文章编号: 1000-7024(2004)11-1985-03 中图分类号: TP311.1 文献标识码: A

基于AOP的代码规则强化方法研究

李 伟, 郑 宁
(杭州电子工业学院, 浙江 杭州 310018)

摘 要: 分析了常见代码规则强化方法中存在的问题, 结合AOP (Aspect-Oriented Programming, 简称AOP) 的特点, 给出了一种基于AOP的代码规则强化方法, 并用方面来表示代码规则, 通过把代码规则织入系统代码, 在编译阶段和运行阶段实现了代码规则强化。

关键词: 代码规则强化; AOP; 方面; 最佳实践

Study of way to enforce code rules based on AOP

LI Wei, ZHENG Ning
(Hangzhou College of Electronic and Engineering, Hangzhou 310018, China)

Abstract: The problems of some common ways to enforce code rules and the features of AOP (aspect-oriented programming) are analyzed, and a new way based on AOP is proposed. This way expresses code rules with aspects, and implements enforcement of code rules in compile phase and runtime phase by weaving code rules into system codes.

Key words: enforce code rule; AOP; aspect; best-practices

1 引 言

软件开发中有许多规则需要遵循, 这些规则包括软件开发中成功的实践经验和模式[1]、各种规范和标准、失败的教训、设计原则和常见技巧等。代码规则强化就是用来保证程序代码遵循这些规则的一种机制, 它是提高代码质量最有效、最直接的手段之一。代码规则强化能够遵循成功的实践, 有效地避免以前容易出现的错误, 提高模块的可移植性和可维护性。使用代码规则强化, 还有利于保持代码的一致性, 形成开发团队统一的风格。

当前主要通过文档、内嵌检查和工具检查等方式来实现代码规则强化, 这些方法在一定程度上满足了代码规则强化的要求, 但是也存在着很多的不足。本文运用AOP给出一种新的代码规则强化方法, 该方法克服了当前代码规则强化方法中存在的诸多问题, 提高了代码规则强化的灵活性和方便性。

2 当前代码规则强化中存在的问题

常见的代码规则强化的方法主要有: 文档法、内嵌校验法、工具校验法等。文档法是把代码规则以文档形式表示, 通过培训等方式落实到每个开发人员的开发过程中; 内嵌校验法是在代码中内嵌校验代码, 以实现代码规则的强化; 工具法是借助合法性检查工具, 如编译器和应用服务器以及第3方工具等进行强化。这些方法在一定程度上实现了代码规则强化, 但是也存在着较大的局限性。

(1) 代码规则无法重用。一个项目中的代码规则实现不能应用到另外的项目中, 这就意味着要做大量的重复性工作, 很容易带来潜在的错误。例如, 文档法强化要依靠开发人员自觉, 这就会带来很大的不确定性, 很难实现重用; 内嵌校验法因为是把代码规则内嵌在系统代码中的, 所以也无法实现重用。

(2) 规则强化代码和系统代码交织在一起, 不易维护。规则强化是系统级别的, 适用于多个模块, 把强化代码和系统代码混杂在一起会引起混淆, 不利于维护。例如, 内嵌校验法把强化代码散落到多个模块, 和系统代码交织在一起, 这样对规则强化代码的修改可能会影响到系统代码, 所以不容易维护。

(3) 规则强化代码是分散的, 灵活性差。因为规则强化代码分散于多个模块, 如果需要修改某一代码规则就要修改每一个相关的模块, 而且当加入新模块时还必须在其中加入所有的规则强化代码。这一点内嵌校验法表现得最为明显, 分散的强化代码很难进行维护。

(4) 实现复杂, 成本高, 效率低。无论是最初的开发阶段, 还是以后的维护阶段, 代码规则强化的工作量是巨大的, 耗费大量的时间, 增加开发的成本。无论是文档法还是内嵌校验法都存在这样的问题, 加大了代码规则强化实现的

收稿日期: 2003-11-17。

作者简介: 李伟 (1979-), 男, 山东高青人, 硕士研究生, 研究方向为电子商务、企业信息系统; 郑宁, 男, 教授, 博士生导师, 研究方向为ICCAD、电信网管和电子商务。

难度。

(5) 强化的能力有限，可靠性差。只能进行基本的、局部的强化，无法胜任复杂的、系统级别上的强化。例如，工具校验法只能进行基本的语法检查，无法定制复杂的代码规则；文档法则过多地依靠开发人员的自觉行为，因而可靠性差。

3 代码规则强化的AOP实现

3.1 AOP简介

软件系统应满足现实世界的一系列需求。在AOP的世界里，一个特定的需求可称为一个关注点(Concern)。例如，在仓储管理系统中存在着以下关注点：出入库管理、移库管理、盘点管理、仓储费用计算、日志管理等。关注点分为两类：核心关注点(Core Concern)和横切关注点(Crosscutting Concern)[2]。核心关注点即一个模块的核心需求；横切关注点是系统级别上的，贯穿于多个模块，如日志管理、安全管理等。

虽然面向对象编程(Object-Oriented Programming, 简称OOP)是最常用的实现核心关注点的方法，但是它却无法有效地处理横切关注点，特别是在复杂的系统中。OOP导致了核心关注点和横切关注点的过分耦合，使得横切特征的加入或修改会影响到所有相关的核心关注点。随着软件开发技术的发展，AOP应运而生。它通过方面(Aspect)来模块化横切关注点，借助方面织入器(Aspect Weaver)把核心关注点和横切关注点组织到一起，形成最终的系统。这样就实现了核心关注点与横切关注点的分离，使得系统更容易设计、实现和维护。

现在有多种AOP的实现如AspectJ、Hyper/J、AspectWerkz、AspectC++等[3]，其中AspectJ发展得比较完善，得到了广泛的应用，所以本文选用的是AspectJ。AspectJ[4]是由AOP的提出者Xerox PARC实现的，现在已归开放源码组织Eclipse(www.eclipse.org)管理。

3.2 AspectJ的基本概念介绍

AspectJ是Java语言的面向方面的扩展，它在Java中加入了如下语言构造使其具有面向方的能力。

(1) 连接点(Join point)[4]：程序执行过程中明确定义的点。连接点可以定义在方法调用、条件检查、循环的开始或者复制动作等处。连接点有一个与之相关的上下文。例如，一个方法调用连接点的上下文可能会包含一个目标对象及调用参数。在AspectJ中主要有以下几种连接点：方法的调用和执行、构造器的调用和执行、属性的读/写访问、异常处理、对象和类的初始化执行等。

(2) 切入点(Pointcut)[4]：用来指定、组合所需连接点并搜集连接点上特定上下文信息的语言构造。可以用它来指明一系列连接点，同时它还可以为在连接点上执行的通知提供上下文信息。

(3) 通知(Adivce)[4]：在符合特定条件的情况下执行的代码。AspectJ提供了3种把通知关联到连接点的方式：before、after及around。before通知在连接点的前面运行；after通知在连接点的后面运行；around通知包在连接点的外面，并有权决定是否运行此连接点，还可以在此处修改连接点上下文环境。

(4) 方面(Aspect)[4]：实现横切关注点的模块化单元，它把切入点和通知封装在一起。和类相似，方面也可以包含方法和属性、从其它类或方面扩展及实现接口等；但与类不同的是，不能用new来创建一个方面的实例。

3.3 代码规则强化的AOP实现

代码规则强化是对整个系统而言的，贯穿于多个模块，所以可以看做是横切关注点，适合于用AOP来实现。用方面来表示代码规则，通过把代码规则方面织入到系统代码中以实现代码规则的强化，AspectJ是以编译方式实现方面织入的。在AspectJ中可以通过两种方式进行代码规则的强化：编译时强化和运行时强化。

(1) 编译时强化：系统在编译阶段(即方面织入时)实现的代码规则强化。Java、C++等强类型语言的编译器已经提供了某些编译时的检查，如类型匹配、访问控制等。而AspectJ的编译器提供了更为灵活、强大的检查，并且允许定制。编译时强化使用了以下两种AspectJ结构[4]：

declare error:<pointcut>:<message>;

declare:warning:<pointcut>:<message>;

其中error和warning定义了两种不同的提示级别。

尽管编译时强化的作用是强大的，但是也存在着一定的局限性：程序执行的某些行为，如参数值的范围检查，是无法在编译时实现的。

(2) 运行时强化：织入代码规则的系统在运行过程中进行的代码规则强化。在Java语言中，虚拟机能够执行某些运行时检查，如类型转换检查、空指针访问检查等，但是这些检查是虚拟机内置的，无法实现定制。而运行时强化允许定制，这就提高了代码规则强化的灵活性。为了及时保存运行时相关的信息，可以通过日志把强化时的上下文记录下来。运行时强化使用的是AspectJ的动态横切结构，如before通知、after通知及around通知。

代码规则强化完成后就可以把代码规则方面移去，用传统的编译器重新编译源代码，这样就得到最终的系统。因为这些方面没有改变源代码，所以移去方面后系统功能不受影响，保证了系统的正常运行。

4 典型的代码规则强化应用模式

上一部分对基于AOP的代码规则强化的原理做了说明，在这里将通过两个典型的应用模式来做阐述。

4.1 接口调用的合法性强化

软件开发过程中经常更新或者调整代码，这往往需要原来的接口或引入新的接口，而以前的接口则不推荐使用。例如，对于Class1有一方法method1(…)，现在做了优化后引入了另外一个方法method2(…)，这就要求以后用到method1(…)的地方要用method2(…)来代替，如果还是使用了method1(…)则应该及时予以提示。

在Java中通常采用文档标签@deprecation在源代码中

标记来实现。但是这种方法会更改原来的代码，不够灵活，更无法实现代码规则的模块化。而通过 AspectJ，可以写一个简单的方面来实现这一代码规则强化：

```
import test.Class1;
aspect ValidateDeprecation {
  declare warning
        : call(void Class1.method1(..))
        : "Method is deprecatory, please replace it with method2";
}
```

把该方面加入到原来的系统中用 AspectJ 的编译器进行编译时，编译器就会指出所有用到 method1 的地方，并提示 method1 已不推荐使用，应该进行修改。

如果要提高警告的级别，可以把“declare warning”改为“declare error”。还可以利用 AspectJ 中切入点丰富的表达能力来进一步精确定义该代码规则。例如，改为只对 test 包中的代码进行强化，则只需要把切入点改为如下。

```
call(void Class1.method1(..))
  &&within(test.*)
```

4.2 最佳化实践原则强化

经过多年的软件开发，无论个人还是团队都有一些最佳化实践——编程技巧和模式，运用这些最佳化实践可以有效避免某些潜在错误，大大提高代码的质量。虽然违反这些最佳化实践并不会立即带来什么问题，但是当问题聚集到一定程度时就很难解决。所以，这些最佳化实践应该从开始就要认真遵循，对没有遵循的应该及时发现和解决。

面向对象中的一条基本的原则是隐藏具体的实现细节。破坏这一原则不会立即产生什么影响，但是随着时间的推移就会形成“蝴蝶效应”：对局部的修改可能导致整个系统的崩溃。尽管这一原则很容易接受，但是开发人员却很难严格遵守。这需要做大量的、不断的培训，因为过一段时间后开发人员可能又会犯同样的错误。借助 AspectJ 则可以很容易地解决这一问题，通过下面的方面就可以保证。

```
aspect ValidateAccess {
  declare warning
        :get(public ! final * *)
          || set(public * *)
        :"Please consider using nonpublic access";
}
```

这个方面能够指出代码中所有可公共访问的、非 final 的属性被赋值的地方，也就是破坏该原则的地方。这样就可以很容易地进行修改。

复杂的最佳化实践也可以用 Aspect 来强化，例如，为了提高系统的灵活性，在创建对象时应采用静态工厂方法而不是用构造函数[5]。这一最佳化实践就可以用方面来进行强化：

```
aspect FlagNonFactoryCreation {
  declare error:call(Class1.new(..))
    && ! withincode(Class1 Class1Factory.createClass1(..))
    :"Only Class1Factory.createClass1can create Class1";
}
```

该方面对代码中所有用 new 创建 Class1 对象的地方（除方法 Class1Factory.createClass1(..)）予以 error 提示：只能用静态工厂方法 Class1Factory.createClass1(..)创建 Class1 对象。

5 基于 AOP 的代码规则强化的优点分析

基于AOP的代码规则强化方法把适用于多个模块中的代码规则看做横切关注点，并用方面来模块化，实现了代码规则的集中管理，解决了当前代码规则强化中存在的问题，具有以下优点：①实现集中管理，易于维护。把作用于多个模块的代码规则抽取出来用方面表示，实现代码规则的集中管理，降低了维护的难度；②易于重用。表示代码规则的方面是与程序代码分离的，具有普遍性，这样可直接用于新的项目，实现项目间的重用。也方便在一个项目的不同开发阶段、不同开发人员之间进行重用；③可以及时、持续、快速地实现强化，效率高，成本低。只要把表示代码规则的方面包含到系统中，用 AspectJ 编译器进行编译就可以快速实现强化。当加入新的模块后，也可以及时地进行强化。从而实现了及时、持续、快速的检查，提高了代码规则强化的效率，同时降低了强化的成本；④减少人的干预，不影响源代码，可靠性高。通过选择不同的编译器就可以灵活地加载或者卸载代码规则，并不会影响到原来的代码，减少了人的干预，提高了强化的可靠性。

6 结束语

本文针对当前代码规则强化中存在的诸多问题，给出基于 AOP 的代码规则强化方法。该方法充分发挥了 AOP 的特点，克服了当前代码规则中存在的问题，为代码规则的强化提供了新的思路。

当然，该方法有待于进一步完善和扩充，例如，建立代码规则库以实现代码规则的有效管理、强化设计模式的实现、结合重构[6]以优化代码结构等。

参 考 文 献:

[1] Erich Gamma. 设计模式:可复用面向对象软件的基础[M]. 北京:机械工业出版社, 2000.

[2] Kiczales G, Lopes C. Aspect-oriented programming[EB/OL]. http://www.parc.xerox.com/spl/projects/aop.

[3] AOSD Steering Committee. Supported systems[EB/OL]. http://aosd.net/technology/research.php.

[4] The aspectJTM programming guide[EB/OL].http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/prog-guide/index.html.

[5] Bloch J. Effective java programming language guide[Z]. 2001.

[6] Martin Fowler. 重构:改善既有代码的设计[M]. 北京:中国电力出版社, 2003.