

[Refactoring]



- Refactoring es una **transformación** que preserva el comportamiento, pero mejora el diseño

Bill Opdyke, PhD Thesis "Refactoring Object-Oriented Frameworks". Univ. of Illinois at Urbana-Champaign (UIUC). 1992. Director: Ralph Johnson.

[Veamos un ejemplo....]

- Imprimir los puntajes de cada set de un jugador en cada partido de tenis de una fecha específica.

Puntajes para los partidos de la fecha 7/5/2023

Partido:

Puntaje del jugador: Federico Delbonis: 6; 5; 7; Puntos del partido: 36

Puntaje del jugador: Guido Pella: 4; 7; 6; Puntos del partido: 34

Partido:

.....

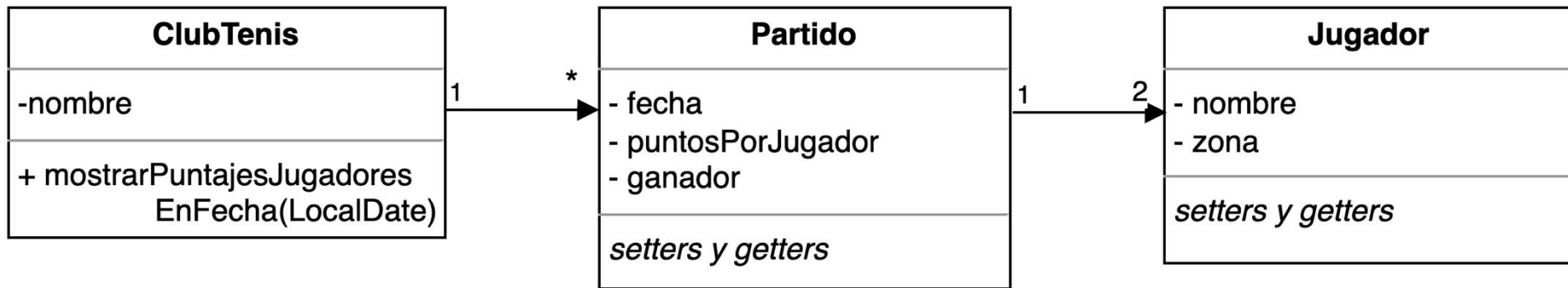
```

public class ClubTennis {
    private String nombre;
    private List<Partido> coleccionPartidos;
    public String mostrarPuntajesJugadoresEnFecha(LocalDate fecha) {
        int totalGames = 0;
        List<Partido> partidosFecha;
        String result = "Puntajes para los partidos de la fecha " + fecha.toString() + "\n";
        partidosFecha = coleccionPartidos.stream().filter(p -> p.fecha().equals(fecha)).collect(Collectors.toList());
        for (Partido p : partidosFecha) {
            totalGames = 0;
            Jugador j1 = p.jugador1();
            result += "Partido: " + "\n";
            result += "Puntaje del jugador: " + j1.nombre() + ": ";
            for (int gamesGanados : p.puntosPorSetDe(j1)) {
                result += Integer.toString(gamesGanados) + ";";
                totalGames += gamesGanados;
            }
            result += "Puntos del partido: ";
            if (j1.zona() == "A") result += Integer.toString(totalGames * 2);
            if (j1.zona() == "B") result += Integer.toString(totalGames);
            if (j1.zona() == "C")
                if (p.ganador() == j1)
                    result += Integer.toString(totalGames);
                else
                    result += Integer.toString(0);

            Jugador j2 = p.jugador2();
            totalGames = 0;
            result += "Puntaje del jugador: " + j2.nombre() + ": ";
            for (int gamesGanados : p.puntosPorSetDe(j2)) {
                result += Integer.toString(gamesGanados) + ";";
                totalGames += gamesGanados;
            }
            result += "Puntos del partido: ";
            if (j2.zona() == "A") result += Integer.toString(totalGames * 2);
            if (j2.zona() == "B") result += Integer.toString(totalGames);
            if (j2.zona() == "C")
                if (p.ganador() == j2)
                    result += Integer.toString(totalGames);
                else
                    result += Integer.toString(0);
        }
        return result;
    }
}

```

[Diagrama inicial]



[Cambios pedidos ...]

- Cambiará la manera de calcular los puntos
- Pueden cambiar las zonas
- Se necesita el resultado en HTML

[Ejemplo del club de tenis]

- ¿Por dónde empezamos?
- ¿Cuáles son los problemas que tiene el código, que atentan contra su extensibilidad y reuso?

```

public class ClubTennis {
    private String nombre;
    private List<Partido> coleccionPartidos;
    public String mostrarPuntajesJugadoresEnFecha(LocalDate fecha) {
        int totalGames = 0;
        List<Partido> partidosFecha;
        String result = "Puntajes para los partidos de la fecha " + fecha.toString() + "\n";
        partidosFecha = coleccionPartidos.stream().filter(p -> p.fecha().equals(fecha)).collect(Collectors.toList());
        for (Partido p : partidosFecha) {
            totalGames = 0;
            Jugador j1 = p.jugador1();
            result += "Partido: " + "\n";
            result += "Puntaje del jugador: " + j1.nombre() + ": ";
            for (int gamesGanados : p.puntosPorSetDe(j1)) {
                result += Integer.toString(gamesGanados) + ";";
                totalGames += gamesGanados;
            }
            result += "Puntos del partido: ";
            if (j1.zona() == "A") result += Integer.toString(totalGames * 2);
            if (j1.zona() == "B") result += Integer.toString(totalGames);
            if (j1.zona() == "C")
                if (p.ganador() == j1)
                    result += Integer.toString(totalGames);
                else
                    result += Integer.toString(0);

            Jugador j2 = p.jugador2();
            totalGames = 0;
            result += "Puntaje del jugador: " + j2.nombre() + ": ";
            for (int gamesGanados : p.puntosPorSetDe(j2)) {
                result += Integer.toString(gamesGanados) + ";";
                totalGames += gamesGanados;
            }
            result += "Puntos del partido: ";
            if (j2.zona() == "A") result += Integer.toString(totalGames * 2);
            if (j2.zona() == "B") result += Integer.toString(totalGames);
            if (j2.zona() == "C")
                if (p.ganador() == j2)
                    result += Integer.toString(totalGames);
                else
                    result += Integer.toString(0);
        }
        return result;
    }
}

```

[

]

MÉTODO LARGO!

[Refactoring Extract Method]

- Motivación :
 - Métodos largos
 - Métodos muy comentados
 - Incrementar reuso
 - Incrementar legibilidad

[Refactoring Extract Method]

■ Mecánica:

1. Crear un nuevo método cuyo nombre explique su propósito
2. Copiar el código a extraer al nuevo método
3. Revisar las variables locales del original
4. Si alguna se usa sólo en el código extraído, mover su declaración
5. Revisar si alguna variable local es modificada por el código extraído. Si es solo una, tratar como query y asignar. Si hay más de una no se puede extraer.
6. Pasar como parámetro las variables que el método nuevo lee.
7. Compilar
8. Reemplazar código en método original por llamada
9. Compilar



[Refactoring Extract method

```
public class ClubTenis {  
  
    public String mostrarPuntajesJugadoresEnFecha(LocalDate  
        fecha) {  
        List<Partido> partidosFecha;  
        String result = "Puntajes para los partidos de la fecha "  
            + fecha.toString() + "\n";  
        partidosFecha = coleccionPartidos.stream().filter(p ->  
            fecha.equals(p.fecha())).collect(Collectors.toList());  
        for(Partido p: partidosFecha)  
            result += this.mostrarPartido(p);  
        return result;  
    }  
  
    private String mostrarPartido(Partido p) {...}
```

eclipse-

***ClubTenis.java** X **Parti**

```
9
10 public String m
11     List<Partid
12     String resu
13     partidosFec
14     for(Partido
15         int tot
16         Jugador
17         result
18         result
19         for (in
20             res
21             tot
22             result
23             if (j1.
24             res
25             if (j1.
26             res
27             if (j1.
28                 if
29
30         els
31
32     }
33     Jugador
34     totalGa
35     result
36     for (in
37         res
38         tot
39         result
40         if (j2.
41         res
42         if (j2.
43         res
44         if (j2.
45             if
46
47         els
48
49 }
```

Save ⌘ S

Open Declaration F3

Open Type Hierarchy F4

Open Call Hierarchy ^⌘ H

Show in Breadcrumb ⌘ B

Quick Outline ⌘ O

Quick Type Hierarchy ⌘ T

Open With >

Show In ⌘ W >

Cut ⌘ X

Copy ⌘ C

Copy Qualified Name

Paste ⌘ V

Raw Paste

Quick Fix ⌘ 1

Source ⌘ S >

Refactor ⌘ T >

Surround With ⌘ Z >

Local History >

References >

Declarations >

Add to Snippets...

Coverage As >

Run As >

Debug As >

Profile As >

Team >

.java - Eclipse IDE

```
Date fecha) {
    la fecha " + fecha.toString() + "\n";
    er(p -> fecha.equals(p.fecha())).collec
    re() + ": ";
    ) {
    ) + ";";
    2);
    es);
```

Move... ⌘ V

Change Method Signature... ⌘ C

Extract Method... ⌘ M

Extract Interface...

Extract Superclass...

Use Supertype Where Possible...

Pull Up...

Push Down...

Extract Class...

Introduce Parameter Object...

[Se requieren ajustes manuales]

- Antes de poder aplicar “Extract Method” automáticamente:
 - El código que queremos extraer en un nuevo método necesita devolver 2 valores: result y totalGames
 - Entonces primero hay que mover la declaración de totalGames al código que vamos a extraer.
- Luego de aplicar “Extract Method”
 - Vemos que no hace falta pasar result como parámetro
 - Eliminamos el parámetro y volvemos a declarar result dentro del nuevo método.

[A tener en cuenta...]

- Testear siempre después de hacer un cambio
 - Sí se cometió un error es más fácil corregirlo
- Definir buenos nombres

[En la clase ClubTenis...]

- A quien pertenece realmente el código de

String mostrarPartido(Partido p) {...}. ?



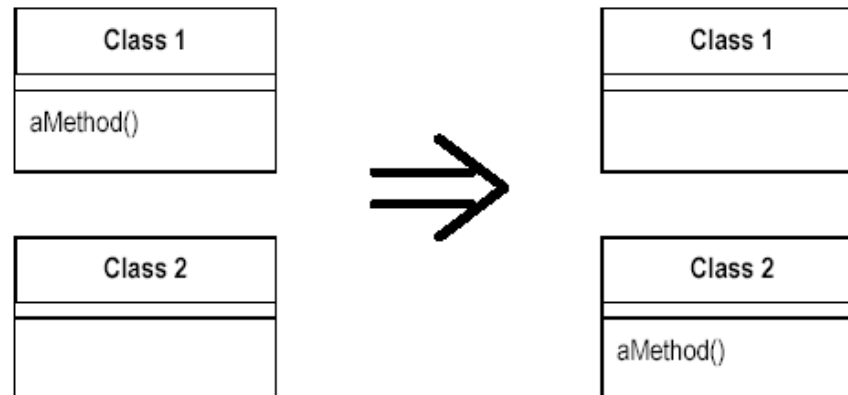
```
public class ClubTenis {  
  
    private String mostrarPartido(Partido partido) {  
        int totalGames = 0;  
        Jugador j1 = partido.jugador1();  
        String result = "Partido: " + "\n";  
        result += "Puntaje del jugador: " + j1.nombre() + ": ";  
        for (int gamesGanados: partido.puntosPorSetDe(j1)) {  
            result += Integer.toString(gamesGanados) + ".";  
            totalGames += gamesGanados; }  
        result += "Puntos del partido: ";  
        if (j1.zona() == "A") result += Integer.toString(totalGames * 2);  
        if (j1.zona() == "B") result += Integer.toString(totalGames);  
        if (j1.zona() == "C")  
            if (partido.ganador() == j1)  
                result += Integer.toString(totalGames);  
            else  
                result += Integer.toString(0);  
        Jugador j2 = partido.jugador2();  
  
        ...  
        ...  
        return result;  
    }  
}
```




RESPONSABILIDAD MAL ASIGNADA

[Refactoring “Move Method”]

- Motivación:
 - Un método esta usando o usará muchos servicios que están definidos en una clase diferente a la suya
- Solucion:
 - Mover el método a la clase donde están los servicios que usa.
 - Convertir el método original en un simple delegación o eliminarlo



[Refactoring Move Method]

```
public class ClubTenis {  
    public String mostrarPuntajesJugadoresEnFecha(LocalDate fecha) {  
        List<Partido> partidosFecha;  
        String result = "Puntajes para los partidos de la fecha " + fecha.toString() + "\n";  
        partidosFecha = coleccionPartidos.stream().filter(...);  
        for(Partido p: partidosFecha)  
            result = p.mostrar();  
        return result;  
    }  
}
```

```
public class Partido {  
    public String mostrar() {  
        int totalGames = 0;  
        Jugador j1 = jugador1();  
        String result = "Partido: " + "\n";  
        result += "Puntaje del jugador: " + j1.nombre() + ": ";  
        ...  
    }  
}
```

[“Move Method” a mano]

■ Mecánica:

1. Revisar las v.i. usadas por el método a mover. Tiene sentido moverlas también?
2. Revisar super y subclases por otras declaraciones del método. Si hay otras tal vez no se pueda mover.
3. Crear un nuevo método en la clase target cuyo nombre explique su propósito
4. Copiar el código a mover al nuevo método. Ajustar lo que haga falta
5. Compilar la clase target
6. Determinar como referenciar al target desde el source
7. Reemplazar el método original por llamada a método en target
8. Compilar y testear
9. Decidir si remover el método original o mantenerlo como delegación

[En la clase Partido...]

- mostrar() ¿es un buen nombre?
- Aplicamos el refactoring Rename Method porque todo buen código debería comunicar con claridad lo que hace, sin necesidad de agregar comentarios. Lo llamamos toString()
- El metodo sigue siendo bastante largo, porque tiene código duplicado
 - más Extract Method!

```
public class Partido {
```

```
@Override
```

```
public String toString() {
```

```
    int totalGames = 0;
```

```
    Jugador j1 = jugador1();
```

```
    String result = "Partido: " + "\n";
```

```
    result += "Puntaje del jugador: " + j1.nombre() + ": ";
```

```
    for (int gamesGanados: puntosPorSetDe(j1)) {
```

```
        result += Integer.toString(gamesGanados) + ",";
```

```
        totalGames += gamesGanados; }
```

```
    result += "Puntos del partido: ";
```

```
    if (j1.zona() == "A") result += Integer.toString(totalGames * 2);
```

```
    if (j1.zona() == "B") result += Integer.toString(totalGames);
```

```
    if (j1.zona() == "C")
```

```
        if (ganador() == j1)
```

```
            result += Integer.toString(totalGames);
```

```
        else
```

```
            result += Integer.toString(0);
```

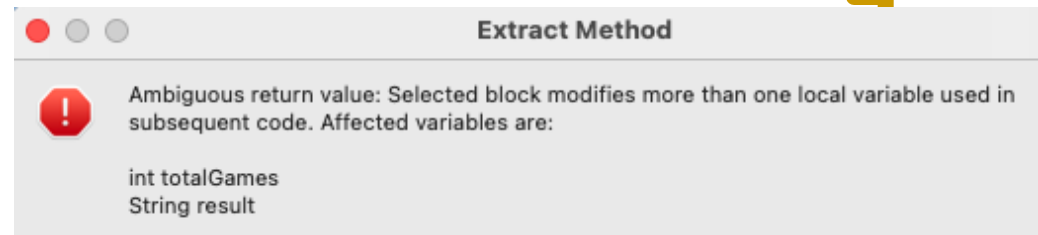
```
    Jugador j2 = jugador2();
```

```
    totalGames = 0;
```

```
    result += "Puntaje del jugador: " + j2.nombre() + ": ";
```

```
...
```

```
...
```



@Override

```
public String toString() {  
    int totalGames = 0;  
    Jugador j1 = jugador1();  
    String result = "Partido: " + "\n";  
    result = puntosJugadorToString(totalGames, j1, result);  
    Jugador j2 = jugador2();  
    totalGames = 0;  
    result = puntosJugadorToString(totalGames, j2, result);  
    return result;  
}
```

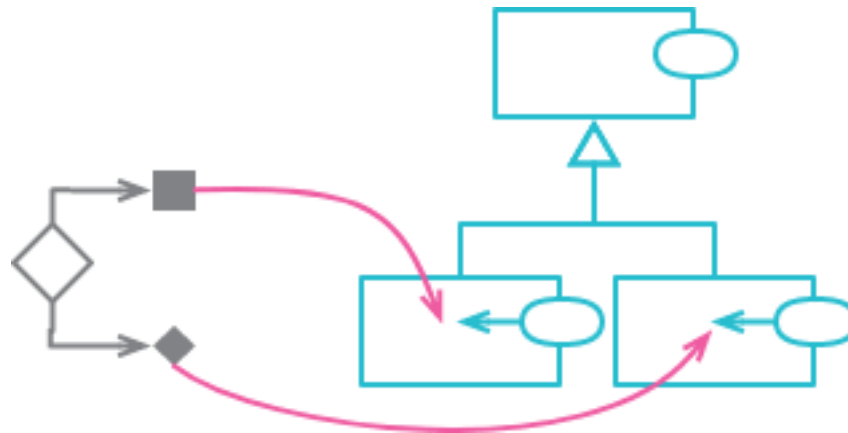
```
public String puntosJugadorToString(int totalGames, Jugador j1, String result) {  
    result += "Puntaje del jugador: " + j1.nombre() + ": ";  
    for (int gamesGanados: puntosPorSetDe(j1)) {  
        result += Integer.toString(gamesGanados) + ";";  
        totalGames += gamesGanados; }  
    result += "Puntos del partido: ";  
    if (j1.zona() == "A")  
        result += Integer.toString(totalGames * 2);  
    if (j1.zona() == "B")  
        result += Integer.toString(totalGames);  
    if (j1.zona() == "C") {  
        ...  
    }  
}
```

Seguimos ajustando!!

```
public String puntosJugadorToString(Jugador unJugador) {  
    int totalGames = 0;  
    String result = "Puntaje del jugador: " + unJugador.nombre() + ": ";  
    for (int gamesGanados: puntosPorSetDe(unJugador)) {  
        result += Integer.toString(gamesGanados) + ";";  
        totalGames += gamesGanados; }  
    result += "Puntos del partido: ";  
    if (unJugador.zona() == "A")  
        result += Integer.toString(totalGames * 2);  
    if (unJugador.zona() == "B")  
        result += Integer.toString(totalGames);  
    if (unJugador.zona() == "C")  
        if (this.ganador() == unJugador)  
            result += Integer.toString(totalGames);  
        else  
            result += Integer.toString(0);  
    return result;  
}
```


[Seguimos teniendo el switch]

- ¿Cómo eliminar el switch?
- ➔ Replace Conditional with Polymorphism

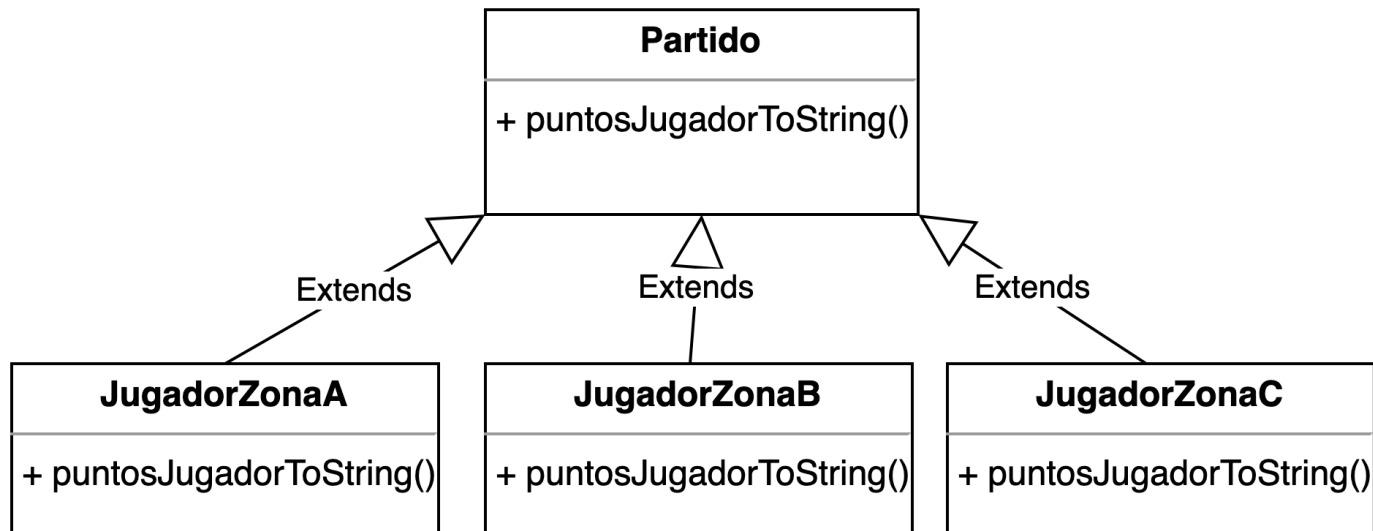
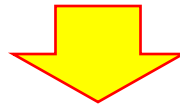


Partido>>puntosJugadorToString(Jugador unJugador)



...

```
if (unJugador.zona() == "A")  
    result += Integer.toString(totalGames * 2);  
if (unJugador.zona() == "B")  
    result += Integer.toString(totalGames);  
if (unJugador.zona() == "C")  
    if (ganador() == unJugador)  
        result += Integer.toString(totalGames);  
    else  
        result += Integer.toString(0);
```





- ¿Tiene sentido hacer subclases de Partido?
¿Corresponde a Partido este cálculo?

[No corresponde a Partido]

- Aplico **Move Method**





Partido>>puntosJugadorToString(Jugador j) a
Jugador>>puntosEnPartidoToString(Partido p)

- Aplico **Replace Conditional with Polymorphism** en

Jugador>>puntosEnPartidoToString(Partido p)

Move Method

New target for 'String puntosJugadorToString(Jugador unJugador)':

Type	Receiver
 Jugador	 unJugador
 Jugador	 ganador

New method name:

New parameter name:

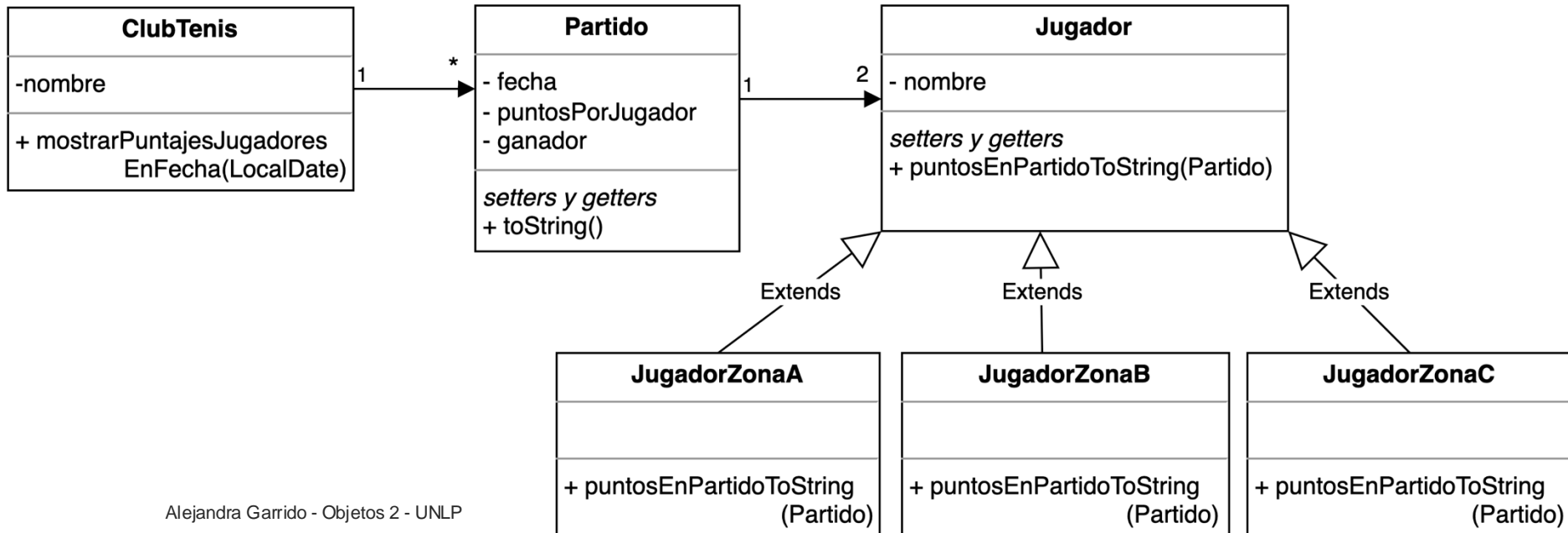
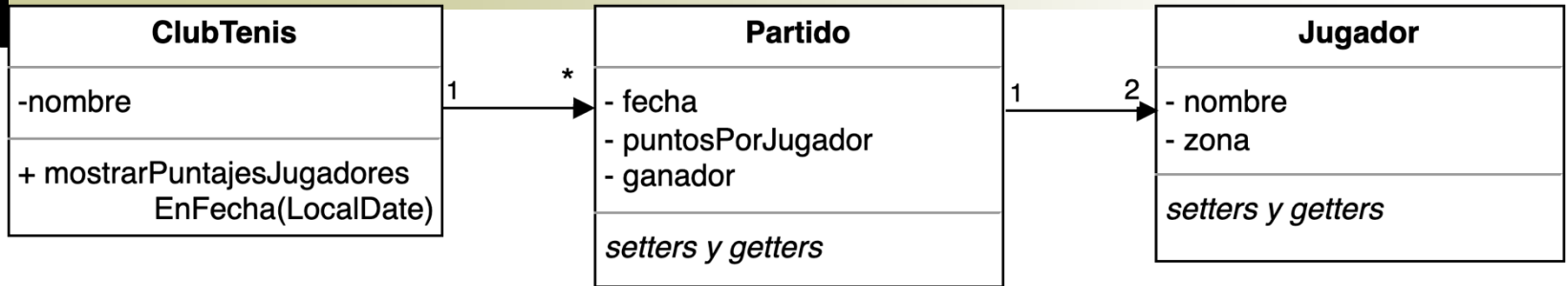
☐ Keep original method as delegate to moved method

☒ Mark as deprecated

Preview >

Cancel

OK



Replace Conditional with Polymorphism

■ Mecánica:

1. Crear la jerarquía de clases necesaria
2. Si el condicional es parte de un método largo: Extract Method
3. Por cada subclase:
 1. Crear un método que sobrescribe al método que contiene el condicional
 2. Copiar el código de la condición correspondiente en el método de la subclase y ajustar
 3. Compilar y testear
 4. Borrar la condición y código del branch del método en la superclase
 5. Compilar y testear
4. Hacer que el método en la superclase sea abstracto

Antes de Replace Conditional with Polymorphism

Jugador>>puntosEnPartidoToString(Partido partido)

...

```
if (zona() == "A")
    result += Integer.toString(totalGames * 2);
if (zona() == "B")
    result += Integer.toString(totalGames);
if (zona() == "C")
    if (partido.ganador() == this)
        result += Integer.toString(totalGames);
    else
        result += Integer.toString(0);
```




[Después de Replace Conditional y ajustes]

```
public class Jugador {  
    public String puntosEnPartidoToString(Partido partido) {  
        int totalGames = 0;  
        String result = "Puntaje del jugador: " + nombre() + ": ";  
        for (int gamesGanados: partido.puntosPorSetDe(this)) {  
            result += Integer.toString(gamesGanados) + ";";  
            totalGames += gamesGanados; }  
        result += "Puntos del partido: ";  
        result +=  
            Integer.toString(this.puntosGanadosEnPartido(partido,  
                                                                totalGames));  
        return result;  
    }  
}  
public class JugadorZonaA {  
    public int puntosGanadosEnPartido(Partido partido, int totalGames) {  
        return totalGames * 2;  
    }  
}
```

[Ajustando (2)]

```
public class Jugador {
    public String puntosEnPartidoToString(Partido partido) {
        int totalGames = 0;
        String result = "Puntaje del jugador: " + nombre() + ": ";
        for (int gamesGanados: partido.puntosPorSetDe(this)) {
            result += Integer.toString(gamesGanados) + ";";
            totalGames += gamesGanados; }
        result += "Puntos del partido: ";
        result +=
            Integer.toString(this.puntosGanadosEnPartido(partido,
                                                                    totalGames));

        return result;
    }
}

public class JugadorZonaA {
    public int puntosGanadosEnPartido(Partido partido, int totalGames) {
        return totalGames * 2;
    }
}
```

Refactoring: Replace Temp with Query

- Motivación: usar este refactoring:
 - Para evitar métodos largos. Las temporales, al ser locales, fomentan métodos largos
 - *Para poder usar una expresión desde otros métodos*
 - Antes de un Extract Method, para evitar parámetros innecesarios
- Solución:
 - Extraer la expresión en un método
 - Reemplazar TODAS las referencias a la var. temporal por la expresión
 - El nuevo método luego puede ser usado en otros métodos

Ajustando(3)

```
public class Jugador {  
    public String puntosEnPartidoToString(Partido partido) {  
        String result = "Puntaje del jugador: " + nombre() + ": ";  
        for (int gamesGanados: partido.puntosPorSetDe(this))  
            result += Integer.toString(gamesGanados) + ";";  
        result += "Puntos del partido: ";  
        result +=  
            Integer.toString(this.puntosGanadosEnPartido(partido));  
        return result; }  
  
    public int totalGamesEnPartido(Partido partido) {  
        int totalGames;  
        for (int gamesGanados: partido.puntosPorSetDe(this))  
            totalGames += gamesGanados;  
        return totalGames;  
    }  
  
    public class JugadorZonaA {  
        public int puntosGanadosEnPartido(Partido partido) {  
            return totalGamesEnPartido(partido) * 2;  
        }  
    }  
}
```

[Sobre la performance]

- La mejor manera de optimizar un programa, primero es escribir un programa bien factorizado y luego optimizarlo
- En el ejemplo, mientras estamos refactorizando no importa que el cambio nos haga recorrer más de una vez la colección de sets de un partido, no solo porque la colección es corta sino que además ganamos en flexibilidad y reusabilidad
 - El código es más flexible porque el día de mañana podemos cambiar la manera de generar los datos que se muestran sin que eso cambie la manera de calcular los puntos; ambos algoritmos pueden cambiar de manera independiente
 - El código es más reusable porque ahora cuento con un método que calcula los puntos de cada jugador y lo puedo invocar también en otros casos donde no necesite devolver un string con los resultados