sont soulignées). Elles devront être déclarées avec le mot-clé `static`. De plus, remarquer que le paramètre de type `string` ou `int` est un paramètre référence (symbole `&`) ; cela signifie que le paramètre est *en sortie* de ces méthodes. En effet, ces deux méthodes ont pour contrat de récupérer un nom et un âge tapés au clavier par l'utilisateur et de le donner en sortie. Ne pas confondre ces deux méthodes avec les méthodes de lecture d'un attribut privé : ici tous les attributs et toutes les méthodes sont *publiques*. Tout au long de ces TPs, nous n'utiliserons pas de méthode d'accès aux attributs.

Afin de tester la classe `Joueur`, modifier le programme principal comme suit :

```
#include <iostream>
#include <string>
#include "joueur.h"

void main ()
{
    cout << "Projet stade" << endl;
    string s;
    Joueur::lireNom(s);
    int a;
    Joueur::lireAge(a);
    cout << "s = <" << s << ">" << endl;
    cout << "a = <" << a << ">" << endl;
    Joueur * j = new Joueur(s, a);
    cout << "Joueur cree a l'adresse " << j << endl;
    j->imprimer();
}
```

Remarquer que le programme demande d'abord le nom et l'age du joueur à l'utilisateur, puis connaissant les valeurs du nom et de l'âge, il crée une instance en appelant le constructeur de la classe `Joueur` avec ces valeurs. Remarquer que lors de l'appel de `lireNom` et `lireAge`, aucune instance de joueur n'existe. Elles sont des méthodes de classe et pas des méthodes d'instance.

Maintenant vous savez donc faire un programme qui lit des propriétés d'une instance d'une classe, qui crée l'instance et qui l'affiche. Afin d'avoir une classe utilisatrice et une classe utilisée comme spécifié, déclarer une classe `Stade` avec une unique méthode de classe dans un fichier `stade.h`. Dans `stade.cpp`, le `main()` appelle cette méthode statique de la classe `Stade`.

TP C++ : CREER DETRUIRE UNE INSTANCE AVEC INTERACTION UTILISATEUR

Introduction

Ce TP reprend la suite du TP dans lequel le programme principal permettait de créer un joueur en demandant à l'utilisateur de taper le nom et l'age du joueur au clavier. Ici, nous remplaçons un joueur par *des ballons*. Nous allons créer une classe `Ballon` avec laquelle l'utilisateur pourra non seulement *créer* des ballons identifiés par une `string`, mais aussi les *afficher* et les *détruire*. Il va donc être nécessaire d'avoir un menu demandant l'action a effectuer (créer, détruire ou afficher) et un *conteneur* contenant les instances créées par l'utilisateur.

Autant l'écriture du menu va être très simple, autant l'utilisation du conteneur d'instances va être un peut « technique » car nous allons utiliser la librairie de patrons, ou 'templates', standards du C++ , la STL, et la notion d'attribut de classe (mot-clé `static`). L'avantage d'utiliser la librairie STL est de ne pas ré-inventer la roue ; mais l'inconvénient est d'utiliser des types paramétrés dont le mode d'emploi n'est pas trivial.

Afficher un menu, saisir le choix de l'utilisateur

Ecrire un programme qui affiche le menu ci-dessous, prend la commande de l'utilisateur au clavier, affiche un message « choix n » (n=1,2,3,0) et recommence tant que l'utilisateur ne veut pas quitter le programme.

1. **créer un ballon**
2. **détruire un ballon**
3. **afficher la liste des ballons**
0. **quitter**

Créer une instance par l'intermédiaire du clavier et de l'écran

Enrichir le programme précédent en remplaçant l'affichage « choix 1 » par l'appel à la fonction membre `creer_une_instance()` de la classe `Ballon`. Pour cela, on pourra avoir une interface de la classe `Ballon` du type suivant :

```
class Ballon {
public:
    string identificateur;             // pour identifier un ballon
    int taille;                        // taille du ballon
    Ballon(string, int);               // constructeur
    static void lireIdentificateur(string &);// dialogue
    static void lireTaille(int &);           // dialogue
    void imprimer();                         // imprimer
    static Ballon * creer_une_instance();    // dialogue puis new
};
```

`identificateur` est un attribut permettant d'identifier de manière unique une instance de la classe `Ballon`.

`taille` est un attribut descriptif d'une instance de la classe `Ballon`.

`lireIdentificateur(string &)` est une fonction qui demande à l'utilisateur de rentrer un identificateur au clavier, qui le récupère avec `cin` et qui le met dans le paramètre de type `string &`.

`lireAge(int &)` est une fonction qui demande à l'utilisateur de rentrer un age au clavier, qui le récupère avec `cin` et qui le met dans le paramètre de type `int &`. Ces deux dernières fonctions sont donc très simples. Pour cette raison, on peut choisir de ne pas les mettre sous forme de fonctions. Elles doivent être exécutées avant d'appeler le constructeur.

`Ballon(string, int)` est le constructeur d'une instance de la classe `Ballon`. On suppose que l'on connaît l'identificateur et la taille du ballon avant de l'appeler. On passe en paramètres l'identificateur et la taille du ballon à créer.

`imprimer()` est la fonction d'affichage à l'écran d'une instance. Par exemple, pour une instance dont l'identificateur vaut « bienGonflé » et dont la taille vaut 4, `imprimer()` peut afficher par exemple :

identificateur = bienGonflé
taille = 4

`Ballon * creer_une_instance()` est la fonction qui exécute les appels à `lireIdentificateur(string &)`, `lireAge(int &)`, `Ballon(string, int)` et `imprimer()`. Elle est appelée lorsque l'utilisateur tape 1 dans le menu Ballon. Elle fait exactement ce que faisait le programme principal du TP précédent.

Afficher les instances à l'écran

Enrichir le programme précédent en remplaçant l'affichage « choix 3 » par l'affichage de l'ensemble des instances de la classe `Ballon`. Pour cela, on enrichit la classe `Ballon` avec le vecteur des instances de la classe `instances` et une méthode d'impression de toutes les instances de la classe `imprimer_instances()`:

```
class Ballon {
public:
    static vector<Ballon*> * instances; // les instances
    static void imprimer_instances();  // imprimer les instances
};
```

Le type `vector<Ballon*>` est un type paramétré. Il déclare un tableau unidimensionnel contenant des `Ballon*`. Sa taille s'adapte dynamiquement au nombre d'éléments du vecteur. La classe `vector<Ballon*>` est définie dans `vector.h` de la STL de C++.

`instances` est donc un pointeur vers un vecteur de `Ballon*`. `instances` sert à stocker les instances de la classe `Ballon`. Le mot-clé `static` est nécessaire pour préciser que cet attribut n'est pas un attribut d'instance mais un attribut de classe.

Pour utiliser correctement `instances`, il est nécessaire de veiller à plusieurs choses :

- car il est déclaré avec `static` dans `ballon.h`, définir `instances` dans `ballon.cpp` :
```
vector<Ballon*> * Ballon::instances;
```

- car son type est un pointeur, initialiser `instances` au début du main() :
```
Ballon::instances = new vector<Ballon*>();
```

- lors de la construction de l'instance, l'insérer dans le conteneur `instances` :
```
instances->insert(instances->begin(), this);
```
(nb : l'insertion est faite au début par choix)

- lors de sa destrcution, enlever l'instance de `instances` :

```
for (vector<Ballon*>::iterator b=instances->begin();
     b !=instances->end() ; b++)
      if (this==*b) { instances->erase(b) ; break ; }
```

(nb : éventuellement mettre cette boucle 'compliquée' dans une méthode ?)

- inclure la définition de la classe `vector` partout où elle est utilisée :

```
#include <vector>
```

Les méthodes :
```
vector<Ballon*>::iterator begin(),
vector<Ballon*>::iterator end(),
insert(vector<Ballon*>::iterator, Ballon*),
erase(vector<Ballon*>::iterator)
```
sont définies implicitement par la classe template `vector` de la STL.

`begin` retourne le premier *itérateur* du vecteur, `end` retourne le dernier.

Un itérateur est une variable permettant de parcourir le vecteur ou de désigner une position dans le vecteur. Ici, son type est `vector<Ballon*>::iterator`. (Selon le compilateur, ce type peut être `Ballon**` (un type pointeur sur le type contenu du vecteur) mais cela n'est pas systématique). Remarquer que la variable de la boucle `for` ci-dessus parcourant le vecteur est de type `vector<Ballon*>::iterator`.

`insert` permet d'insérer un élément à une position (au début dans l'exemple ci-dessus) et `erase` permet d'enlever un élément du vecteur connaissant sa position.

Détruire une instance par l'intermédiaire du clavier et de l'écran

Enrichir le programme précédent en remplaçant l'affichage « choix 2 » par l'appel à la fonction membre `detruire_une_instance()` de la classe `Ballon`. Pour cela, on pourra ajouter les membres suivants à la classe `Ballon` :

```
class Ballon {public:
    ~Ballon();                                  // destructeur
    static Ballon * getInstance(string);        // retourne le ballon
    void toString(string &);                    // l'instance en string
    static bool detruire_une_instance();        // dialogue puis detruire
};
```

`~Ballon()` est le destructeur d'une instance de la classe `Ballon`.

`toString(string &)` est la fonction qui met le descriptif d'un objet (identificateur + taille) dans une chaine de caractères de sorte que le contenu de `imprimer()` se réduise à :

```
    string s ; toString(s) ; cout << s ;
```

Normalement, la classe `string` permet d'effectuer les opérations suivantes de manière agréable :

```
    string s1 = "A l'impossible, ";
    string s2 = s1;
    string s3 = s2 + "nul n'est tenu." ;
```

`Ballon * getInstance(string)` est la fonction qui retourne l'instance de la classe `Ballon` possédant un identificateur égal à la `string` passée en paramètre. Elle est appelée à chaque fois que l'on cherche une instance connaissant son identificateur. Elle retourne `NULL` si aucune instance ne correspond à la `string`. Elle retourne l'instance adéquate sinon. Pour écrire son contenu, on utilisera une boucle du style :

```
for (vector<Ballon*>::iterator b=instances->begin();
     b !=instances->end() ; b++)
          if ((*b)->identificateur==s) return *b ;
    return NULL;
```

`bool detruire_une_instance()` est la fonction qui exécute les appels à `lireIdentificateur(string&)`, `getInstance(string)`, `~Ballon()` et retourne `true` si la destruction est faite et `false` sinon. Elle est appelée lorsque l'utilisateur tape 2 dans le menu Ballon.

TP C++ ASSOCIER DISSOCIER AVEC INTERACTION UTILISATEUR

Introduction

Ce TP reprend la suite du TP dans lequel l'utilisateur pouvait créer, détruire, afficher les instances d'une classe (la classe `Ballon`). Ici, nous allons permettre en plus à l'utilisateur d'*associer* et *dissocier* des instances appartenant à des classes différentes.

Pour cela, il nous faut plusieurs classes et un menu général permettant de diriger l'utilisateur vers la classe de son choix. Ce TP suppose qu'une classe `But` et que la classe `Joueur` ont été construites sur le principe de la classe `Ballon`. Pour commencer, il y a donc une partie importante d'édition et de couper-coller à faire dans ce TP. Ce TP suppose aussi que la classe `vector` de la STL est connue (cf TP précédent).

Ensuite seulement, la partie intéressante du TP (associer, dissocier des instances) pourra être abordée.

Plusieurs classes...

Ecrire un programme qui gère plusieurs classes, par exemple la classe `Joueur`, la classe `Ballon` et la classe `But`. En particulier, il affiche le menu ci-dessous :

1. **Joueur**
2. **Ballon**
3. **But**
0. **Quitter**

Puis, par exemple, le menu

1. **créer un ballon**
2. **détruire un ballon**
3. **afficher la liste des ballons**
0. **précédent**

si l'utilisateur a tapé 2 dans le premier menu.

Associer deux instances une à une

Enrichir le programme précédent en rajoutant la ligne suivante dans les menus `Ballon` et `But` :

4. **associer un ballon à un but.**

En particulier, on ajoutera un attribut et deux méthodes à la classe `Ballon` :

```
But * monBut;
static void associer_but_instance() {} // dialogue puis associer
void associer_but(But * b) {}      // associer ballon a but
```

Analogue pour la classe `But` :

```
Ballon * monBallon;
static void associer_ballon_instance() {} // dialogue puis associer
void associer_ballon(Ballon * b) {}      // associer but a ballon
```

Associer une instance avec plusieurs instances

Enrichir le programme précédent en rajoutant la ligne suivante dans le menu `Ballon` :

5. associer un ballon à des joueurs.

Et la ligne suivante dans le menu `Joueur`:

5 associer un joueur à un ballon.

En particulier, on ajoutera un attribut `mesJoueurs` à la classe `Ballon` :

```
vector<Joueur*> * mesJoueurs ;
```

Et un attribut à la classe `Joueur` :

```
Ballon * monBallon ;
```

Remarquer qu'il est nécessaire d'utiliser un attribut de type `vector<Joueur*>` dans la classe `Ballon` pour associer des joueurs et un ballon car la spécification UML du projet STADE (en premier page) indique que l'ordre de multiplicité de l'association est 0 ou plusieurs. Par contre, cela n'est pas nécessaire dans l'autre sens.

Dissocier des instances

Enrichir le programme précédent en rajoutant la ligne suivante dans le menu `Ballon` :

6. dissocier un ballon d'un joueur.

Et la ligne suivante dans le menu `Joueur`:

6 dissocier un joueur d'un ballon.

Compléter votre programme pour que tout lien puisse être dissocié.

Imprimer des instances avec leurs liens

Pour visualiser les instances du programme, il est nécessaire d'avoir deux méthodes qui transforment une instance en `string`. Une première méthode qui transforme une instance en une `string` identifiant l'instance (nous l'appellerons `toIdent()`) et une seconde méthode qui transforme l'instance en une `string` contenant tous les noms des attributs de l'instance avec leur valeurs.

Ecrire la méthode `toString()` et la méthode `toIdent()` pour toutes les classes de votre application. Par exemple, pour la classe `Ballon`, le corps de ces méthodes est :

```
void Ballon::toString(string & s) {
  char ss[16]; sprintf (ss, "%d", taille);
  s = "Identificateur " + identificateur + "\nTaille " + ss + "\nBut ";
  if (monBut) s = s + monBut->identificateur;
  else s = s + "null ";
  s = s + "\nJoueurs: ";
  string sss = "";
  for (Joueur ** j = mesJoueurs->begin(); j != mesJoueurs->end(); j++) {
    ((Joueur*)*j)->toIdent(sss);
  }
  s = s + sss + "\n";
}

void Ballon::toIdent(string & s) {
  s = s + identificateur + " ";
}
```

Rendre le programme correct, robuste et plus facile à utiliser

A ce niveau du TP, vous avez une première version du programme qui permet de créer, détruire, associer, dissocier et afficher les instances, sans plus.

Vous pouvez rendre robuste votre programme en vérifiant les tests suivants :

- créer et associer des instances, détruire l'une d'elle. afficher les instances non détruites. Que se passe-t-il ? Corriger le programme.

- associer une instance avec une instance déjà associée : le programme doit refuser l'association si elle est un-à-un.

- l'utilisateur veut détruire, associer ou dissocier une instance, le programme propose-t-il une liste des instances possibles ? Si la réponse est négative, améliorer le programme pour qu'il propose des listes d'instances à l'utilisateur lorsque cela est nécessaire.

Introduction

Ce TP reprend la suite du TP dans lequel l'utilisateur pouvait créer, détruire, associer, dissocier des instances de différentes classes. A chaque session, l'utilisateur devait repartir de zéro. S'il avait créé des instances à la session précédente, il ne les retrouvait pas à la session suivante. Le but de ce TP est donc de *sauver* et *charger* sur *fichier* les informations contenues dans une classe. Ainsi, au début du nouvelle session, l'utilisateur pourra repartir des données de la fin de la session précédente.

Ecrire dans un fichier les instances d'une classe

Le but de ce paragraphe est d'écrire un programme qui écrit dans un fichier les instances d'une classe situées dans la liste d'instances de la classe.

Pour ouvrir un fichier `Joueur.txt` en écriture, on peut utiliser le constructeur de la classe `ofstream` puis utiliser l'instance construite pour sauver la liste des joueurs dans le fichier, puis refermer le fichier avec `close()`.

```
cout << "sauvegarde sur fichier: debut" << endl;
ofstream * ofile;

ofile = new ofstream("Joueur.txt", ios::out);
if (!(*ofile)) cout << "Ouverture Joueur.txt impossible." << endl;
else {
      Joueur::sauver_instances(ofile);
      ofile->close();
      cout << "sauver Joueur.txt fin normale." << endl;
}
```

Ajouter un `#include <fstream>` ; c'est nécessaire pour utiliser la classe `ofstream` en C++.

Ecrire la fonction `sauver()` de `Joueur`.

Mettre au point le programme principal. Tester avec quelques exemples. Vous devez obtenir un fichier `Joueur.txt` avec un contenu ressemblant à :

```
Nom HARPO
Age 45
Nom GROUCHO
Age 56
Nom TYPO
Age 99
```

Lire un fichier contenant les instances d'une classe

Le but de ce paragraphe est de mettre au point un programme qui :

1) lit des informations dans un fichier à propos d'une classe et crée les instances de cette classe en mémoire.

2) lit des informations tapées au clavier par l'utilisateur et crée d'autres instances de la classe en mémoire.

3) écrit la liste des instances de la classe dans le fichier.

2 et 3 ont déjà été faits avant, il suffit de faire 1. On va donc rajouter le chargement du fichier `Joueur.txt` au début du programme :

```
void Stade::main ()
{
      cout << "Projet stade" << endl;
      // chargement des objets
      cout << "chargement des fichiers: debut" << endl;
      ifstream * ifile;
      ifile = new ifstream("Joueur.txt", ios::in);
      if (!(*ifile)) cout << "Ouv. Joueur.txt impossible." << endl;
      else {
            Joueur::charger_instances(ifile);
            ifile->close();
            cout << "charger Joueur.txt fin normale." << endl;
      }
      cout << "Joueur.instances: " << endl;
      Joueur::imprimer_instances();
      // l'ancien programme main avec 2 et 3
}
```

Ecrire la méthode `Joueur::charger_instances()`.

Mettre au point ce programme. Tester avec le fichier `Joueur.txt` donné ci-avant et enrichir ce fichier à chaque exécution. Attention ! lorsque vous mettez au point ce programme, une fausse manipulation peut vous écraser le fichier `Joueur.txt`, donc sauvez `Joueur.txt` avant chaque nouvelle exécution…

Voilà, vous savez désormais utiliser des fonctions qui permette de sauver des instances dans un fichier et de les charger depuis un fichier. Il ne reste plus qu'a faire la même chose pour les classes `But` et `Ballon`.

Ecrire/Lire des fichiers contenant des instances associées de classes différentes

Lorsque les instances sont associées à d'autres le mécanisme précédent doit être amélioré avec deux passes. En effet lors de la lecture du fichier des instances, les instances associées à une instance ne sont pas nécessairement déjà construites. Il faut donc une première passe de lecture des fichiers en construisant les instances sans les associations, puis une deuxième passe sans la construction mais avec les associations.