Laboratorio 5

El objetivo del laboratorio es:

Implementar una pequeña versión de la aplicación de fútbol en C++.

Entrega:

- Código del proyecto (Programas en C++)
- Documentación de la implementación.

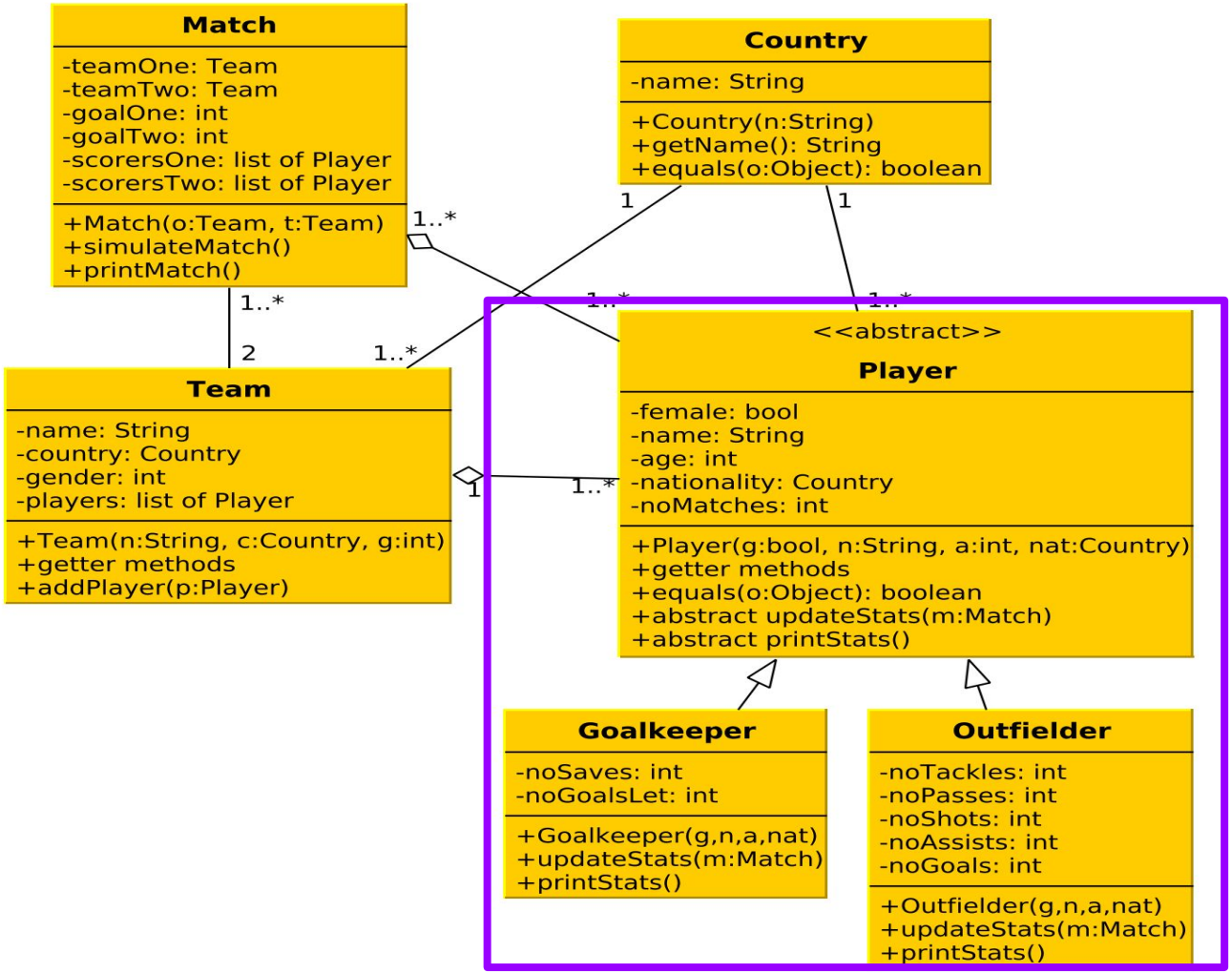
Documentación

1. Una **introducción** donde se describe el problema.
   a. ¿Qué hace el programa?
   b. ¿Qué clases se definieron?
   c. ¿Cuáles métodos se implementaron?
2. Una **descripción** de **soluciones alternativas** que se discutieron, y una descripción de la **solución elegida**, así como el motivo de la elección de esta solución en lugar de otras. También es una buena idea mencionar los **aspectos teóricos** de los conceptos de programación orientada a objetos que se aplicaron como parte de la solución.
3. Una **conclusión** que describa el **funcionamiento** de la práctica, es decir, ¿las pruebas mostraron que las clases se implementaron correctamente? Se puede mencionar cualquier **dificultad** durante la implementación, así como la solución.

El código fuente y la documentación deben cargarse en un directorio de nombre **Lab5** de su repositorio Git ***antes del 11 de diciembre***. ***Importante, el nombre del directorio deberá ser exactamente Lab5.***

Universitat Pompeu Fabra Barcelona

Diagrama propuesto



Universitat Pompeu Fabra Barcelona

Clase Team

```java
import java.util.LinkedList;

public class Team {
    public enum Gender { MALE, FEMALE, MIXED }

    protected String name;
    protected Country country;
    protected Gender gender;
    protected LinkedList<Player> players;

    public Team(String n, Country c, Gender g) {
        name = n;
        country = c;
        gender = g;
        players = new LinkedList<Player>();
    }

    public String getName() {
        return name;
    }

    public Country getCountry() {
        return country;
    }

    public Gender getGender() {
        return gender;
    }

    public LinkedList<Player> getPlayers() {
        return players;
    }

    public void addPlayer(Player p) {
        if ((gender == Gender.MALE && !p.isFemale()) ||
            (gender == Gender.FEMALE && p.isFemale()) ||
            gender == Gender.MIXED)
            players.add(p);
    }
```

```cpp
#ifndef _TEAM_
#define _TEAM_

#include "Country.hpp"
#include "Player.hpp"

class Team {

public:

    enum Gender { MALE, FEMALE, MIXED };

private:

    std::string name;
    Country * country;
    Gender gender;
    std::list<Player *> players;

public:

    Team(std::string n, Country * c, Gender g) {
        name = n;
        country = c;
        gender = g;
    }

    std::string getName() {
        return name;
    }

    Country * getCountry() {
        return country;
    }

    Gender getGender() {
        return gender;
    }

    std::list<Player *> & getPlayers() {
        return players;
    }

    void addPlayer(Player * p) {
        if ((gender == Gender::MALE && !p->isFemale()) ||
            (gender == Gender::FEMALE && p->isFemale()) ||
            gender == Gender::MIXED)
            players.push_back(p);
    }
};

#endif
```

Universitat Pompeu Fabra Barcelona

Clase Country

```java
public class Country {
    private String name;

    public Country(String n) {
        name = n;
    }

    public String getName() {
        return name;
    }
```

```cpp
#ifndef _COUNTRY_
#define _COUNTRY_

#include <cstdlib>
#include <iostream>
#include <list>

class Country {

private:

    std::string name;

public:

    Country(std::string n) {
        name = n;
    }

    std::string getName() {
        return name;
    }

};

#endif
```

Universitat Pompeu Fabra Barcelona

Clase Match

```java
import java.util.LinkedList;
import java.util.Random;

public class Match {
    protected Random random;

    protected Team one;
    protected Team two;
    protected int goalsOne;
    protected int goalsTwo;
    protected LinkedList<Player> scorersOne;
    protected LinkedList<Player> scorersTwo;

    public Match(Team o, Team t) {
        random = new Random();
        one = o;
        two = t;
        scorersOne = new LinkedList<Player>();
        scorersTwo = new LinkedList<Player>();
    }

    public Team getTeamOne() {
        return one;
    }

    public Team getTeamTwo() {
        return two;
    }

    public int getGoalsOne() {
        return goalsOne;
    }

    public int getGoalsTwo() {
        return goalsTwo;
    }

    public LinkedList<Player> getScorersOne() {
        return scorersOne;
    }

    public LinkedList<Player> getScorersTwo() {
        return scorersTwo;
    }

    protected void simulateScorers(Team t, LinkedList<Player> l, int g) {
        for (int i = 0; i < g; ++i) {
            int ix = random.nextInt(t.getPlayers().size());
            Player p = t.getPlayers().get(ix);
            l.add(p);
        }
    }

    public void simulateMatch() {
        goalsOne = random.nextInt(bound:6);
        goalsTwo = random.nextInt(bound:6);
        simulateScorers(one, scorersOne, goalsOne);
        simulateScorers(two, scorersTwo, goalsTwo);
    }

    public void printMatch() {
        System.out.print(one.getName() + "-" + two.getName() + ": ");
        System.out.println(goalsOne + "-" + goalsTwo);
    }
}
```

```cpp
#ifndef _MATCH_
#define _MATCH_

#include "Player.hpp"
#include "Team.hpp"

class Match {

private:

    Team * teamOne;
    Team * teamTwo;
    int goalOne;
    int goalTwo;
    std::list<Player *> scorersOne;
    std::list<Player *> scorersTwo;

    void simulateScorers(Team * t, std::list<Player *> & l, int g) {
        for (int i = 0; i < g; ++i) {
            int ix = rand() % t->getPlayers().size();
            std::list<Player *>::iterator it = t->getPlayers().begin();
            std::advance(it, ix);
            l.push_back(*it);
        }
    }

public:

    Match(Team * o, Team * t) {
        teamOne = o;
        teamTwo = t;
    }

    Team * getTeamOne() {
        return teamOne;
    }

    Team * getTeamTwo() {
        return teamTwo;
    }

    int getGoalOne() {
        return goalOne;
    }

    int getGoalTwo() {
        return goalTwo;
    }

    std::list<Player *> & getScorersOne() {
        return scorersOne;
    }

    std::list<Player *> & getScorersTwo() {
        return scorersTwo;
    }

    void simulateMatch() {
        goalOne = rand() % 6;
        goalTwo = rand() % 6;
        simulateScorers(teamOne, scorersOne, goalOne);
        simulateScorers(teamTwo, scorersTwo, goalTwo);
    }

    void printMatch() {
        std::cout << teamOne->getName() + "-" + teamTwo->getName() + ":
        std::cout << goalOne << "-" << goalTwo << "\n";
    }

};

#endif
```

Universitat Pompeu Fabra Barcelona

Clase Player

- Como se indica en las clases de Match y Team, la definición de la clase de Player debería aparecer en un archivo de encabezado Player:hpp. A diferencia de Java, no es obligatorio almacenar la definición de una clase X en un archivo de encabezado X:hpp, aunque sigue siendo bueno practica para hacerlo.
- A diferencia de Java, si dos o más clases incluyen el mismo archivo de encabezado en C++ (por ejemplo, Team y Match que incluyan Player:hpp), entonces, de forma predeterminada, todas las definiciones en el archivo de encabezado está duplicado, lo que normalmente provoca errores del compilador. Para evitar duplicación, los archivos de encabezado de C++ generalmente definen directivas de inclusión de la siguiente manera:

```
#ifndef _PLAYER_
#define _PLAYER_
// todo el código del archivo de encabezado va aquí
#endif
```

Directivas de inclusión

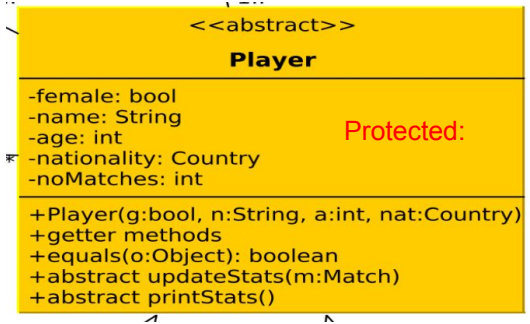
- Dado que la **clase Player** tiene un atributo de tipo Country, debe **incluir** el archivo de encabezado **Country:hpp**. La clase Player también contiene un método abstracto updateStats(m: Match). Esto normalmente indicaría que el jugador necesita **incluir también** el archivo de encabezado **Match:hpp**.
- Sin embargo, en dependencias cíclicas el compilador marca error. Por ejemplo, la clase Match incluye el archivo de encabezado Player:hpp y Player incluye Match:hpp.
- Para resolver el problema de las dependencias cíclicas, C++ tiene un concepto llamado **declaración directa**. Concretamente, en la clase Player, en lugar de incluir Match:hpp, se puede escribir una definición vacía de la clase Match (es decir, una declaración directa):

```
#include "Country.hpp"
class Match;
```

- Dado que la clase Player nunca intenta acceder a un atributo o método de Match, la declaración directa es suficiente para definir el método abstracto updateStats.

Atributos y métodos

- Dado que se va a heredar de Player, es buena idea dar los atributos de visibilidad protegida (en lugar de privada). Es importante ver que en C++, casi siempre se usan punteros cuando un atributo (u otra variable) se refiere a una instancia de otra clase. Por tanto, el atributo nacionalidad debería ser un puntero de tipo Country.
- Implemente todos los atributos y métodos de la clase Player. Tener en cuenta que el método isFemale necesita tener exactamente este nombre y devolver un valor booleano, ya que la clase Team llama a este método desde addPlayer.

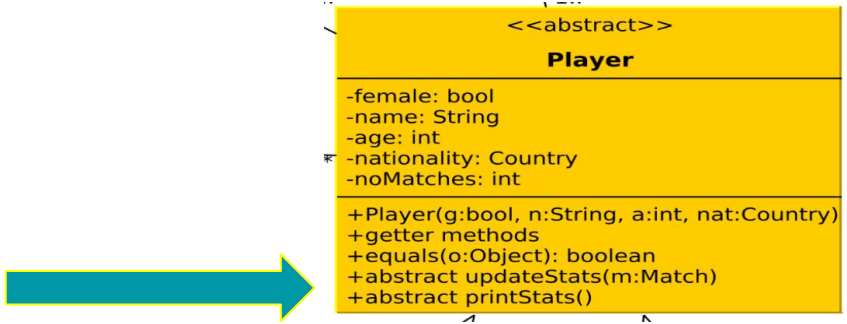


Clases y métodos abstractos

Dado que la clase Player tiene dos métodos abstractos updateStats y printStats, por defecto, la clase Player también es abstracta. Sin embargo, C++ no tiene palabra clave abstracta, por lo que no es necesario indicar en el encabezado de la clase que Player es abstracto (el compilador se si se intenta crear un instancia de Player).

Para crear un método abstracto en C++, necesitamos usar la palabra clave virtual y establecer el puntero del método igual a 0:

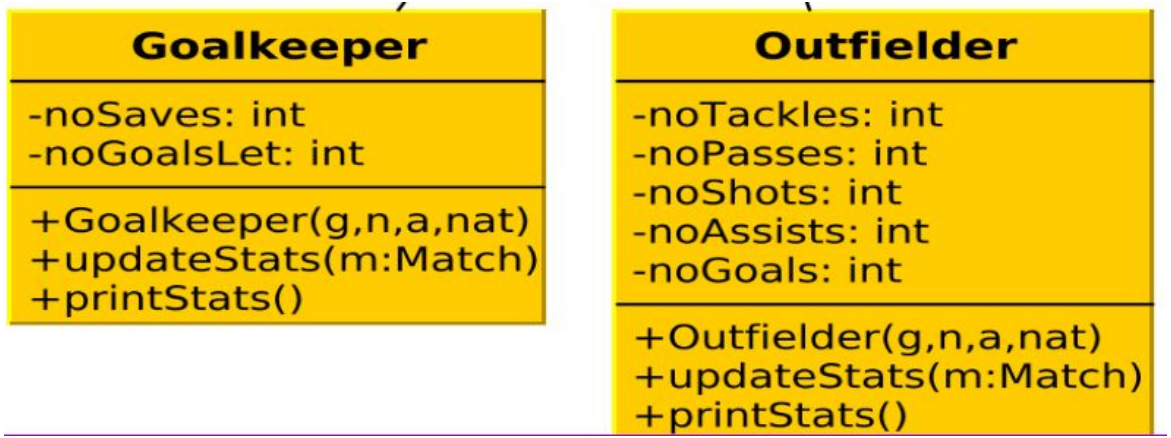
```
virtual void updateStats(Match * m) = 0;
virtual void printStats() = 0;
```



Clases Goalkeeper y Outfielder

- Toca implementar las dos clases Goalkeeper y Outfielder. Estas clases, deben incluir tanto **Player:hpp** como **Match:hpp**, ya que el método updateStats necesita llamar a métodos de Match. Tener en cuenta que esto no crea una dependencia cíclica, ya que Match no incluye ni al portero:hpp ni al jardinero:hpp, solo al Player:hpp.

#include



Clases Goalkeeper y Outfielder

- Para que el Goalkeeper herede del Player, se debe incluir el siguiente encabezado de clase:

```
class Goalkeeper : public Player {
```

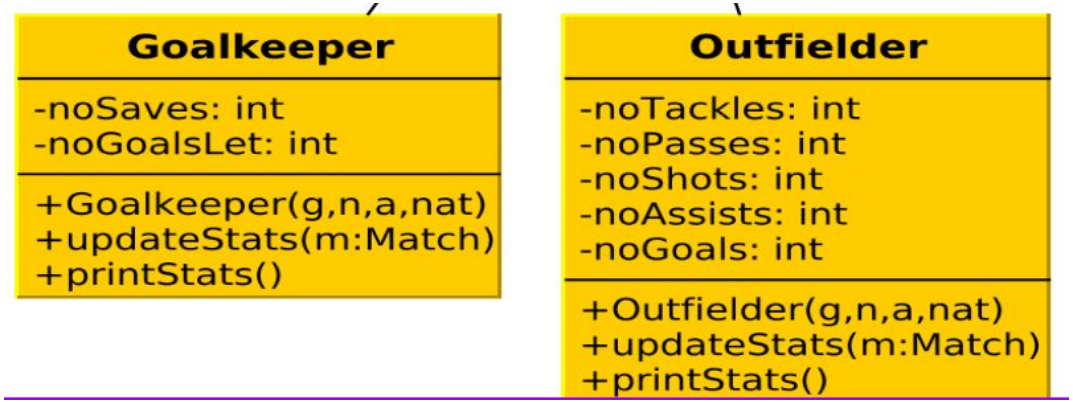
- En lugar de super, C++ usa el nombre de la superclase para llamar a su constructor en la misma línea que el propio constructor:

```
public Goalkeeper(boolean f, String n, int a, Country c) {
    super(f, n, a, c);
```

```
Goalkeeper(...) : Player(...) {
```

Clases Goalkeeper y Outfielder

- Como Goalkeeper y Outfielder no son clases abstractas, es necesario sobrescribir los dos métodos updateStats y printStats, deberán ser similares a los métodos implementados en la práctica de laboratorio anterior.



Programa principal

- Finalmente, se ha de crear una función principal main() para poder probar las clases.
- En C++, la función principal siempre va en un archivo de código con **extensión :cpp**.
- El archivo de código debe **incluir todos los archivos de encabezado**, aunque la inclusión es recursiva: si A:hpp incluye B:hpp, entonces cualquier archivo que incluya A:hpp también incluirá B:hpp. Por eso sólo es estrictamente necesario **incluir Goalkeeper:hpp y Outfielder:hpp** en el archivo de código, ya que esos encabezados incluyen recursivamente todos los demás encabezados.

Programa principal

- La función principal debería al menos hacer lo siguiente:
  - Crear algunos países.
  - Crea algunos outfielders de diferentes países.
  - Crea dos equipos que incluyan algunos outfielders cada uno.
  - Crear un partido entre los dos equipos, simular el partido e imprimir el resultado.
  - Actualizar las estadísticas de cada jugador como resultado de disputar el partido.
  - Imprimir las estadísticas de cada jugador.
- Si ya se tiene un código similar de laboratorios anteriores (por ejemplo, laboratorio 2 o 3), debería ser fácil adaptar el código para C++. Tenga en cuenta que la palabra **new** siempre **crea** un **puntero de la clase correspondiente**. **(enviar un puntero &)**

Outfielder p1(false, "Messi", 36, &c1);