面向 Aspect 的程序设计——一种新的编程范型*)

曹东刚　梅　宏
(北京大学软件研究所　北京 100871)

Aspect Orientation——A New Approach to Programming*)

CAO Dong-Gang　MEI Hong
(Software Institute, Peking University, Beijing 100871)

Abstract Currently the main stream programming paradigm is Object Oriented Programming: OOP, which has gotten great success. The advantage of OOP is that it provides effective modularity support, and enables direct mapping from requirement space to design space, etc. However, there are still some specific requirement and design issues, e.g., security and logging, which are hard to be implemented with clear modularity using either procedural or OOP techniques. In fact, the code of this kind of concerns are often scattered through out the whole system, resulting in some ugly scattering code and tangling code. Such kinds of programs are often difficult to develop and maintain. Recently a new programming paradigm——Aspect Oriented Programming: AOP, which aims at solving the crosscutting concerns, has gained much popularity. This paper is an overview of AOP.

Keywords Programming, Modularity, Crosscutting concern, Separation of concerns

1 引言

面向 Aspect 的程序设计(Aspect Oriented Programming:AOP)[1],其概念的出现不过几年的时间,却体现了解决问题的非常简单而深刻的"分而治之"的思想。"分而治之"是指把一个复杂的问题分解成若干个简单的问题,然后逐个解决。这种朴素的思想来源于人们生活与工作的经验,也同样适合于软件技术领域。软件人员在实施分而治之的时候,应该着重考虑:复杂问题分解后,每个子问题能否分别用程序实现?所有程序最终能否集成为一个软件系统并有效解决原始的复杂问题?可以说,目前的程序设计技术已经极大地提高了开发效率,在支持程序员通过"分而治之"的途径解决复杂的软件开发问题方面有了很大进展。然而,一些复杂问题分解后形成的子问题,特别是系统的"贯穿特性",在目前的程序设计技术框架下,并不能很好地加以程序模块化实现。软件的复杂性并没有被控制在令人满意的范围内。AOP 正是在这样的背景下出现的。

目前关于 AOP 的研究和 20 年前的 OOP 研究现状很相似。基本概念开始形成,一些研究小组已经取得了较好进展,AOP 所蕴含的基本思想在程序设计领域已经得到一定范围的认同。当然,现在的研究工作尚处于初步阶段,特别是缺少大型实际应用的验证。

1.1 贯穿特性示例

文[6]给出了一个简单的使用面向对象方法设计的图元编辑器的示例,如图 1 所示。在该图元编辑器中,抽象图元类 FigureElement 有两个具体图元子类 Point 和 Line,分别对点和线进行管理。这两个类体现了良好的模块性,类中方法的源代码都紧密相关,内聚度很高,并且每个类的接口都很清晰。现在有这样一个显示更新的需求:不论图元何时移动、移动到哪里,都要通知屏幕管理器(Display)其位置发生了改变。采用面向对象的设计方法,典型的做法是在每一个移动图元的操作代码中,都插入一段通知 Display 其位置发生了改变的代码(调用 Display.Update()方法),如图 2 所示。图 1 中的 DisplayUpdating 框图表示了逻辑上的"显示更新"需求,它的实现代码并不是模块化的,因为它不能被图 1 中的任何类封装成单独的模块单位。从图 2 的类代码中可以看出,实际上 DisplayUpdating 需求由 Point 和 Line 两个类的四种方法联合实现。这样的需求,就是所谓的"贯穿特性"(crosscutting concerns)。

从上面的例子可以看出,采用面向对象方法实现 DisplayUpdating 需求,是比较笨拙的。该需求的实现代码不得不散布于系统的所有更新图元位置的操作中,难以用独立的对象或者其他模块化的方式进行处理。而图元类的 SetX/ SetY 和 SetP1/SetP2 方法的代码中又要实现多个需求,彼此相互纠缠交织,破坏了类方法原来的模块性。

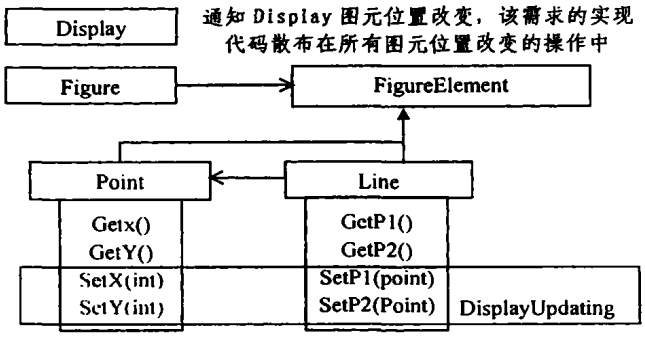

图 1　图形元素类示意图

```
class Line{
    private Point-p1,-p2;
```

*)基金项目:国家杰出青年科学基金课题(60125206);国家自然科学基金(60043002, 60103001);国家高技术研究发展计划(863 计划,2001AA113060);教育部重大项目(重大 0214)。

```
    Point getP1() {reture_p1;}
    Point getP2() {reture_p2;}
    void setP1(Point p1){
        this._p1=p1;
        Display.update();
    }
    void setP2(Point p2){
        this._p2=p2;
        Display.update();
    }
}

class Point{
    private int_x1,_x2;
    int getX1() {return_x1;}
    int getX2() {return_x2;}
    void setX1(int x1){
        this._x1=p1;
        Display.update();
    }
    void setX2(int x2){
        this._x2=x2;
        Display.update();
    }
}
```

图 2　图形元素类的部分实现代码

理想的做法是既能保持系统原来的良好模块性，又能将 DisplayUpdating 需求的实现代码模块化。目前使用 OOP 技术难以满足上述要求，而 AOP 技术却提供了这样的方法和机制。在 AOP 中，通过 Aspect 将 DisplayUpdating 的实现代码封装成独立的模块单元，而不必和其他模块的实现代码交织。Aspect 是和对象地位相同的独立设计单元，因此，程序员在设计阶段就可以从“Aspect”的角度思考如何解决问题。

实际上，图 2 的简单例子可以引发更深入的思考：为什么会出现贯穿特性？是因为设计不当，还是因为需求分析不够深入？下面将对此具体分析。

1.2　软件设计原则—分离关注点

计算机软件设计的一个重要原则，就是要清晰分离各种关注点(separation of concerns)，然后分而治之，各个击破，最后形成统一的解决方案。所谓关注点，是指一个特定的目标、概念或者兴趣域。从过程的角度，典型的开发关注点包括需求分析、设计、编码、测试和维护。从技术的角度，一个典型的软件系统分别包含若干核心级和系统级的关注点。核心级关注点就是系统要完成的业务功能；而系统级关注点是完成核心关注点所必须的配套设施，这些配套设施通常被认为是整个系统的系统特性，或者是业务功能的功能约束。例如，信用卡处理系统的核心关注点是处理付款，而系统级的关注点则包括日志、事务、认证、安全和性能等等。这种关注点的分离，首先体现在对系统需求的分析方面。人们对需求进行规约，然后针对需求规约进行系统设计，最后用编程语言分别实现每一项需求。图 3[11]给出了一个形象的软件系统关注点分离的示例。

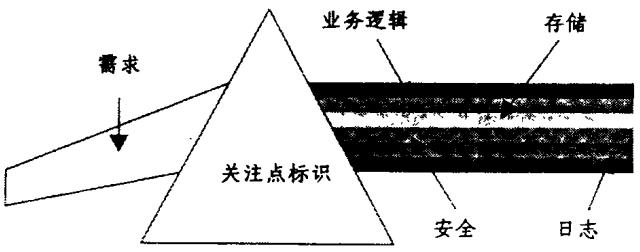

图 3　软件系统的关注点分离

将复杂系统中的多种关注点分离，然后分而治之，被经验证明为人们解决问题的一种有效方法。尤其是在软件系统中，这种方法有利于目标系统的演化，可以提高系统的可理解性，提供更好的适应性、可变性和复用性。

在目前的技术框架下，从一个软件系统的任何结构化实现形式中，都会找到一些关注点，它们能够很好地用单独的模块实现；同时还有另外一些关注点，它们的实现代码却贯穿于若干不同模块中。通常系统级的关注点在逻辑上相互之间彼此正交(相互独立)，同时在实现上趋向于和若干核心模块交织。例如，信用卡处理系统的每一个核心业务关注点，都和安全、日志等系统关注点相关联。核心业务关注点多数情况下可以被很好地分解，并通过编程语言模块化地实现。但有时核心业务关注点彼此之间也相互交织。例如 1.1 节的“显示更新”和“图元移动”两个关注点就互相交织。

如果某个关注点的实现代码无法模块化，而是散布于系统其他模块中，和若干其他关注点相互交织，我们就称其为系统的“贯穿特性”。贯穿特性的出现，是程序设计人员所不乐见的，因为在目前的程序设计技术下，具有贯穿特性的目标系统难于实现，难于理解，难于演化。

1.3　面向对象等技术的不足

从认识论的角度看，软件开发可以归结为两种活动：人们对问题域的认识和基于这种认识所进行的描述(实现)。开发人员对问题域的认识是通过自然语言进行的，而描述(实现)则是通过编程语言进行的，它们之间存在着很大的“语言鸿沟”[13]。语言鸿沟的存在意味着问题空间和解空间的失配，导致人们设计开发出来的许多软件或者没有正确地实现需求，或者实现方式笨拙别扭，容易产生很多错误。为了解决软件开发中的实现问题，人们总结并提出了程序设计方法学，力图以一种符合人类思维模式的、自然而连贯的系统构造方式，实现从系统需求到程序结构的映射。从早期原始的机器代码和汇编语言程序设计，历经过程型程序设计、结构化程序设计和抽象数据类型程序设计等，直至今天的面向对象程序设计(OOP)，程序设计技术已经发生了深刻的演变[5]，大大缩短了认识和描述之间的语言鸿沟，提高了人们对问题域的描述能力，使人们能够创建愈来愈复杂的系统。

编程语言提供了程序员和机器通信的方式，它和程序设计方法及过程紧密相关。程序设计过程通过清晰分离各种关注点，将系统分解为更小的子单元；编程语言则提供一种抽象机制，允许程序员对系统子单元进行抽象和定义，最后以不同的方式将这些抽象结果组合成最终的运行系统。当编程语言提供的抽象和组合机制能够很好地支持程序设计过程对系统的分解时，程序设计过程和编程语言就可以很好地协调工作。随着编程语言的发展，其提供的抽象机制从原始的机器代码、汇编代码，发展到过程和函数，直到现在的类和对象，抽象功能越来越强。

在传统的程序设计范型中，“子程序”的思想占有重要地位。自 Fortran 语言以来的所有编程语言，都能通过创建并显式调用子程序的方法，来部分地分离系统关注点。然而，并不是所有的关注点都能很利索地通过子程序实现[5]。当一个系统关注点的实现代码和其他系统关注点的实现代码交织糅杂在一起的时候，程序的模块性被破坏，复杂性增加。并且，进行子程序调用的程序员，必须对被调用的子程序有足够的了解。就是说，一段程序必须知道要显式地调用另一段子程序，并知道怎样去调用。这极大限制了程序员并行独立解决子问题的能力。

目前，主流的程序设计技术是面向对象的程序设计(OOP)——一种通过将问题域分解为对象，然后编写这些对象的实现代码，通过组装对象来构造软件系统的思想。对象将数据及其相关的行为抽象为单独的概念或物理实体，表现了良好的封装性和模块性。在编写复杂程序，比如图形用户界面、操作系统和分布应用程序等，以及维护源代码的可理解性方面，OOP 表现了强大的能力。OOP 已被公认为是一种能极大地提升软件开发效率和质量的技术。

但是人们也发现，对象模型和问题域有时候并不能很好地吻合，因此 OOP 技术并不能全部清晰地实现系统所必须的所有重要关注点。有时候一个需求或关注点经常由若干对象共同实现，而这些对象又实现了若干其他的需求或关注点。例如，在 1.1 节的贯穿特性示例中，DisplayUpdating 需求由 Point 和 Line 两个类的所有更新图元操作实现，而 Point 和 Line 两个图元类还实现了其它的图元显示和管理需求。换言之，对象技术不能很好地处理如下问题：将牵涉到全局限制和总体行为的系统责任局部化；适当地分解需求和划分系统责任，使之能够模块化地实现；应用领域特定知识。实际上，不管是采用 OOP 技术还是过程型技术，都不能很好地解决这些问题，尤其是“贯穿特性”的问题。

上述问题的出现，并不能简单地归咎于分析和设计。实际上，这是面向对象建模技术的内在局限性。出现贯穿特性的技术原因，在于目前的实现技术只提供了一维方法学实现系统的需求和关注点，强迫实现代码沿着某单一维度映射需求。该单一维度一般是核心需求和关注点的模块化实现，其它类型的需求也被迫和该主导维度一致。换句话说，问题空间是 n 维的，而解空间却是一维的，这种失配必然导致需求和实现之间的失配，如图 4 所示。对象技术尽管在建模能力以及控制复杂度的能力方面有了很大增强，但是仍未跳出此局限。

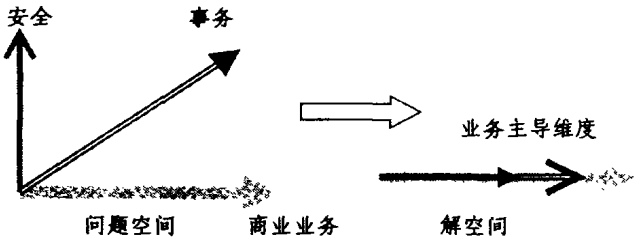

图 4　问题空间和解空间的失配

本质上，用现在标准语言实现的软件，是线性结构的。正如一本书只有分解为章节和段落的一种方式一样，软件也只采用一种分解为模块(类)的方式。这就是目前主流的关注点分离技术。分解之后形成的模块，有效地封装了若干概念(如一些对象的表示和实现细节被封装在类里)。但是还有一些关注点不能被封装于核心需求的模块中，结果是它们的实现代码分布于许多模块中，和核心模块互相交织。从最后结果看，这类关注点即是所谓的贯穿特性。

AOP 的出现，正是为了改进上述程序设计方法学的不足。AOP 被视为是后面向对象时代的一种新的重要的程序设计技术。

2　面向 Aspect 的程序设计

2.1　什么是 Aspect

所谓的 Aspect，就是 AOP 提供的一种程序设计单元，它可以将上文提到的那些在传统程序设计方法学中难于清晰地封装并模块化实现的设计决策，封装实现为独立的模块。

Aspect 是 AOP 的核心，它超越了子程序和继承，是 AOP 将贯穿特性局部化和模块化的实现机制。通过将贯穿特性集中到 Aspect 中，AOP 就取得一种单一的结构化行为--该行为在传统程序中分布于整个代码中--这样就使 Aspect 代码和系统目标都易于理解。在 AOP 中，Aspect 是 AOP 中的一阶实体。Aspect 之于 AOP，正如类之于 OOP。现有对 Aspect 的认识有错误校验策略、设计模式、同步策略、资源共享、分布关系和性能优化等。

Aspect 的实现和传统开发方法中模块的实现不同。Aspect 之间是一种松耦合的关系，各 Aspect 的开发彼此独立。主代码的开发者甚至可能没有意识到 Aspect 的存在。只是在最后系统组装时刻，才将各 Aspect 代码和主代码编排融合在一起。因此，主代码和 Aspect 之间可以是一种不同于传统“显式调用”关系的“隐式调用”[5]。在软件复杂性日益增加的今天，隐式调用有巨大的优点。因为某一应用的领域专家，不太可能对分布、认证、访问控制、同步、加密、冗余等问题的复杂的实现机制很熟悉，所以就不能保证他们在程序中进行正确的调用。在当前强调程序演化的情况下，这一点尤其重要，因为开发人员很难正确预见到未来对程序的新需求。

如何保证 Aspect 和 Aspect 组合的语义正确性，对于 Aspect 的成功应用至关重要。在模块化系统中，如何保证模块自己正确并且在组合后能正确交互，一直没有满意的方案。AOP 语言提供的基于连接点的组合较之其它模块化机制提供的基于接口或者消息的连接更为丰富，这就使组合规约和组合测试更加复杂。即使是一个单独 Aspect 的规约和单元测试也需要继续深入研究。

在目前强调软件复用的环境下，Aspect 作为一种模块单位，其复用理所当然成为 AOP 的一个研究热点。如何保证可复用 Aspect 的正确性？保证可复用 Aspect 在特定语境下正确工作的合理假定是什么，其表达方式是怎样的？同时，研究人员需要在发展较大规模的可复用 Aspect 库方面努力。相关的问题是大量 Aspect 怎样协同工作？它们怎样组合？怎么评价这样的设计？应该使用什么符号描述它们？所有这些相关问题有待于进一步研究。

2.2　AOP 定义

面向 Aspect 的程序设计是一种关注点分离技术，通过运用 Aspect 这种程序设计单元，允许开发者使用结构化的设计和代码，反映其对系统的认识方式。和所有其它的关注点分离技术一样，AOP 的目标是使设计和代码更加模块化和更具结构性，使关注点局部化而不是分散于整个系统中，同时和系统其他部分保持良好定义的接口，从而真正达到“关注点分离，分而治之”的目的。

AOP 构建在已有的技术基础之上，同时提供了自己的一套额外机制，也就是 Aspect 机制，对系统进行描述、设计和实现。AOP 要保证这些机制在概念上简洁，执行效率高。其基本思想是通过分别描述系统的不同关注点(属性或者兴趣)及其关系，以一种松耦合的方式实现单个关注点，然后依靠 AOP 环境的支撑机制，将这些关注点组织或编排成最终的可运行程序。关注点包括普通关注点和系统的贯穿特性。通常可以使用传统的结构化方法和面向对象方法提供的机制，对普通关注点进行处理；使用 Aspect 机制，对贯穿特性进行处理。

系统的贯穿特性范围包括了从高层的关注目标，比如安全和服务质量，到低层的关注目标比如缓存处理等。贯穿特性

可以是功能性的，比如事务规则，也可以是非功能的，比如同步和交易管理等。AOP 将传统方法学中分散处理的贯穿特性实现为系统的一阶元素——Aspect，并将它们从类结构中独立了出来，成为单独的模块。

2.3 AOP 机制

图 5 给出了一个 AOP 系统运用 Aspect 机制开发软件的过程的简单抽象。首先针对需求规约进行 Aspect 分解，这需要对系统的 Aspect 进行标识。然后对于标识出的 Aspect 分别通过程序机制实现。最后用 Aspect 编排器将所有的单元编排重组在一起，形成最终的可运行系统。

为了支持上述的系统实现和运行过程，AOP 系统首先必须提供一种声明 Aspect 的机制，包括 Aspect 如何声明，连接点(Aspect 代码与其它代码的交互点)如何定义，Aspect 代码如何定义，Aspect 的参数化程度和可定制性等。

其次，需要提供一种将 Aspect 代码和基础代码组合编排(Weaving)在一起的机制，包括定义编排语言和规则，解决 Aspect 之间潜在的冲突，为组装和执行建立外部约束等。

最后，必须有生成可运行系统的实现机制，包括系统提供什么组合机制，是编译时刻静态组装，还是运行时动态决定；对程序单元分别进行编译的模块化编译机制；对 AOP 机制和现有系统兼容性的规约；对组装结果的验证机制等。

AOP 系统还需要其他辅助机制的支持，包括 Aspect 系统调试机制，Aspect 复用机制，Aspect 软件过程指导，Aspect 开发方法学等。

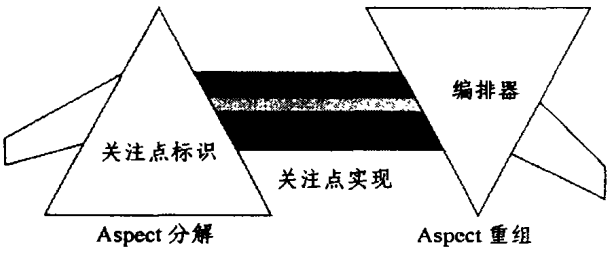

图 5 AOP 的 Aspect 机制

2.4 AOP 和 OOP 的关系

考察程序设计技术的发展历程，如表 1[6]所示。结构化编程提供了显式控制结构的概念，与之对应的程序结构体是 DO..WHILE，FOR 等循环体。其后出现的程序设计技术都包含了结构化编程的控制结构概念和循环体结构。之后模块化编程、抽象数据类型等新技术都继承了以前形成的有效概念和程序结构。面向对象编程提供了类、对象等和以前迥然不同的程序结构，然而循环体、接口、类型等依然存在。

表 1 程序设计技术及其关键概念和构造体

程序设计技术	概念	程序结构
结构化编程	显式的控制结构	循环体
模块化编程	信息隐藏	良好定义的模块接口
抽象数据类型	数据表示隐藏	类型
面向对象编程	对象，分类，特化	类，对象，多态

因此，AOP 也不会抛弃现有的 OOP 和结构化程序设计等技术的思想精髓。AOP 构建在 OOP 系统上，可以说 OOP 是 AOP 的技术基础，AOP 是对 OOP 的继承和发展。相比于以前的程序设计方法学，AOP 更好地将关注点分离，从而可以模块化地实现贯穿特性。当使用 AOP 编程时，可以结合需求/问题的特点，选择合适的实现结构：过程、对象或者 Aspect。

AOP 方法弥补了 OOP 处理贯穿特性的缺陷，是程序设计方法学的下一个明显进步。

3 主流 AOP 技术

AOP 是许多研究人员独立研究路径的交点。目前关于 AOP 的相关的工作有 AspectJ[8]、自适应程序设计[4]、组合过滤器[3]、多维关注点分解[2]等。各种 AOP 技术的一个主要的区别，在于其将程序和 Aspect 结合的方法不同。这里将分别简单介绍这些技术，其中主要介绍 AspectJ 技术。

3.1 AspectJ

3.1.1 概述 AspectJ 是 Xerox PARC 开发的基于 Java 语言的 AOP 扩展，它既是一种规约语言，也是一种 AOP 的实现语言。AspectJ 使用 Java 语言实现单个关注点，并通过对 Java 进行扩展，提供了编排(Weaving)规则。

作为规约语言，AspectJ 定义了支持“面向 Aspect”概念的如下语言结构及语义：

- Joinpoints：AspectJ 的核心概念，预定义好的程序的特定执行点。例如，对某个类的某个方法的调用入口。
- Pointcuts：对 jointpoints 进行声明的语言结构。
- Advices：要在 pointcuts 执行的 Aspect 代码。例如，在执行某个 jointpoint 前，将一条消息记录到日志中。
- Aspect：上述三者的结合。以类似于类(class)的概念，将 pointcut 和 advice 组合在一起，形成一个程序单元。

AspectJ 语言结构扩展了 Java 语言，因此所有合法的 Java 程序同样也是合法的 AspectJ 程序。AspectJ 编译器生成和标准 Java 字节代码一致的 .class 文件，任何标准 JVM 都可以解释执行其生成的代码。由于选择了 Java 作为自己的语言基础，AspectJ 就具有 Java 语言的所有优点，方便 Java 程序员的使用。

作为一种实现语言，AspectJ 为程序员提供了编译、调试等工具，包括 Aspect 编排器(Aspect 编译器)；Aspect 调试器；文档自动生成工具；独立的 Aspect 浏览器，可以以可视化的形式显示“advice”代码在系统中编排后的情况。此外，AspectJ 可以很好地和一些流行的 IDE 环境集成，包括 Sun 的 Forte，Borland 的 Jbuilder，还有 Emacs 等。这为 Java 程序员提供了比较方便和强大的 AOP 支持工具。

AspectJ 的 Aspect 机制很强大，可以引入新的数据成员和新的方法，甚至可以声明一个新的类去实现其它基类和接口。AspectJ 的编译器，将不同 aspect 组装到一起，生成的系统可以在任何 JVM 上运行。

由于 AspectJ 是从标准 Java 语言扩展而来，类似于 C++ 之于 C，因此，AspectJ 可以较容易地获得广大 Java 程序员的支持。又由于其提供了一套编译、调试等工具，可以较容易地应用于实际系统中，可以说，AspectJ 是目前较为成熟的一种 AOP 技术。

3.1.2 应用示例 图 6 给出了 AspectJ 对 1.1 节 DisplayUpdating 贯穿特性的解决方法。

```
aspect DisplayUpdating{
pointcut move();
    call(void Line.setP1(Point))||
    call(void Line.setP2(Point))||
    call(void Point.setX(int))||
    call(void Point.setY(int));
    after() returning:move(){
        Display.update();
```

```
    }
}

class Line{
    private Point_p1,_p2;
    Point getP1() {return_p1;}
    Point getP2() {return_p2;}
    void setP1(Point p1){
        this._p1=p1;
    }
    void setP2(Point p2){
        this._p2=p2;
    }
}

class Point{
    private int_x1,_x2;
    int getX1() {return_x1;}
    int getX2() {return_x2;}
    void setX1(int x1){
        this._x1=p1;
    }
    void setX2(int x2){
        this._x2=x2;
    }
}
```

图6 AspectJ 实现 DisplayUpdating 贯穿特征

在图6中，DisplayUpdating 需求由“DisplayUpdating”aspect 实现。该 aspect 包括两部分内容：首先声明 pointcut move ( ) 为对 Point. setX 等4种方法的调用，“call (void Line. setP1(point))”表示调用 Line 对象的 setP1(Point)方法，“||”符号表示这4种调用声明是逻辑或的关系。其次要给出在该 pointcut 声明的 jointpoint 处要执行的 advice 代码，定义为“after() returning: move() {Display. update();}”，意思是从 move 声明的 jointpoint 处执行完原来的计算代码返回时，要调用一次操作：“Display. update()”，用该操作完成屏幕更新。

Line 和 Point 的实现代码如图6下框图所示，可以看出，Line 和 Point 中已经没有了图2中在每次更新图元操作后对“Display. Update()”的调用。这样，各个模块单元分别很好地实现了自己的关注点，避免了贯穿特性在传统方法中的“代码散布”和“代码交织”现象。

最后，AspectJ 通过编译器，将 aspect 代码和类代码编排到一起，形成最终系统。

3.2 自适应方法

美国西北大学 Demter 项目小组的开发人员，提出了一种“自适应方法”(Adaptive Methods)，解决面向对象技术对贯穿特性的处理问题。他们认为，对象只应了解和自己紧密相关的其他对象，因此在新的程序结构的工作方面，将对象行为概念的表达从对象结构概念中分离出来，通过抽象类的结构，来避免贯穿特性所产生的“代码散布”和“代码交织”问题。

自适应方法的核心是“遍历＋访问”，即把操作的行为表达为对如何到达计算参与者(遍历策略)，加上到达参与者时采取什么动作(自适应访问)的高层描述。遍历策略和自适应访问都只涉及参与操作的类的最小集合，这些类之间的复杂关系被抽象出来。Java 的反射机制为这种抽象提供了很好的方法：类的结构可以在运行时直接得到。

Demeter 小组提供了 DJ 库——一个 Java 代码库，来支持自适应方法。DJ 使用遍历策略语言标识对象之间的导航；使用 Visitor 模式支持对象访问。DJ 允许命名遍历和访问者，并可以复用。

DJ 的一个显著特点是对遍历的编程非常简单。然而 DJ 过分依赖于程序语言的反射机制，执行速度很慢。此外，DJ 是纯 Java 的，没有对 Java 做任何语法扩展。因此，DJ 是 Java 程序员理想的一个快速原型工具，简单易用。

3.3 多维关注点分解

IBM 的一个研究小组提出了多维关注点分解(Multi-Dimensional Separation of Concerns：MDSOC)的思想，并提供了相关支持工具 Hyper/J。

文[2]认为，分离关注点在软件工程中占有核心地位。不同的开发者，不同的角色，软件生命周期的不同阶段各自有不同类型的关注点。他们把关注点的类型称为关注点的维度。因此，关注点分解主要就是将软件在不同的维度上进行分解。清晰的关注点分解可以降低软件复杂性，提高程序的可理解性，保持软件制品在整个软件生命周期内的可追踪性，可以限制修改的变动范围，利于演化和复用。而现代的程序设计语言和方法学只允许按照一种维度进行分解，无法达到上述目标。MDSOC 的提出，正是为了克服这种“主流分解技术”的不足。

对于一个待开发对象，开发者可以从多个角度进行描述和理解。比如，反映对象内在结构和属性的“数据关注点”；反映对象行为特征的“特征关注点”；对象的“事务规则”，对象的“非功能特性”等等。尽管它们有重叠，但因为它们是从不同方面塑造系统，所以所有的 Aspect 可以和原来的模块共存。这种方法最终突破了传统的线性结构的编程。

应用 MDSOC，开发者不需要知道或者关心和他们自己的活动不相干的关注点或者维度。开发者除了可以将贯穿特性模块化，还可以同时标识和封装不同种类不同维度的关注点，所有维度的地位相同。开发者还可以在软件生命周期的任何时候标识新的关注点，从而递增地增加新维度。MDSOC 最后提供了将各种不同维度的关注点集成到一个最终系统的机制。

Hyper/J 是一个支持 MDSOC 的语言工具，它同样建立在 Java 语言上。

3.4 组合过滤器

Twente 大学的 Mehmet Aksit 及其小组是基于过滤器(Compositional Filter)的 AOP 方法最早和最突出的倡导者。他们在上世纪80年代晚期，开发了过滤器方法，以表达一种通用的数据抽象机制。后来对之进行扩展，可以用一种可组装的形式表达各种系统关注点，比如同步、对象间通信、实时规约等。通过对基本构件进行过滤器包装，在显式的过滤器中体现 Aspect。

组合过滤器建立在对象间的本地消息传递机制上，基本思想是通过操纵对象接收或发出的消息来对贯穿特性进行编码，并将非贯穿特性留给对象实现。这种方法非常类似于一种称为调用截取器(Interceptor)的设计模式。

结束语 AOP 起源于程序设计中“贯穿特性”引发的“代码散布”和“代码交织”问题。

贯穿特性是大多数复杂系统的固有问题。在目前的程序设计技术下，程序中的贯穿特性无法避免，其实现代码相互纠缠在一起，是软件复杂性的来源之一。当程序员们要实现一个贯穿特性时，他们被迫编写交织代码。AOP 就是要能分离出那些隐含的、相互交织纠缠的系统关注点，并使之明确，这样开发者就可以看到他们在干什么。使用 AOP 为程序员们提供的新的模块化武器，贯穿特性的代码就可以局部化，程序的结构很明白，代码也常常很简短。

然而，如果 AOP 对系统关注点的分离粒度太细，则会适得其反，导致各模块之间关系更为复杂，反而使复杂度增加。

因此，应用 AOP 时仍然要遵从良好的设计原则，只分离需要分离的关注点。

关于贯穿特性的一个重要问题是理解谁"贯穿"了谁。实际上，发生"贯穿"关系的双方是互相贯穿的。分别从商业业务功能关注点的角度和安全、日志等系统关注点的角度，会得到完全相反的"贯穿"结果。由于目前人们对系统进行分解时，多数将商业业务功能作为主导维度，因此，安全、日志等系统关注点就是主要的"贯穿特性"。

AOP 正在迅速发展，其思想已经超越了程序设计阶段，正应用于软件生命周期的不同阶段。AOP 正在许多领域得到应用，比如中间件、安全、容错、服务质量、操作系统等。不过，AOP 目前并没有完全成熟，各种概念和实现机制尚在不断完善中。在实践中，AOP 没有得到工业界的大规模推广，没有经历开发大型系统的实际考验，因此，目前断言 AOP 有效与否尚为之过早。我们需要更深入地研究和实验，回答下述若干问题：AOP 对大型项目有效吗？它最适合解决什么问题？什么构造形式最适于声明"贯穿特性"？这些问题的回答，需要学术界和工业界的广泛参与，需要更多应用的支持。

我们北京大学软件研究所在研究基于软件体系结构的构件组装过程中，将 Aspect 的概念引入到软件体系结构和构件运行平台中。通过在 ABC/ADL 中加入相应的语言元素，在设计阶段描述系统贯穿特性[10]。同时，在构件运行平台 PKUAS 上[14]，通过构件容器的截取器(Interceptor)机制，提供在运行时刻动态组装编排 Aspect 的机制。PKUAS 中的安全、事务等，都通过这种方式实现。

可以肯定的是，AOP 不会是处理关注点分离问题的最后解决方案。新的更有效的程序设计技术仍将在现有的基础上涌现，从而推动软件工程不断进步。

参 考 文 献

1 Kiczales G, Lamping J, et al. Aspect-Oriented Programming. In: Proc. of ECOOP'97, 1997
2 Ossher H, Tarr P. Using Multi-dimensional Separation of Concerns to (Re)Shape Evolving Software. CACM, 2001, 44(10)
3 Bergmans L, Aksit M. Composing Crosscutting Concerns Using Composition Filters. CACM, 2001, 44(10)
4 Lieberherr K, Orleans D, Ovlinger J. Aspect-Oriented Programming With Adaptive Methods. CACM, 2001, 44(10)
5 Elrad T, Filman R E, Bader A. Aspect-Oriented Programming. CACM, 2001, 44(10)
6 Elrad T, Aksit M M, Kiczales G, Lieberherr K, Panelists H O. Discussing Aspects of AOP. CACM, 2001, 44(10)
7 Pace J A D, Campo M R. Analyzing The Role of Aspects in Software Design. CACM, 2001, 44(10)
8 Kiczales G, et al. Getting Started with AspectJ. CACM, 2001, 44(10)
9 Constantinides C A, et al. Designing an Aspect-Oriented Framework in an Object-Oriented Environment. ACM Computing Surveys (CSUR), March 2000
10 Mei Hong, et al. ABC/ADL: An ADL Supporting Component Composition. In: Proc. of ICFEM 2002
11 Laddad R. I want my AOP! http://www.javaworld.com/javaworld/ jw-01-2002/jw-0118-aspect.html
12 Aspect-Oriented Software Development Web site. http://aosd.net
13 邵维忠，杨芙清. 面向对象的系统分析. 北京：清华大学出版社，1998
14 黄罡，王千祥，曹东刚，梅宏. PKUAS：一种面向领域的构件运行支撑平台. 电子学报，已录用

(上接第4页)

类别的文本进行识别。文档聚类是将文档集合分成若干簇，要求同一簇内的文档内容的相似度尽可能地大，而不同簇间的相似度尽可能地小。互联网动态信息挖掘主要挖掘互联网内容、结构和访问模式的变化。

用户信息挖掘是从用户对互联网的使用方式和行为中挖掘有用的模式，用于对互联网信息的合理组织和服务质量的改进。挖掘的结果通常是用户群体的共同行为和兴趣，以及用户个人的检索偏好、习惯和模式等。通过用户信息挖掘，可以进行主动服务，改善服务质量。

多策略数据挖掘平台 MSMiner[7]是由中国科学院计算技术研究所智能信息处理重点实验室研制的，它提供决策树、支持向量机、粗糙集、模糊聚类、基于范例推理、统计方法、神经计算、可视化等多种数据采掘算法，支持特征抽取、分类、聚类、预测、关联规则发现、统计分析等数据采掘功能，并支持高层次的决策分析功能。

结束语 智能互联网是互联网发展的一种模式，它有效地将智能融入到互联网中，让具有智能的计算机程序在互联网这种动态开放的无限网络环境中运作，共享网络资源，为人们提供高质量的服务。智能互联网将在语义网格的架构下，利用主体、资源管理、服务管理、数据挖掘等技术，开创互联网的新局面，提供智能化的信息基础设施[11]。

参 考 文 献

1 Berners-Lee T, Oralassila J H. The Semantic Web. Scientific American, May, 2001
2 Zhong Ning, Liu Jiming, Yao Yiyu. In Search of the Wisdom Web. Computer, IEEE CS, 2002, 35(11): 27～31
3 Baader F, Nutt W. Basic description logic. In: F. Baader et al. eds. The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press, 2002
4 Dong Mingkai, Zhang Haijun, Shi Zhongzhi. Dynamic Description for Agent Programming. In: Proc. of CRBCIIP 2002. 11～19
5 史忠植. 智能主体及其应用. 科学出版社, 2000
6 http://www.intsci.ac.cn/
7 史忠植. 知识发现. 清华大学出版社, 2001
8 Czajkowski K, Foster I. A Resource Management Architecture for Metacomputing Systems. In: Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, 1998
9 The DAML Services Coalition, DAML-S: Web Service Description for the Semantic Web. In: The First Intl. Semantic Web Conf. (ISWC), Sardinia, Italy, June, 2002
10 蒋运承，张海俊，董明楷，史忠植. 多主体系统中的动态服务匹配. 已投软件学报, 2002
11 史忠植. 高级计算机网络. 电子工业出版社, 2001