Extreme Programming Tour

刘光聪

2016.10

内容

1. 提出问题
2. 编程环境
3. 面向对象
4. 函数式
5. 语义模型
6. 参考文献

提出问题

需求

老师说出 3 个特殊数，例如 3, 5, 7，让 100 个学生依次报数

1. 如果所报数字是「第一个特殊数 (3)」的倍数时说 Fizz；如果所报数字是「第二个特殊数 (5)」的倍数时说 Buzz；如果所报数字是「第三个特殊数 (7)」的倍数时说 Whizz；
2. 如果所报数字同时是「两个特殊数」的倍数，也要特殊处理。例如，如果是「第一个 (3)」和「第二个 (5)」特殊数的倍数，那么也不能说该数字，而是要说 FizzBuzz。以此类推，如果同时是三个特殊数的倍数，那么要说 FizzBuzzWhizz；
3. 如果所报数字包含了「第 1 个 (3)」特殊数时，忽略规则 1 和 2，直接说 Fizz。例如，要报 13 的同学应该说 Fizz；要报 35，它既包含 3，同时也是 5 和 7 的倍数，要说 Fizz，而不能说 BuzzWhizz；
4. 否则，直接说出要报的数字。

形式化

```
r1:
- times(3) -> Fizz
- times(5) -> Buzz
- times(7) -> Whizz
r2:
- times(3) && times(5) && times(7) -> FizzBuzzWhizz
- times(3) && times(5) -> FizzBuzz
- times(3) && times(7) -> FizzWhizz
- times(5) && times(7) -> BuzzWhizz
r3:
- contains(3) -> Fizz
- the priority of contains(3) is highest
rd:
- num -> "num"
```

编程环境

build.gradle

```
apply plugin: 'java'
apply plugin: 'groovy'

jar {
  baseName = 'fizz-buzz-whizz'
  version = '1.0.0'
}

repositories {
  mavenCentral()
}

dependencies {
  compile 'org.codehaus.groovy:groovy-all:2.4.1'
  testCompile 'org.spockframework:spock-core:1.0-groovy-2.4'
}
```

环境需求

- 测试： Spock(Groovy)
- 实现： Java8

Spock

```
import spock.lang.Specification

class RuleSpec extends Specification {
  def "should fail"() {
    expect:
    1 == 2
  }
}
```

Gradle

```
$ gradle wrapper
$ ./gradlew test
```

面向对象

第一个测试用例

```
import spock.lang.Specification

class RuleSpec extends Specification {
  def "times(3) -> Fizz"() {
    expect:
    new Times(3, "Fizz").apply(3 * 2) == "Fizz"
  }
}
```

通过测试

```java
public class Times {
  public Times(int n, String word) {
  }

  public String apply(int m) {
    return "Fizz";
  }
}
```

实现 Times

```
public class Times {
  private final int n;
  private final String word;

  public Times(int n, String word) {
    this.n = n;
    this.word = word;
  }

  public String apply(int m) {
    return m % n == 0 ? word : "";
  }
}
```

第 2 个测试用例

```groovy
import spock.lang.Specification

class RuleSpec extends Specification {
  def "contains(3) -> Fizz"() {
    expect:
    new Contains(3, "Fizz").apply(13) == "Fizz"
  }
}
```

实现 Contains

```
import static java.lang.String.valueOf;

public class Contains {
  private final int n;
  private final String word;

  public Contains(int n, String word) {
    this.n = n;
    this.word = word;
  }

  public String apply(int m) {
    return valueOf(m).contains(valueOf(m)) ? word : "";
  }
}
```

第 3 个测试用例

```
import spock.lang.Specification

class RuleSpec extends Specification {
  def "default: 2 -> str(2)"() {
    expect:
    new Default().apply(2) == "2"
  }
}
```

实现 Default

```java
public class Default {
  public String apply(int m) {
    return String.valueOf(m);
  }
}
```

提取抽象

```
@FunctionalInterface
public interface Rule {
  String apply(int n);
}
```

提取抽象: Times

```
public class Times implements Rule {
  private final int n;
  private final String word;

  public Times(int n, String word) {
    this.n = n;
    this.word = word;
  }

  @Override
  public String apply(int m) {
    return m % n == 0 ? word : "";
  }
}
```

提取抽象: Contains

```java
import static java.lang.String.valueOf;

public class Contains implements Rule {
  private final int n;
  private final String word;

  public Contains(int n, String word) {
    this.n = n;
    this.word = word;
  }

  @Override
  public String apply(int m) {
    return valueOf(m).contains(valueOf(m)) ? word : "";
  }
}
```

提取抽象：Default

```
public class Default implements Rule {
  @Override
  public String apply(int m) {
    return String.valueOf(m);
  }
}
```

第 4 个测试用例

```
import spock.lang.Specification

class RuleSpec extends Specification {
  def "times(3) && times(5) -> FizzBuzz"() {
    expect:
    new AllOf(
        new Times(3, "Fizz"),
        new Times(5, "Buzz")
    ).apply(3*5) == "FizzBuzz"
  }
}
```

实现 AllOf

```
public class AllOf implements Rule {
  private Rule[] rules;

  public AllOf(Rule... rules) {
    this.rules = rules;
  }

  @Override
  public String apply(int n) {
    StringBuilder result = new StringBuilder();
    for (Rule rule : rules) {
      result.append(rule.apply(n));
    }
    return result.toString();
  }
}
```

第 5 个测试用例

```
import spock.lang.Specification

class RuleSpec extends Specification {
  def "times(3) || times(5) -> Fizz || Buzz"() {
    expect:
    new AnyOf(
        new Times(3, "Fizz"),
        new Times(5, "Buzz"),
    ).apply(3*5) == "Fizz"

    new AnyOf(
        new Times(5, "Buzz"),
        new Times(3, "Fizz"),
    ).apply(3*5) == "Buzz"
  }
}
```

实现 AnyOf

```java
public class AnyOf implements Rule {
  private Rule[] rules;

  public AnyOf(Rule... rules) {
    this.rules = rules;
  }

  @Override
  public String apply(int n) {
    for (Rule rule : rules) {
      String result = rule.apply(n);
      if (!result.isEmpty())
        return result;
    }
    return "";
  }
}
```

引入工厂：times

```
import spock.lang.Specification

import static fizz.buzz.whizz.Rule.times

class RuleSpec extends Specification {
  def "times(3) -> Fizz"() {
    expect:
    new Times(3, "Fizz").apply(3 * 2) == "Fizz"
  }

  def "factory: times(3) -> Fizz"() {
    expect:
    times(3, "Fizz").apply(3 * 2) == "Fizz"
  }
}
```

实现工厂：Rule.times

```java
public interface Rule {
  String apply(int n);

  static Rule times(int n, String word) {
    return new Times(n, word);
  }
}
```

匿名内部类

```java
public interface Rule {
  String apply(int n);

  static Rule times(int n, String word) {
    return new Rule() {
      @Override
      public String apply(int m) {
        return m % n == 0 ? word : "";
      }
    };
  }
}
```

引入工厂：contains

```
import spock.lang.Specification

import static fizz.buzz.whizz.Rule.contains

class RuleSpec extends Specification {
  def "contains(3) -> Fizz"() {
    expect:
    new Contains(3, "Fizz").apply(13) == "Fizz"
  }

  def "factory: contains(3) -> Fizz"() {
    expect:
    contains(3, "Fizz").apply(13) == "Fizz"
  }
}
```

实现工厂：Rule.contains

```
public interface Rule {
  String apply(int n);

  static Rule contains(int n, String word) {
    return new Contains(n, word);
  }
}
```

匿名内部类

```
import static java.lang.String.valueOf;

public interface Rule {
  String apply(int n);

  static Rule contains(int n, String word) {
    return new Rule() {
      @Override
      public String apply(int m) {
        return valueOf(m).contains(valueOf(m)) ? word : "";
      }
    };
  }
}
```

引入工厂：defaults

```
import spock.lang.Specification

import static fizz.buzz.whizz.Rule.defaults

class RuleSpec extends Specification {
  def "default: 2 -> str(2)"() {
    expect:
    new Default().apply(2) == "2"
  }

  def "factory(default): 2 -> str(2)"() {
    expect:
    defaults().apply(2) == "2"
  }
}
```

实现工厂：Rule.defaults

```
public interface Rule {
  String apply(int n);

  static Rule defaults() {
    return new Default();
  }
}
```

匿名内部类

```
public interface Rule {
  String apply(int n);

  static Rule defaults() {
    return new Rule() {
      @Override
      public String apply(int m) {
        return String.valueOf(m);
      }
    };
  }
}
```

引入工厂：allof

```
import spock.lang.Specification

import static fizz.buzz.whizz.Rule.allof

class RuleSpec extends Specification {
  def "factory: times(3) && times(5) -> FizzBuzz"() {
    expect:
    allof(
        times(3, "Fizz"),
        times(5, "Buzz")
    ).apply(3*5*7) == "FizzBuzz"
  }
}
```

实现工厂：Rule.allof

```
public interface Rule {
  String apply(int n);

  static Rule allof(Rule... rules) {
    return new AllOf(rules);
  }
}
```

匿名内部类

```java
public interface Rule {
  String apply(int n);

  static Rule allof(Rule... rules) {
    return new Rule() {
      @Override
      public String apply(int n) {
        StringBuilder result = new StringBuilder();
        for (Rule rule : rules) {
          result.append(rule.apply(n));
        }
        return result.toString();
      }
    };
  }
}
```

引入工厂：anyof

```
import spock.lang.Specification

import static fizz.buzz.whizz.Rule.anyof

class RuleSpec extends Specification {
  def "factory: times(3) || times(5) -> Fizz || Buzz"() {
    expect:
    anyof(
        times(3, "Fizz"),
        times(5, "Buzz"),
    ).apply(3*5) == "Fizz"

    anyof(
        times(5, "Buzz"),
        times(3, "Fizz"),
    ).apply(3*5) == "Buzz"
  }
}
```

实现工厂：Rule.anyof

```java
public interface Rule {
  String apply(int n);

  static Rule anyof(Rule... rules) {
    return new AnyOf(rules);
  }
}
```

匿名内部类

```java
public interface Rule {
  String apply(int n);

  static Rule anyof(Rule... rules) {
    return new Rule() {
      @Override
      public String apply(int n) {
        for (Rule rule : rules) {
          String result = rule.apply(n);
          if (!result.isEmpty())
            return result;
        }
        return "";
      }
    };
  }
}
```

测试规格

```
def spec() {
  Rule r1_3 = times(3, "Fizz")
  Rule r1_5 = times(5, "Buzz")
  Rule r1_7 = times(7, "Whizz")

  Rule r1 = anyof(r1_3, r1_5, r1_7)

  Rule r2 = anyof(
    allof(r1_3, r1_5, r1_7),
    allof(r1_3, r1_5),
    allof(r1_3, r1_7),
    allof(r1_5, r1_7)
  )

  Rule r3 = contains(3, "Fizz"))
  Rule rd = defaults()

  anyof(r3, r2, r1, rd)
}
```

完备测试集

```
class RuleSpec extends Specification {
  def "fizz buzz whizz"() {
    expect:
    spec().apply(n) == expect

    where:
    n             | expect
    3             | "Fizz"
    5             | "Buzz"
    7             | "Whizz"
    3 * 5 * 7     | "FizzBuzzWhizz"
    3 * 5         | "FizzBuzz"
    3 * 7         | "FizzWhizz"
    (5 * 7) * 2   | "BuzzWhizz"
    13            | "Fizz"
    35 /* 5*7 */  | "Fizz"  /* not "BuzzWhizz" */
    2             | "2"
  }
}
```

函数式

重构 times

```
public interface Rule {
  String apply(int n);

  static Rule times(int n, String word) {
    return (int m) -> {
      return m % n == 0 ? word : "";
    };
  }
}
```

类型推演

```
public interface Rule {
  String apply(int n);

  static Rule times(int n, String word) {
    return m -> m % n == 0 ? word : "";
  }
}
```

重构 contains

```
import static java.lang.String.valueOf;

public interface Rule {
  String apply(int n);

  static Rule contains(int n, String word) {
    return m -> valueOf(m).contains(valueOf(m)) ? word : "";
  }
}
```

重构 defaults

```java
import static java.lang.String.valueOf;

public interface Rule {
  String apply(int n);

  static Rule defaults() {
    return m -> valueOf(m);
  }
}
```

重构 allof

```
import static java.lang.String.valueOf;

public interface Rule {
  String apply(int n);

  static Rule allof(Rule... rules) {
    return m -> {
      StringBuilder result = new StringBuilder();
      for (Rule rule : rules) {
        result.append(rule.apply(m));
      }
      return result.toString();
    };
  }
}
```

重构 anyof

```java
import static java.lang.String.valueOf;

public interface Rule {
  String apply(int n);

  public static Rule anyof(Rule... rules) {
    return m -> {
      for (Rule rule : rules) {
        String result = rule.apply(m);
        if (!result.isEmpty())
          return result;
      }
      return "";
    };
  }
}
```

结构性重复

```java
import static java.lang.String.valueOf;

public interface Rule {
  String apply(int n);

  static Rule times(int n, String word) {
    return m -> m % n == 0 ? word : "";
  }

  static Rule contains(int n, String word) {
    return m -> valueOf(m).contains(valueOf(m)) ? word : "";
  }

  static Rule defaults() {
    return m -> true ? valueOf(m) : "";
  }
}
```

匹配器

```java
import static java.lang.String.valueOf;

@FunctionalInterface
public interface Matcher {
  boolean matches(int n);

  static Matcher times(int n) {
    return m -> m % n == 0;
  }

  static Matcher contains(int n) {
    return m -> valueOf(m).contains(valueOf(n));
  }

  static Matcher always() {
    return m -> true;
  }
}
```

执行器

```
@FunctionalInterface
public interface Action {
  String to(int n);

  static Action to(String word) {
    return n -> word;
  }

  static Action nop() {
    return n -> String.valueOf(n);
  }
}
```

方法引用

```java
@FunctionalInterface
public interface Action {
  String to(int n);

  static Action to(String word) {
    return n -> word;
  }

  static Action nop() {
    return String::valueOf;
  }
}
```

改善表达力

```
import static fizz.buzz.whizz.Matcher.*
import static fizz.buzz.whizz.Action.*
import static fizz.buzz.whizz.Rule.*

class RuleSpec extends Specification {
  private static def spec() {
    Rule r_n1 = atom(times(3), to("Fizz"))
    Rule r_n2 = atom(times(5), to("Buzz"))
    Rule r_n3 = atom(times(7), to("Whizz"))

    Rule r3 = atom(contains(3), to("Fizz"))
    Rule r2 = allof(r1_3, r1_5, r1_7)
    Rule rd = atom(always(), nop())

    anyof(r3, r2, rd)
  }
}
```

规则库：atom

```java
@FunctionalInterface
public interface Rule {
  String apply(int m);

  static Rule atom(Matcher matcher, Action action) {
    return m -> matcher.matches(m) ? action.to(m) : "";
  }
}
```

规则库: allof

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.joining;

@FunctionalInterface
public interface Rule {
  String apply(int m);

  static Rule allof(Rule... rules) {
    return m -> stream(rules)
      .map(r -> r.apply(m));
      .collect(joining());
  }
}
```

规则库: anyof

```java
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.joining;

@FunctionalInterface
public interface Rule {
  String apply(int m);

  static Rule anyof(Rule... rules) {
    return m -> stream(rules)
      .map(r -> r.apply(m))
      .filter(s -> !s.isEmpty())
      .findFirst()
      .orElse("");
  }
}
```

语义模型

语义模型

```
Rule:    int -> String
Matcher: int -> boolean
Action:  int -> String
```

匹配器

```
Matcher: times | contains | always
```

执行器

```
Action: to | nop
```

规则库

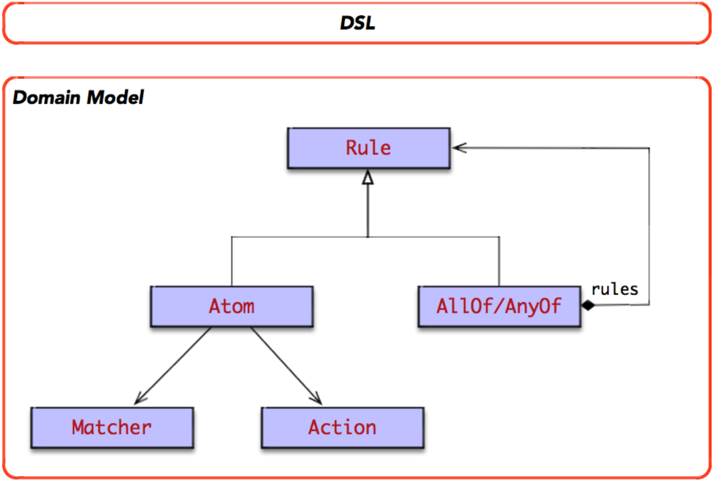
规则库

```
Rule: atom | allof | anyof
```

隐式树

```
atom: (Matcher, Action) -> String
allof(rule1, rule2, ...): rule1 && rule2 && ...
anyof(rule1, rule2, ...): rule1 || rule2 || ...
```

DSL


DSL
Domain Model
Rule
Atom
AllOf/AnyOf
rules
Matcher
Action

参考文献

推荐书籍

- Extreme Programming Explained: Embrace Change, 2th, Kent Beck.
- Agile Software Development: Principles, Patterns and Practices, Robert C. Martin.

联系我

- Email: horance@aliyun.com
- Github: https://github.com/horance-liu
- Blog: http://www.jianshu.com/users/49d1f3b7049e

致谢

Thanks for Attending