devonfw

Coding Dojo – Sesion 2

Octubre 2017

Capgemini

CONSULTING.TECHNOLOGY.OUTSOURCING

Aclaraciones

Aclaraciones sobre la anterior sesión

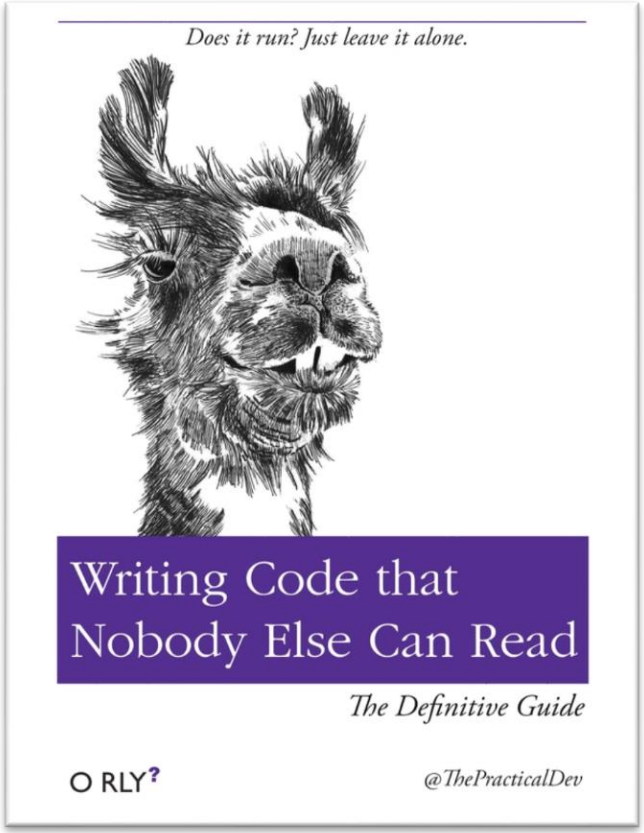
Lo que se explica está exagerado:

- Switch case
- Comentarios en código
- Métodos largos

Aclaraciones sobre la anterior sesión

Lo que se explica está exagerado:

- Switch case
- Comentarios en código
- Métodos largos



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

- Martin Fowler -

Aclaraciones sobre la anterior sesión

Lo que se explica está exagerado:

- Switch case
- Comentarios en código
- Métodos largos

No hace falta escribir tantos jUnits:

- En la kata se usan para fines educativos
- Deberíamos llegar a un compromiso

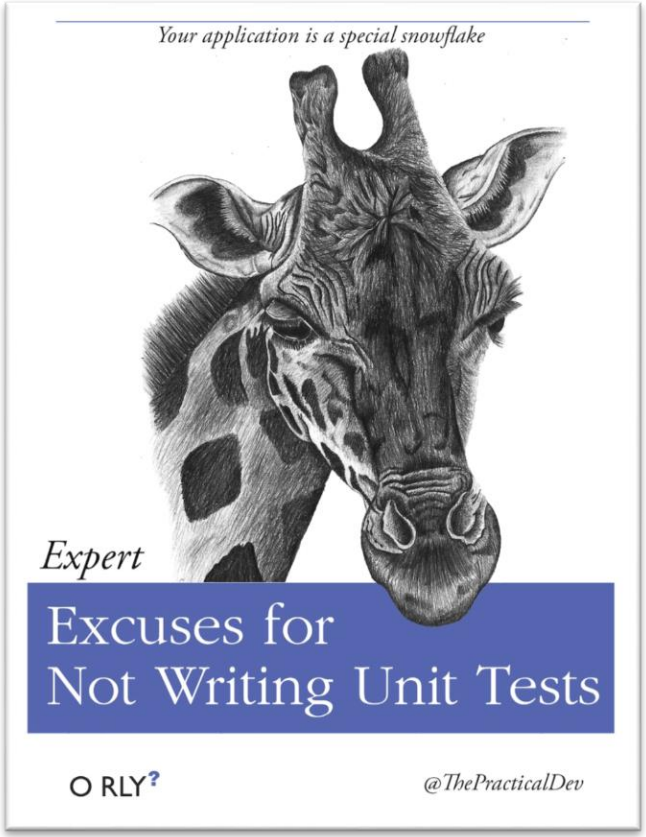
Aclaraciones sobre la anterior sesión

Lo que se explica está exagerado:

- Switch case
- Comentarios en código
- Métodos largos

No hace falta escribir tantos jUnits:

- En la kata se usan para fines educativos
- Deberíamos llegar a un compromiso



Práctica

Kata: Rock Paper Scissors Lizard Spock

Rock Paper Scissors Lizard Spock (Consejos)

Antes de comenzar

Haremos sólo una tarea a la vez.

El truco está en aprender a trabajar de forma incremental.

Nos aseguraremos de testear únicamente las entradas correctas.

No es necesario testear las entradas incorrectas para esta kata.

Llegaremos hasta donde nos de tiempo.

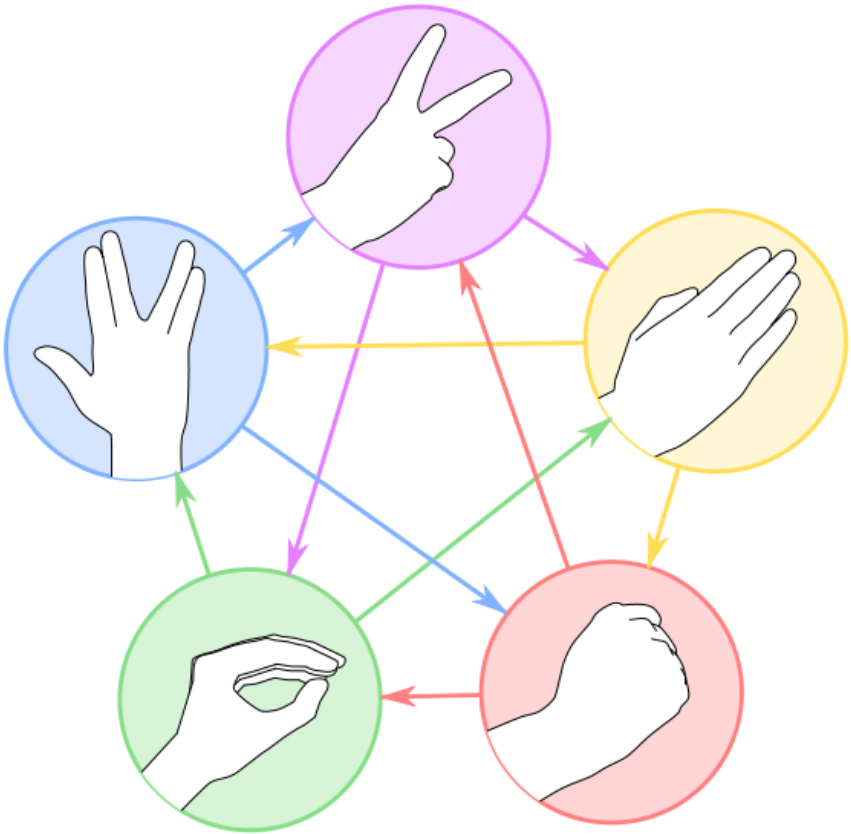
No se trata de terminar la Kata entera sino en aprender durante el proceso.

Aprendizajes que intentaremos conseguir con esta Kata:

Adaptabilidad ante los cambios.
Uso correcto del TDD.
Aprender a hacer refactoring.
Refactoring de switch statement.

Rock Paper Scissors Lizard Spock (Reglas)

Rock Paper Scissors Lizard Spock (Reglas)



Tests

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

Switch Statments

Refactoring

Code Smell

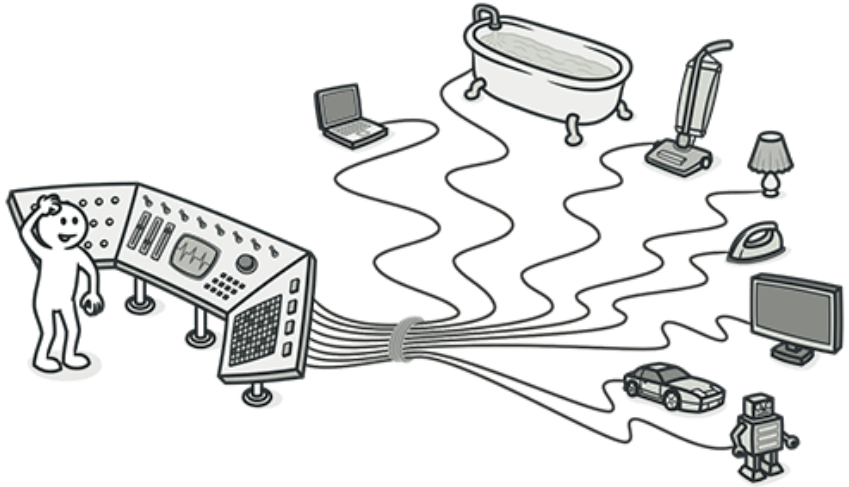
Switch Statement

El uso de operadores switch es uno de los sellos distintivos de la programación orientada a objetos.

El problema que se presenta con este tipo de código es doble.

Por un lado dificulta la lectura del código, en caso de que existan muchas clausulas switch, ya que obliga al desarrollador a memorizar el stack de las clausulas anteriores.

Por otro lado, es muy 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.



Tip #1. Decompose Conditional (Semantic)

(Extract Method, Extract Variable, Magic Number)

Problema

Existe un condicional complejo con muchas condiciones difíciles de entender y retener.

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

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

Solución

Descomponer las partes complicadas del condicional y separarlo en métodos diferentes: condición, then y else.

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

```
if (notSummer(date)) {
  charge = winterCharge(quantity);
}
else {
  charge = summerCharge(quantity);
}
```

Tip #2. Replace Conditional with Guard Clauses

Problema

Existen muchos condicionales anidados y es dificil seguir el flujo normal de ejecución.

```
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

Identificar las cláusulas de guarda que conducen a un punto terminal (excepción o devolución inmediata de valor).

Reorganizar el código con el objetivo de “aplanar” la estructura de código.

↓ Se introduce más de un punto de “return”, pero se consigue un código más legible y más fácil de entender y modificar.

```
public double getPayAmount() {
  if (isDead){
    return deadAmount();
  }
  if (isSeparated){
    return separatedAmount();
  }
  if (isRetired){
    return retiredAmount();
  }
  return normalPayAmount();
}
```

Tip #3. Substitute Algorithm

(Maps, Lists, Functions)

Problema

Existe una algoritmia clara entre la condición y lo que hace el bloque de código.

```
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 que es mucho más simple y eficiente. En ese caso simplemente hay que reemplazar el algoritmo viejo por el nuevo.

A veces también, ese algoritmo es incorporado en alguna librería o framework y se puede reemplazar el código viejo directamente.

```
String foundPerson(String[] people){
  List candidates =
    Arrays.asList(new String[] {"Don", "John", "Kent"});
  for (int i=0; i < people.length; i++) {
    if (candidates.contains(people[i])) {
      return people[i];
    }
  }
  return "";
}
```

Tip #4. Replace Conditional with Polymorphism

(State Pattern, Strategy Pattern)

Problema

Existe una condición que realiza una acción diferente dependiendo del tipo de objeto o de las propiedades. Se trata de un switch 'selector'.

Si aparece una nueva propiedad o tipo de objeto, se debe buscar y cambiar en cada una de las apariciones de este 'selector.

```
class Bird {
  //...
  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");
  }
}
```

Solución

Crear subclases que coincidan con las ramas del condicional.

En cada clase crear un método compartido (por interface) y mover el código de la rama correspondiente al nuevo método.

Si aparece una nueva propiedad o tipo de objeto, simplemente añada una nueva subclase con su implementación.

```
abstract class Bird {
  //...
  abstract double getSpeed();
}
```

```
// Somewhere in client code
speed = bird.getSpeed();
```

```
class NorwegianBlue extends Bird {
  double getSpeed() {
    return (isNailed) ? 0 : getBaseSpeed(voltage);
  }
}
```

```
class European extends Bird {
  double getSpeed() {
    return getBaseSpeed();
  }
}
```

```
class African extends Bird {
  double getSpeed() {
    return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
  }
}
```

Code Smell

Switch Statement

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.

devonfw