面向切面编程 1.0.1

**声明：** 这份文档翻译于ECOOP官方[英文版](http://www.ifs.uni-linz.ac.at/~ecoop/cd/papers/1241/12410220.pdf)。此文档用 Markdown 格式编写，你可以到[这里](https://github.com/runninghack/Aspect-Oriented_Programming/blob/master/Aspect-Oriented%20Programming_zh.md)查看它的[源文件](https://github.com/runninghack/Aspect-Oriented_Programming/blob/master/Aspect-Oriented%20Programming_zh.md)。--By @[Ramiel](file:///C:\\Users\\Ramiel\\Desktop\\写作\\Aspect-Oriented_Programming\\zl.ramiel@gmail.com) / 本项目托管于[Github](https://github.com/runninghack/Aspect-Oriented_Programming)

我们发现很多编程问题，无论是用过程式编程语言还是面向对象编程语言，都不足以清晰的捕获程序必须实现的重要决策设计。这就迫使那些设计决策必须分散在代码中，产生了难以开发和维护的混乱代码。我们分析了代码难以清晰捕获特定决策设计的原因。我们将这些决策所表达的性质称为*切面*（即*aspect*），并揭露了它们难以被捕获的原因——它们是对系统功能的*横切*（*cross-cut*）。我们所呈现的是一种新的编程技术，名为面向切面编程（aspect-oriented programming，简称aop）。这种技术可以清晰的表达这些切面：适当的隔离、切面代码的布置和重用。接下来的讨论是基于我们用aop搭建的系统。

1. 简介

面向对象编程可以从根本上给软件工程以支撑，因为底层的对象模型更适应真实的领域问题。但是我们发现很多编程问题是，使用OOP技术难以清晰的捕获程序必须实现的重要决策设计。相反，似乎有一些编程问题既不适合 OOP 方法，也不适合过程式方法。

这篇论文展示了我们研究的开发程序技术，这种技术能 OOP （或 POP ）所不及，可以清晰表达 OOP 不能清晰表达的程序。我们分析了代码难以清晰捕获特定决策设计的原因。我们将这些决策所表达的问题称为*切面*（即*aspect*），并展示了它们难以被捕获的原因——它们是对系统功能的*横切*（*cross-cut*）。我们所呈现的是一种新的编程技术，名为面向切面编程（aspect-oriented programming，简称aop）。这种技术可以清晰的表达这些切面：适当的隔离、切面代码的布置和重用。

我们认为现阶段 AOP 的研究正如20年前的 OOP 研究。基本的概念正在成型期，不断有更多的研究者把它们投入项目中 [1, 4, 13, 28]。此外，对于 AOP 来说 OOP 是一个新概念，许多现存系统都有类似于 AOP 的特性。本篇论文的贡献在于分析 AOP 将要解决的问题，以及初次明确提出用来支持 AOP 系统设计所必须的名词和概念。

本篇论文用示例来展示 AOP，概括和定义不是在其前面给出，而是从示例中获取。第3节使用一个中等规模的示例展示 AOP 所解决的切面复杂问题；这一节最后给出了切面术语的定义。第4节展示了几个更小的切面示例。第5、6节各展示了一个完整的 AOP 系统示例。余下的章节展示了前景、相关工作和结论。

2. 背景假设

本章概述编程语言和软件设计过程之间的关系，此关系亦即本篇论文所述内容的基础。

软件设计过程和编程语言相互支持。设计过程把系统分解为越来越小的部件。编程语言使得程序员能够定义系统下级构件的抽象，然后将这些抽象以不同方式组合成整个系统。当编程语言提供的抽象和组合机制可以简洁地支持设计过程产生的子构件时，设计过程和编程语言两者才称得上是合作良好。

这样看来，许多现存编程语言，包括面向对象语言、过程式语言和函数式语言，它们在关键抽象和组合机制有共同的机理，都源于对一些形式的归纳过程。为顺应本篇论文的论点，我们把这些都称为归纳过程（generalized-procedure，简称GP）语言。（我们并非不知道OOPL的许多优势！只是为了本篇论文的论点，只关注所有GP语言的共同点会简单一些。）

GP 语言所演化出的设计方法把系统划分为行为或功能构件。这种形式称为功能分解（functional decomposition）[25-27]。分解的确切性质随过程语言实例不同而不同，但是每个部件浓缩进一个过程/函数/对象。对各种情况，都把系统的功能压缩为构件，只关注构件会让人感觉更轻松。最后一点有些老生常谈，以至于感觉多余。但这种不厌其详的关注是重要的，因为这篇论文将要考虑的是非功能性系统分解。

3. 什么是切面？

为了更好的理解复杂问题，以及 AOP 是如何解决它们的，这一节将主要围绕一个具体实例，这个实例是一个我们使用过的现实程序[18,22]。这个现实程序的实现有三种：易理解低效率、难理解高效率、和一个易理解高效率的 AOP 实现。这里展示的是三个相似但经过简化的实现。

3.1 基础功能

实现第一个目标相对比较容易。优秀的传统过程编程可以实现程序，并做到整洁、简明、与领域模型一致。通过这种途径，过滤器可被定义为接受图像和输出图像的过程。一组原始流程可实现基本过滤器，高级过滤器定义于原始过滤器之上。例如，一个原始 or! 过滤器，接受两个图像，返回它们的逻辑逐像素 or，也就是实现为：

(defun or! (a b)

(let ((result (new-image)))

(loop for i from 1 to width do

(loop for j from 1 to height do

(set-pixel result i j

(or (get-pixel a i j)

(get-pixel b i j)))))

result))

从 or! 和其他原始过滤器开始，程序员定义了一个过滤器选择水平边上的黑色像素，返回包含边界像素的新图片。

|  |  |
| --- | --- |
| **functionality** | **implementation** |
| pixelwise logical operations | written using loop primitive as above |
| shift image up, down | written using loop primitive;  slightly different loop structure |
| difference of two images | (defun remove! (a b)  (and! a (not! b))) |
| pixels at top edge of a region | (defun top-edge! (a)  (remove! a (down! a))) |
| pixels at bottom edge of a region | (defun bottom-edge! (a)  (remove! a (up! a))) |
| horizontal edge pixels | (defun horizontal-edge! (a)  (or! (top-edge! a)  (bottom-edge! a))) |

注意，只有原始过滤器和图像中的像素的循环密切相关。高级过滤器，比如 horizontal-edge!，由原始过滤器进行简洁的定义。返回的代码容易阅读、推理、debug和扩展，简而言之，实现了最初目标。

3.2 优化内存使用

但是这个简单的实现没有完成优化内存使用的第二目标。每次过程被调用，都要循环输入图片产生新图片。输出图片经常被创建，却往往刚创建就被其他循环使用了。频繁的内存检索和空间分配的结果导致了 cache 丢失、页面错误和糟糕的表现。

这个问题的一般解决方法是在程序中使用更多的全局变量，筹划最终输入别的过滤器的中间结果，新版本的程序适当的融合循环，务必在实现原有功能的前提下创造尽量少的中间图形。修改后的代码 horizontal-edge! 大概是这样的：

(defun horizontal-edge! (a)

(let ((result (new-image))

(a-up (up! a))

(a-down (down! a)))

(loop for i from 1 to width do

(loop for j from 1 to height do

(set-pixel result i j

(or (and (get-pixel a i j)

(not (get-pixel a-up i j)))

(and (get-pixel a i j)

(not (get-pixel a-down i j)))))))

result))

和原版本相比，这段那代码全都混在一起了。horizontal-edge! 把所有不同的过滤器合并在一起，它把很多，而不是全部，的循环合并在一起了。（up! 和 down! 的循环没有合并，因为它们使用了不同的循环结构。）简而言之，为了高效，重写代码使得原代码失去了简洁的组成结构。

当然，这只是一个非常简单的例子，少量的复杂代码我们依然能够处理。但是在实际项目中这种混乱带来的复杂性迅速膨胀，足以变成代码开发和维护的巨大障碍。这个例子的来源是一个视觉字母识别系统的重要子构件。实际系统如果采用前面所述的整洁代码，只有768行代码；但如果采用复杂实现（使用整合优化，例如中间结果缓存、编译时内存分配、专门中间数据结构），则代码有35213行。这样的复杂代码非常难以维护，功能小小的调整也需要先解开复杂代码，然后重新混合。

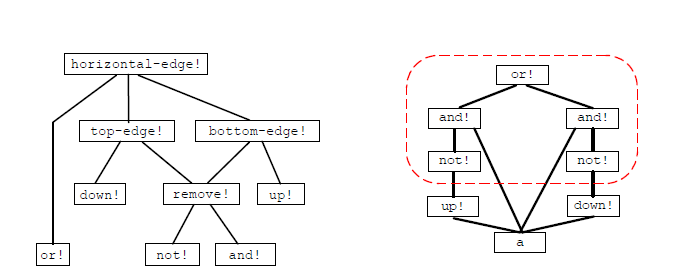


图1：未优化的 horizontal-edge! 过滤器图解

左边是功能分解，可以发现和功能模型非常一致。右边的是数据流图，框内的是原始过滤器，线是过滤器在运行时的数据流。标着 a 的方框是输入图片。

3.3 横切

回到最初的示例代码，图1提供了理解代码混合的基础。左图是过滤器功能的分层结构。右边是原始未优化版本 horizontal-edge! 的数据流图。在这个图中，方框和线表示原始过滤器和过滤器之间的数据流。虚线椭圆表示融入优化版 horizontal-edge! 中单循环的边界。

我们可以发现 horizontal-edge! 并不都在融合椭圆之内。实际上，它没有和左边的分层构件联合。当这两种属性实施——功能和循环混合——都始于相同的原始过滤器，它们必须像过滤器一样有差别的组合。功能以传统形式分层组合。但是循环融合的形式为把具有相同循环结构并且在数据流图中相邻的原始过滤器组合。单独看每种组合规则各自使用的图都非常简单。但是两种组合关系从根本上相切，以至于从另一个的图中很难看到自己。

这种横切现象是代码混乱的直接原因。语言提供给我们的单一组合机制——过程调用——很适合创建非优化功能构件。但是它不能帮我们同时创建功能构件和循环复合，因为那要求不同的组合规则相互组合。这个断层迫使我们完全手动组合属性——这就是上面的混合代码所发生的。

一般当两个属性的编码有不同的组合，又必须相互协调，我们说它们相互*横切*（*cross-cut*）。由于 GP 语言只提供了一种组合机制，程序员必须手动进行组合，就导致了代码的复杂和混乱。

我们现在给两个名词以精确的定义：

对于 GP 语言实现的系统，一个属性必须实现：

**构件（component），即能以整洁的形式压缩进通用过程的部分**。所谓整洁，指的是容易定位，如有必要可以容易的访问和组合。构件一般是系统的功能分解构件，比如图形过滤器，银行账户和 GUI 构件。

**切面（aspect），即不能以整洁的形式压缩进通用过程的部分**。切面一般不是系统的功能分解构件，而是关于性能或系统整体部件的语义。例如内存访问模式和同步对象的同步。（第4节提供了切面的更多示例。）

有了这些名词，就可以清楚的表达 AOP 的目标了：帮助程序员清晰的区分构件和切面，提供了一个机制可以抽象和组合构件和切面，从而生成整个系统。相比 GP 编程，GP 只能让程序员区分不同的部件，通过抽象组合构件这种机理实现系统。

4. 切面横切构件的其他例子

在介绍 AOP 以及它解决代码切面混乱的方法之前，这一节简要叙述几个切面和构件的例子。下面表格中对每个例子我们列出一个程序、一种能获取程序的构件结构然后完成工作的 GP 语言、如果使用这种语言可能有的构件结构、以及可以横切构件结构的切面。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **application** | | **GP language** | **components** | **aspects** | |
| image  processing | | procedural | filters | loop fusion  result sharing  compile-time memory  allocation | |
| digital  library | | object-oriented | repositories,  printers,  services | minimizing network  traffic  synchronization  constraints  failure handling | |
| matrix  algorithms | | procedural | linear algebra  operations | matrix representation  permutation  floating point error | |
|  | | | |

有些切面非常普通，能够不涉及具体领域而分出来。最好的一个例子是错误处理。对于这种现象我们很熟悉：给一个系统原型添加错误处理的支持，结果需要给系统进行许多细小的添加和修改。这是因为可能导致错误的不同动态上下文和涉及如何处理错误，是对系统功能的横切。

许多性能相关的问题是切面，因为性能优化通常跨构件对运行上下文进行信息采集。

5. AOP 的第一个示例

这一节我们回到图像处理的例子，使用它来概括程序的 AOP 重构。接下的演示基于我们开发的一个系统，但经过了一定程度的简化。完整系统的讨论见[22]。本节的目标是快速获取表格中基于 AOP 实现的完整结构，而不是对结构进行详尽的解释。详尽的解释在第6节的内容中。

一个程序基于 AOP 实现的结构与基于 GP 的实现是相似的。但是基于 GP 的程序实现包含：(i) 一种语言，(ii)一个编译器（或解释器）和(iii) 用该语言写的改程序的实现；基于 AOP 的程序包括：(i.a) 一种用来编写构件的构件语言，(i.b) 一种或多种编写切面的切面语言，(ii) 一个对所结合语言的切面织入器，(iii.a) 一个构件程序，它使用构件语言来实现构件，和(iii.b) 一个或多个切面程序，它们使用切面语言来实现切面。就像 GP 语言，AOP语言和织入器即可设计成把织入工作推迟到运行时（RT weaving），也可以设计成编译时（CT weaving）。

5.1 构件语言和程序

在当前的例子中我们使用一种构件语言和一种切面语言。构件语言类似于前面所述的流程语言，只有少量改变。首先，过滤器不一定是过程了。然后，原始循环尽可能写得清晰。使用新的构件语言则 or! 过滤器是这样的：

(define-filter or! (a a)

(pixelwise (a b) (aa bb) (or aa bb)))

pixelwise结构是一个迭代器，在这里在一步中经过a和b，把aa和bb的绑定到像素值上，并返回包含结果的图像。四个相似的结构提供了系统所需的像素的聚合、分配、转移和组合。引入这些高级循环结构很有意义，使得切面语言可以更容易的检测、分析和融合循环。

5.2 切面语言和程序

这个程序中切面语言的设计是基于图1的数据流图——这个图让所需的循环融合更容易理解。该切面语言就是提供了数据流图节点简单操作的普通过程语言。切面程序可以直接找到需要融合的循环，然后施行所需的融合。接下来的代码段是该切面程序源代码的一部分——它解决了第5节讨论的融合。该代码检测一个数据流线所连的两个节点是否有逐像素结构，是否融合到一个循环仍具有逐像素结构，将输入、循环变量和两个循环体进行适当的合并。

(cond ((and (eq (loop-shape node) ’pointwise)

(eq (loop-shape input) ’pointwise))

(fuse loop input ’pointwise

:inputs (splice …)

:loop-vars (splice …)

:body (subst …))))

描述实际系统中的五种循环的组合规则和融合结构需要一组类似的条件来描述融合的时机和方式。这也是系统为什么不用优化编译器来融合的原因之一——程序分析和理解太重要了，编译器不够可靠。（虽然许多编译器能优化这个优化我们的简单例子。）另一个难题是实际系统处理的其他切面，包括中间结构的分享和保持运行时内存不超标。

5.3 织入

切面织入器获取构件和切面程序当做输入，生成的输出是一个c 程序。这项工作从三个不同的方面进行，如图2所示。

图1中织入器使用展开从构件程序中生成一个数据流图。在这个图中，节点表示原始过滤器，边表示原式过滤器之间图像的流动。每个节点包含一个循环结构。所以，比如我们用 #<...>表示连接节点的边，A节点包含下面的循环结构：

(pointwise (#<edge1> #<edge2>) (i1 i2) (or i1 i2))

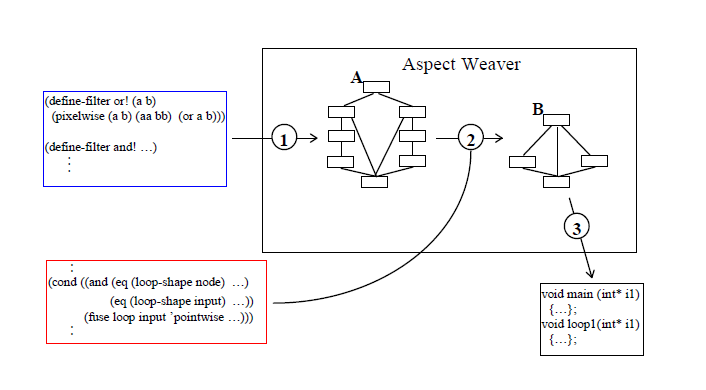


图2：从三个不同的角度对图形处理程序进行切面织入

图2中切面程序运行编辑图像，它把节点折叠在一起，并相应调节实体。结果是比循环结构有比图2中更多的原始像素操作。例如，B 节点在原图中有五个循环的融合，代码如下所示：

(pointwise (#<edge1> #<edge2> #<edge3>) (i1 i2 i3)

(or (and (not i1) i2) (and (not i3) i2))))

最后，在第三步，一个简单的代码生成器通过融合图像，对每个循环节点产生一个C 函数，并产生一个主函数依次调用循环函数，从上一个循环中传递相应的结果。代码生成器非常简单，这是由于每个节点包含单一的循环结构，每个循环结构的代码包含像素的全部操作。

该系统的一个残酷的特性是织入器不是“智能”编译器，很难设计和构建。使用 AOP，我们使用适当的切面语言，为所有的重要实现策略决策做了准备——所有智能——提供给程序员。织入器的工作是集成，而不是鼓动。

5.4 结果

真实系统比这要复杂的多。比如有两个额外的切面程序，一个负责共享公共子运算，另一个确保同一时间尽可能少的图像被分配。在这个系统中，三个切面程序都用同一中切面语言编写。

在这个例子中，AOP 的重构实现了最初的设计目标——程序代码容易分析、开发和维护，同时又有高效率。程序员很容易理解构件及其组合方式。程序员很容易理解最终代码里面切面程序的作用。过滤器构件或融合切面的修改都很容易通过简单的重新织入来部署到整个系统。修改输出代码的细节对程序员来说并不简单。AOP 的方法就是织入器负责处理这些细节，而程序员不需要手动乱搞。

我们程序的 AOP 重新实现有1039行代码，包括了构件程序和全部的三个切面程序。切面织入器本身包含一个可重用的代码生成器构件，有3520行（织入器的核心代码有1959行）。重新实现的程序性能可以与35213行代码的手动混合代码相比（时间效率差一些但空间效率更高）。

就行很多其他软件工程项目一样，不用大型试验来研究，很难量化使用 AOP 的优势，这样的试验需要不同的程序员同时使用 AOP 和传统技术来开发维护不同的系统[6,21,36]。虽然我们希望将来可以做一个这样的试验，但这样的研究超过了我们当前研究的视野。与此同时，我们开发了一种度的衡量方法可以说明 AOP 对程序的简化。这中衡量方法比较了相同程序的一种 GP 实现和一种 AOP 实现。它表明 AOP 实现的代码比非 AOP 实现的代码更加简明。以下所列是对一个特定程序的这种衡量方法的等式和数值：

在这个度量中，比1大任何的数均表示使用 AOP 的正面效果。这个程序说明使用 AOP 获得了最惊人的收益，其他的程序中这个收益最小为2 [8,14,22]。也可以说织入器本身的大小也应该加到分母上。这一点尚存争议，因为任意的图形处理程序都可以使用这个织入器，而不仅仅是表处理器这一个。但是即使把织入器也加进去，我们发现这个量依然有9这么大。

任何单一的度量标准都有功能局限。我们相信这个例子中的度量是有用的，因为在性能的其他重要领域，AOP 实现的程序都足以匹敌非 AOP 实现。第7节我们提供了 AOP 应用定量标准度量的一些要求。

6. AOP的第二个示例

本节用 AOP 实现系统的第二个例子来详述构件编程设计、切面编程设计和织入。和前面一样，这个例子是真实系统的简化版，见[14]。这个例子的来源是文档处理领域，我们想要实现一种分布式字符库，可以以不同格式储存文档，并且对文档提供了很多操作。在本节，构件语言、切面语言和切面织入器以通用的方式呈现，而不是像前面章节的高专业领域的例子。

这个系统的功能用面向对象模型可以很好的捕获。用这种方法，对象就是文档、仓库、文档的不同打印格式（pdf,ps,rip...）、打印机、服务器等。关注点有这些：

* 交互。也就是控制程序使用的网络带宽的交互和远程方法调用复制的对象和子对象的交互。例如，我们想确认当一个书的对象在一个远程方法中被调用时，除非接收方法需要，不同的打印表现并不进行传输。
* 协作约束。也就是用来保证构件程序在多线程控制下正确性的同步规则。
* 错误处理。也就是以适当的上下文敏感方法来处理分布式系统的各种错误。

现在，我们将只关注交互切面。使用 AOP 进行交互和协作都在[14]进行讨论。使用 AOP 进行错误处理是将来的目标。

6.1 构件语言和程序

设计 AOP 系统我们要搞清楚构件语言处理什么、切面语言处理什么以及两种语言共同处理什么。构件语言必须运行程序员写构件程序来实现系统功能，同时确保这些程序在切面程序需要控制的时候不乱插手。切面语言必须自然而精确的支持要求切面的实现。构件和切面语言有不同的抽象和组合机制，但是他们也必须有共同术语，这样才能使织入器组合两种不同的程序。

为了不让通用术语成为争论点，切面语言（而不是构件语言）必须解决不同的问题。在图像处理系统中，把低级循环换成高级原始循环就是确保构件程序不先占切面程序的例子。这种修改让切面程序更容易检测和实现循环融合的机会。

在这个示例中，构件程序必须实现图书、仓库、打印机等元素。为了让交互切面程序能够处理交互，构件程序必须避免处理交互。在这里 Java 作为构件语言做的很好。Java 提供了一个实现适当构件的对象模型，同时避免处理交互切面。因此，使用 Java 作为构件语言，对两个简单类——book和book的repository——的定义如下所示：

public class Book {

String title, author;

int isbn;

OCR ocr;

PDF pdf;

Postscript ps;

RIP rip;

public String get\_title(){

return title;

}

public String get\_author(){

return author;

}

public int get\_isbn() {

return isbn;

}

}

public class Repository {

private Book books[];

private int nbooks = 0;

public Repository (int dbsize)

{

books = new Book[dbsize];

}

public void register (Book b)

{

books[nbooks++] = b;

}

public void unregister(Book b)

{ … }

public Book lookup (String s)

{ … }

}

6.2 切面语言和程序

交互切面程序需要能在远程调用的时候控制所产生变量的复制。要做到这点，切面程序必须能有效进入方法调用的运行，检测它是本地还是远程的，然后对每种情况作出适当数量的复制。

一种实现方法是提供方法调用的运行时反射访问。如[7,23,35,37]所示，反射访问可以控制分布式对象系统的交互切面。但是这种反射访问太过强大，以至于使用起来即困难又危险。所以对这种情况，我们使用一种高级切面语言，对控制远程方法调用复制的特定切面更量体裁衣。

我们设计的交互切面语言运行程序员详细描述对象从远程调用接受参数时应该复制几次。使用这种语言，下面交互切面程序的片段说明了 book 被注册到 repository 时，他们的子对象是如何复制的；当它们取消注册或以一个检查结果返回，只复制 ISBN 号码。除非以后用到，其余的 book 及其包括的大量的子对象（比如可打印描述）不复制。

remote Repository {

void register (Book);

void unregister (Book: copy isbn);

Book: copy isbn lookup(String);

}

6.3 切面织入器

切面织入器必须处理构件和切面语言，合理组织它们来生成系统的所有操作。切面织入器的关键功能是*切入点（join point）*概念，也就是切面程序要处理的构件语言的语义要素。

在图形处理程序的例子中，切入点就是构件程序的数据流。在分布式对象例子中，切入点则是构件程序的运行时方法调用。这两个例子揭示了切入点的关键——它们都不是构件语言必须实现的组织。确切的说，数据流图中的节点和运行时方法调用都是清晰的，而非含蓄的，构件程序的语义元素。

切面织入器工作方式是生成一个构件程序的切入点描述，然后据此执行（或编译）切面程序。在字符库示例中，切入点描述包括动态方法调用的信息，例如所获变量的具体类和类的位置。对构件语言，切入点描述可以在运行时使用反射生成。用这种方法，切面语言的原理就像元程序一样，调用每个方法调用，使用切入点信息和切入程序，来获知集结变量的合适方法。因而我们设计的高级切面语言是基于低级别语言的，这在 GP 语言中很常见。

在图像处理程序中，切入点描述非常简单。其实就是数据流图、获取节点内容的操作和操作图像的操作。

7. 尚未解决的问题

作为一种具体的编程方法，AOP 还很年轻。我们的预计的工作已经主要关注设计和实现面向切面的编程语言，以及使用这些语言开发原型程序。这个以编程为中心的关注点是自发的，正如早期 OOP 的开发一样。但是有更多的工作尚待解决：获得 AOP 的全面应用、与现有想法之间关系的进一步理解、以及让用户广泛应用的进一步开发。

一个重要的目标是对 AOP 应用的定量评估。在实际程序中有多大帮助？我们能否发明一种方法衡量在程序中的帮助有多大？这是很难的一个问题，就像定量评估对于 OOP 也同样困难，但是我们相信现在对这个问题的研究是很重要的，因为取得实在的结果尚需时日。

我们认为对寻找现有系统中像 AOP 的元素进行系统化的研究也是非常重要的。通过提供一种粗略的经验证据，而不必重新创建大型新系统，我们把这当做加快 AOP 思想发展的一个途径。

另一个重要的探索方向是不同构件语言和切面语言的设计。我们能否开发一组构件语言和切面语言，它们可以在不同程序中以不同的方法相互插入？我们能否使用元级框架[2,3,20,38]来创建这样的语言组？

对于AOP会产生什么理论支持？什么理论能最好的描述构件和切面的相互作用和织入方式？我们能否训练人们识别切面？清晰的分别切面？编写切面程序？为 AOP 系统debug？为 AOP 系统编写文档？

另一个探索方向是将 AOP 和现有的策略、方法、工具和开发流程做结合。就像这篇论文展示的例子一行，AOP 可以额对现有技术进行提升。为了实现这个目标，AOP 必须找到一个方法与其他技术进行合并。

8. 相关工作

本节对我们的工作进行简要的论述。首先介绍关联性比较强的，然后是关系不是太密切的。

8.1 对 AOP 的详尽研究

有另外的几个组织已经考虑对 AOP 进行详尽的研究。他们包括：

* Twente 大学的 Mehmet Aksit 等，已经研究出结构过滤器对象模型，这对对象的信息接收和发送提供了控制[1]。根据他们的研究，构件语言是传统的 OOP，结构过滤器机制提供了一种切面语言来控制包括同步和交互的切面。大部分的织入发生在运行时；切入点是到达对象发送和接收的动态信息。
* Oregon Graduate 研究所的 Calton 等，他们对 Synthetix 的研究开发出了高性能、轻型、可适应的操作系统内核[19,28]。在他们的研究中，构件时操作系统内核熟悉的功能元素。切面是基于服务使用方法的常量的主要优化。他们的织入器技术使用了部分赋值来有效专攻部分用例的内核代码。他们的代码在常量为 true 或终止为 ture 时被组织为切入点。
* Northeastern 大学的 Karl Lieberherr 等正在研究的是试图让面向对象程序在任务演化过程[13,15,31]中更具可重用性、更加健壮。根据他们的研究，构件语言包括现在的面向对象语言，比如 C++ 和 Java。精要遍历规范（Succinct traversal specifications）[13]和上下文对象（context objects）[31]提供了可以对付横切变化问题的切面语言。使用 Succinct traversal specifications 的切面程序的织入是面向运行时的，切入点描述粗略来说就是类图。使用上下文对象的切面程序的织入更面向运行时，切入点就是动态方法和函数调用。

8.2 反射和元对象协议

面向切面编程与计算性反射和元对象协议[11,20,24,32,24,28]有很深的联系。反射系统提供了基本语言和（一或多个）元语，用来控制基本语言的语义和实现。元语提供了基本语构件所看不到的计算性的视图，比如整个执行栈，或者指定类的对象的全部调用。因而，它们横切基本层计算。在 AOP 术语中，元语是低级切面语言，它们的切入点是反射系统提供的“钩子”。AOP 是一个目标，而反射是一个强力的工具。

我们已经探讨了 AOP 研究的巨大优势。在原型化 AOP 系统的时候，我们通常始于开发构件语言的简单元对象协议，然后使用它们原型化紧要切面程序。之后，一旦我们搞清切面程序需要做什么，就可以对此开发更详尽的切面语言。

两者的联系片面的证明了第6节所述的，我们提供的切面语言在反射架构之上。类似地，第5节的循环融合切面可以使用CLOS 元对象协议[11,33]的设备的方法组合在一定效率下实现。这种联系也是第8.1节中研究的证据；Demeter 研究和组合组成过滤器研究都被描述为反射设备[16]。

8.3 程序转化

程序转化的目标和 AOP 的目标类似。它们希望能在高级语言中编写正确的程序，然后把程序机械化的转化为具有指定行为但更高效的程序。在这种编程中，程序员想要实现的一些性质写在最开始的程序中。其他性质的添加要依靠不同的转化程序来传递。这种区别在精神上类似于构件和切面程序的区别。

但是对 AOP 来说，构件和切面的概念是新的。他们的术语给系统设计增加了额外价值。而且，有一些切面的转化很自然，有些则不是。转化程序趋向于操作被转化程序的语义。如果需要其他切入点，转化程序就有责任去显示。因为，既然能把一些切面程序放在程序转化基之上，那么就可以从实现工作中把它分离出来。

我们希望对于这个团体研发的转化做一个系统化分析，看哪一个能够提供给不同的切面语言。

8.4 主题编程（Subjective Programming）

很自然的要问：主题编程是不是 AOP 呢？我们认为 AOP 和主题编程有重要的不同。类似于面向对象编程能为不同对象的相同信息自动选择方法，主题编程对不同主题的给定通知可以自动对方法进行组合。这两者所涉及的方法都是 AOP 层面上的构件，因为它们都存在于普通的过程里。甚至有可能在通常过程语言之上同时用面向对象风格和面向主题风格，而不会有明显的混乱。对 AOP 来说就截然不同了。因此，AOP 的切面倾向于关注程序的性能和构件语义，而主题编程的主题倾向于对其他主题添加特性。我们认为主题编程是对 AOP 的补充和兼容。

8.5 其他工程准则

许多工程准则都建立在对整体的多方位分解上。例如，力学工程师使用相同系统的静态、动态和热力模型来设计它。不同的模型互相横切，因为一个系统的不同特性有不同的构成。类似的，一些软件开发工具对多方面分解有良好支持：OMT [29,30]方法的工具使程序员能对对象的工作画出不同的图。

9. 结论

我们已经追溯了现有代码在几种实现的属性上的根本性区别的复杂性。构件系统这样的性质：它的实现可以干净的封装进普通的过程。切面是不能被封装进普通过程的性质。切面和横切构件在系统实现上相互横切。

基于这个分析，我们可以开发面向切面的编程技术来同时支持构件和切面的抽象和构成。AOP 和其他方法的关键区别在于 AOP 让构件语言和切面语言有不同的抽象和组合机制。一个特殊的语言处理器也叫做一个切面织入器，用来协调切面和构件和共同构建。

我们在几个使用 AOP 的测试程序中获得了可喜的成功。AOP 概念框架帮助我们设计系统，AOP 的实现让我们更容易地开发和维护，这是相比传统技术的复杂代码更有效率的实现。

鸣谢

感谢马坤老师的指导。

参考文献

1. Aksit M., Wakita K., et al., *Abstracting object interactions using composition*

*filters*, in proc. *ECOOP'93 Workshop on Object-Based Distributed*

*Programming*, pp. 152-184, 1993.

2. Bobrow D. G., DeMichiel L. G., et al., *Common Lisp Object System Specification*,

in *SIGPLAN Notices*, vol. 23, 1988.

3. Chiba S., *A Metaobject Protocol for C++*, in proc. *Conference on Object-*

*Oriented Programming Systems, Languages, and Applications (OOPSLA*

*95)*, Austin, 1995.

4. Consel C., *Program Adaptation based on Program Transformation*, in

proc. *ACM Workshop on Strategic Directions in Computing Research*,

1996.

5. Harrison W. and Ossher H., *Subject-oriented programming (a critique of*

*pure objects)*, in proc. *Conference on Object-Oriented Programming:*

*Systems, Languages, and Applications*, pp. 411--428, Washington D.C.,

1993.

6. Henry S. and Kafura D., *Software Structure Metrics Based on Information*

*Flow*, in *IEEE Transactions on Software Engineering*, vol. SE-7: 509--518,

1981.

7. Ichisugi Y., Matsuoka S., et al., *Rbc1: A reflective object-oriented concurrrent*

*language without a run-time kernel*, in proc. *International Workshop*

*on New Models for Software Architecture '92; Reflection and Meta-*

*Level Architecture*, pp. 24--35, 1992.

8. Irwin J., Loingtier J.-M., et al., *Aspect-Oriented Programming of Sparse*

*Matrix Code*, Xerox PARC, Palo Alto, CA. Technical report SPL97-007

P9710045, February, 1997.

9. Kiczales G., *Foil for the Workshop on Open Implementation*, Xerox

PARC, Web pages, http://www.parc.xerox.com/spl/eca/oi/workshop-

94/foil/main.html

10. Kiczales G., *Why are Black Boxes so Hard to Reuse?*, Invited Talk,

OOPSLA'94, Video tape, Web pages,

http://www.parc.xerox.com/spl/eca/oi/gregor-invite.html

11. Kiczales G., des Rivères J., et al., *The Art of the Metaobject Protocol*.

Book published by MIT Press, 1991.

12. Kiczales G., Lamping J., et al., *Open Implementation Design Guidelines*,

in proc. *International Conference on Software Engineering*,

(Forthcoming), 1997.

13. Lieberherr K. J., Silva-Lepe I., et al., *Adaptive Object-Oriented Programming*

*Using Graph-Based Customization*, in *Communications of the ACM*,

vol. 37(5): 94-101, 1994.

14. Lopes C. V. and Kiczales G., *D: A Language Framework for Distributed*

*Programming*, Xerox PARC, Palo Alto, CA. Technical report SPL97-010

P9710047, February, 1997.

15. Lopes C. V. and Lieberherr K., *Abstracting Process-to-Function Relations*

*in Concurrent Object-Oriented Applications*, in proc. *European Conference*

*on Object-Oriented Programming*, pp. 81-99, Bologna, Italy, 1994.

16. Lopes C. V. and Lieberherr K., *AP/S++: Case-Study of a MOP for Purposes*

*of Software Evolution*, in proc. *Reflection'96*, pp. 167-184, S. Francisco,

CA, 1996.

17. Maeda C., Lee A., et al., *Open Implementation Analysis and Design*, in

proc. *Symposium on Software Reuse (To Appear, May 1997)*, 1997.

18. Mahoney J. V., *Functional Visual Routines*, Xerox Palo Alto Research

Center, Palo Alto SPL95-069, July 30, 1995, 1995.

19. Massalin H. and Pu C., *Threads and Input/Output in the Synthesis Kernel*,

in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*

:pp 191-201, 1989.

20. Matsuoka S., Watanabe T., et al., *Hybrid Group Reflective Architecture*

*for Object-Oriented Concurrent Reflective Programming*, in *European*

*Conference on Object Oriented Programming* :pp 231-250, 1991.

21. McClure C., *A Model for Program Complexity Analysis*, in proc. *3rd International*

*Conference on Software Engineering*, Los Alamitos, CA, 1978.

22. Mendhekar A., Kiczales G., et al., *RG: A Case-Study for Aspect-Oriented*

*Programming*, Xerox PARC, Palo Alto, CA. Technical report SPL97-009

P9710044, February, 1997.

23. Okamura H., Ishikawa Y., et al., *Al-1/d: A distributed programming system*

*with multi-model reflection framework*, in proc. *International Workshop*

*on New Models for Software Architecture '92; Reflection and Meta-*

*Level Architecture*, pp. 36--47, 1992.

24. Okamura H., Ishikawa Y., et al., *Metalevel Decomposition in AL-1/D*, in

proc. *International Symposium on Object Technologies for Advanced Software*,

pp. 110-127, 1993.

25. Parnas D. L., *Designing Software for Extension and Contraction*, in proc.

*3rd International Conference on Software Engineering*, pp. 264-277, 1978.

26. Parnas D. L., *On a 'Buzzword': Hierarchical Structure*, in proc. *IFIP Congress*

*74*, pp. 336-339, 1974.

27. Parnas D. L., *On the Criteria to be Used in decomposing Systems into*

*Modules*, in *Communications of the ACM*, vol. 15(2), 1972.

28. Pu C., Autrey T., et al., *Optimistic Incremental Specialization: Streamlining*

*a Commercial Operating System*, in proc. *15th ACM Symposium on*

*Operating Systems Principles (SOSP'95)*, 1995.

29. Rational, *Rational Web pages*, Rational Software Corporation, Web pages,

http://www.rational.com

30. Rumbaugh J., Blaha M., et al., *Object-Oriented Modeling and Design*.

Book published by Prentice Hall, 1991.

31. Seiter L. M., Palsberg J., et al., *Evolution of Object Behavior Using Context*

*Relations*, in proc. *Fourth ACM SIGSOFT Symposium on the Foundations*

*of Software Engineering*, pp. 46--57, San Francisco, 1996.

32. Smith B. C., *Reflection and Semantics in a Procedural Language LCS*

*Technical Report*, M.I.T., Cambridge, MA, 1982.

33. Steele G. L., *Common LISP: The Language, 2nd Edition*. Book published

by Digital Press, 1990.

34. Wand M. and Friedman D. P., *The Mystery of the Tower Revealed: A Non-*

*Reflective Description of the Reflective Tower*, in *Proceedings of the*

*ACM Conference on LISP and Functional Programming* :pp 298-307.

ACM, 1986.

35. Watanabe T. and Yonezawa A., *Reflection in an object-oriented concurrent*

*language*, in proc. *ACM Conference on Object-Oriented Programming*

*Systems, Languages, and Applications (OOPSLA 88)*, pp. 306--315,

San Diego, CA, 1988.

36. Yau S. and Collofello J., *Some Stability Measures for Software Maintenance*,

in *tse*, vol. SE-6: 545--552, 1980.

37. Yokote Y., *The Apertos Reflective Operating System: The Concept and its*

*Implementation*, in proc. *Conference on Object-Oriented Programming:*

*Systems, Languages, and Applications*, 1992.

38. Yonezawa A. and Watanabe T., *An Introduction to Object-Based Reflective*

*Concurrent Computation*, in *Proceedings of the ACM SIGPLAN*

*Workshop on Object-Based Concurrent Programming*, *SIGPLAN Notices,*

*24(4)*, Agha G., Wegner P., et al., Eds., 1989.