Capgemini

TDD Switch Easy

Coding Dojo

Kata Rock Paper Scissors Lizard Spock

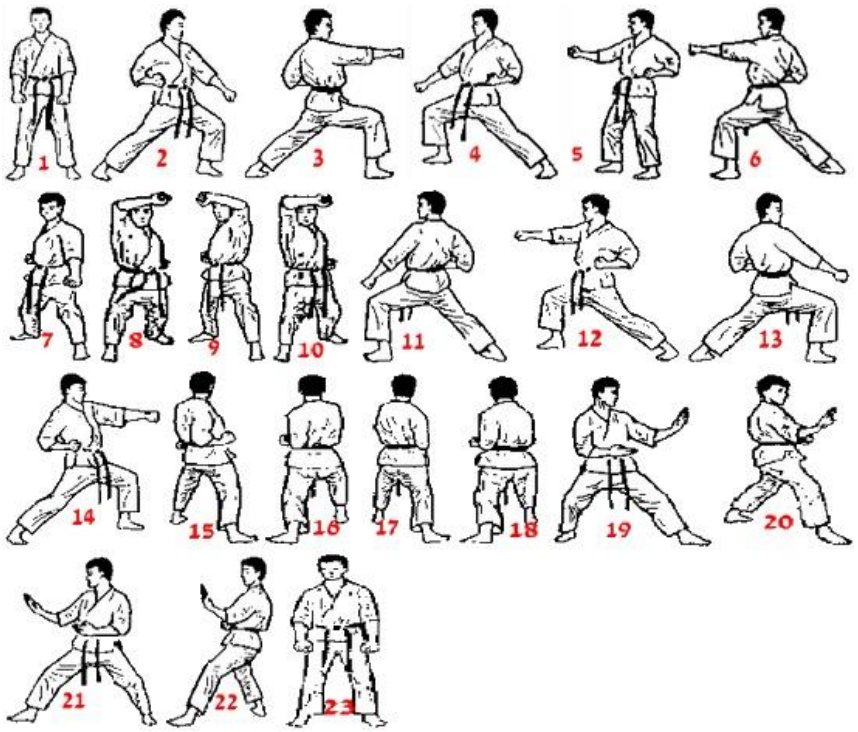
Antes de empezar (¿Qué es coding dojo?)

Dojo es el lugar donde se reúne la gente para practicar y entrenar artes marciales.

CODING DOJO

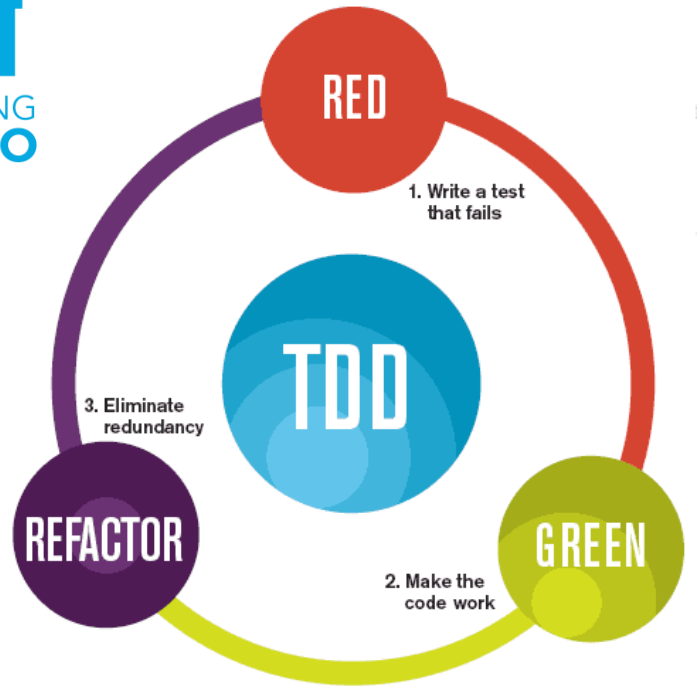
Lugar donde nos reunimos para practicar y entrenar buenas formas de programación.

Antes de empezar (¿Qué es coding dojo?)



Ejecutan Katas para aprender los movimientos y las técnicas.

CODING DOJO



Usamos TDD para el desarrollo

Ejecución por consenso

Antes de empezar (¿Qué es Refactor?)

"Cambios en el código para hacerlo más fácil de entender y más barato de modificar, sin alterar su comportamiento observable"

- Martin Fowler -

Antes de empezar (¿Cómo sabemos que está “mal escrito”?)

Basic smells

COMMENTS

MAGIC NUMBER

LONG METHOD

DUPLICATE METHOD

LARGE CLASS

LONG PARAMETER LIST

Refactor Actions

EXTRACT CLASS

INTRODUCE EXPLAINING VARIABLE

EXTRACT METHOD

SELF ENCAPSULATE FIELD

INLINE METHODS

ENCAPSULATE COLLECTIONS

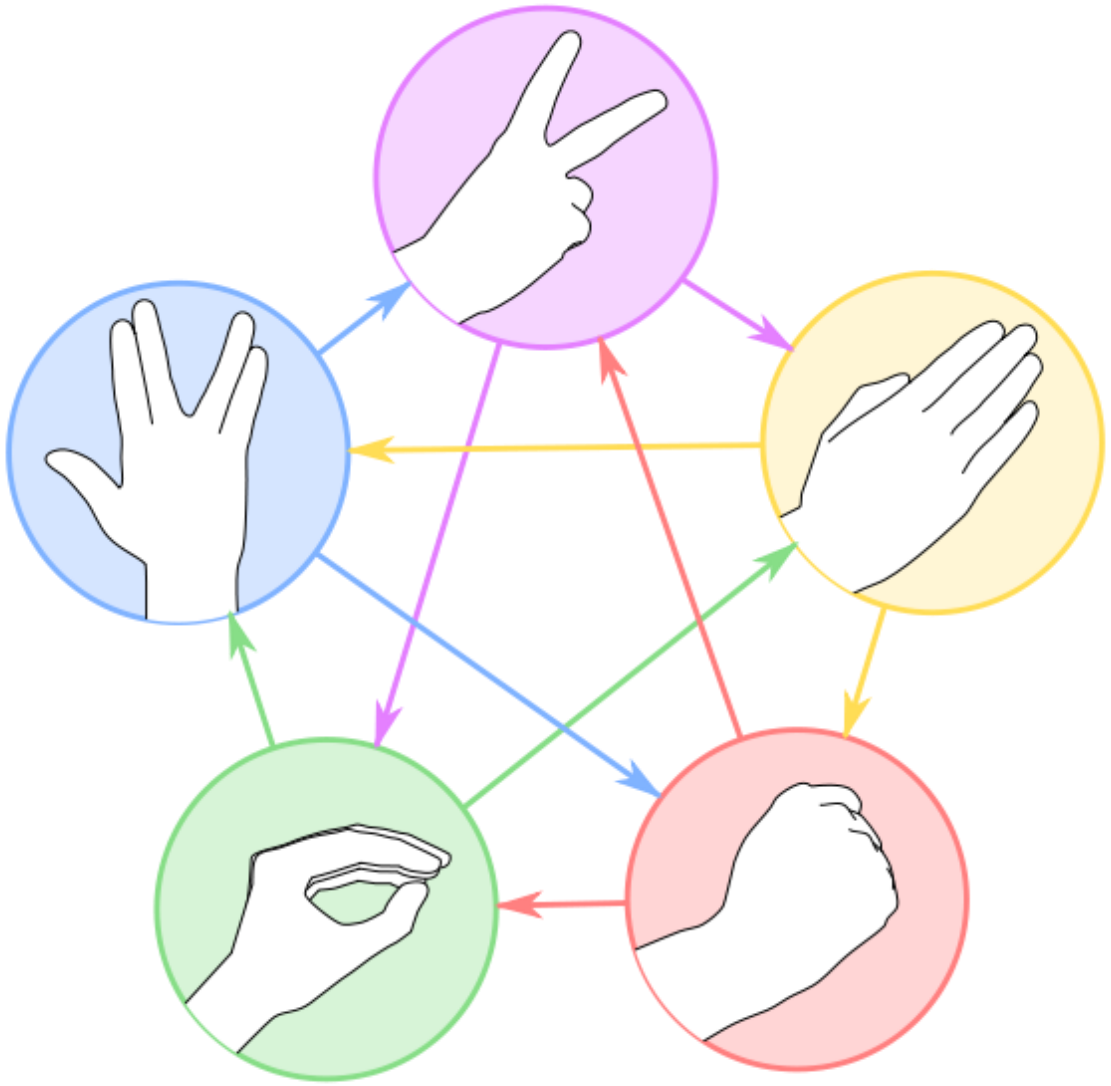
https://refactoring.guru/refactoring/smells

Práctica

Kata Rock Paper Scissors Lizard Spock

Rock Paper Scissors Lizard Spock (Enunciado)

Rock Paper Scissors Lizard Spock (Enunciado)



Reglas

Scissors cuts Paper
Paper covers Rock
Rock crushes Lizard
Lizard poisons Spock
Spock smashes Scissors

Scissors decapitates Lizard
Lizard eats paper
Paper disproves Spock
Spock vaporizes Rock
Rock crushes Scissors

It is always a draw because everyone choose "Spock"

https://www.youtube.com/watch?v=iapcKVn7DdY

Rock Paper Scissors Lizard Spock (Enunciado)

Consejos:

- Intentaremos no leer el enunciado completo.
  - Iremos leyendo poco a poco.
- Haremos solo una tarea a la vez.
  - Tenemos que aprender a trabajar de forma incremental.
- Para esta kata solo vamos a testear las entradas correctas.
  - Por agilidad en la session.
- Llegaremos hasta donde nos de tiempo.
  - No se trata de terminarlo, sino de aprender durante el proceso.
- Programaremos entre todos.
  - No existe una solución correcta y una incorrecta, se puede hacer de muchas formas.

No te olvides de pasar los tests a cada cambio que se efectue del código, para verificar que funciona todo correctamente y que no se ha roto nada más.

A refactorizar!

Kata Rock Paper Scissors Lizard Spock

Code Smell

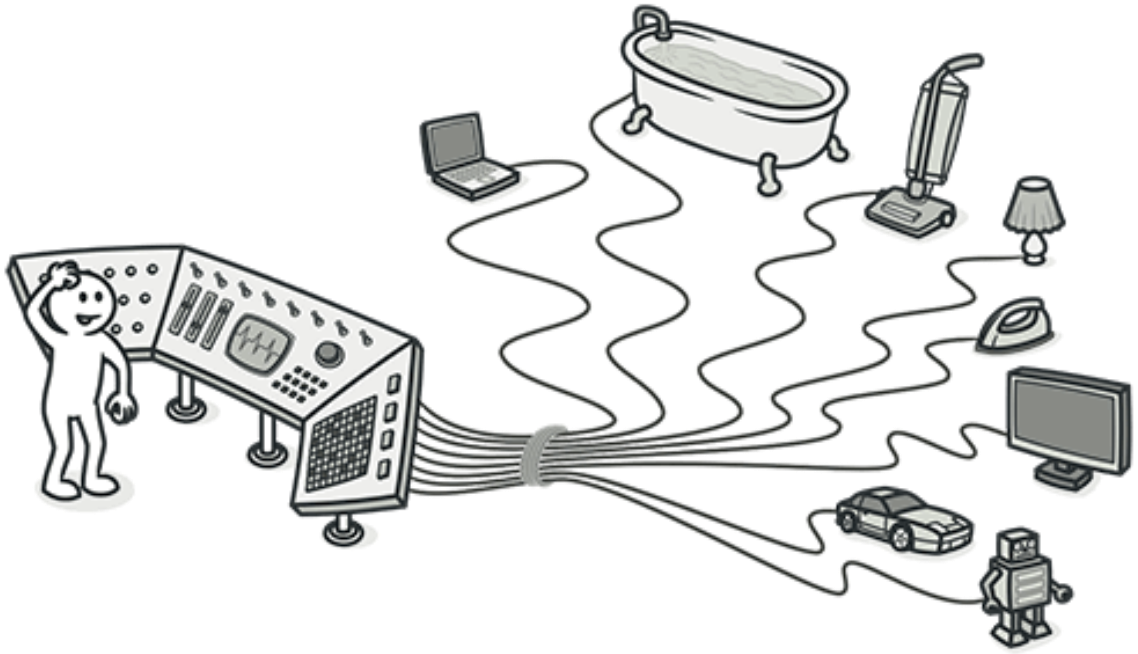
Switch Statement

Casi todos nuestros algoritmos, tarde o temprano, van a hacer uso de operadores switch / if.

Problema

El problema que nos puede traer este operador es doble:

- Dificulta la lectura del código ya que obliga al desarrollador a memorizar el stack de las clausulas anteriores.
- Suele ser frecuente que si tenemos un operador switch, ese operador esté repartido en más sitios del código y por tanto, a la hora de modificar una condición, deberemos acordarnos de modificarlo varias veces.



https://refactoring.guru/es/smells/switch-statements

Extract Method | Extract Variable | Magic Number

Tip #1. Decompose Conditional (Semantic)

Se da cuando vemos un condicional que es complejo o tiene muchas condiciones aunque sean sencillas.

Problema

Mientras estás ocupado averiguando lo que hace el código de dentro del bloque, olvidas cual era la condición. Y mientras estás ocupado averiguando cual era la condición, olvidas lo que hace el código de dentro del bloque.

Obliga al cerebro a releer varias veces el mismo código hasta asimilarlo por completo. Por ejemplo:

```
if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
  charge = quantity * winterRate + winterServiceCharge;
}
else {
  charge = quantity * summerRate;
}
```

Solución

Una posible solución es descomponer las partes complicadas del condicional y separarlo en métodos diferentes: condición, then, else.

Al extraer el código a métodos correctamente nombrados, le estamos dando un valor semántico a las condiciones y por tanto facilitando su lectura.

```
if (isNotSummer(date)) {
  charge = quantity * winterRate + winterServiceCharge;
}
else {
  charge = quantity * summerRate;
}
```

Extract Method | Extract Variable | Magic Number

Tip #1. Decompose Conditional (Semantic)

Se da cuando vemos un condicional que es complejo o tiene muchas condiciones aunque sean sencillas.

```
if (moveOne.equals(PlayMove.SCISSORS) && (moveTwo.equals(PlayMove.PAPER) || moveTwo.equals(PlayMove.LIZARD)))
  return true;

if (moveOne.equals(PlayMove.PAPER) && (moveTwo.equals(PlayMove.ROCK) || moveTwo.equals(PlayMove.SPOCK)))
  return true;

if (moveOne.equals(PlayMove.ROCK) && (moveTwo.equals(PlayMove.LIZARD) || moveTwo.equals(PlayMove.SCISSORS)))
  return true;
```

Mient del
dentr en,
ocupa
códig

Al extraer el código a métodos correctamente nombrados, le

Obliga al cerebro a iones y por tanto
asimilarlo por com

```
if (moveOne.equals(PlayMove.SCISSORS) && isScissorsWin(moveTwo))
  return true;

if (moveOne.equals(PlayMove.PAPER) && isPaperWin(moveTwo))
  return true;

if (moveOne.equals(PlayMove.ROCK) && isRockWin(moveTwo))
  return true;
```

Add return Add throws

Tip #2. Replace with Guard Clauses

Se da cuando vemos condicionales anidados que parecen simples y siguen un flujo de respuesta en bloque.

Problema

El problema que nos aparece es que al estar anidados, es muy difícil seguir el flujo normal de ejecución.

Obliga al cerebro a tener que volver atrás muchas veces, para leer condiciones e identaciones de las clausulas. Por ejemplo:

```
public double getPayAmount() {
  double result;

  if (isDead) {
    result = deadAmount();
  }
  else {
    if (isSeparated) {
      result = separatedAmount();
    }
    else {
      if (isRetired) {
        result = retiredAmount();
      }
      else {
        result = normalPayAmount();
      }
    }
  }

  return result;
}
```

Solución

Hay que identificar cuales son las clausulas que conducen a un punto terminal (generalmente con una excepción o una devolución inmediata del valor).

Una vez identificadas, se debe reorganizar el código con el objetivo de *aplanar* la estructura de código, e introducir un punto de return en las clausulas de guarda.

```
public double getPayAmount() {

  if (isDead) {
    return deadAmount();
  }

  if (isSeparated) {
    return separatedAmount();
  }

  if (isRetired) {
    return retiredAmount();
  }

  return normalPayAmount();
}
```

Tip #2. Replace with Guard Clauses

Add return Add throws

```
private boolean isPlayWinner(PlayMove moveOne, PlayMove moveTwo) {

  if (moveOne.equals(PlayMove.SCISSORS) && isScissorsWin(moveTwo))
    return true;

  if (moveOne.equals(PlayMove.PAPER) && isPaperWin(moveTwo))
    return true;

  if (moveOne.equals(PlayMove.ROCK) && isRockWin(moveTwo))
    return true;

  if (moveOne.equals(PlayMove.LIZARD) && isLizardWin(moveTwo))
    return true;

  if (moveOne.equals(PlayMove.SPOCK) && isSpockWin(moveTwo))
    return true;

  return false;

}
```

Maps Lists Functions

Tip #3. Substitute Algorithm

Se da cuando hay una algoritmia clara o una correlación entre la condición y lo que hace el bloque de código.

Problema

Las condiciones ya nos están diciendo claramente lo que va a hacer el código. Hay una especie de duplicidad innecesaria.

```
public String foundPerson(String[] people) {

  for (int i = 0; i < people.length; i++) {
    if (people[i].equals("Don")) {
      return "Don";
    }
    if (people[i].equals("John")) {
      return "John";
    }
    if (people[i].equals("Kent")) {
      return "Kent";
    }
  }

  return "";
}
```

Solución

Tal vez existe un algoritmo mucho más simple y eficiente que pueda realizar esa misma funcionalidad.

A veces también se puede sustituir por List, Map o por librerías o frameworks que ya realizan esa funcionalidad.

```
private List candidates = Arrays.asList(
      new String[] {"Don", "John", "Kent"});

public String foundPerson(String[] people) {

  for (int i = 0; i < people.length; i++) {
    if (candidates.contains(people[i])) {
      return people[i];
    }
  }

  return "";
}
```

Maps Lists Functions

Tip #3. Substitute Algorithm

```
Map<PlayMove, List<PlayMove>> WIN_MOVES = new HashMap<PlayMove, List<PlayMove>>() {
  {
    put(PlayMove.SCISSORS, Arrays.asList(new PlayMove[] { PlayMove.PAPER,    PlayMove.LIZARD }));
    put(PlayMove.PAPER,    Arrays.asList(new PlayMove[] { PlayMove.ROCK,     PlayMove.SPOCK }));
    put(PlayMove.ROCK,     Arrays.asList(new PlayMove[] { PlayMove.LIZARD,   PlayMove.SCISSORS }));
    put(PlayMove.LIZARD,   Arrays.asList(new PlayMove[] { PlayMove.SPOCK,    PlayMove.PAPER }));
    put(PlayMove.SPOCK,    Arrays.asList(new PlayMove[] { PlayMove.SCISSORS, PlayMove.ROCK }));
  }
};

private boolean isPlayWinner(PlayMove moveOne, PlayMove moveTwo) {

  return this.WIN_MOVES.get(moveOne).contains(moveTwo);

}
```

State Pattern Strategy Pattern

Tip #4. Replace with Polymorphism

Cuando tenemos que cada una de las condiciones realizan una acción diferente dependiendo del tipo de condición. Además, puede ser que esta misma estructura la tengamos varias veces repetidas en el código.

Problema

Si aparecen nuevos tipos de condiciones debemos crear nuevas acciones y recordar que debemos modificar todas las estructuras condionales que tengamos iguales.

```
class Bird {

  public double getSpeed() {
    switch (type) {
      case EUROPEAN: return getBaseSpeed();
      case AFRICAN: return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
      case NORWEGIAN_BLUE: return (isNailed) ? 0 : getBaseSpeed(voltage);
    }

    throw new RuntimeException("Should be unreachable");
  }

  public String getColor() {
    switch (type) {
      case EUROPEAN: return BROWNISH;
      case AFRICAN: return ELECTRIC_YELLOW;
      case NORWEGIAN_BLUE: return BLUE;
    }

    throw new RuntimeException("Should be unreachable");
  }

  public double getSize() {
    ...
  }

}
```

Solución

Utiliando la herencia se pueden crear subclases que implementen sus propio comportamiento, mientras que en la clase base implementaremos el comportamiento estandar. Se puede implementar el State Pattern o el Strategy Pattern.

Hay que crear una clase base con el caso general y luego una subclase con cada una de las condiciones del condicional. Luego hay que implementar los métodos necesarios en cada subclase y mover el código del condicional a la subclase.

Si aparecen nuevos tipos solo hay que implementar una nueva subclase con su comportamiento.

State Pattern Strategy Pattern

Tip #4. Replace with Polymorphism (II)

Problema

```
class Bird {

  public double getSpeed() {
    switch (type) {
      case EUROPEAN: return getBaseSpeed();
      case AFRICAN: return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
      case NORWEGIAN_BLUE: return (isNailed) ? 0 : getBaseSpeed(voltage);
    }

    throw new RuntimeException("Should be unreachable");
  }

  public String getColor() {
    switch (type) {
      case EUROPEAN: return BROWNISH;
      case AFRICAN: return ELECTRIC_YELLOW;
      case NORWEGIAN_BLUE: return BLUE;
    }

    throw new RuntimeException("Should be unreachable");
  }

  public double getSize() {
    ...
  }

}
```

Solución

```
class Bird {

  public double getSpeed() {

    throw new RuntimeException("Should be unreachable");
  }

  public String getColor() {

    throw new RuntimeException("Should be unreachable");
  }

  public double getSize() {
    ...
  }
}
```

```
class NorwegianBlue extends Bird {

 public double getSpeed() {

    return getBaseSpeed();
  }

  public String getColor() {

    return (isNailed) ? 0 : getBaseSpeed(voltage);
  }

  public double getSize() {
    ...
  }
}
```

```
class European extends Bird {

 public double getSpeed() {

    return getBaseSpeed();
  }

  public String getColor() {

    return BROWNISH;
  }

  public double getSize() {
    ...
  }
}
```

```
class African extends Bird {

 public double getSpeed() {

    return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
  }

  public String getColor() {

    return ELECTRIC_YELLOW;
  }

  public double getSize() {
    ...
  }
}
```

State Pattern Strategy Pattern

Tip #4. Replace with Polymorphism (II)

```
public abstract class PlayMove {
  public static PlayMove ROCK = new MoveRock();

  public static PlayMove PAPER = new MovePaper();

  public static PlayMove SCISSORS = new MoveScissors();

  public static PlayMove LIZARD = new MoveLizard();

  public static PlayMove SPOCK = new MoveSpock();

  abstract boolean isPlayWinner(PlayMove move);

}
```

```
public class MoveRock extends PlayMove {

  @Override
  boolean isPlayWinner(PlayMove move) {

    return move.equals(PlayMove.LIZARD) || move.equals(PlayMove.SCISSORS);
  }

}
```

```
public class MovePaper extends PlayMove {

  @Override
  boolean isPlayWinner(PlayMove move) {

    return move.equals(PlayMove.ROCK) || move.equals(PlayMove.SPOCK);
  }

}
```

```
public class RuleEngine {
  ...

  private boolean isPlayWinner(PlayMove moveOne, PlayMove moveTwo) {

    return moveOne.isPlayWinner(moveTwo);

  }

  ...
}
```

```
public class MoveScissors extends PlayMove {

  @Override
  boolean isPlayWinner(PlayMove move) {

    return move.equals(PlayMove.PAPER) || move.equals(PlayMove.LIZARD);
  }

}
```

```
public class MoveLizard extends PlayMove {

  @Override
  boolean isPlayWinner(PlayMove move) {

    return move.equals(PlayMove.SPOCK) || move.equals(PlayMove.PAPER);
  }

}
```

```
public class MoveSpock extends PlayMove {

  @Override
  boolean isPlayWinner(PlayMove move) {

    return move.equals(PlayMove.SCISSORS) || move.equals(PlayMove.ROCK);
  }

}
```

State Pattern Strategy Pattern

Tip #4. Replace with Polymorphism (II)

```
public enum PlayMove {

  ROCK {
    @Override
    boolean isPlayWinner(PlayMove move) {

      return move.equals(PlayMove.LIZARD) || move.equals(PlayMove.SCISSORS);
    }

  },

  PAPER {
    @Override
    boolean isPlayWinner(PlayMove move) {

      return move.equals(PlayMove.ROCK) || move.equals(PlayMove.SPOCK);
    }

  },

  SCISSORS {
    @Override
    boolean isPlayWinner(PlayMove move) {

      return move.equals(PlayMove.PAPER) || move.equals(PlayMove.LIZARD);
    }

  },

  LIZARD {
    @Override
    boolean isPlayWinner(PlayMove move) {

      return move.equals(PlayMove.SPOCK) || move.equals(PlayMove.PAPER);
    }

  },

  SPOCK {
    @Override
    boolean isPlayWinner(PlayMove move) {

      return move.equals(PlayMove.SCISSORS) || move.equals(PlayMove.ROCK);
    }
  };

  abstract boolean isPlayWinner(PlayMove move);
}
```

```
public class RuleEngine {
  ...

  private boolean isPlayWinner(PlayMove moveOne, PlayMove moveTwo) {

    return moveOne.isPlayWinner(moveTwo);

  }

  ...
}
```

Code Smell

Switch Statement

Resultado

Hemos conseguido tener el código más organizado, más legible y por tanto más fácil de entender y modificar.
Además, es muy probable que hayamos disminuido el código duplicado.

Excepciones

- Cuando un switch realiza acciones muy simples y no se repite en el código, no es necesario refactorizar en profundidad.
- A menudo, los operadores switch se utilizan en los patrones 'Factory Method' o 'Abstract Factory', para seleccionar la clase a crear.

https://refactoring.guru/es/smells/switch-statements

People matter, results count.

About Capgemini

With more than 190,000 people, Capgemini is present in over 40 countries and celebrates its 50th Anniversary year in 2017. A global leader in consulting, technology and outsourcing services, the Group reported 2016 global revenues of EUR 12.5 billion. Together with its clients, Capgemini creates and delivers business, technology and digital solutions that fit their needs, enabling them to achieve innovation and competitiveness. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model.

Learn more about us at

www.capgemini.com