重构-改善既有代码的设计：简化条件表达式（七）

1.Decompose Conditional 分解条件表达式

你有一个复杂的条件语句。**从 if、then、else 三个段落中分别提炼出独立函数**。

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge
else charge = quantity * _summerRate;
```

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

程序之中，复杂的条件逻辑是最常导致复杂度上升的地点之一。你必须编写代码来检查不同的条件分支、根据不同的分支做不同的事，然后，你很快就会得到一个相当长的函数。大型函数自身就会使代码的可读性下降，而条件逻辑则会使代码更难阅读。在带有复杂条件逻辑的函数中，代码（包括检查条件分支的代码和真正实现功能的代码）会告诉你发生的事，当常常让你弄不清为什么会发生这样的事，这就说明代码的可读性的确大大降低了。

和任何大块头代码一样，你可以将它分解为多个独立函数，根据每个小块代码的用途，为分解的新函数命名，并将原函数中对应的代码改为调用新建函数，从而更清楚的表达自己的意图。对于条件逻辑，将每个分支条件分解为新函数还可以给你带来更多好处：可以突出条件逻辑，更清楚地表明每个分支的作用，并且突出每个分支的原因。

2.Consolidate Conditional Expression 合并条件表达式

你有一系列条件测试，都得到相同结果。**将这些测试合并为一个条件表达式，并将这个条件表达式提炼为一个独立函数。**

```
double disabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
    // compute the disability amount
```

```
double disabilityAmount() {
    if (isNotEligableForDisability()) return 0;
    // compute the disability amount
```

有时你会发现这样一串条件检查 :检查条件各不相同 ,最终行为却一致。如果发现这种情况，就应该使用“逻辑或”和“逻辑与”将它们合并为一个条件表达式。

之所以要合并条件表达式，有 2 个重要原因。首先，合并后的条件代码会告诉你“实际上只有一次条件检查，只不过有多个并列条件需要检查而已”，从而使这一次检查的用意更清晰。当然，合并前和合并后的代码有着相同的结果，但原先代码传达出的信息却是“这里有一些各自独立的条件测试，它们只是恰好同时发生”。其次，这项重构往往可以为你使用 Extract Method（提炼方法）做好准备。将检查条件提炼成一个独立函数对于厘清代码意义非常有用，因为它把描述“做什么“的语句换成了“为什么这样做”。

条件语句的合并理由也同时指出了不要合并的理由：如果你认为这些条件检查的确彼此独立，的确不应该被视为同一次检查，那么就不要使用本项重构。因为在这种情况下，你的代码已经清晰表达出自己的意义。

3.Consolodate Duplicate Conditional Fragments 合并重复的条件片段

在条件表达式的每个分支上有着相同的一段代码。**将这段重复代码移到条件表达式之外。**

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
}
else {
    total = price * 0.98;
    send();
}
```

```
if (isSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;
send();
```

一组条件表达式的所有分支都执行了相同的某段代码。你应该将这段代码搬移到表达式外面。这样，代码才能更清楚地表明哪些东西随条件变化而变化、哪些东西保持不变。

4.Remove Control Flag 移除控制标记

在一系列布尔表达式中，某个变量带有“控制标记’的作用。

以 break 或 return 语句取代控制标记。

```
void checkSecurity(String[] people) {
    String found = foundMiscreant(people);
    someLaterCode(found);
}

String foundMiscreant(String[] people) {
    String found = "";
    for (int i = 0; i < people.length; i++) {
        if (found.equals("")) {
            if (people[i].equals("Don")) {
                sendAlert();
                found = "Don";
            }
            if (people[i].equals("John")) {
                sendAlert();
                found = "John";
            }
        }
    }
    return found;
}
```

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

在一系列条件表达式中，常常会看到用以判断何时停止条件检查的控制标记。这样的标记带来的麻烦超过了它所带来的便利。人们之所以会使用这样的控制标记，因为结构化编程原则告诉他们：每个子程序只能有一个入口和出口。“单一出口“原则会让你在代码中加入让人讨厌的控制标记，大大降低条件表达式的可读性。这就是编程语言提供 break 和 continue 语句的原因：用它们跳出复杂的条件语句。去掉控制标记所产生的效果往往让你大吃一惊：条件语句真正的用途会清晰得多。

5.Replace Nested Conditional with Guard Clauses 以卫语句取代嵌套条件表达式

函数中的条件逻辑使人难以看清正常的执行途径。**使用卫语句表现所有特殊情况。**

```
double getPayAmount() {
  double result;
  if (_isDead) result = deadAmount();
  else {
      if (_isSeparated) result = separatedAmount();
      else {
          if (_isRetired) result = retiredAmount();
          else result = normalPayAmount();
      };
  }
return result;
};
```

```
double getPayAmount() {
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) return retiredAmount();
  return normalPayAmount();
};
```

条件表达式通常有 2 种表现形式。第一：所有分支都属于正常行为。第二：条件表达式提供的答案中只有一种是正常行为，其他都是不常见的情况。

这 2 类条件表达式有不同的用途。如果 2 条分支都是正常行为，就应该使用形如 if.....else.....的条件表达式；如果某个条件极其罕见，就应该单独检查该条件，并在该条件为真时立刻从函数中返回。这样的单独检查常常被称为“卫语句”。

Replace Nested Conditional with Guard Clauses（以卫语句取代嵌套条件表达式）的精髓是：给某个分支以特别的重视。它告诉阅读者：这种情况很罕见，如果它真的发生了，请做一些必要的整理工作，然后退出。

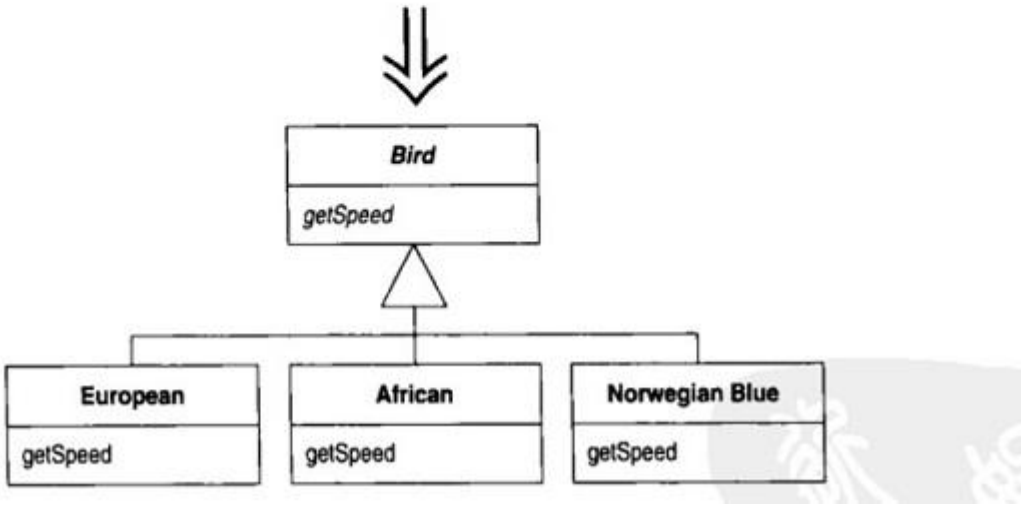
“每个函数只能有一个入口和一个出口”的观念，根深蒂固于某些程序员的脑海里。现今的编程语言都会强制保证每个函数只有一个入口，至于“单一出口”规则，其实不是那么有用。保持代码清晰才是最关键的：如果单一出口能使这个函数更清晰易读，那么就使用单一出口；否则就不必这么做。

（卫语句就是把复杂的条件表达式拆分成多个条件表达式，比如一个很复杂的表达式，嵌套了好几层的 if - then-else 语句，转换为多个 if 语句，实现它的逻辑，这多条的 if 语句就是卫语句.）

6.Replace Conditional with Polymorphism 以多态取代条件表达式

你手上一个条件表达式，它根据对象类型的不同而选择不同的行为。**将这个条件表达式的每个分支放进一个子类的覆写函数中，然后将原始函数声明为抽象函数。**

```
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");
}
```



多态的最根本的好处是：如果你需要根据对象的不同类型而采取不同的行为，多态使你不必编写某些的条件表达式。

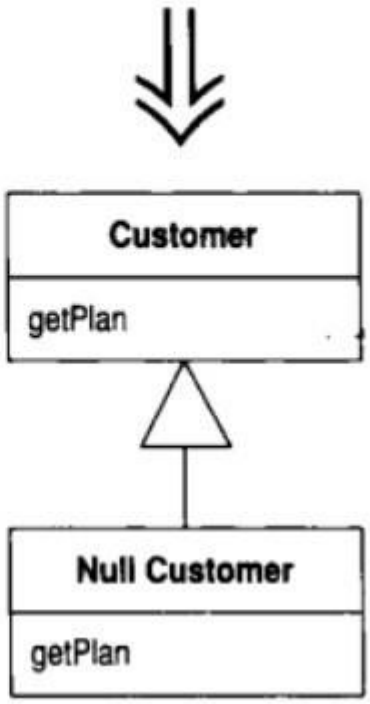
正因为有了多态，所以你会发现：“类型吗的 switch 语句”以及 ” 基于类型名称的 if-then-else 语句“在面向对象程序中很少出现。

多态能够给你带来很多好处。如果同一组条件表达式在程序的许多地点出现，那么使用多态的收益是最大的。使用条件表达式时，如果你想添加一种新类型，就必须查找并更新所有条件表达式。但如果使用多态，只需建立一个新的子类，并在其中提供适当的函数就行了。类的用户不需要了解这个子类，这就大大降低了系统各部分之间的依赖，使系统升级更加容易。

7. Introduce Null Object 引入 Null 对象

你需要再三检查某对象是否为 null。**将 null 值替换为 null 对象。**

```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```



多态的最根本好处在于：你不必再向对象询问“你是什么类型”而后根据得到的答案调用对象的某个行为-你只管调用该行为就是了，其他的一切多态机制会为你安排妥当。当某个字段内容是 null 时，多态可扮演另一个较不直观的用途。

8. Introduce Assertion 引入断言

某一段代码需要对程序状态做出某种假设。**以断言明确表现这种假设。**

```
double getExpenseLimit() {
    // should have either expense limit or a primary project
    return (_expenseLimit != NULL_EXPENSE) ?
        _expenseLimit:
        _primaryProject.getMemberExpenseLimit();
}
```

```
double getExpenseLimit() {
    Assert.isTrue (_expenseLimit != NULL_EXPENSE || _primaryProject != null);
    return (_expenseLimit != NULL_EXPENSE) ?
        _expenseLimit:
        _primaryProject.getMemberExpenseLimit();
}
```

常常会有这样一段代码：只有当某个条件为真时，该段代码才能正常运行。

这样的假设通常并没有在代码中明确表现出来，你必须阅读整个算法才能看出。有时程序员会以注释写出这样的假设。可以使用断言明确标明这些假设。

断言是一个条件表达式，应该总是为真。如果它失败，不是程序员犯了错误。因此断言的失败应该导致一个非受控异常。断言绝对不能被系统的其他部分使用。实际上，程序最后的成品往往将断言删除。因此，标记“某个东西是个断言”是很重要的。

断言可以作为交流与调试的辅助。在交流的角度上，断言可以帮助程序阅读者了解代码所做的假设；在调试的角度上，断言可以在距离 bug 最近的地方抓住它们。