

Integrantes:

Katherine Alexandra Tumbaco Sellan	202104485
Cesar Alejandro Arana Castro	202101663
Melisa Nathalia Ayllon Gutiérrez	202109641

Contenido

Sección A	3
Identificación.....	3
<i>a) Malos Olores</i>	<i>3</i>
<i>b) Técnicas de refactorización que utilizaría para mejorar el código.....</i>	<i>5</i>
Sección B.....	6
Identificación de violación a principios SOLID	6
Malos olores de programación que tiene el código.	6
Refactorización	10

Sección A

Identificación

Señalar los malos olores e indicar su nombre. Expliqué por qué sería un problema.

a) *Malos Olores*

- 76 Long method

El método calcularCobroProducto es demasiado extenso, esto influye negativamente a la lectura.

```
public double calcularCobroProducto(){
    double cobro = 0;
    switch (getTipoProducto()) {
        case Fragil:
            // 500 es el cobro base y 0.55 el factor de distancia
            cobro = 500 + getDistanciaDestino()*0.55;
            break;
        case Regular:
            // 250 es el cobro base, 0.25 es el factor de distancia
            // y 10 es el factor de recargo por peso
            cobro = 250 + getDistanciaDestino()*0.25+getPeso()*10;
            break;
        case Pesado:
            if (peso > 30)
                //cobro base es 999 y factores son 0.9 y 9
                cobro = 999+getDistanciaDestino()*0.9+getPeso()*9;
            else
                //cobro base es 750 y factor de distancia es 0.6
                cobro = 750 + getDistanciaDestino()*0.6;
            break;
    }
    return cobro;
}
```

- Comments

Existen comentarios en el código, esto impide seguir con el flujo de lectura del código, y además se lo agrega cuando no está del todo claro el código.

```
81         // 250 es el cobro base, 0.25 es el factor de distancia
82         // y 10 es el factor de recargo por peso
```

- Alternative Classes with Different Interfaces

Existen estas dos clases que, aunque estén heredando de la misma clase, tienen sus propios métodos para crear su formato y en los parámetros se puede ver que solicitan los mismo, esto causa duplicación de código y la similitud de la forma de implementar y declarar el método.

```

109 public class ReporteHtml extends Reporte{
110     public String generarFormatoHtml(int cantidadProductos,
111                                     double ingresos) {
112         String html = "<HTML><TITLE>Reporte en HTML</TITLE>";
113         html += "<HEAD>Reporte del flete "+formatearFecha()+"</HEAD>";
114         html += "<BODY><P>Este flete transporta "+cantidadProductos;
115         html += " productos.</P>\n";
116         html += "<P>Estos productos dejan un ingreso de "+ingresos;
117         html += " USD.</P>\n</BODY>\n<HTML>";
118         return html;
119     }
120 }

122 public class ReporteTexto extends Reporte{
123     public String generarFormatoTexto(int cantidadProductos,
124                                     double ingresos) {
125         String texto = "Reporte del flete "+formatearFecha()+"\n ";
126         texto += "Este flete transporta "+cantidadProductos;
127         texto += " productos. Estos productos dejan ";
128         texto += "un ingreso de "+ ingresos + " USD.";
129         return texto;
130     }
131 }
132 }

```

- Switch Statements línea 76 a línea 92

La utilización de muchos casos genera conflicto durante la lectura y organización.

```

switch (getTipoProducto()) {
    case Fragil:
        // 500 es el cobro base y 0.55 el factor de distancia
        cobro = 500 + getDistanciaDestino()*0.55;
        break;
    case Regular:
        // 250 es el cobro base, 0.25 es el factor de distancia
        // y 10 es el factor de recargo por peso
        cobro = 250 + getDistanciaDestino()*0.25+getPeso()*10;
        break;
    case Pesado:
        if (peso > 30)
            //cobro base es 999 y factores son 0.9 y 9
            cobro = 999+getDistanciaDestino()*0.9+getPeso()*9;
        else
            //cobro base es 750 y factor de distancia es 0.6
            cobro = 750 + getDistanciaDestino()*0.6;
        break;
}

```

-Switch Statements

La utilización de muchos casos genera conflicto durante la lectura y organización.

```

64  public void setValue(Object value){
65      if (value instanceof TipoProducto)
66          tipoProducto = (TipoProducto)value;
67      else if(value instanceof Double)
68          peso = ((Double)value).doubleValue();
69  }

```

-Temporary Field línea 23 a la línea 24

La variable ingreso es creada de forma innecesaria, pues solo almacena un dato que será directamente utilizado en la siguiente línea de código.

```

23      double ingreso = producto.calcularCobroProducto();
24      totalIngresos += ingreso;

```

b) Técnicas de refactorización que utilizaría para mejorar el código.

-Reemplazar switch con Polimorfismo línea 75 a la línea 96

Esta técnica de refactorización se ajusta a tener como nuestro atributo una clase específica para él y no los distintos casos.

Y al estar asociado nuestro objeto a la clase, podemos asociarle el método de cobro. Y ya no depender del switch.

-Form Template Method línea 123 y 110 deberían moverse a la clase Flete

Esta técnica de refactorización nos enuncia que al tener un método similar en dos clases distintas podemos asociarlo como uno solo a una clase padre.

Esto es conveniente porque de ese modo ya no va a requerir más nombres y se puede llegar a confusión.

-Replace parameter with Explicit Methods línea 64 a 69

Esta técnica de refactorización nos enuncia que tenemos en un método con casos condicionales, pero a cada condición se le asocian distintas tareas independientes entre sí, es por ello que se pueden crear métodos separados para cada una de estas tareas.

Esto es conveniente porque mejorar la lectura del código y no tiene que pasar por cada condicional.

-Introduce Parameter Object línea 51 a la línea 57

Para este caso se agrupan atributos compatibles, los cuales son peso y tipoProducto, en una nueva clase llamada InfoProducto. Al constructor se pasará entonces el id, el objeto de tipo InfoProducto y distanciaDestino, reduciendo el número de parámetro. Del objeto se extraen los atributos peso y tipoProducto.

-Inline Temp línea 23 a la línea 24

El dato obtenido en la variable ingreso puede ser pasada directamente a sumar al campo totalIngresos, borrando una línea innecesaria de código.

Sección B

Identificación de violación a principios SOLID

Open Closed Principle

En la clase Player el método playerChoice(), la elección del jugador está limitada a tres opciones, y que, en caso de querer agregar más opciones, o en su defecto modificar las que ya hay habría que modificar el método. Adicionalmente, en la clase RPMGame los resultados que puede tener el juego están limitados a tres objetos, lo que haría necesario cambiar el código si se necesita agregar un objeto más en el juego.

Clase de la línea 4, 15-56 SRP

Single Responsibility Principle

En la clase RPMGame se viola por completo este principio debido a que toda la lógica del juego está implementada dentro de esta clase, de esta manera al tener varias responsabilidades, se vuelve más difícil de entender, mantener y reutilizar.

Malos olores de programación que tiene el código.

Comments en línea 2, 9,15,29,61,64

En estas líneas se presentan comentarios para explicar el funcionamiento del código, estos comentarios pueden llegar a ser innecesarios ya que si el código se implementa de forma correcta debe ser entendido sin la necesidad de implementar comentarios en el mismo.

```
1  /*
2   * Simulate a game of Rock, Paper, Scissors
3   */
4  public class RPMGame {
5      public static void main(String args[]) {
6          Player p1 = new Player();
7          Player p2 = new Player();
8          boolean gameWon = false;
9          int roundsPlayed = 0; // Number of rounds played
10         int p1Wins = p1.wins;
11         int p2Wins = p2.wins;
12         int draw = 0;
13         String p1Choice;
14         String p2Choice;
15         // Game Loop
16         do {
17             System.out.println("***** Round: " +
18                 roundsPlayed + " *****\n");
19             System.out.println("Number of Draws: " +
20                 draw + "\n");
21             p1Choice = p1.playerChoice();
22             System.out.println("Player 1: " + p1Choice +
23                 "\t Player 1 Total Wins: " + p1Wins);
24             p2Choice = p2.playerChoice();
25             System.out.println("Player 2: " + p2Choice +
26                 "\t Player 2 Total Wins: " + p2Wins);
27             if((p1Choice.equals("rock"))&&(p2Choice.equals("paper"))){
28                 System.out.println("Player 2 Wins");
29                 p2Wins++; // trying a couple different ways to get count to work
30             } else if((p1Choice.equals("paper"))&&(p2Choice.equals("rock"))){
31                 p1Wins++;
32                 System.out.println("Player 1 Wins");
33             }
34         } while (true);
35     }
36 }
```

```

60 class Player{
61     int wins;        // # of wins
62     int winTotal;
63     /**
64      * Randomly choose rock, paper, or scissors
65      */
66     public String playerChoice(){

```

Switch Statements línea 66 a la línea 80 y en el bloque Do de la línea 27 a la línea 54

En esta parte del código se hace del código se hace uso de la sentencia switch y de varias sentencias if y else if. Para la clase Player, en el método playerChoice() se refleja este mal olor en la sentencia switch para evaluar la elección de jugada. Entonces, considerando la explicación del mal olor en ese punto, podría decirse que de tener una implementación distinta las elecciones de jugadas (ser tratados como objetos de una clase y no como un tipo de dato primitivo) la sentencia switch no sería necesaria. Con esta modificación y agregando un método win dentro de las diferentes implementaciones de las elecciones del jugador se puede eliminar todo el bloque do con las sentencias if, else if implementadas.

```

do {
    System.out.println("**** Round: " +
        roundsPlayed + " ****\n");
    System.out.println("Number of Draws: " +
        draw + "\n");
    p1Choice = p1.playerChoice();
    System.out.println("Player 1: " + p1Choice +
        "\t Player 1 Total Wins: " + p1Wins);
    p2Choice = p2.playerChoice();
    System.out.println("Player 2: " + p2Choice +
        "\t Player 2 Total Wins: " + p2Wins);
    if((p1Choice.equals("rock"))&&(p2Choice.equals("paper"))) {
        System.out.println("Player 2 Wins");
        p2Wins++; // trying a couple different ways to get count to work
    } else if((p1Choice.equals("paper"))&&(p2Choice.equals("rock"))) {
        p1Wins++;
        System.out.println("Player 1 Wins");
    } else if((p1Choice.equals("rock"))&&(p2Choice.equals("scissors"))) {
        p1Wins = p1.setWins();
        System.out.println("Player 1 Wins");
    } else if((p1Choice.equals("scissors"))&&(p2Choice.equals("rock"))) {
        p2Wins = p2.setWins();
        System.out.println("Player 2 Wins");
    } else if((p1Choice.equals("scissors"))&&(p2Choice.equals("paper"))) {
        p1Wins = p1.setWins();
        System.out.println("Player 1 Wins");
    } else if((p1Choice.equals("paper"))&&(p2Choice.equals("scissors"))) {
        p2Wins = p2.setWins();
        System.out.println("Player 2 Wins");
    }
    if(p1Choice==p2Choice) {
        draw++;
        System.out.println("\n\t\t\t Draw \n");
    }
    roundsPlayed++;
    if((p1.getWins()>=3) || (p2.getWins()>=3)) {
        gameWon = true;
        System.out.println("GAME WON");
    }
}

```

```

66     public String playerChoice(){
67         String choice = "";
68         int c = (int)(Math.random()*3);
69         switch(c) {
70             case 0:
71                 choice = ("rock");
72                 break;
73             case 1:
74                 choice = ("paper");
75                 break;
76             case 2:
77                 choice = ("scissors");
78                 break;
79         }
80         return choice;
81     }

```

Long Method main a partir de la línea 5 a la 58

En la clase RPMGame, el método main() consta de un cuerpo “monstruoso”, contiene más de 10 líneas de código cuando lo recomendable es no excederse de ese límite.

Esto es resultado de una inexistente división de tareas en métodos más pequeños y concretos en realizar una tarea.


```

4 public class RPSGame {
5     public static void main(String args[]) {
6         Player p1 = new Player();
7         Player p2 = new Player();
8         boolean gameWon = false;
9         int roundsPlayed = 0; // Number of rounds played
10        int p1Wins = p1.wins;
11        int p2Wins = p2.wins;
12        int draw = 0;
13        String p1Choice;
14        String p2Choice;
15        // Game Loop
16        do {
17            System.out.println("***** Round: " +
18                roundsPlayed + " *****\n");
19            System.out.println("Number of Draws: " +
20                draw + "\n");
21            p1Choice = p1.playerChoice();
22            System.out.println("Player 1: " + p1Choice +
23                "\t Player 1 Total Wins: " + p1Wins);
24            p2Choice = p2.playerChoice();
25            System.out.println("Player 2: " + p2Choice +
26                "\t Player 2 Total Wins: " + p2Wins);
27            if((p1Choice.equals("rock"))&&(p2Choice.equals("paper"))) {
28                System.out.println("Player 2 Wins");
29                p2Wins++; // trying a couple different ways to get count to work
30            } else if((p1Choice.equals("paper"))&&(p2Choice.equals("rock"))) {
31                p1Wins++;
32                System.out.println("Player 1 Wins");
33            } else if((p1Choice.equals("rock"))&&(p2Choice.equals("scissors"))) {
34                p1Wins = p1.setWins();
35                System.out.println("Player 1 Wins");
36            } else if((p1Choice.equals("scissors"))&&(p2Choice.equals("rock"))) {
37                p2Wins = p2.setWins();
38                System.out.println("Player 2 Wins");
39            } else if((p1Choice.equals("scissors"))&&(p2Choice.equals("paper"))) {
40                p1Wins = p1.setWins();
41                System.out.println("Player 1 Wins");
42            } else if((p1Choice.equals("paper"))&&(p2Choice.equals("scissors"))) {
43                p2Wins = p2.setWins();
44                System.out.println("Player 2 Wins");
45            }
46            if(p1Choice==p2Choice) {
47                draw++;
48                System.out.println("\n\t\t\t Draw \n");
49            }
50            roundsPlayed++;
51            if((p1.getWins()>=3) || (p2.getWins()>=3)) {
52                gameWon = true;
53                System.out.println("GAME WON");
54            }
55            System.out.println();
56        } while(gameWon != true);
57    }

```

Temporary field línea 62 y línea 82 a 85

En la clase Player se crea una variable que se utiliza para un propósito específico durante un tiempo limitado, que solamente se implementa cuando se utiliza el método setWins(), con lo que este campo sería innecesario y puede afectar la legibilidad, mantenibilidad y rendimiento del código en caso se llegué a realizar una implementación más extensa.

```

62     int winTotal;

82     public int setWins() {
83         int winTotal = wins++;
84         return winTotal;
85     }

```

Técnicas de refactorización

Reemplazar switch con polimorfismo línea 66 a 81

Esta técnica de refactorización se ajusta a tener como nuestro atributo una clase específica para él y no los distintos casos.

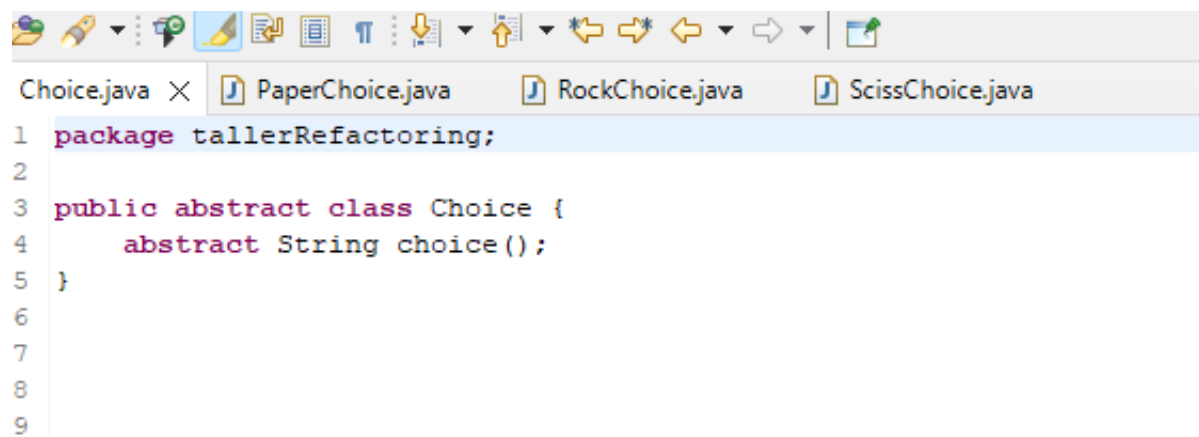
Con esta técnica se puede reemplazar las elecciones (Rock, Scissors y Paper) en subclases de forma que cada una tenga la manera de establecer una elección para el jugador (Player).

Refactorización

Reemplazar switch con polimorfismo línea 66 a 81

A continuación, se realiza la implementación con la técnica de refactorización descrita previamente.

Se crea una clase abstracta de la que heredaran las distintas elecciones de jugada.



```
1 package tallerRefactoring;
2
3 public abstract class Choice {
4     abstract String choice();
5 }
6
7
8
9
```

Las clases que heredan de Choice:

```
Choice.java PaperChoice.java X RockChoice.java ScissChoice.java
1 package tallerRefactoring;
2
3 public class PaperChoice extends Choice{
4
5     @Override
6     String choice() {
7         // TODO Auto-generated method stub
8         return "paper";
9     }
10
11 }
12
```

```
Choice.java PaperChoice.java RockChoice.java X ScissChoice.java
1 package tallerRefactoring;
2
3 public class RockChoice extends Choice{
4
5     @Override
6     String choice() {
7         // TODO Auto-generated method stub
8         return "rock";
9     }
10
11 }
12
```

```
Choice.java PaperChoice.java RockChoice.java ScissChoice.java X
1 package tallerRefactoring;
2
3 public class ScissChoice extends Choice{
4
5     @Override
6     String choice() {
7         // TODO Auto-generated method stub
8         return "scissors";
9     }
10 }
11
```

Extraer metodo

Se puede extraer el código del método playerChoice() en un método separado para mejorar la legibilidad y reutilización del código.

```
public class Player {
    private static final List<Choice> eleccion = new ArrayList<>();
    int wins;

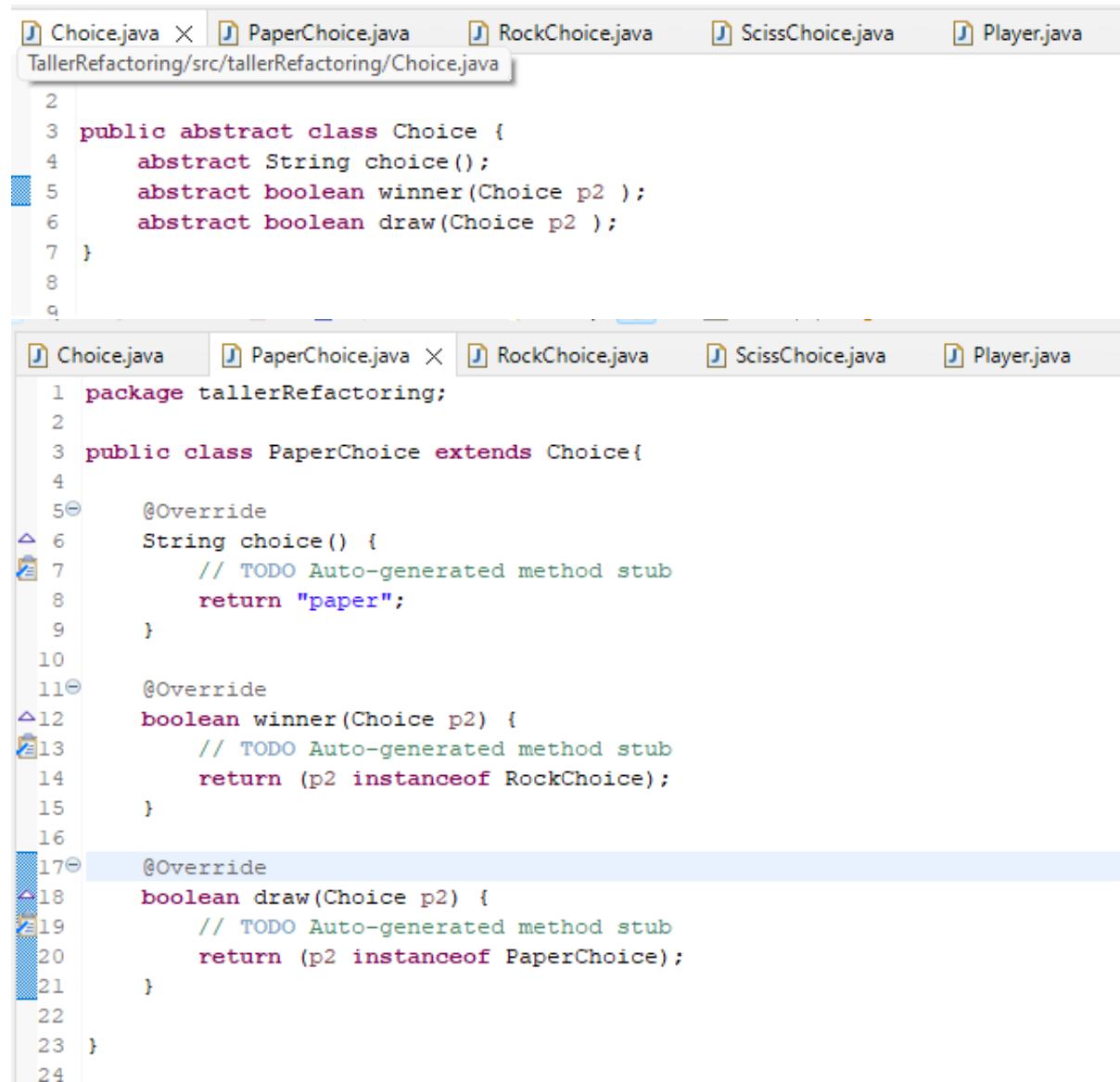
    public Player() {
        eleccion.add(new RockChoice());
        eleccion.add(new PaperChoice());
        eleccion.add(new ScissChoice());
    }

    public Choice playerChoice() {
        int op = (int) (Math.random()*3);
        return eleccion.get(op);
    }
}
```

Con esta implementación se reduce considerablemente el código y la implementación del método.

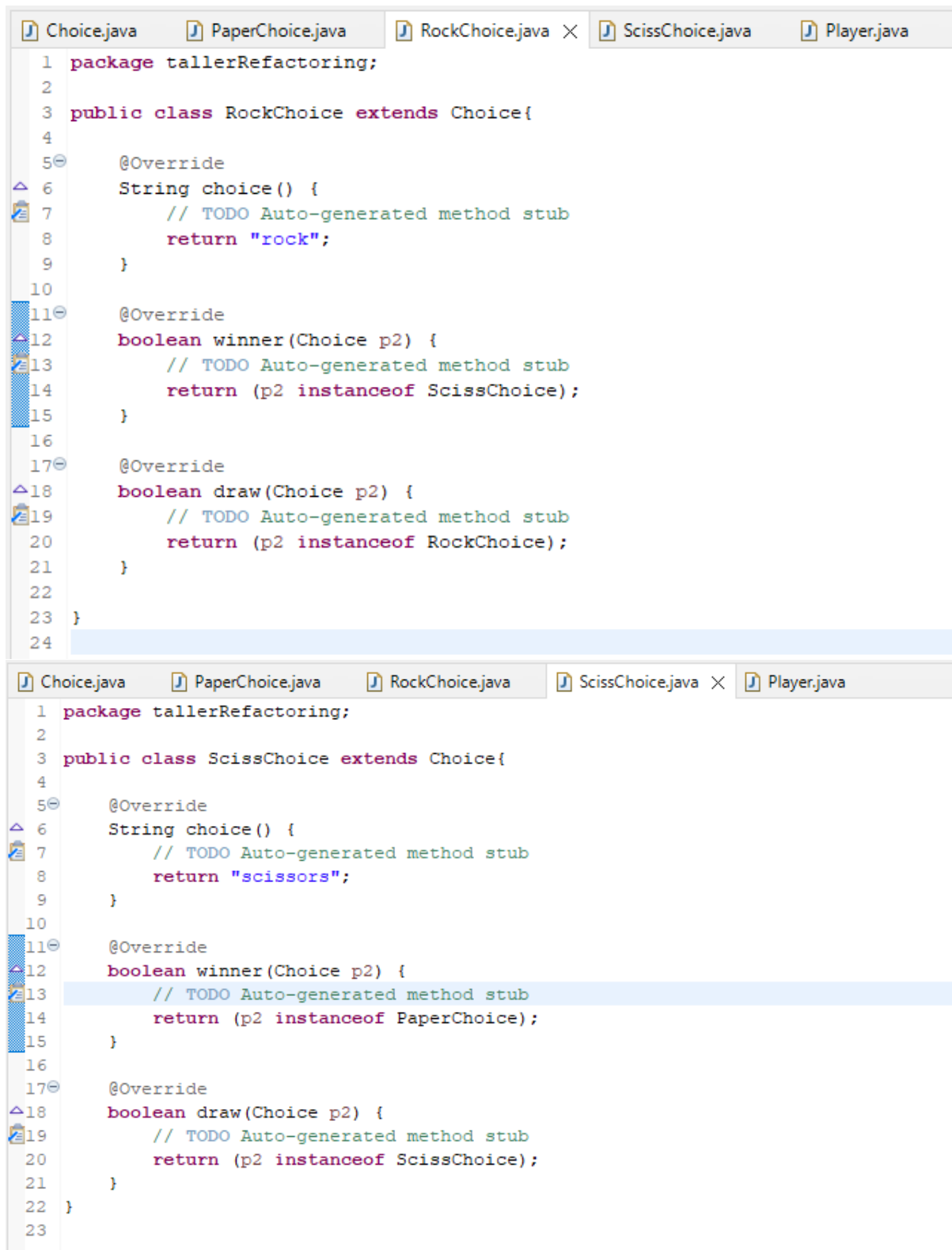
Reemplazar switch con polimorfismo

Porque para las diferentes opciones se establece una clase que contenga métodos específicos los cuales determinarán si ambas opciones son iguales o distintas y poder determinar quién es el jugador de cada partida.



```
Choice.java
2
3 public abstract class Choice {
4     abstract String choice();
5     abstract boolean winner(Choice p2 );
6     abstract boolean draw(Choice p2 );
7 }
8
9

PaperChoice.java
1 package tallerRefactoring;
2
3 public class PaperChoice extends Choice{
4
5     @Override
6     String choice() {
7         // TODO Auto-generated method stub
8         return "paper";
9     }
10
11     @Override
12     boolean winner(Choice p2) {
13         // TODO Auto-generated method stub
14         return (p2 instanceof RockChoice);
15     }
16
17     @Override
18     boolean draw(Choice p2) {
19         // TODO Auto-generated method stub
20         return (p2 instanceof PaperChoice);
21     }
22
23 }
24
```



```
1 package tallerRefactoring;
2
3 public class RockChoice extends Choice{
4
5     @Override
6     String choice() {
7         // TODO Auto-generated method stub
8         return "rock";
9     }
10
11     @Override
12     boolean winner(Choice p2) {
13         // TODO Auto-generated method stub
14         return (p2 instanceof ScissChoice);
15     }
16
17     @Override
18     boolean draw(Choice p2) {
19         // TODO Auto-generated method stub
20         return (p2 instanceof RockChoice);
21     }
22 }
23
24
```

```
1 package tallerRefactoring;
2
3 public class ScissChoice extends Choice{
4
5     @Override
6     String choice() {
7         // TODO Auto-generated method stub
8         return "scissors";
9     }
10
11     @Override
12     boolean winner(Choice p2) {
13         // TODO Auto-generated method stub
14         return (p2 instanceof PaperChoice);
15     }
16
17     @Override
18     boolean draw(Choice p2) {
19         // TODO Auto-generated method stub
20         return (p2 instanceof ScissChoice);
21     }
22 }
23
```

El mal olor: Temporary field, se va a refactorizar con la técnica Extract Class.

Esta técnica además de permitirnos que la clase Player siga el principio SRP, dejando de lado el uso de una variable temporal que se usaba en el único caso específico de ganar, pero este método era mejor que estuviera separado de mi clase jugador.

```

/**
 *
 * @author melis
 */
public class Play {
    private Player jugador1;
    private Player jugador2;

    public void jugar() {
        Player ganador = null;

        //Logica del juego

        ganadorPuntos(ganador);
    }

    public void ganadorPuntos(Player jugador) {
        jugador.sumarJugada();
    }

}

class Player {
    private int wins;

    public void sumarJugada() {
        this.wins++;
    }

}

```

Se creará otra clase la cual tendrá métodos solo para mostrar en pantalla distinta información. Gracias a esto se aplica la técnica “Extraer métodos” porque son fragmentos de código que puedan agruparse. Así mismo, para acceder a los atributos se hace uso de los métodos getter, para que no se pueda acceder directamente y modificarlos.

```

package tallerRefactoring;

public class Round {
    public static void printRounds(int roundplayed, int draw){
        System.out.println("**** Round ****"+roundplayed+ "*****");
        System.out.println("Number of draws:" + draw+ "*****");
    }

    public static void playersInfo(Player p1, Player p2){
        System.out.println("Player 1: " + p1.getOpcion()+"Player 1 Total Wins: " + p1.getWins());
        System.out.println("Player 2: " + p2.getOpcion()+"Player 2 Total Wins: " + p2.getWins());
    }
}

```

Con la implementación de los métodos “winner” e “draw” en las opciones de juego, resulta posible aplicar la técnica conocida como Reemplazar condicionales con Polimorfismo en la clase RPMGame. Sin embargo, aún existirán sentencias condicionales. Con la diferencia que estas no aumentarán a largo plazo, porque siempre en un juego ambos competidores “empatan” o “ganan”.

```

package tallerRefactoring;

public class RPMGame {
    public static void main(String[] args) {
        Player p1 = new Player();
        Player p2 = new Player();
        boolean gameWon = false;
        int roundsPlayed = 0;
        int draw = 0;

        do {

            Choice p1Choice = p1.playerChoice();
            String eleccionp1 = p1Choice.choice();
            p1.setOpcion(eleccionp1);

            Choice p2Choice = p2.playerChoice();
            String eleccionp2 = p2Choice.choice();
            p2.setOpcion(eleccionp2);

            Round.printRounds(roundsPlayed, draw);
            Round.playersInfo(p1, p2);

            if(p1Choice.draw(p2Choice)) {
                draw++;
                System.out.print("\n\t\t\t Draw \n");
            }else {
                if(p1Choice.winner(p2Choice)) {
                    p1.setWins();
                    System.out.print("Player 1 Wins");
                }else{
                    p2.setWins();
                    System.out.print("Player 2 Wins");
                }
            }
        }
    }
}

```

```

Choice p1Choice = p1.playerChoice();
String eleccionp1 = p1Choice.choice();
p1.setOpcion(eleccionp1);

Choice p2Choice = p2.playerChoice();
String eleccionp2 = p2Choice.choice();
p1.setOpcion(eleccionp2);

Round.printRounds(roundsPlayed, draw);
Round.playersInfo(p1, p2);

if(p1Choice.draw(p2Choice)) {
    draw++;
    System.out.print("\n\t\t\t Draw \n");
}else {
    if(p1Choice.winner(p2Choice)) {
        p1.setWins();
        System.out.print("Player 1 Wins");
    }else{
        p2.setWins();
        System.out.print("Player 2 Wins");
    }
}
roundsPlayed++;
if((p1.getWins()>=3) || (p2.getWins()>=3)){
    gameWon=true;
    System.out.print("\n\t\t\t Game WON\n");
}
System.out.println();
}while(gameWon!=true);

    }
}

```