Refactoring

Alejandra Garrido
Objetos 2
Facultad de
Informática - UNLP

En la clase pasada…

- Los elementos distintivos de la arquitectura de un sistema no surgen hasta *después* de tener código que funciona
- Construir el sistema perfecto es imposible
- Los errores y el cambio son inevitables
- No se trata sólo de agregar, sino de *adaptar*, *transformar.*
- Hay que aprender del **feedback**

Nos ponemos ágiles

- http://agilemanifesto.org/

- Dos prácticas ágiles esenciales:
  - Refactoring
  - Testing

Refactoring

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

  - Framework Choices. Qué cambios ocurren de una iteración a otra?

Surge el refactoring en la OO

- Restructurings in a class hierarchy
  E.g. “Create an abstract superclass”
  - “Creating Abstract Superclasses by Refactoring”. Opdyke & Johnson. ACM Conf. Computer Science. 1993

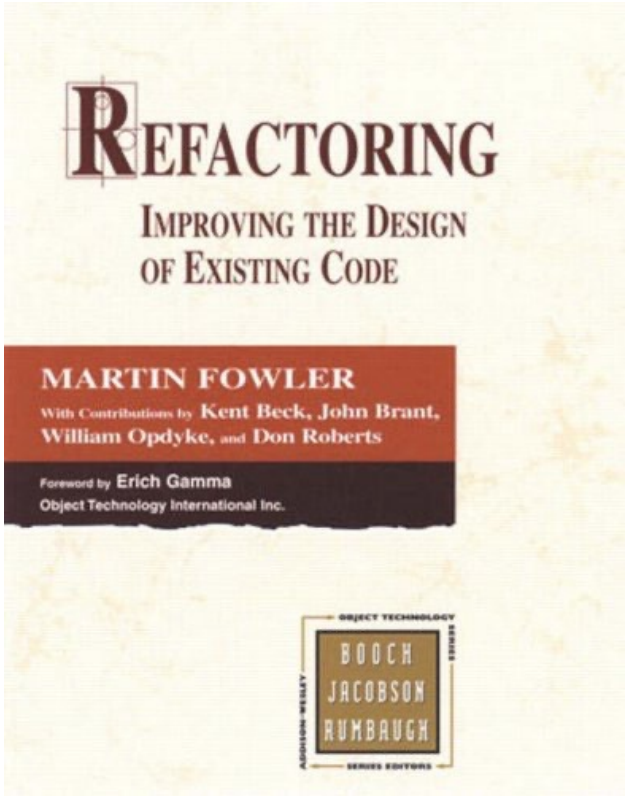
- Restructurings between components
  E.g. “Converting inheritance into aggregation”
  - “Refactoring and Aggregation”. Johnson & Opdyke. ISOTAS 1993.

Refactoring

- Refactoring como una transformación que preserva el comportamiento

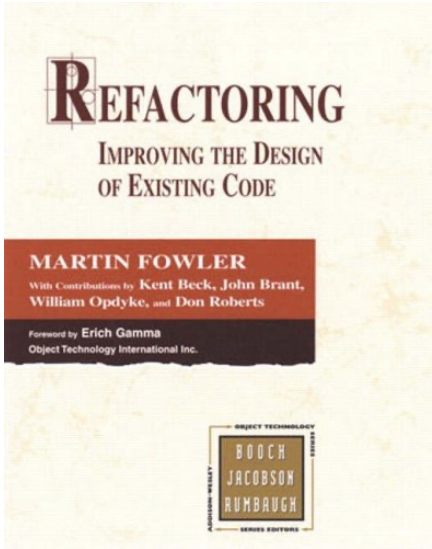
Beautifying code

Refactoring by Martin Fowler

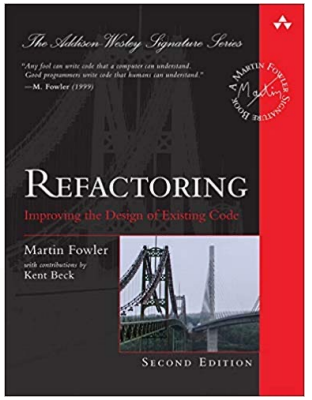


Refactoring

- Es el proceso a través del cual se cambia un sistema de software mejorando la organización, legibilidad, adaptabilidad y mantenibilidad del código luego que ha sido escrito….
  - que **NO altera** el comportamiento externo del sistema,
  - que ***mejora*** su estructura interna



Refactoring by Fowler



- *Refactoring* (sustantivo): cada uno de los cambios catalogados
  - “A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior”.
  - Con un nombre específico y una receta (mecánica)

- *Refactor* (verbo): el proceso de aplicar refactorings
  - “To restructure software by applying a series of refactorings without changing its observable behavior”

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 15/3/2022

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:

.....

Ejemplo. ClubTenis class

```
public class ClubTenis
    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…. //siguientes 2 slides
      return result;
   }
```

Ejemplo ClubTenis (2)

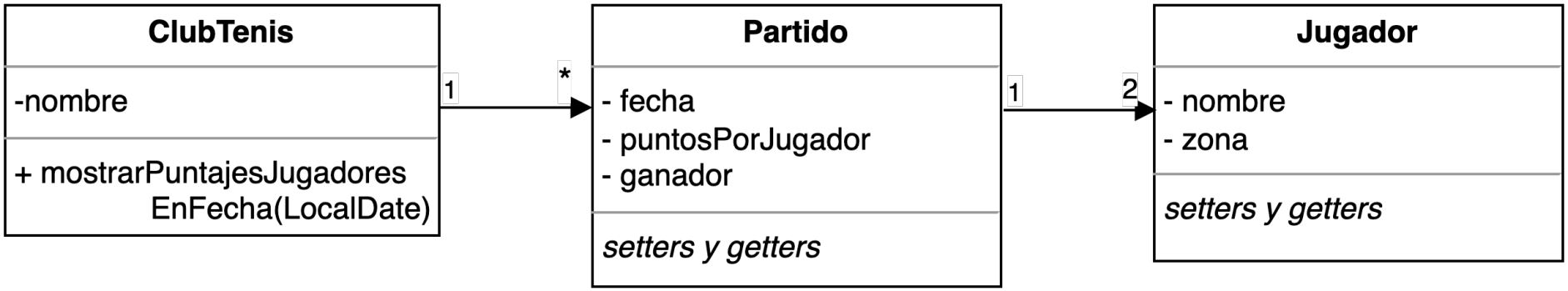
```
for(Partido p: partidosFecha) {
        Jugador j1 = p.jugador1();
        result += “Partido: ” + “\n”;
        result += “Puntaje del jugador: “ + j1.nombre() + “: “;
        for (int gamesGanados: p.puntosPorSetDe(j1)) {
                result += Integer.toString(gamesGanados) + “;”;
                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);
        }
//sigue prox. slide
```

Ejemplo ClubTenis (3)

```
//viene de slide anterior
        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);
        }
…
```

```
public class ClubTenis
    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) {
                    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

Ejemplo del club de tenis

- Por dónde empezamos?
- Cuáles son los problemas que tiene el código?

MÉTODO LARGO!

Refactoring Extract Method

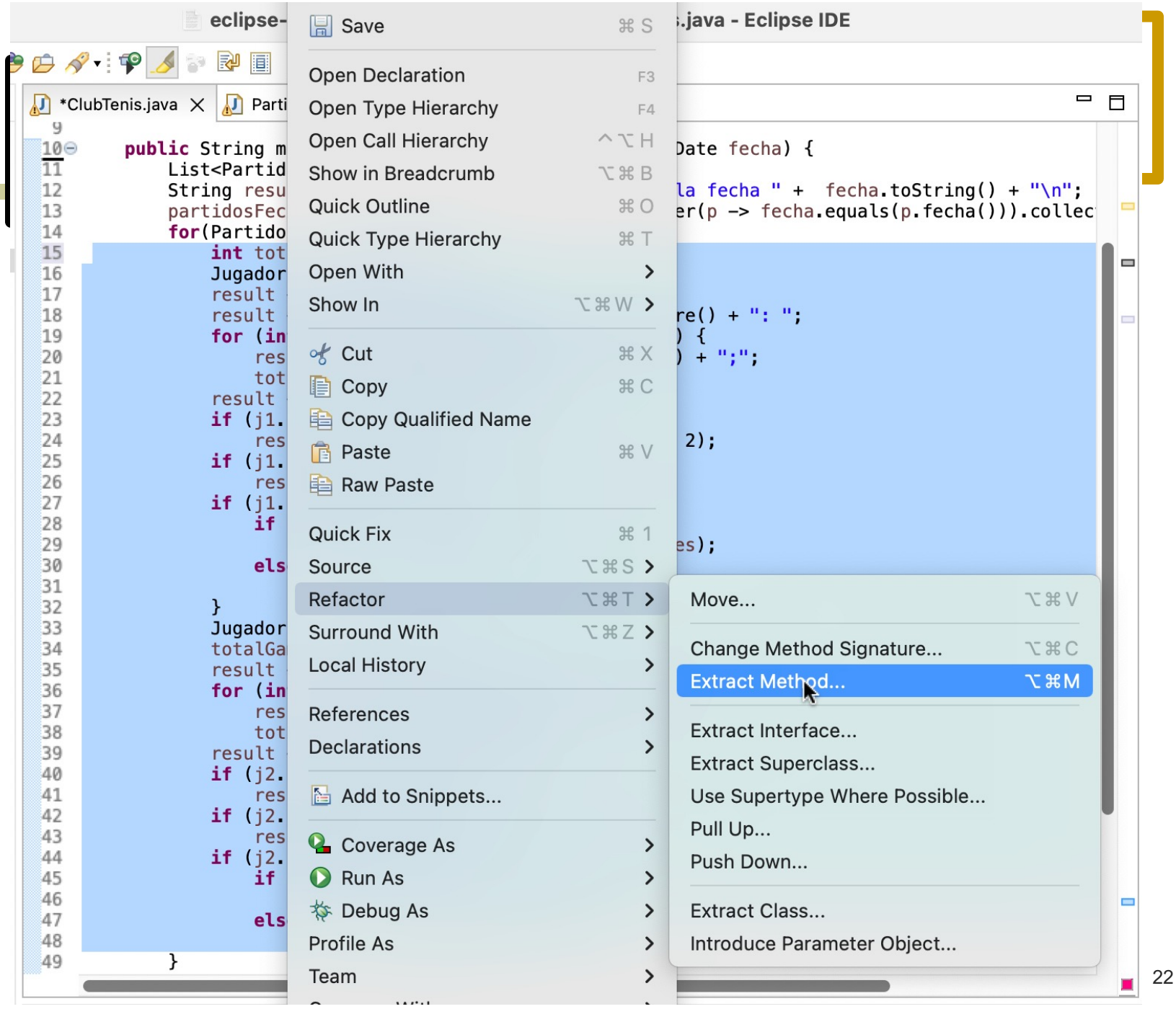
- Motivación :
  - Métodos largos
  - Métodos muy comentados
  - Incrementar reuso
  - Incrementar legibilidad

Refactoring Extract Method

- **Mecánica**:
  - Crear un nuevo método cuyo nombre explique su propósito
  - Copiar el código a extraer al nuevo método
  - Revisar las variables locales del original
  - Si alguna se usa sólo en el código extraído, mover su declaración
  - Revisar si alguna variable local es modificada por el código extraido. Si es solo una, tratar como query y asignar. Si hay más de una no se puede extraer.
  - Pasar como parámetro las variables que el método nuevo lee.
  - Compilar
  - Reemplazar código en método original por llamada
  - 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 -> p.fecha().
                                    equals(fecha)).collect(Collectors.toList());
            for(Partido p: partidosFecha)
                        result += this.mostrarPartido(p);
            return result;
}

  private String mostrarPartido(Partido p) {…}
```


eclipse-
.java - Eclipse IDE
*ClubTenis.java
Parti
public String m
List<Partid
String resu
partidosFec
for(Partido
int tot
Jugador
result
result
for (in
res
tot
result
if (j1.
res
if (j1.
res
if (j1.
if
els
}
Jugador
totalGa
result
for (in
res
tot
result
if (j2.
res
if (j2.
res
if (j2.
if
els
}
Date fecha) {
la fecha " + fecha.toString() + "\n";
er(p -> fecha.equals(p.fecha())).collec
re() + ": ";
) {
) + ";";
2);
es);
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
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...

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 codigo de

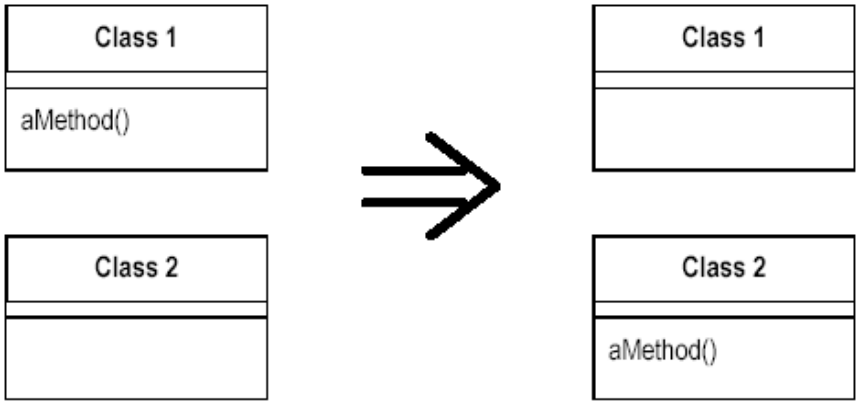
String mostrarPartido(Partido p) {…}. **?**

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

“Move Method” a mano

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

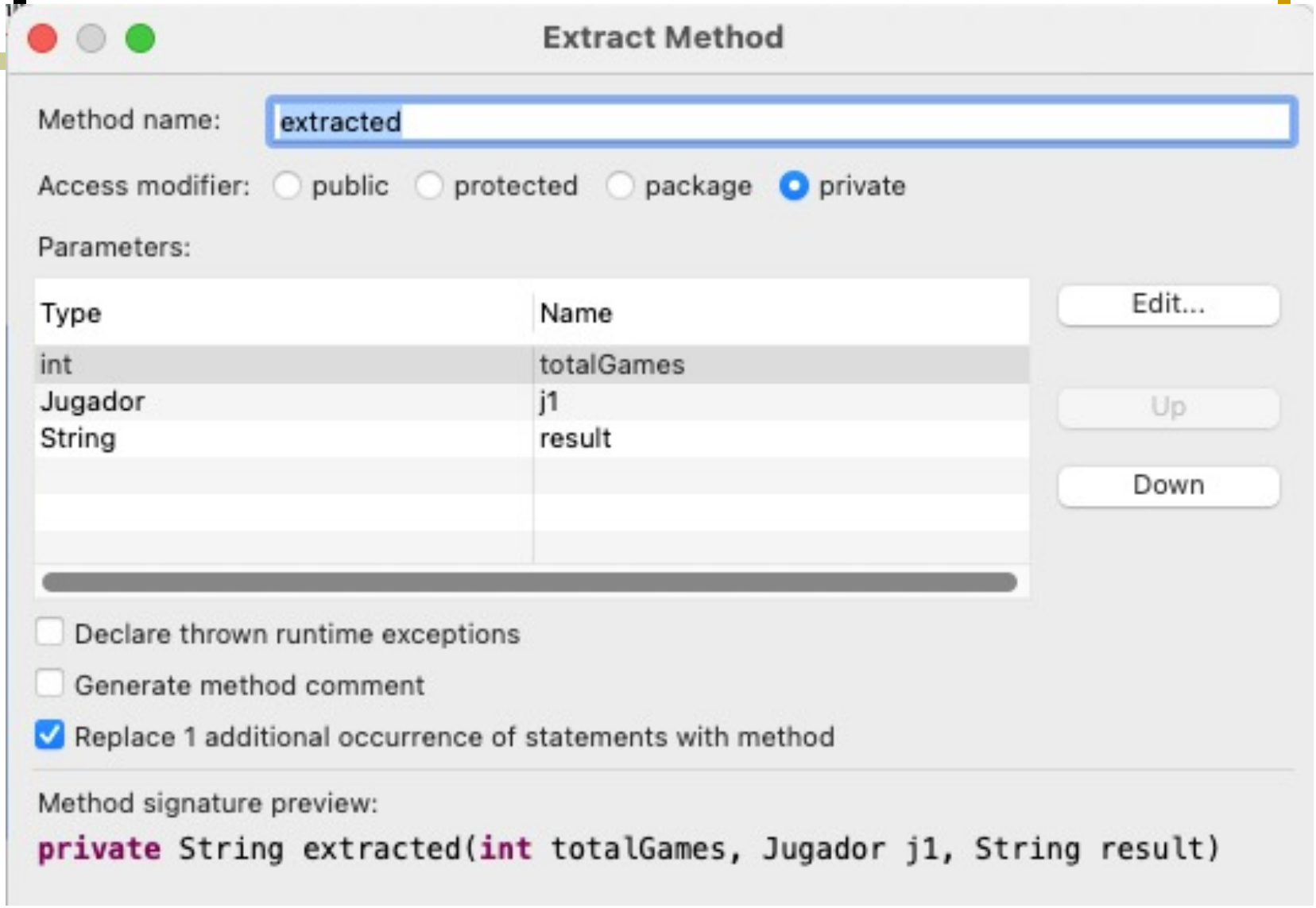
En la clase Partido…

- 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!

```
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() + “: “;
…       ...
```

Extract con Eclipse



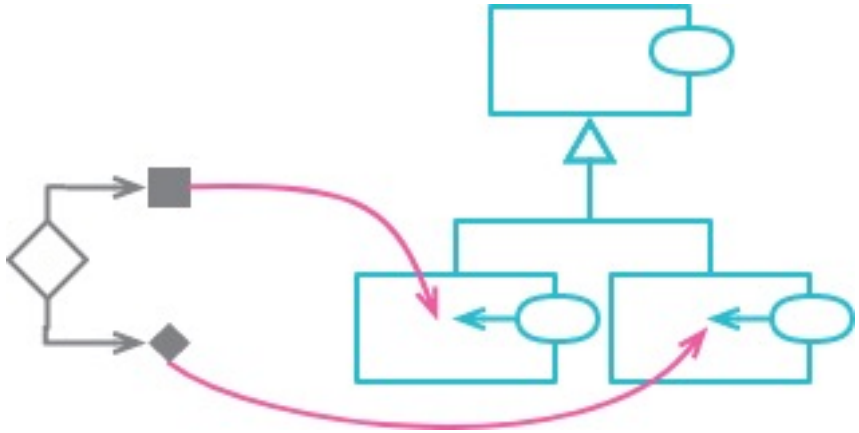
```java
@Override
public String toString() {
  int totalGames = 0;
  Jugador j1 = jugador1();
  String result = "Partido: " + "\n";
  result = puntosJugadorToString(totalGames, j1, result);
  Jugador j2 = jugador2();
  totalGames = 0;
  result = puntosJugadorToString(totalGames, j2, result);
  return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

private 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!!

```
private 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 (ganador() == unJugador)
            result += Integer.toString(totalGames);
        else
            result += Integer.toString(0);
  return result;
}
```

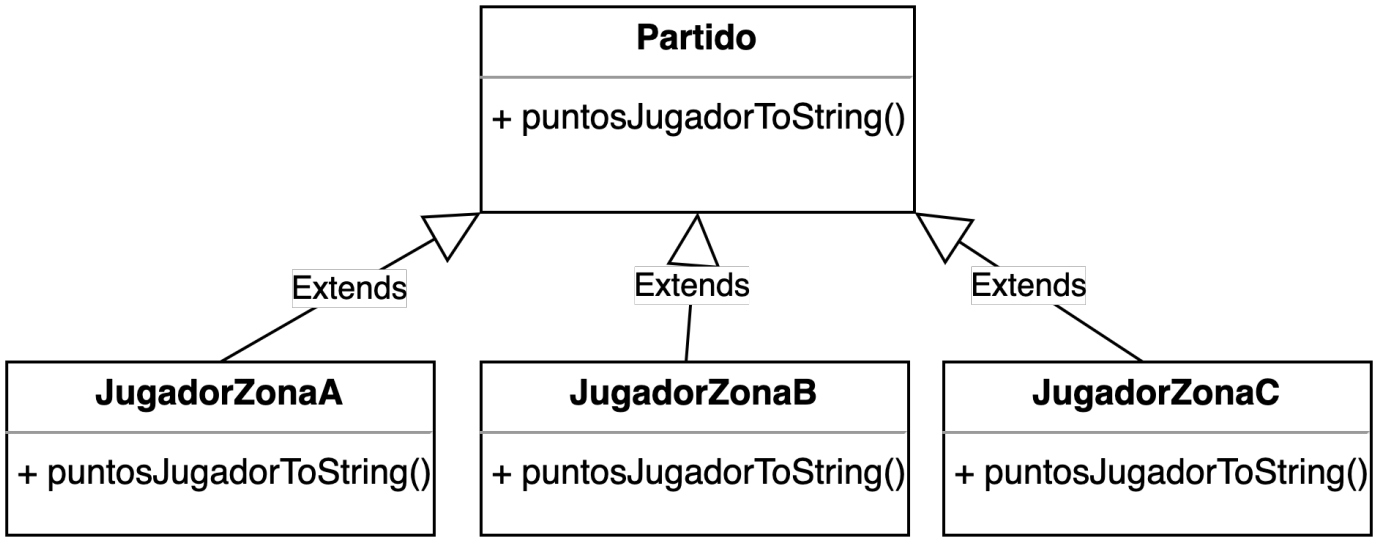
Seguimos teniendo el switch

- ¿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);
```

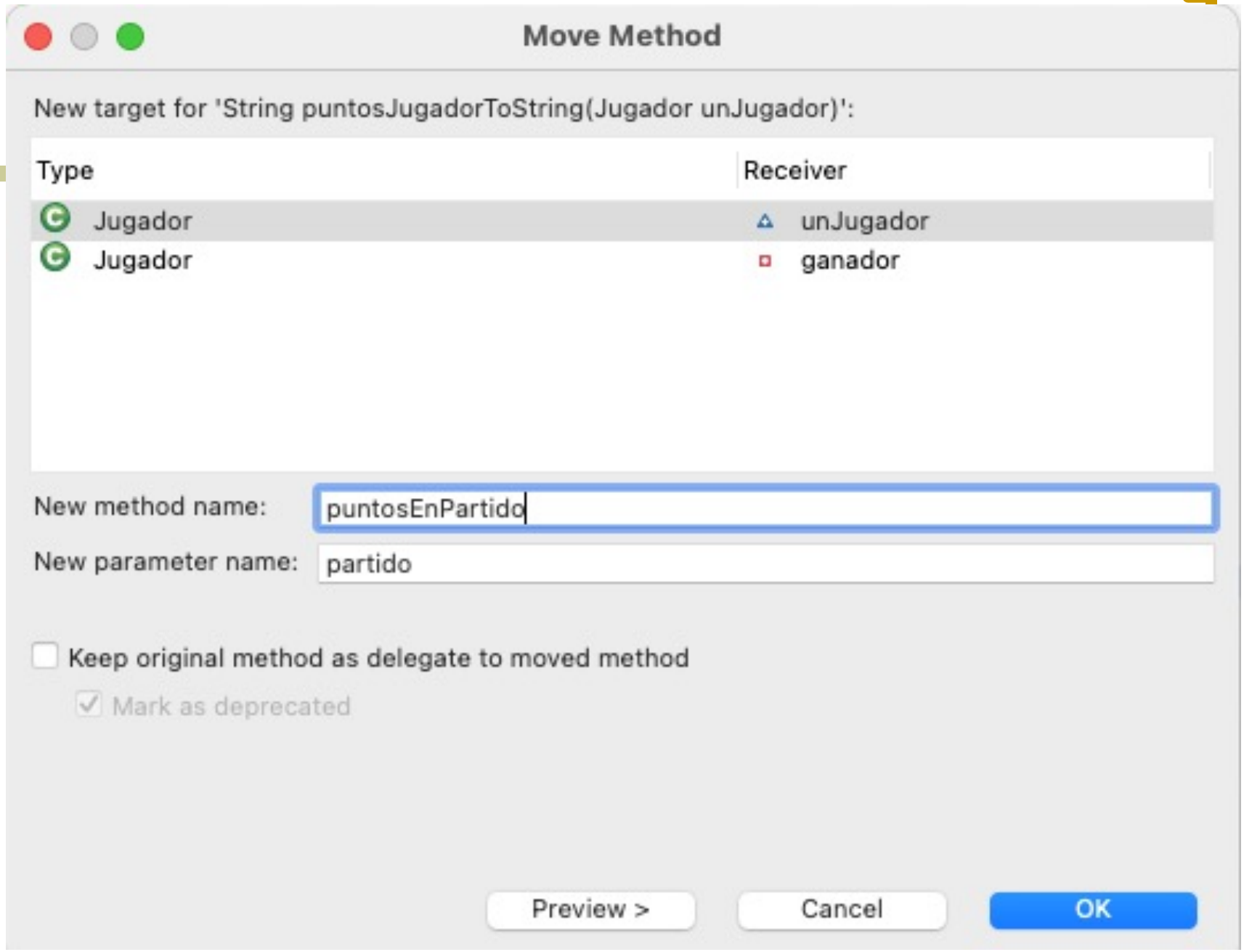


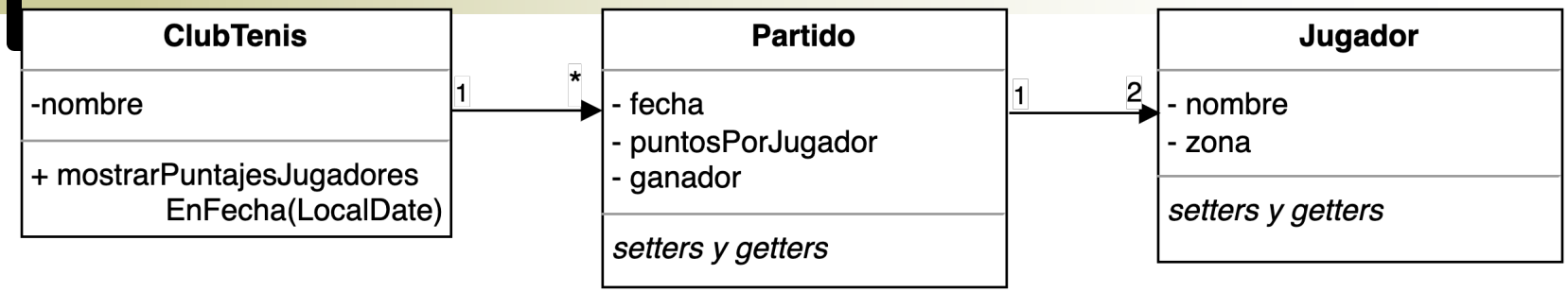
- ¿Tiene sentido hacer subclases de Partido? ¿Corresponde a Partido este cálculo?

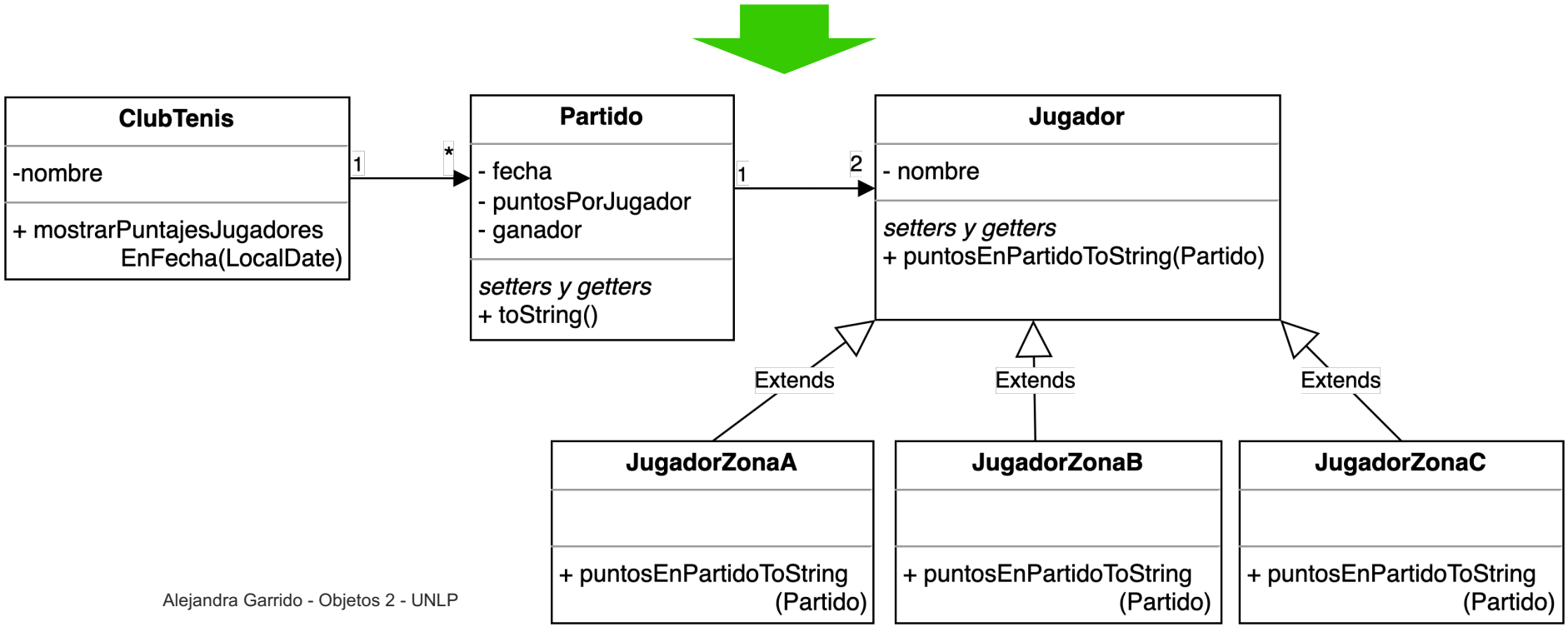
No corresponde a Partido

- 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:
puntosEnPartido
New parameter name:
partido
Keep original method as delegate to moved method
Mark as deprecated
Preview >
Cancel
OK


ClubTenis
-nombre
+ mostrarPuntajesJugadores EnFecha(LocalDate)
1
*
Partido
- fecha
- puntosPorJugador
- ganador
setters y getters
1
2
Jugador
- nombre
- zona
setters y getters


ClubTenis
-nombre
+ mostrarPuntajesJugadores EnFecha(LocalDate)
1
*
Partido
- fecha
- puntosPorJugador
- ganador
setters y getters
+ toString()
1
2
Jugador
- nombre
setters y getters
+ puntosEnPartidoToString(Partido)
Extends
Extends
Extends
JugadorZonaA
+ puntosEnPartidoToString (Partido)
JugadorZonaB
+ puntosEnPartidoToString (Partido)
JugadorZonaC
+ puntosEnPartidoToString (Partido)

Replace Conditional with Polymorphism

- Mecánica:
  - Crear la jerarquía de clases necesaria
  - Si el condicional es parte de un método largo: Extract Method
  - Por cada subclase:
    - Crear un método que sobreescribe al método que contiene el condicional
    - Copiar el código de la condición correspondiente en el método de la subclase y ajustar
    - Compilar y testear
    - Borrar la condición y código del branch del método en la superclase
    - Compilar y testear
  - 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 {
 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(puntosGanadosEnPartido(partido,
                                                      totalGames));
    return result;
}
public class JugadorZonaA {
 private int puntosGanadosEnPartido(Partido partido, int
totalGames) {
    return totalGames * 2;
 }
```

Ajustando (2)

```
public class Jugador {
 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(puntosGanadosEnPartido(partido,
                                                      totalGames));
    return result;
}
public class JugadorZonaA {
 private 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
  - Remplazar 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 {
 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(puntosGanadosEnPartido(partido));
   return result;   }

 int totalGamesEnPartido(Partido partido) {
   int totalGames;
   for (int gamesGanados: partido.puntosPorSetDe(this))
       totalGames += gamesGanados;
   return totalGames;
 }

public class JugadorZonaA {
 private 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 podriamos refactorizar luego para usar streams de Java 8