รายงานวิจัยฉบับสมบูรณ์

โครงการสนับสนุนการออกแบบระบบปฏิบัติการที่สามารถขยายและยืดหยุ่นได้ด้วยโครงร่างเชิงลักษณะ

โดย ผู้ช่วยศาสตราจารย์ ดร.ปณิธิ เนตินันทน์

กรกฎาคม 2549

รายงานวิจัยฉบับสมบูรณ์

โครงการสนับสนุนการออกแบบระบบปฏิบัติการที่สามารถขยาย และยืดหยุ่นได้ด้วยโครงร่างเชิงลักษณะ

โดย ผู้ช่วยศาสตราจารย์ ดร.ปณิธิ เนตินันทน์

กรกฎาคม 2549

สัญญาเลขที่ MRG4780168

รายงานวิจัยฉบับสมบูรณ์

โครงการสนับสนุนการออกแบบระบบปฏิบัติการที่สามารถขยาย และยืดหยุ่นได้ด้วยโครงร่างเชิงลักษณะ

ผู้ช่วยศาสตราจารย์ ดร.ปณิธิ เนดินันทน์
มหาวิทยาลัยกรุงเทพ
คณะวิทยาศาสตร์และเทคโนโลยี
ภาควิชาวิทยาการคอมพิวเตอร์

สนับสนุนโดยสำนักงานคณะกรรมการการอุดมศึกษา
และสำนักงานกองทุนสนับสนุนการวิจัย

กิตติกรรมประกาศ

ขอกราบขอบพระคุณบิดาและมารดาของกระผม ที่ให้ความอนุเคราะห์และเป็นกำลังใจให้ดลอดเสมอมา ภรรยาผู้ร่วมสุขทุกข์เขียงข้างมา 4 ปีและบุตรที่น่ารัก อันเป็นแรงบันดาลใจให้วิริยะ อุตสาหะและมานะพากเพียรในการทำงานวิจัยชิ้นนี้ให้สำเร็จด้วยดีตลอด 2 ปีในการทำวิจัยนี้ ขอขอบพระคุณ โดยเฉพาะ ศาสตราจารย์การวิจัย (Research Professor) Tzilla Elrad แห่งสถาบัน Illinois Institute of Technology, Illinois, U.S.A. ที่คอยให้ความช่วยเหลือ ให้คำแนะนำการดำเนินการวิจัยและตรวจสอบบทความตีพิมพ์เผยแพร่ในการประชุมทางวิชาการนานาชาติ ตลอดจนขอขอบพระคุณสำนักงานคณะกรรมการการอุดมศึกษาและสำนักงานกองทุนสนับสนุนการวิจัยที่ให้การสนับสนุนทุนวิจัยในโครงการวิจัยนี้ สุดท้ายขอขอบพระคุณผู้บริหารมหาวิทยาลัยกรุงเทพ ที่ให้ความอนุเคราะห์สนับสนุนเรื่องเวลาการทำวิจัยและค่าใช้จ่ายในการเดินทางไปเสนอผลงานการวิจัยต่อที่ประชุมทางวิชาการทั้งในประเทศและต่างประเทศตลอดมา

ผศ.ดร.ปณิธิ เนตินันทน์

30 ส.ค. 2549

บทคัดย่อ

รหัสโครงการ MRG4780168

ชื่อโครงการ การสนับสนุนการออกแบบระบบปฏิบัติการที่สามารถขยาย
และยืดหยุ่นได้ด้วยโครงร่างเชิงลักษณะ

ชื่อนักวิจัย ผู้ช่วยศาสตราจารย์ ดร.ปณิธิ เนตินันทน์
มหาวิทยาลัยกรุงเทพ

E-mail Address: paniti.n@bu.ac.th

ระยะเวลาโครงการ 1 กรกฎาคม 2547 – 30 มิถุนายน 2549

ในการพัฒนาระบบซอฟแวร์ เช่น ระบบปฏิบัติการ การมีปฏิกิริยากับส่วนประกอบต่าง ๆ ทำให้มีซับซ้อนมาก และยังเป็นข้อจำกัดให้การนำกลับมาใช้ การปรับแต่งระบบ ตลอดจนตรวจสอบการออกแบบและความถูกต้องของระบบเป็นไปด้วยความยากลำบาก ส่งผลให้ความต้องการใหม่ ๆ ที่จะเพิ่มลงไปในระบบไม่สามารถหลีกเหลี่ยงการออกแบบระบบใหม่หมด เป็นความเข้าใจผิดที่ว่าการนำกลับมาใช้ การปรับแต่งระบบ ตลอดจนตรวจสอบการออกแบบและความถูกต้องของระบบนั้นไม่จำเป็น แต่ซอฟแวร์ระบบต้องถูกออกแบบให้มีความสามารถโดยเฉพาะการนำกลับมาใช้ การขยาย และการปรับแต่ง อย่างไรก็ตามการสนับสนุนในเรื่องดังกล่าวเป็นเรื่องยากที่สามารถทำให้สำเร็จได้โดยใช้หลักการของการเขียนโปรแกรมเชิงวัตถุ (Object-Oriented Programming) การเขียนโปรแกรมเชิงโครงข่าย (Aspect-Oriented Programming) เป็นอย่างหนึ่งที่เสนอขึ้นเพื่อมุ่งที่การแยกส่วนประกอบและลักษณะต่างๆ ในซอฟแวร์ออกจากกันตั้งแต่การเริ่มต้นของวงจรการออกแบบซอฟแวร์แล้วทำการรวมส่วนประกอบและลักษณะต่างๆ เข้าด้วยกันในขั้นตอนของการดำเนินการสร้าง นอกจากนั้นการเขียนโปรแกรมเชิงโครงข่ายสนับสนุนการแยกโครงข่ายต่าง ๆ ในส่วนประกอบของซอฟแวร์ได้อย่างเป็นธรรมชาติ แต่กระนั้นก็ดีวิศวกรรมซอฟแวร์โครงร่างเชิงลักษณะสามารถถูกสนับสนุนได้เป็นอย่างดีหากมีระบบปฏิบัติการที่ถูกสร้างขึ้นบนพื้นฐานของการออกแบบเชิงลักษณะ โครงการวิจัยนี้แสดงให้เห็นความเป็นไปได้ในการใช้โครงร่างเชิงลักษณะช่วยให้การออกแบบระบบสามารถเข้าใจได้ง่าย เพื่อสนับสนุนในการออกแบบระบบปฏิบัติการที่สามารถยืดหยุ่นและขยายได้ โครงร่างเชิงลักษณะใช้หลักสามมิติในการออกแบบ ประกอบด้วยส่วนประกอบย่อย (component) คุณลักษณะ (aspect) และระดับชั้น (layer) ซึ่งกระบวนการนี้สามารถสนับสนุนการนำกลับมาใช้ การปรับแต่ง และการยืดหยุ่นขยาย

คำหลัก ยืดหยุ่นได้ ขยายได้ โครงร่าง โครงร่างเชิงลักษณะ วิศวกรรมซอฟแวร์

Abstract

Project Code: MRG4780168

Project Title: Supporting the Design of Extensible and Adaptable Operating System Using Aspect-Oriented Framework

Investigator: Assistant Professor Dr.Paniti Netinant

Bangkok University

E-mail Address: paniti.n@bu.ac.th

Project Period: 1 July 2004 – 30 June 2006

With software systems such as operating systems, the interaction of their components becomes more complex. This interaction may limit reusability, adaptability, and make it difficult to validate the design and correctness of the system. As a result, re-engineering of these systems might be inevitable to meet future requirements. There is a general feeling that OOP promotes reuse and expandability by its very nature. This is a misconception as none of these issues is enforced. Rather, system software must be specifically designed for reuse, expandability, and adaptability. However, such support is difficult to accomplish using object-oriented programming (OOP). Aspect-Oriented Programming (AOP) is a paradigm proposal that aims at separating components and aspects from the early stages of the software life cycle, and combines them together at the implementation phase. Besides, Aspect- Oriented Programming promotes the separation of the different aspects of components in the system into their natural form. However, Aspect-Oriented software engineering can be supported well if there is an operating system, which is built based on an aspect- oriented design. This research will show an Aspect-Oriented Framework which simplifies system design by expressing its design at a higher level of abstraction, for supporting the design of adaptable and extensible operating systems. Aspect-Oriented Framework is based on a three-dimensional design that consists of components, aspects, and layers. This approach can support reusability, adaptability, and extensibility.

Keywords: Adaptability, Extensibility, Framework, Aspect Orientation, Software Engineering

ผลลัพธ์ (Output) โครงการที่ได้รับทุนจาก สกอ. และ สกว.

1. ผลงานตีพิมพ์ในวารสารวิชาการนานาชาติ

   อยู่ในระหว่างการพิจารณาของ International Journal of Software Engineering and Knowledge Engineering. Skokie, USA.

2. การนำผลงานวิจัยไปใช้ประโยชน์

   -เชิงสาธารณะ

   การวิจัยได้รับความสนใจในระดับนานาชาติ มีการจัดตั้งโครงการร่วมมือในการทำการวิจัยเรื่องอื่น ๆ ที่เกี่ยวข้องระหว่าง Bangkok University และสถาบัน Illinois Institute of Technology, Concurrent Programming research Group ประเทศสหรัฐอเมริกา

   -เชิงวิชาการ

   ได้รับเชิญให้เป็นประธานกลุ่มในการประชุมทางวิชาการนานาชาติ ณ ประเทศสหรัฐอเมริกา หัวข้อ Aspect-Orientation เมื่อเดือนมิถุนายน 2549 นอกจากนั้นยังได้นำความรู้ ทักษะ และผลการวิจัยที่ได้ไปใช้ในการเรียนการสอนวิชา Operating Systems ให้กับนักศึกษาระดับปริญญาตรี สาขาวิชาวิทยาการคอมพิวเตอร์

3. บทความที่ได้รับการตีพิมพ์เผยแพร่ในการประชุมทางวิชาการนานาชาติจำนวน 6 บทความ

   - Paniti Netinant and Tzilla Elrad. "A Framework for Extensible and Adaptable System Software" in *Proceedings of the International Conference on Programming Languages and Compilers (PLC 2005)*, Las Vegas, Nevada, USA, June 2005.
   - Paniti Netinant. "Component + Aspect = an Extensible and Adaptable System Software" in *Proceedings of the International Conference on Software Engineering Research and Practices(SERP 2005)*, Las Vegas, Nevada, USA, June 2005.
   - Paniti Netinant. "Extensibility Aspect-Oriented Framework to Build Agent-Based System Software" in *Proceedings of the 15th International Conference on Software Engineering and Data Engineering (SEDE 2006)*, Los Angeles, California, USA, July 2006.
   - Paniti Netinant. "Extensible and Adaptable System Software" in *Proceedings of the International Conference on Programming Languages and Compilers (PLC 2006)*, Las Vegas, Nevada, USA, June 2006.
   - Paniti Netinant. "Supporting Separation of Concerns to Automation of Code Generation" in *Proceedings of the International Conference on Software Engineering Research and Practices (SERP 2006)*, Las Vegas, Nevada, USA, June 2006.
   - Paniti Netinant. "Building Agent-Based System Software Using Aspect-Oriented Framework" in *Proceedings of the 2006 Electrical Engineering/Electronics, Computer, Telecommunication, and Information Technology (ECTI) International Conference*, Thailand, May 2006.

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACS – Adaptive Computing Systems
AO – Aspect Oriented
AOD – Aspect-Oriented Design
AODSM – Aspect-Oriented Domain-Specific Modeling
AOP – Aspect-Oriented Programming
AOSD – Aspect-Oriented Software Development
AP – Adaptive Programming
API – Application Program Interface
ASDL – Abstract Syntax Description Language
ASOC – Advanced Separation of Concerns
AST – Abstract Syntax Tree
ATR – Automatic Target Recognition
C3I – Command, Control, Communication, and Information
CCM – CORBA Component Model
CDL – Contract Description Language
CF – Composition Filters
CLOS – Common Lisp Object System
CORBA – Common Object Request Broker Architecture
CSS – Cascading Style Sheet
DLL – Dynamic Link Library
DOC – Distributed Object Computing
DOM – Document Object Model
DSL – Domain-Specific Language
DSM – Domain-Specific Modeling
DSVL – Domain-Specific Visual Language
DTD – Document Type Definition
ECBS – Engineering of Computer-Based Systems
ECL – Embedded Constraint Language
ECOOP – European Conference on Object-Oriented Programming
GME – Generic Model Editor
GP – Generative Programming
GUI – Graphical User Interface
ICSE – International Conference on Software Engineering
IDE – Integrated Development Environment
IP – Intentional programming
ISIS – Institute for Software Integrated Systems
JSP – JavaServer Pages
JTS – Jakarta Tool Suite
KWIC – Key Word in Context
MCL – Multigraph Constraint Language
MDA – Model-Driven Architecture
MDSOC – Multi-Dimensional Separation of Concerns
MIC – Model-Integrated Computing
MOBIES – Model-Based Integration of Embedded Software
MOP – Metaobject Protocol
NCI – National Compiler Infrastructure
OCL – Object Constraint Language
OMG – Object Management Group

OO – Object Oriented
OOP – Object-Oriented Programming
OOPSLA – Object-Oriented Programming, Systems, Languages, and Applications
PCCTS – Purdue Compiler Construction Tool
PCES – Program Composition for Embedded Systems
QoS – Quality of Service
SOP – Subject-Oriented Programming
StratGen – Strategy Code Generator
SUIF – Stanford University Intermediate Format
UAV – Unmanned Aerial Vehicle
UML – Unified Modeling Language
WCET – Worst Case Execution Time
XML – Extensible Markup Language
XSLT – Extensible Stylesheet Transformations
YACC – Yet Another Compiler-Compiler

CHAPTER I

INTRODUCTION

In any engineering endeavor, a key requirement is the ability to compose large structures from a set of primitive elements. This is true for children who are constructing toy models of bridges and buildings using Lego" or Erector" sets. This is true, on a larger scale, for civil engineers who design and supervise the construction of skyscrapers.

This is especially true for software engineers who compose increasingly complex systems from components, classes, and methods. An important difference between the engineering of software, and the other undertakings enumerated above, is the recognition that the set of available core elements for software construction is often significantly larger. The composition of these elements can be specified at a much finer level of granularity. As a contrast, the bricks used to build Lego" houses, or the steel beams used in the construction of a bridge, come in but a few different shapes and sizes, and are composed using a simple standard interface (e.g., the prong and receptacle parts of a Lego" block have been unchanged since 1932 [Lego, 2002]; likewise, since around 1850, the standard dimensions for an air cell masonry brick in the United States has been 2.5 x 3.75 x 8 inches [Chrysler and Escobar, 2000]).

Separation of Concerns

Furthermore, the compositional permutations and dynamic interactions that are possible with software elements are several orders of magnitude richer than those found in other engineering activities. For example, a generic function can be parameterized with a seemingly unlimited number of other elements (e.g., a template function that can sort any data type using numerous factors). Parametric polymorphism is but one factor that contributes to the exponential state explosion problem that makes the composition of software so difficult. A reason for this complexity is that the essence of software elements is expressed as logical abstractions, as opposed to physical materials, which results in the generation of an enormous state-space that must be tested. In fact, the core of Brooks No Silver Bullet essay is a commentary that the molding of complex conceptual entities is the essence of software construction [Brooks, 1995].

It has been a longstanding understanding among software engineering researchers that the proverbial Gordian knot has appeared as a consequence of the exponential complexities involved in composing a set of software building blocks, or modules. Separation of concerns has emerged at the center of many helpful techniques for loosening the grip of this knot. Separation of concerns is not a new idea. In fact, over the past quarter-century, issues related to concern separation have been at the heart of the intersection of software engineering and programming language design research. A *concern* is generally defined as some piece of a problem whose isolation as a unique conceptual unit results in a desirable property. Concerns arise as intentional artifacts of a system. They are the primary stimulus for structuring software into localized modules.

The *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems* defines a concern as, those interests that pertain to the systems development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders. Concerns include system considerations such as performance, reliability, security, distribution, and evolvability [IEEE 1471, 2000]. Other researchers have defined a concern to be, any matter of interest in a software system [Sutton and Rouvellou, 2001], and

a slice through the problem domain that addresses a single issue [Nelson et al., 2001]. Concerns are a central point of interest at any stage of the development cycle. A criterion for decomposition abstraction is doing just what our small minds need: making it possible for us to think about important properties of our program its behavior without having to think about the entirety of the machinations [Kiczales, 1992].

Modularity, abstraction, information hiding, and variability are important topics in software engineering that are associated with separation of concerns [Schach, 2002]. A clean separation of concerns provides a system developer with more coherent and manageable modules. From the structured paradigm of the 1960s and 1970s, to the Object-Oriented (OO) paradigm of the past few decades, there has always been an interest in creating new abstraction mechanisms that provide improved separation of concerns. There are several new paradigms on the horizon, as will be discussed in the next subsection (Advanced Separation of Concerns), to assist in further separation.

The most influential paper related to the study of modularization, and perhaps even in all of software engineering, is David Parnas on the Criteria to Be Used in Decomposing Systems into Modules [Parnas, 1972]. Parnas criteria aid a designer in achieving module independence. Parnas recognized that the decomposition of a system into its constituent parts must be performed with several specific goals in mind. To illustrate the consequences and tradeoffs from different design decisions, Parnas introduced a simple indexing program called KWIC (Key Word in Context). From a comparison of two separate modularizations for KWIC, Parnas suggested that modules be composed with the following objectives: changeability, independent development, and comprehensibility. The criterion of information hiding was shown by Parnas to be important in all three of these objectives.

Criteria for Decomposition

Changeability is the way to evaluate a modular decomposition, particularly one that claims to rest on information hiding, is to ask what changes it accommodates [Hoffman and Weiss, 2001].

A change to a module should not necessitate numerous invasive changes to many other modules. Parnas work has revealed that the structure of a system has a direct effect on the cost of change and maintenance. The potential that a module will undergo change should always be kept in mind when considering several different possibilities for modularization. Those implementation decisions that have the possibility of being changed, or those decisions that offer the most degree of flexibility in adaptation, should be hidden from the client of that module. This observation was a key toward the discovery of the properties of encapsulation and information hiding, where abstraction is the principal idea for delimiting what from the how. Designs that are created with the principle of information hiding permit the substitution of different implementations for the same abstraction. This improves the capacity to make changes based upon different desiderata (e.g., the typical time versus space arguments in data structure implementation).

Modularity is about separation: When we worry about a small set of related things, we locate them in the same place. This is how thousands of programmers can work on the same source code and make progress [Gabriel and Goldman, 2000].

As the complexity and size of software system soars, the ability of developers to independently work on separate modules becomes increasingly important. This is a vital attribute of the open-source community, where multiple developers work independently on a common collection of source code. The task of modularization, then, turns out to be a type of work assignment for each developer. The details of the design decisions and responsibilities of each developer should be hidden behind an exposed abstract interface. The interface

supplies the only means of access to the services offered by the module.

Comprehensibility in many pieces of code the problem of disorientation is acute. People have no idea what each component of the code is for and they experience considerable mental stress as a result [Gabriel, 1995]. When Microsoft first began conducting usability studies in the late 1980s to figure out how to make their products easier to use, their researchers found that 6 to 8 out of 10 users couldn't understand the user interface and get to most of the features [Maguire, 1994].

Comprehensibility can be negatively affected, within any context, by a poorly designed interface. Comprehensibility is a major goal of modular reasoning; that is, it should be possible for a developer to study one module at a time without being overwhelmed with the details of extraneous implementation information defined outside of the module context. Several popular ideas in software engineering (e.g., Dijkstra's Go To Statement Considered Harmful [Dijkstra, 1968], and Wulf and Shaws Global Variables Considered Harmful [Wulf and Shaw, 1973]), were in fact arguments made from the perspective of comprehensibility. An early result of object-oriented research demonstrated a strong link between comprehensibility and low coupling [Lieberherr and Holland, 1989].

Cohesion and Coupling

Cohesion and coupling are an obvious connection exists between highly cohesive and lowly coupled modules, and the objectives identified by Parnas. The seminal definitions of cohesion and coupling were provided within the context of structured design [Stevens et al., 1974]. A measure of cohesion and coupling can often provide an assessment of the quality of a design. Cohesion represents the degree of functional correlation between the individual pieces of a module (i.e., the extent to which a module is concentrated on a specific, well-defined concept). A method that exhibits low cohesion often contains code to perform several tasks that are conceptually different (e.g., a stack class where the push method also computes a square root). In a highly cohesive module, the various relationships within the module can be easily discerned because of the distinct focus of the module. This is a great attribute for supporting independent development.

Coupling can be described as the extent to which modules are connected with each other. Highly coupled modules are very brittle because a change to one module often requires the modification of a number of other modules. This also negatively affects independent development because highly coupled modules will often reveal their underlying internal implementation details to other modules. The comprehensibility of such modules is reduced, too, because several different modules must be examined to understand the intent of a module. Coupling is, to a large extent, the opposite of good modularity.

Advanced Separation of Concerns

Even though the general notion of separation of concerns is an old idea, one can witness the nascence of a research area devoted to the investigation of new techniques to support advanced separation of concerns. Recall that the opening paragraphs of this chapter highlighted the importance of modular composition within several engineering activities. It has been recognized by numerous researchers that the software modularization constructs developed over the past quarter-century are sometimes inadequate for capturing certain types of concerns. This has serious consequences with respect to modular composition.

Previously defined modularization constructs are most beneficial at separating concerns that are orthogonal [Tarr et al., 1999]. However, these constructs often fail to capture the isolation of concerns that are non-orthogonal. Such concerns are said to be

crosscutting, and their representation is scattered across the description of numerous other concerns. Crosscutting concerns are denigrated to second-class citizens in most languages (i.e., there is no explicit representation for modularization of crosscutting concerns). As a result, crosscutting concerns are difficult to compose and change without invasively modifying the description of other concerns (i.e., crosscuts are highly coupled with other concerns). The three objectives of changeability, independent development, and comprehensibility are sacrificed in the presence of crosscutting concerns because of the lack of support for modularization (see [Gudmundson and Kiczales, 2001] for an evaluation of these objectives in the context of newly proposed modularization constructs). The latest research efforts, under the general name of Aspect-Oriented Software Development (AOSD) [AOSD, 2002], explore fundamentally new ways to carve a system into a set of elemental parts in order to support crosscutting concerns. The goal is to capture crosscuts in a modular way with new language constructs called *aspects*. A large portion of the second chapter thoroughly explains the problem of crosscutting concerns and surveys solution techniques.

The next section is not about AOSD, but rather shows how crosscutting enters into other areas of human life, as well.

Organization Theory

Thus my central theme is that complexity frequently takes the form of hierarchy and that hierarchic systems have some common properties independent of their specific content [Simon, 1996]. Various types of organizations encompass elaborate hierarchies. The subject of organizational hierarchy has been studied for nearly a century. Within the disciplines of management and administration sciences, there is a popular corpus known as organization theory. Organization theory has a basis for comparison with software organizational hierarchy has been studied for nearly a century. Within the disciplines of management and administration sciences, there is a popular corpus known as organization theory. Organization theory has a basis for comparison with software development whenever a hierarchic approach to software decomposition is adopted. It is worth noting that some of the influential work in organization theory was conducted by a Turing Award winner Herbert Simon who also received the Nobel Prize for his work on decision-making in organizations. This section will offer a short assessment of organization theory as it relates to software construction

Since Adam Smiths, The Wealth of Nations [Smith, 1776], the concept of division of labor has been an important topic within the discourse of economics, and the study of supporting institutions. A keen contribution by Smith was a quantifiable justification for the benefits that division of labor and specialization garner vis-à-vis efficiency and productivity. Division of labor is to a large extent correlated to the general objectives of separation of concerns as it relates to information hiding and the independent development of modules. Parnas actually gave a definition for the term module that would support such an assertion, as he stated, in this context module is considered to be a responsibility assignment rather than a subprogram [Parnas, 1972]. The responsibility assignment of a module to a programmer relates to the specialization of effort that exists in division of labor. Interdependent Organizations display degrees of internal interdependence. Changes in one component or subpart of an organization frequently have repercussions for other parts the pieces are interconnected [Daft et al., 1987]. After an organization is hierarchically constructed (as a result of the specialization of labor division), it is almost assured that the boundaries of the hierarchy will be broken as a result of interdependence among the different divisions. Large organizations naturally have certain kinds of concerns that are non-orthogonal to the hierarchic structure. Such facets of the organization increase the coupling of each division of

the organization and expose particular characteristics of the division specialization (an example of this is provided in the next section, within the context of a student requesting a transcript). These are the crosscutting concerns of the organization. Studies have been conducted on the mechanisms by which organizations have the ability to adapt to feedback [Daft et al., 1987]. These self-correcting behaviors are analogous to the reflective methods that are surveyed in Chapter 2. Hierarchic decomposition is a tool for accomplishing goals and objectives within an organization. It is normal for organizations to have multiple goals, some of which may be conflicting [Hall, 1998]. The multiple rules that are spread throughout the hierarchy of an organization are the result, in many cases, of the implementation of some policy, or protocol. A policy is a mechanism that coordinates specific objectives across a set of dislocated organizational units. A policy, and the rules that implement it, could be considered a type of crosscutting concern within the organization. The pejorative meaning of red-tape is tied to the frustrations that result from bureaucratic rules of policy implementation. In order for the policy to be realized, the specialization of many different organizational departments is needed. Intriguingly, the initial concept of bureaucracy, as proposed by [Weber, 1946], was promoted as the best structure for dealing with a changing environment today, it is mostly associated with a negative connotation. An interesting case study is presented in [Perrow, 1986], where a formal process at the University of Wisconsin was scrutinized. The policy that was examined corresponded to the process for a university faculty member to make a formal suggestion, or complaint. It was discovered that a complete review of the formal request would require that it pass through over fifteen levels of the university hierarchy. This example is comparable to crosscutting concerns in software implementations that execute a protocol across a large code base. As will be shown in a later chapter (see Figure 9 through Figure 11), the communication path in a hierarchy can introduce unnecessary overhead in both organizations and software. The concept of an Independent Integrator has been advocated as a coordinator of the policies involving myriad interdependent departments [Dessler, 1986]. An integrator is the closest entity within organization theory that has a relation to techniques for advanced separation of concerns. The role of an integrator is to step outside the hierarchical bounds and assist in the weaving of a crosscutting policy throughout the organization.

Research Objectives

This research is about advanced separation of concerns at the system modeling level, and the construction of support methodology for system software that facilitate the elevation of crosscutting modeling concerns to first-class citizens (i.e., explicit constructs for the representation of such concerns) where adaptability and extensibility can be achieved. The contributions described in this research can be summarized by two research objectives: Raise Aspect-Oriented (AO) concepts for supporting the design of adaptable and extensible system software, such as operating systems, to a higher level of abstraction. An aspect orientation can be beneficial at different stages of the software lifecycle and at various levels of abstraction; that is, it also can be advantageous to apply aspect orientation at levels closer to the problem space (e.g., analysis, design, and modeling), as opposed to the solution space (e.g., implementation and coding). Whenever the description of a software artifact exhibits crosscutting structure, the principles of modularity espoused by aspect orientation offer a powerful technology for supporting better separation of concerns, which is ease of reuse, adaptability, extensibility, and comprehensibility. This has been found to be true also in the area of domain-specific modeling [Gray et al., 2000]. Although there have been other efforts that explore AO at the design and analysis levels (see Chapter 2 for more details), the work described in [Gray et al., 2001a] represents the first occurrence in the literature of an actual

aspect-oriented weaver (see Figure 2.6 in Chapter 2) that is focused on system modeling issues, rather than topics that are applicable to traditional programming languages.

The research assists in the creation of new weavers using a generative framework. Because the syntax and semantics of each modeling domain are unique, a different weaver is needed for each domain. These two objectives provide a contribution toward the synergy of AOSD and Model-Integrated Computing (MIC) (see [Sztipanovits and Karsai, 1997] for an overview of MIC). This union assists a modeler in capturing concerns that, heretofore, were very difficult, if not impossible, to modularize. A key benefit is the ability to explore numerous scenarios by considering crosscutting modeling concerns as aspects that can be rapidly inserted and removed from a model.

This research use the Aspect-Oriented Framework (AOF) developed by Netinant and Elrad to design an operating system built on separation of aspectual system properties from basic functionalities. We believe this is a solid break through and innovative approach to advance understanding and capabilities of system software development and utilization in operating system area. The project will investigate the potential of building Aspects and Components-Oriented Operating Systems (ACOOS) with respect to the following demands.

1. The impact of the aspect-oriented framework called component, adaptability, and layers (CAL) to support the design of operating systems on the extendibility and adaptability of current and potential new systems features.

2. The impact of potential use of aspect orientation approach to operating system design and implementation.

3. The impact of the design and implementation for the aspect and component-oriented operating systems on the ease of implementation and extensibility.

4. The impact of the design and implementation for the aspect and component-oriented operating systems on the ease of implementation and adaptability.

The goal of this two-year project is a development of an open architecture, a prototype of an aspect and component-oriented operating system called ACOOS using aspect-oriented frameworks (CAL) where both basic functional components and crosscutting system properties are designed separately from each other in each layer. Their composition is formally supported to ensure correctness. This separation of concerns allows for reusability and enables the building of software systems that are comprehensible, adaptable, and extendable.

Our research concentrates on the design of extensible and adaptable operating systems using aspect-oriented frameworks. We need to address the following two issues: what should be done in aspects and how it should be done. Based on the current state of the art using an aspect-oriented design framework

Outline

A background survey of related literature can be found in Chapter 2. The chapter reviews several techniques that have been used over the past decade to provide the variability needed to support clean separation of concerns. That chapters overview begins by examining topics such as reflection and metaprogramming. The Chapter 2 also provides the incentive for, and summary of, the emerging research efforts in advanced separation of concerns. Within the general context of generative programming, a cornucopia of topics is summarized at the end of the second chapter. This encompasses a brief synopsis of the literature on object-oriented frameworks, code generators, and domain-specific languages.

In Chapter 3, the framework is introduced. Chapter 4 is about concluding and remarks of the framework. Finally, Chapter 5 is conclusion of this research. A comprehensive bibliography is included at the end of this report.

CHAPTER II

BACKGROUND

This chapter contains a broad survey of many techniques that have been found useful for supporting modularization of software (e.g., reflection and metaobjects, advanced separation of concerns, generative programming, and frameworks). These techniques also are effective at providing the capability needed for software compositions to adapt and change to evolving requirements. The contributions of this research in Chapters 4 are extensions of several of these ideas.

Reflection and Metaobjects

Industry increasingly demands that systems be adaptable and extensible. This demand may be manifested in various forms, including:

- The malleability of an application with respect to a set of changing user requirements (i.e., the degree of difficulty to affect change in an application's source code implementation);
- The degree of adaptability within a system in the presence of a changing environment (i.e., the capacity of an application to examine itself and modify its own internal state during run-time).

Reflection and metaprogramming provide powerful techniques for extensibility by separating the program's computation (the base level) from the specifics of how the program is interpreted (the metalevel). This separation permits the modification of the underlying implementation semantics (through changes to the metalevel) at run-time. These techniques have been shown to provide great flexibility in systems that must adapt to changing environments [Robertson and Brady, 1999]. A philosophical definition of reflection has been given as, "...the capacity to represent our ideas and to make them the object of our own thoughts" [Clavel, 2000]. As used in this sense, reflection was first introduced in logic as a way to extend theories [Hoftstadter, 1979]. Reflection also has been an active research area within the context of programming languages. Various forms of reflection are even appearing in popular programming languages like Java.

Procedural Reflection

The work of Brian Cantwell Smith provided the seminal ideas for formally applying reflection to programming languages [Smith, 1982]. Smith defined procedural reflection as the concept of a program knowing about its implementation and the context in which it is executed (later, Smith would prefer the term introspection in place of procedural reflection). A reflective system is capable of reasoning about itself in the same way that it can reason about the state of some part of the external world. Introspection offers the capability of dynamically adjusting the way that programs are executed. A reflective system has a causally connected self-representation [Smith, 1982]. Thus, a reflective system has access to the structures that are used to represent it. Depending on the level of support for reflection, these internal representations can be inspected and even manipulated. Here, the term "causally connected" means that a manipulation of the internal representation structures directly affects the observable external behavior.

Smith identified three conditions that must be satisfied in order for a system to be considered introspective:

1. The system must be able to represent a description of its internal structure in such a way that it can be inspected and modified by facilities within the system.

2. The self-representation must be causally connected to the structure and behavior of the system. Each event and state in the system must be self-described and modifications to the description must result in a change in structure or behavior.

3. The self-representation must be at the proper level of abstraction. It must be low enough such that meaningful modifications can be made. Yet, it must not be so low-level that a programmer gets bogged down in a morass of detail.

Metacircular Interpreters

Smith also described a language, called 3-Lisp that supported his model of reflection. In 3-Lisp, the notion of a reflective tower of metacircular interpreters [Steele and Sussman, 1978] supports the incremental changes to layers of interpreters. A *metacircular* interpreter is a program that is written in the same language that it interprets [Abelson and Sussman, 1996]. The reflective tower is an infinitely ascending stack of interpreters. All interpreters in this tower are implemented in 3-Lisp. Each new layer in the tower is interpreted by the layer above it. The interpreter at the very bottom of the layer is the traditional program that processes user input. In 3-Lisp, as is typical of most Lisp or Scheme implementations, an expression, an environment, and a continuation argument capture the state of an interpreter. The layers in the tower are connected by reification and reflection. Reification is the inverse of reflection – it is about the ability to consider an abstract concept as concrete. Sobel and

Friedman distinguish the two processes as, "...converting some component of the interpreter's state into a value that may be manipulated by the program is called *reification*; the process of converting a programmatically expressed value into a component of the interpreter's state is called *reflection*" [Sobel and Friedman, 1996]

Object Reflection

The first effort to incorporate "Smithsonian" reflection into an object-oriented language is described in [Maes, 1987]. Building on the foundation of procedural reflection, an object-oriented reflective architecture divides the object part from the reflective part. The object part describes and manipulates the application domain and the reflective part describes and manipulates the object computation semantics.

The reflective operations provided by some object-oriented programming languages are limited. For example, the model of reflection provided in Java is much weaker than that found in Smalltalk and the Common Lisp Object System (CLOS). The reflection mechanism in Java does not permit the modification of the internal representation [Anderson and Hickey, 1999], [Sullivan, 2001]. It only provides a type of "read-only" examination facility that allows run-time inspection of the internal representation of an object. A further limitation is that the reflective methods in Java are marked final, which prohibits their extension. Therefore, the reflective model provided in Java is not of the Smithsonian style because it does not provide the adaptation needed for being causally connected. The definition of introspection is presented slightly differently in [Bobrow et al., 1993]. They define *introspection* as a program's ability to observe and reason about its own state. They define *intercession* as the more powerful capability of modifying the internal state to affect the underlying semantics. Using these definitions, Java can be said to provide support for introspection, but not intercession.

Metaobjects

Meta means that you step back from your own place. What you used to do is now what you see. What you were is now what you act on. Verbs turn to nouns. What you used to think of as a pattern is now treated as a thing to put in the slot of another pattern. A metafoo is a foo into whose slots you can put parts of a foo [Steele, 1998]. As Steele observes, the prefix meta is used to denote a description that is one level higher than the standard frame of perception. Meta is also used to mean "about," "between," "over," or "after." Hence, a metaprogram is usually defined as a program that modifies or generates other programs. A compiler is an example of a metaprogram because it takes a program in one notation as input and produces another program (usually object code) as output. Reflection is considered a form of metaprogramming where the target of the modification is the metaprogram itself. Metaprogramming can be a complex activity sometimes because there can be a blur between the base level and the metalevel.

Metaobject Protocols

Maes appears to be the first to introduce the notion of a metaobject [Maes, 1987]. In an object reflection system, a metaobject is just like any other object during run-time. Every object in the language has a corresponding metaobject and every metaobject has a pointer to its corresponding implementation object [Maes, 1988]. The metaobject contains information about its language object, such as details on its implementation and interpretation. During the execution of a system, the language objects may request information about their state, and even perform a modification on the internal representation. Metaobject Protocols (MOPs) facilitate the modification of the semantics of the underlying implementation language [Kiczales et al., 1991]. Manipulating the interfaces that the MOP provides can incrementally modify the behavior and implementation of the underlying language. For example, CLOS has a MOP that specifies a set of generic functions [Steele, 1990].

There are five categories of functions that represent the core elements of CLOS (i.e., classes, slots, methods, generic functions, and method combination). A metaobject represents each of these core elements. Each metaobject has a metaclass. The metaclasses behave like any other class such that the semantics of a metaobject can be adapted by modifying its metaclass. A programmer can alter the semantics of CLOS by using standard object-oriented techniques, like subclassing. The instance of each metaobject can be adapted at run-time. The behavior of the system at any particular time is dependent on the configuration of the set of metaobjects. The protocol, in this case, represents the interfaces of the metaclasses. Any modification to the behavior of the system must adhere to the interface definitions. MOPs gain their adaptive power from a synergy of reflection and Object-Oriented Programming (OOP). As described in [Kiczales et al., 1991], there are three attributes of a metaobject protocol:

1. The core programming elements of a language are represented as objects. For example, the syntax and semantics for method calls, the rules for handling multiple-inheritance, and the rules of method lookup are all represented as objects.
2. The behavior of the language is encoded in a protocol based on these objects. The protocol is the interface of the metaclasses.
3. A default object is created for each kind of metaobject.

Concerning the first attribute from above, an example of the ability to modify multiple-inheritance rules is shown in [Kiczales et al., 1991]. A generic function called compute-class-precedence-list returns the rules that determine the resolution of conflicts due to multiple-

inheritance. The programmer can modify this list so that new rules of conflict resolution are used. As another example, objects are created in CLOS by calling make-instance. The implementation of this method can be redefined at runtime to perform specialized adaptations during object creation. Although the majority of the literature on reflection and metaprogramming is described in some dialect of Lisp, there have been efforts to apply these techniques to other languages. For example, [Chiba and Masuda, 1993] describe a basic metaobject protocol for a language called Open C++. A more detailed description of a MOP for C++ is given in [Forman and Danforth, 1999]. While not analogous to MOPs, *per se*, there has also been research in C++ on an idea called static metaprogramming. A variant of this, which relies on C++ templates, provides a compile-time facility for generating code and component configuration [Czarnecki and Eisenecker, 2000].

Metaobjects also can be used in assisting in the separation of concerns in areas other than programming languages. Research at IBM recognized that, within middleware, there is an intermixing of application code and protocol code [Atsley et al., 2001]. The lack of modularity affects the ability to maintain and customize the middleware. A metaobject protocol cleanly separates the policy and protocol code from the underlying application. Some example metaobjects that were defined to represent communication events are transmit (what happens when a component sends a message), deliver (what happens when a message is received by a component), and dispatch (the received message a component decides to process). Nonfunctional system properties like security and persistence [Rashid, 2002] can be cleanly separated from the base level program to improve reuse. This has been termed *implementational reflection* in [Rao, 1991].

Within the scope of distributed object computing and middleware, the technique of CORBA interceptors is closely related to metaobject protocols. Interceptors are defined as, "non-application components that can alter application behavior" [Narasimhan et al., 1999]. An interceptor can transparently modify the behavior of an application by attaching itself to the invocation path of a client and server object. Interceptors have been shown to be useful in enhancing CORBA by providing adaptability with respect to profiling, protocol adaptation, scheduling, and fault tolerance [Narasimhan et al., 1999].

Evaluating MOPs

A detailed evaluation of the practical use of MOPs can be found in [Lee and Zachary, 1995]. In this study, a MOP was applied to a geometric CAD tool in order to add persistence to the CLOS implementation objects. The project was described as being very ambitious and a much more complicated application of MOPs than previously studied. Much of the evaluation was positive. Because the majority of the effort to extend CLOS related to objects, the metaobject protocol provided a useful resource. However, the effort had several difficulties. Although the CLOS MOP is very useful when extension is based on a property of an object, the protocol is not helpful when there is a requirement to augment a feature that is not captured as an object property. For example, in CLOS, arrays and several other composite values are native to Common Lisp and are not available for extension in the MOP. Another difficulty was found with respect to performance. In several experiments, it was found that object creation was sixteen times slower than the prior implementation that did not use a MOP. Similarly, write access using the MOP was found to be about seven times slower. Performance has always been a problem for reflective approaches. Consider the following observation, with respect to Java-based reflection, "As of release 1.4, reflective method invocation was forty times slower on my machine than normal method invocation. Reflection was re-architected in release 5 for greatly improved performance, but is still twice as slow as normal access, and the gap is unlikely to narrow" [Bloch, 2001]. The performance penalty

resulting from many dynamic calls in a reflective implementation will often rule-out reflection as an implementation alternative in some contexts.

Open Implementations

Traditionally, black-box abstraction states that a software module should expose its interface, but hide its implementation details. This is a corollary to [Parnas, 1972], and is similar to the *Open-Closed Principle*, described in [Meyer, 1997], which states that a module should be open for extension, yet closed for modification. However, the idea of an open implementation disagrees with this principle when applied fundamentally. Research in the area of open implementations has found that, in some cases, software can be more reusable when a client is allowed to control a module's implementation strategy [Kiczales, 1996]. Open implementation proponents agree that the base level should remain closed like a black-box. It is the metapart that they advocate opening to extension [Kiczales, 1992]. In fact, the initial motivation behind MOPs was a desire to open the language in such a way that better control could be exerted over the selection of the implementation with respect to certain performance concerns [Kiczales et al., 1993].

Advanced Separation of Concerns

In Chapter 1, the importance of separation of concerns was motivated. During the latter part of the 1990s, research in this area increased with an invigorated interest. This was due, in part, to the recognition that the languages and tools used to develop software hampered the proper isolation of specific categories of concerns. The inadequacies of modern programming languages (with respect to separating certain concerns) prompted many researchers to take a fresh look at modularization constructs and extensions/complements to current languages. The focus of the problem can be discerned from the observation that programming languages are often used in a linear process. However, the things that we want to express in a language, and our conceptualization of key abstractions as a supporting mechanism, are certainly not linear. This section provides the initial motivation and problems that are being solved by a new area of research entitled Advanced Separation of Concerns (ASOC).

A Survey of Some Concerns and Their Separation Before initiating the impetus behind advanced separation of concerns at the implementation level, it may be beneficial to first notice the various methods that have been suggested for managing concerns in other contexts. The examples in this section represent concerns that are typically identified outside of the milieu of traditional programming language research.

Database Triggers

Assume that the following business rule is to be consistently enforced within a database: "Every time an employee's salary is increased by 25%, log the employee's social-security number, previous salary, and new salary into an audit table." The implementation of this business rule requires that some action be taken every time that an update to the salary column occurs. This business rule is an archetype for a crosscutting concern. Without triggers, the realization of this rule would require that the concern be placed in *all* of the stored procedures that update the employee's salary. That is, the delta of a salary increase must be computed for each update and checked against the specified 25% rate increase. This could result in the insertion of redundant code throughout all stored procedures that are affected by this business rule. The problem is compounded when the salary update occurs

within embedded SQL in a base programming language. In that case, the check must be made outside of the database in every location of the base program that implements this business rule. Fortunately, a trigger mechanism facilitates a cleaner solution. A trigger-based solution, like that found in Figure 1, would provide a single location from which changes could be made to the semantics of the concern. The trigger solution does not need access to metalevel control in order to capture the intent of the concern (i.e., it is not necessary to redefine the underlying semantics of the table update definition). As will be shown later, this is similar to the way that AspectJ captures a concern without resorting to metaprogramming techniques (i.e., aspects and non-aspects are all at base-level code – there is no reference to the metalevel within AspectJ). This is an important point in differentiating triggers, and even aspect languages, from pure metaprogramming techniques. Later in this chapter, the constitutive parts of an aspect language will be described. A preview of these is now given in a comparison of aspect languages and triggers.

```
CREATE OR REPLACE TRIGGER salary_audit
AFTER UPDATE OF salary ON employee
FOR EACH ROW
WHEN (new.salary > 1.25 * old.salary)
CALL log_salary_audit(:new.ssn, :old.salary, :new.salary);
```

Figure 2.1: A Trigger for Logging Salary Increases

On the second line of Figure 1, the "AFTER UPDATE" statement indicates the point of execution when the trigger statement is applied. Using BEFORE/AFTER, an Oracle database trigger is able to influence the dynamic execution of a database server whenever certain operations (DELETE, INSERT, UPDATE) are executed on a database table. There are six different variations that can be given, resulting from the permutation of {BEFORE, AFTER} × {DELETE, INSERT, UPDATE}. Also, on the second line, the "OF salary ON employee" is similar to the pointcut idea in aspect languages. This construct identifies a particular point in the database table (e.g., a row and a table) that is affected by the trigger. The "when condition" syntactical construct on line 4 has some likeness to the "if" pointcut designator in AspectJ. The executable statement that is associated with the trigger (this is the action that occurs when the trigger is fired), found on the last line of Figure 1, is akin to the concept of "advice" in AspectJ. The definition of these aspect-oriented terms will be clarified in a subsequent section. Even though the database trigger mechanism permits the capture of crosscutting business rules within a database, it has several weaknesses when compared to pure aspect languages. The most evident limitation is the lack of the ability to create compositions of triggers. The trigger approach allows only the naming of a single table. It does not permit the logical composition of table property descriptions. That is, the type of pointcut model used within triggers is not composable in the same way as AspectJ. Triggers also do not support the concept of wildcards within the naming of a pointcut. For example, the second line from above could not be written as "OF sal* ON emp*" in order to designate multiple columns and tables that are affected by the trigger.

Mail Merge

Mail merge is an office automation tool that supports the separation of the form of a document from a data source of merge fields. By this separation, the insertion of each instance throughout the document can be better managed (see Figure 2). Consider the task of a lawyer who specializes in commercial foreclosures. He, or she, will typically need to process fifteen different documents in order to execute a foreclosure (according to

information obtained from a personal conversation with a Nashville attorney). Furthermore, five or more different parties (with separate contact information) are typically involved. Their contact addresses, and other pertinent information, are diffused across the space of the various legal documents. By separating the instance from the form, the author of the document is spared from the tedious task of visiting multiple locations in the document in order to make each change. Although the mail merge tool assists in a specific type of concern separation, it requires the document designer initially to visit every instantiation point in order to insert a field designator (because of this, the process is somewhat similar to the LaTeX macro command).

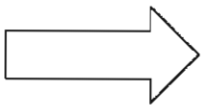
Style Sheets

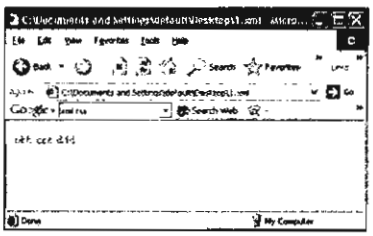
Within the context of web publishing, style sheets are a useful technique for separating the content of a document from its presentation style [Meyer, 2000]. Such a separation provides a method for making seamless global changes to the appearance of a document without the need for visiting numerous individual locations in the document. In a Cascading Style Sheet (CSS), a rendering engine visits each node of a document. As the traversal proceeds over the document's hierarchy, the rendered attempts to match the current element with a pattern specified as a CSS rule. A CSS rule consists of two parts: a selector, which names the type of the element to which the style will be applied, and a declaration, which represents the type of style to be applied.

XML Text

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet href="style1.css" type="text/css"?>

<FOO>
  <BAR1>bbb</BAR1>
  <BAR2>ccc
    <BAR3>ddd</BAR3>
  </BAR2>
</FOO>
```





CSS Stylesheet (style1.css)

```
BAR1  {color:red}
BAR2  {color:blue}
BAR3  {color:green}
```

Figure 2.2: A Cascading Stylesheet Example

An illustration of the application of a CSS rule is shown in Figure 2. The top-left of the figure contains the content of a document as represented in the Extensible Markup Language (XML). The information regarding the name of the specific style that is to be applied (in this case, the style sheet named style1.css) is located within the preamble of this document. The specification of style1.css is listed in the bottom-left of the figure. As can be seen, this style sheet has a rule asserting that all elements of type BAR1 are to be rendered in the color red. In this example, it should be understood that the rendering engine resides within the browser.

Literate Programming and WEB

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do [Knuth, 1984]. The idea of literate programming was initially described by Donald Knuth and implemented with a tool called WEB [Knuth, 1984]. In WEB, a single program is a combination of source code, documentation text, and WEB commands. Literate programming assists a programmer in assembling programs that are more easily read by a human. This is done by treating the construction of documentation and source code as a simultaneous activity. The aim is to make the construction of programs more like the creation of a literary work. The formal expression of a concern is so closely tied to the informal description that tools are needed to separate the two representations so that they are consumable by different parties (e.g., a compiler and a human). In WEB, source code is produced from the TANGLE tool, and documentation is formed by the WEAVE tool (see Figure 3). It is interesting to note that the structure of the process for creating WEB programs is almost opposite to that seen in Figure 1 and Figure 2. In those contexts, the concept of weaving a document entailed the notion of bringing separated entities together as one (where the separation provided some desirable property that assisted in change maintenance and comprehensibility). In literate programming, however, the concept of weaving represents the task of separating concerns of interest (e.g., the visual presentation of documentation) from an existing tightly coupled document.

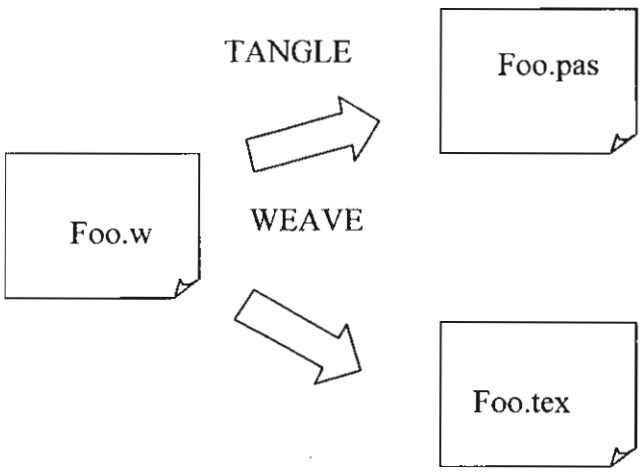


Figure 2.3: Separation of Concerns in WEB

The preceding subsections provided several examples of concern separation. Two of the four examples were in contexts not associated with software development (e.g., mail-merge and stylesheets). A common topic in each of these examples was the existence of an integration tool for assisting in the conceptual separation. In the following sections, the problems associated with crosscutting concerns are motivated, along with the need for a new type of software integration tool – a weaver.

Problems with Scattered Code

It is organization which gives birth to the dominion of the elected over the electors, of the mandataries over the mandators, of the delegates over the delegators. Who says organization, says oligarchy [Michels, 1915]. Non-orthogonal concerns can be described as crosscutting, because such concerns tend to be scattered across the traditional modularity boundaries provided by a development paradigm. In programming languages, two concerns crosscut when the modularity constructs of a language allow one concern to be captured separately, but only to the detriment of another concern that must be captured in a way that is not cleanly localized. This has been referred to as the "tyranny of the dominant decomposition" [Tarret al., 1999]. The "Iron Law of Oligarchy," quoted above from Michels, suggests that bureaucratic hierarchy tends to result in oligarchy; that is, those at the top of an organization are those that rule. In Chapter 1, an allusion was made to this tyranny under the Organization Theory section that described Interdependence. With respect to the dominant decomposition, this also seems to be true with traditional methods for software modularization. Crosscutting has the potential to destroy modularity. The crosscutting phenomenon can occur in structured programming, where the procedure, function, and module delimit the modularity boundaries. It is also prevalent in object-oriented programming, where classes, methods, and inheritance define the boundaries of encapsulation.

Persistence

Crosscutting concerns provide difficulties for a programmer because the implementation of the concern is scattered throughout the code; the concern is not localized in a single module. This can be a source of potential error when modifications are required. Comprehensibility is negatively affected in two ways [Tarr et al., 1999]:

- The scattering problem: The ability to reason about the effect of a concern is decreased because a programmer must visit numerous modular units in order to understand the intent of a single concern. The problem is that a concern often touches many different pieces of code.
- The tangling problem: Within a module, the tangling of numerous concerns decreases cohesion, and raises coupling. This reduces a programmer's ability to understand the core intent of a particular module. The problem is that many concerns may touch a single piece of code.

Programmers are often forced to keep track of crosscutting concerns in their heads. This is an error-prone activity, because even medium-sized programs can have hundreds of different crosscutting issues [Tristram, 2001]. Another problem of crosscutting concerns is maintenance. It is often the case that the global spreading of a concern, and the ramifications of its modifications, are not intuitive to those who inherit the code for maintenance. Maintenance becomes more of an archaeological metaphor, where a programmer must search through rubble in order to uncover a useful artifact [Hunt and Thomas, 2002]. The Parnasian objectives, found in Chapter 1, are usually sacrificed in the presence of non-orthogonal concerns.

Figure 4 provides an illustration of scattering and tangling. The three individual units (Unit A, B, and C) would be considered highly cohesive, if it were not for the tangling of the three concerns of logging, synchronization, and persistence. Furthermore, the scattering of these concerns would make it difficult to change their behavior, especially if the example were scaled to a much larger problem with thousands of units.

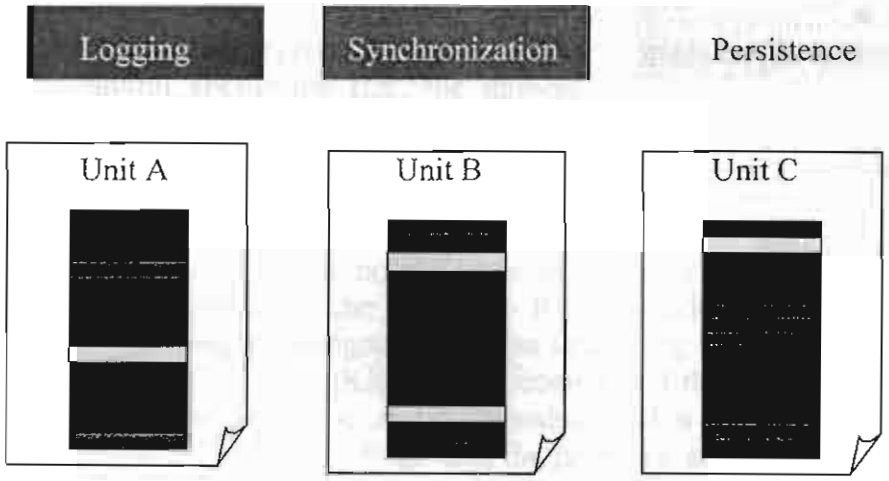


Figure 2.4: Crosscutting Concerns

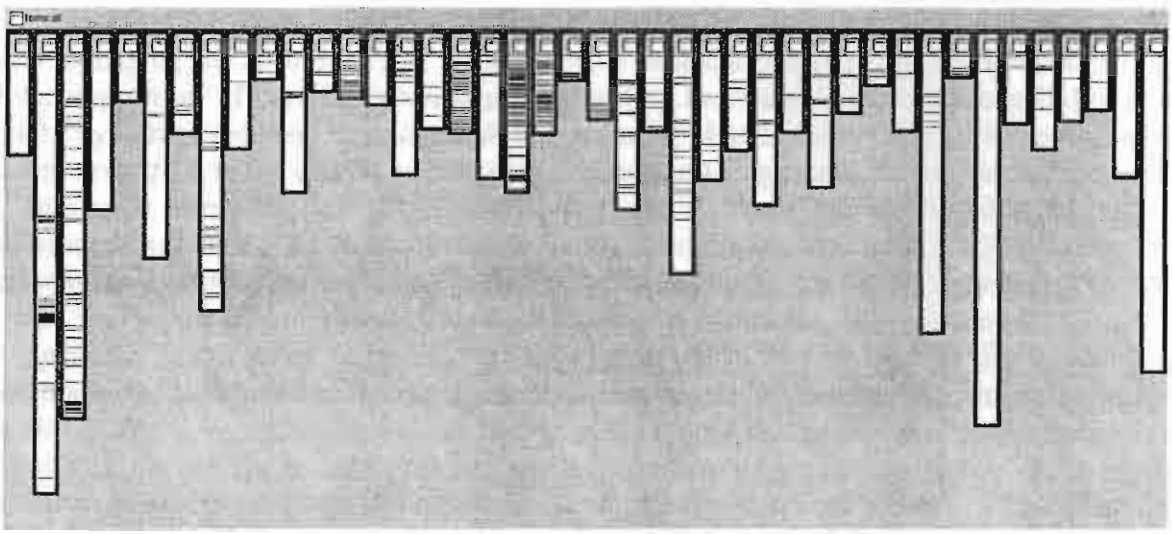


Figure 2.5: A Pictorial Representation of Crosscutting

This figure represents a piece of the Apache Tomcat code. Tomcat is an implementation of the Java Servlet and JavaServer Pages (JSP) specifications. Tomcat can run as a standalone, or it can be integrated into the Apache Web Server. The white vertical boxes represent a few of the classes in a subset of the Tomcat implementation. The highlighted lines designate the lines of code related to the concern of logging. Notice that the implementation of the logging concern is spread across the various classes. It is not located in a single spot. In fact, it is not even located in a small number of places. As reported in [Robillard and Murphy, 2002], a modification to the logging concern, "would require the developer to consider 47 of the 148 (32%) Java source files comprising the core of Tomcat." In this example, if the type of information to be logged is changed, then a developer may be required to make modifications to each of these 47 individual source files. From a software engineering viewpoint, this is not desirable. There is no cohesive module for representing the concept of logging – that concept is coupled among all of the other concerns. To highlight the importance of this, forget for a moment that the highlighted code in Figure 5 represents

logging. Assume, instead, that it represents all of the code for implementing the concerns of an employee in a payroll application (i.e., the implementation of employee features is scattered across multiple source files, in different modules). In that situation, it is easy to see that the basic principles of cohesion and coupling are being violated. The same can be said, then, when the highlighted concern is understood to be logging.

The problem just described is not the fault of a programmer who is guilty of poor design [Simonyi, 2001]. There is simply no traditional programming language construct that would permit a better localization of the concern – it is, "a lack of expressibility in the technology available to the original designer to express interacting or overlapping concerns" [Robillard and Murphy, 2002]. Gregor Kiczales has commented that, "Many people, when they first see AOP, suggest that concerns...could be modularized in other ways, including the use of patterns, reflection, or 'careful coding.' But the proposed alternatives nearly always fail to localize the crosscutting concern. They tend to involve some code that remains in the base structure" [Kiczales, 2001]. These alternatives require that the code related to the concern be placed in numerous locations.

Aspect-Oriented Programming

Programming language support for separation of concerns has long been a core aid toward managing the complexity of large software projects. Support for the modularization and decomposition of certain dimensions of a system has improved comprehensibility and evolvability during software development. For example, objects support the decomposition of a system according to the dimensions of data abstraction and generalization (via inheritance), and structured programming techniques focus on a functional decomposition. Other dimensions of concern often concentrate on features that are crosscutting (e.g., persistence is a crosscutting feature) [Tarr et al., 1999]. Most modularization constructs, however, provide for the separation of concerns along only one dimension. The dominant form of decomposition forces other dimensions of the system to be scattered across other modules. When non-orthogonal concerns are spread out across multiple modules, the system becomes more difficult to develop, maintain, and understand. Moreover, reusability of such concerns is not possible due to the crosspollination of one concern into many modules; there is no localized container to capture the concern. As implied in the first section of this chapter, reflection and metaprogramming were an early attempt at resolving crosscutting. These techniques were somewhat low-level, but provided a lot of expressive power. With MOPs, for instance, there is a blurred distinction between language user and language designer. Therefore, a more practical use of the techniques by less experienced programmers would require modularization constructs that offered more disciplined control over this power. As these techniques evolve, a new breed of programming languages is emerging to assist in the modularization of crosscutting concerns.

Aspect-Oriented Programming (AOP) provides a strategy for dealing with emergent entities that crosscut modularity [Kiczales et al., 1997]. AOP recognizes that crosscuts are inherent in most systems and are generally not random. The goal of AOP is to provide new language constructs that allow a better separation of concerns for these aspects. An *aspect*, therefore, is a piece of code that describes a recurring property of a program that crosscuts the software application (i.e., aspects capture crosscutting concerns). AOP supports the programmer in cleanly separating components and aspects from each other by providing mechanisms that make it possible to abstract and compose them to produce an overall system.

Gregor Kiczales and his colleagues at Xerox PARC developed the seminal ideas behind AOP in the mid-1990s. In *MIT Technology Review*, AOP was featured as one of the top 10 "Emerging Technologies That Will Change the World" [Tristram, 2001] and has been

the subject of a special issue of *Communications of the ACM* [Elrad et al., 2001]. Notably, object-oriented guru Grady Booch labeled AOP as, "something deeper, something that's truly beyond objects...a disruptive technology on the horizon" [Booch, 2001].

Aspects – A Complement to Traditional Paradigms

In the structured paradigm, modular block structures were used to provide scope for separating the boundaries of concerns. The "go-to" statements that often resulted in tangled and scattered concerns were replaced with procedure calls [Dijkstra, 1968]. This improved the control flow of a program and enhanced its modularization. The Object-Oriented (OO) paradigm represents the generation that followed the structured paradigm. In OO, the key modularization technique focused on hierarchical structuring through classes and inheritance. Another key feature of OO, a polymorphism permits variation of behavior within a class hierarchy.

Each new generation of modularity technology builds upon the previous generation. AOP should be evaluated within the context of being another technology for supporting separation of concerns. The ideas of AOP should be viewed as a counterpart to procedures, packages, objects, and methods to the extent that they all support different ways of modularizing certain kinds of concerns. In this sense, AOP can be regarded as a complement to both the structured and OO paradigm, or any other paradigm for software construction (e.g., logic programming [De Volder and D'Hondt, 1999]). In AOP, the focus is on capturing, in a modular way, the crosscutting concerns of a system. The crosscuts will still exist, but the problems of scattered and tangled code are removed by encapsulating the crosscut in a single module. To quote a personal communication with Gregor Kiczales, "OO made inheritance explicit in language. AO makes crosscutting explicit in language. OO makes its bet on hierarchical structures, but AOP makes its bet on crosscutting structures."

AOP has been defined in terms of its ability to provide quantification and obliviousness. *Quantification* is the notion that a programmer can write single, separated statements that introduce effects across numerous locations in the source code. Thus, quantification would provide the capability for saying the following: "In programs P, whenever condition C arises, perform action A" [Filman, 2001]. This can be stated more formally as: C [A], where the crosscutting nature is captured in the universal quantifier and the action to be performed within the concern is the parameterized action. The property of *obliviousness* holds when the quantified locations do not require modification in order to incorporate the effects of the quantification. As stated by the authors of this definition, "AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers" [Filman and Friedman, 2000].

The idea of quantification does suggest a special property of aspect languages, but quantification also exists within pure metaprogramming techniques. Even though metaprogramming is one way to capture crosscutting concerns, and AOP has its roots in metaprogramming, it should be understood that there are some important differences. Perhaps a better characterization of aspect languages, in order to avoid confusion, would be those languages that provide constructs for quantification, yet do not refer to metalevel concepts.

In a first exposure to AOP, many compare it to macro expansion. However, this comparison is far from accurate. Although there are similarities with respect to code being inserted or expanded, the AOP model is much more powerful. A limitation to the strength of macros is the fact that the transformations that are performed are textually local [Kiczales et al., 1992]. For instance, to use a macro, a programmer must visit numerous locations in the source code and insert the name of the macro. If a change needs to be made, or the macro

needs to be removed from a specific context, then the programmer must visit all of these points in the code. Macros do not exhibit quantification. Aspects, on the other hand, operate under the property of reverse inheritance (also known as inversion of control2). The behavior of an aspect is specified outside of the context where it is applied. Aspects, and their quantification, are described in one location – a programmer does not have to visit and insert code in any other place. This makes the addition and removal of aspects effortless.

It should be noted that the same distinction that has been made between AOP and macros could also be made in comparing AOP and *mixins* [Bracha and Cook, 1990]. A mixin is a class that is not intended to be instantiated. It provides some desired behavior (e.g., persistence) that is imported into other classes via inheritance. Mixin-based inheritance does not provide quantification and obliviousness. If a programmer wants to include mixin behavior in a class, the mixin must be explicitly imported within the purview of the class's predecessors. Mixin based inheritance is also missing the reverse inheritance property that can be provided through the kind of quantification available in aspect languages.

In comparing aspects to classes, there is almost an inverse relation between the way inheritance works in OO and the way aspects work in AOP. As stated in [Viega and Voas, 2000], "With inheritance, classes choose what functionality they wish to subsume from other objects. Aspects, on the other hand, get to choose what functionality other objects subsume."

Examples of Commonly Recurring Crosscuts

There are several commonly recurring crosscutting concerns that have been identified from a wide variety of different systems. For example, the software described in Figure 5 highlighted the fact that the common concern of logging is often scattered across the code base.

The study of operating systems code is ripe for the mining and understanding of crosscutting concerns. As pointed out in [Coady et al., 2001b], many of the key elements of operating systems crosscut. As an illustration, the prefetching activity that is performed in OS code is often highly scattered and tangled. As Coady and colleagues discovered, the FreeBSD v3.3 implementation of prefetching was spread across 260 lines of code in 10 clusters in 5 core functions from two subsystems. A refactoring of the prefetching implementation using an aspect language demonstrated an increased comprehensibility of the code with respect to independent development, as well as the ability to (un)plug different modes of prefetching [Coady et al., 2001a]. Their future research focus is in the investigation of other crosscutting concerns in FreeBSD; namely, scheduling, communication protocols, and the file system. It is also often the case that the implementation of specific protocols lead to tangled code, as does code that is introduced into the system to improve some performance optimization. This also can be true in implementations that provide resource sharing among a set of objects. The various policies, or protocols, contained within an operating system are typically implemented in a crosscutting manner. This is similar to the observation made in Chapter 1 concerning policy implementations that have been studied in organization theory.

Perhaps the two most commonly observed crosscutting concerns are synchronization and exception handling. Both of these are also evident in the case studies of Appendix A. A detailed analysis has been performed on the ability of AOP to remove redundant code in exception handling [Lippert and Lopes, 2000]. This study looked at the code for JWAM, a framework for interactive business applications, which is implemented in over 614 Java classes in 44,000 lines of code. It was discovered that 11% of the overall code was focused on the concern of exception handling. The core of their work involved a refactoring of the exception handling code into AspectJ. The benefits of this refactorization are obvious. In many types of exceptions, they were able to reduce the amount of redundant code by a factor

of 4. Of the top five types of exceptions in the JWAM application, over 90% of the number of catch statements was removed. For example, the number of catches of the generic Exception type went from 77 in the original code to only 7 catches in the refactored AspectJ code. Similarly, the number of catches of the SQLException type went from 46 catches in the original code to only 2 in the aspectized code. Because the JWAM application was written using *Design by Contract* [Meyer, 1997], there are many assertions that test the pre- and post-conditions for a particular method. Lippert and Lopes found that over 375 post-conditions contained an assertion of "result != null" – this redundant assertion represented 56% of all post-conditions (here, redundancy referes to the replication of a single statement at the end of multiple methods). There were also 1,510 pre-conditions that contained the assertion of "arg != null"; using AspectJ, that number was cut down to 10. That is, the 1,510 pre-conditions were separated into 10 aspects, where each aspect contained a concise specification of the methods that were to contain the assertion.

The idea of superimposition, which is related to the "diffusing computation" concept initially proposed in [Dijkstra and Scholten, 1980], has recently been compared to aspect-orientation. A superimposition has been found helpful in distributed systems for maintaining and changing the global properties related to a distributed computation (e.g., deadlock detection, or the snapshot algorithm in [Chandy and Lamport, 1985]).Typically, the implementation that manages each globally distributed property is scattered in two ways: it is scattered across the processes that perform the distributed computation, and it is scattered across the source code implementation that is charged with the task of maintaining the global property. It has been noted that, "Algorithms which were intentionally designed to superimpose additional functionality on a basic program have a long history in distributed systems research, probably starting with algorithms to detect termination of basic algorithms" [Katz and Gil, 1999]. Like aspect orientation, superimpositions impose additional functionality to a base program through quantification.

Enforcing Programmer Discipline

Aspects can be used to enforce certain properties of a system that would typically be left to programmer discipline. To understand this point, reconsider the trigger example from Figure 1. Rather than using a trigger, a database administrator could have written a stored procedure, called UpdateSalary, which provides a single point of control for updating the salary field of the employee table. The UpdateSalary stored procedure could then contain, in one location, the semantics for implementing the business rule.

This solution, however, does not provide any guarantee that others will obey the rule for using only this stored procedure. There is nothing to prevent a user or developer from updating the table through means other than the stored procedure. The reliance on programmer discipline is unfeasible in large systems, and it is quite likely that certain system properties are violated when there is no direct way to enforce the concern. Aspects can be helpful in enforcing that a particular policy, or protocol, is observed in a way that does not rely on the programmer remembering to conform to a large set of unverifiable rules.

AspectJ

Early aspect languages, like COOL and RIDL [Lopes, 1997], dealt with specific types of concerns (e.g., synchronization and distribution). The most mature language, however, is a general aspect language (called AspectJ) that is an extension to Java. It is described as being general because it is not tied to capturing a particular kind of concern; instead, it provides general constructs that allow a programmer to capture a wide variety of different kinds of

concerns. The language definition has undergone many changes since the first description in [Kiczales et al., 1997] to the most recent implementation, as documented in [Kiczales et al., 2001a] and [Kiczales et al., 2001a]. This section highlights some of the key characteristics of AspectJ. AspectJ is being used in commercial development. CheckFree.com, which provides financial services for e-commerce, uses AspectJ [Miller, 2001]. An interesting anecdote is reported from this effort. A senior engineer at CheckFree stated that AspectJ allowed his team to implement a crosscutting feature in four programmer-hours. The same feature, implemented in a previous version of the application in C++, is reported to have taken two programmer-weeks [Tristram, 2001]. It has been proposed that there are three critical parts to an aspect composition language: a join point model, a way of denoting joins points, and the ability to specify behavior at those join points [Kiczales et al., 2001b].

Join Points and Pointcuts

In AOP languages like AspectJ, a *join point* denotes the location in the program that is affected by a particular crosscutting concern. This location can be either the static location of a specific line of source code, or it can represent a dynamic point during the execution of the program. There are many potential join points in a program. A *pointcut* specifies a collection of join points. The AOP literature does not provide the etymology of this term. Perhaps the intent of the terminology comes from graph theory, where the notion of a cutpoint represents a vertex in a graph whose removal would leave the graph in a disconnected state. It is a point of separation between nodes in a graph. Analogously, a pointcut is a place of potential separation for non-orthogonal concerns. A pointcut designator is declarative and permits the composition of join points using logical operators. There are many different types of pointcut designators. Several designators that will be used in a later example are:

- this(T): all join points where the currently executing object is an instance of class T
- target(T): all join points where the target object of a call is an instance of class T
- call(S): all join points (in a calling object) that are matched by a call specified by signature S
- cflow(C): this powerful designator selects all join points within the control flow of pointcut C

Advice

Whereas a join point represents a location where an aspect adds behavior, *advice* represents the behavior to add (Note: The name "advice" was chosen because it is similar to the advice feature in early Lisp machines). Advice represents a type of method that can be attached to pointcuts. The definition of an advice relates a pointcut with specific code, contained in the advice body, which takes care of the crosscutting concern. The body of the advice is normal Java code. There are three different designators for specifying the point of execution for advice: before, after, and around. The choice of these names appears to have been borrowed from CLOS [Steele, 1990]. In before advice, the advice body is executed prior to the execution of the join point's computation. The opposite is true with after advice; the advice runs after the join point computation. There are even three different kinds of after advice:

- After the successful execution of the join point (after returning);
- After an error was encountered during the execution of the join point (after throwing);
- Either of the above two cases (after).

Separation of concerns often necessitates subsequent integration. Whereas AOP provides the capability of separating numerous concerns during development, the effects of the crosscuts must be integrated back into the target code. The goal of the separation is to improve the conceptual ability of programmers during development – the end result at run-time, however, will certainly have crosscutting concerns that are transparent. As David Weiss states, in his introductory comments to one of Parnas' papers, "At run-time, one might not be able to distinguish what criteria were used to decompose the system into modules" [Hoffman and Weiss, 2001]. In AOP, a translator called a weaver is responsible for taking code specified in a traditional programming language, and additional code specified in an aspect language, and merging the two together. Because the aspect code describes numerous behaviors that crosscut a system, the concerns must eventually be integrated into the base code. This is the purpose of a weaver – it integrates aspects into the base code. In Figure 2.6, the weaving process is depicted using the previous example in Figure 2.4.

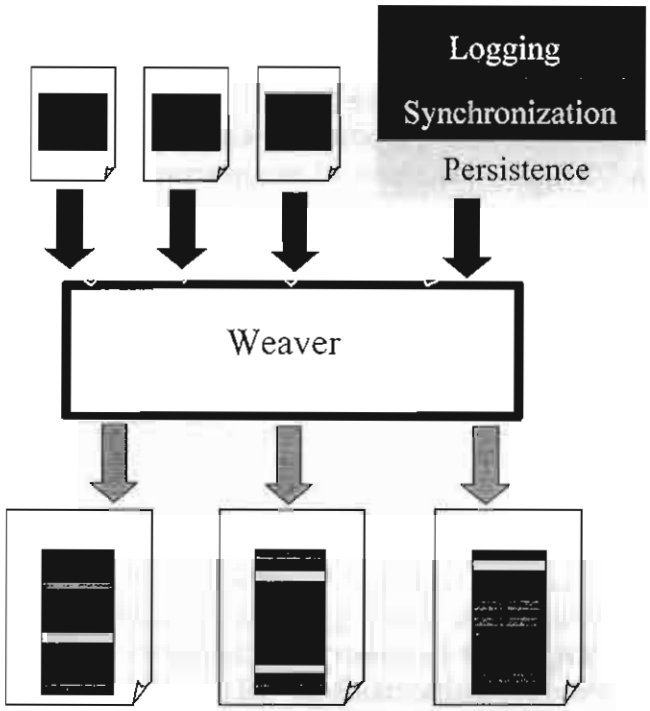


Figure 2.6: The Weaving Process

Other Work in Aspect-Oriented Software Development (AOSD)

Several researchers are working in the area of AOSD to provide new language constructs to support crosscutting concerns [Tarr et al., 1999]. Aside from AOP, other examples of specific research in this area are Subject-Oriented Programming (SOP) [Osher et al., 1996], variants of Adaptive Programming (AP) [Lieberherr et al., 2001], and Composition Filters (CF) [Bergmans and Aksit, 2001]. A hybrid approach to applying these techniques has been suggested in [Rashid, 2001]. Several of these research areas can be considered a part of generative programming, the topic of the next section.

Multi-Dimensional Separation of Concerns (MDSOC)

Another successful approach for dealing with crosscutting concerns is Subject-Oriented Programming (SOP), a research effort at IBM Research. In this approach, it is recognized that objects have different roles that they represent. These different roles can be composed into system features [Ossher et al., 1996], [Ossher and Tarr, 2001]. For example, in an Employee class, an employee object plays different roles depending on whether the Employee is being sent to the payroll subsystem (where salary and tax information are pertinent) versus the same Employee being sent to the human resources, or personnel, subsystem (where years of service and address are appropriate). The separation of these roles into isolated views is referred to as a "hyperslice" [Tarr et al., 1999]. Hyperslices assist a team of programmers in independently developing different concerns that may apply to a single class. Note that this capability was one of the Parnas' criteria described in the first chapter [Parnas, 1972].

Earlier work on subdivided procedures provided a basis for the approach adopted in SOP [Harrison and Ossher, 1990]. Subdivided procedures promote extensible programming by separating the multiple cases of procedure bodies. A procedure that dispatches from a large case statement would be an example application of subdivided procedures. In such instances, the individual cases that comprise the procedure are somewhat related to the notion of a hyperslice. An interesting comparison can be made between AOP and SOP. With AOP, the focus has always been on crosscutting concerns that are spread across multiple modules. A focus of SOP, however, has been the ability to capture several views of a single class. The separation of these views, it is argued, permits a better understanding of the implementation of each view in isolation so that the views do not become tangled. In the SOP literature, a translator called a *compositor* has numerous similarities to a weaver in AOP. A programmer creates composition rules that direct the output of the compositor [Ossher et al., 1996]. A tool called Hyper/J has been developed to support the idea of hyperslices in Java.

Adaptive Programming

The structure of objects within a class hierarchy has been found to be a type of crosscutting concern. In Adaptive Programming (AP), a key focus is the separation of behavior from structure. To aid in the modularization of this concern, visitor and traversal strategies are used [Lieberherr, 1996]. This modularization prevents the knowledge of the program's class structure from being tangled throughout the code, a desirable property that is called "structure shyness." Traversal strategies can be viewed as a specification of the class graph that does not require the hardwiring of the class structure throughout the code [Lieberherr et al., 2001]. An example of a traversal/visitor language for supporting such modularization is described in [Ovlinger and Wand, 1999]. The AP community considers their research as a special case of AOP. The motivation for AP came from the earlier work on the *Law of Demeter*, which offered a set of heuristics for improving the cohesion and coupling of object-oriented programs (the motto of this work was the anti-social message of "Talk only to your immediate friends") [Lieberherr and Holland, 1989]. In previous work at ISIS, an adaptive programming approach was used to solve a tool integration problem for a large aerospace firm [Karsai and Gray, 2000]. The domain for the integration focused on fault-analysis tools, where each tool persistently stored a model in either a database or a textual format (e.g., either comma-separated values, or a proprietary format). In that work, a model from one tool was translated into the representation of another tool. To accomplish this, semantic translators were used to traverse the graph of an internal representation of a

model. In a semantic translator, the specification of the traversal, and the actions to be performed at each traversed node, are separated.

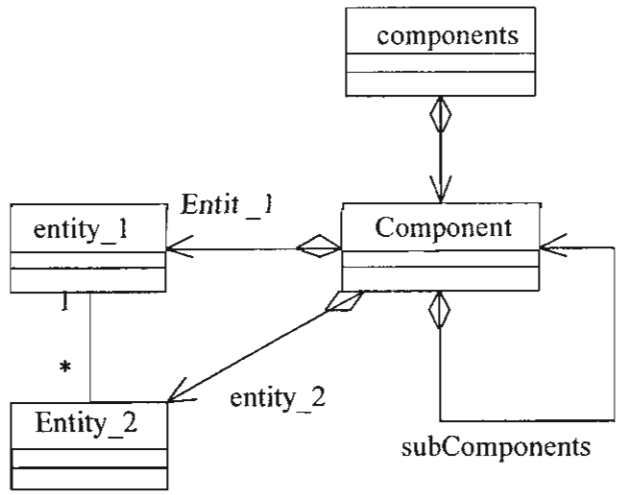


Figure 2.7: A Simple UML Tool Model Specification

The illustration in Figure 2.7 represents a simple model that is specified in the Unified Modeling Language [Booch et al., 1998]. A domain-specific language (DSL) for textually representing this diagram is presented in [Karsai and Gray, 2000]. Another DSL is shown in Figure 2.8, which demonstrates the traversal/visitor specifications that appear within a translator. During a translation, the process begins with the top model and follows along the traversal specifications. At visitor nodes, a specific action is performed that executes the

```
visitor Visitor                    visitor Visitor
{                                  {
  at Component[...]                  at Component[...]
  <<...>>                            <<...>>
   traverse[...];                     traverse[...];

  at Entity_1[...]                   at Entity_1[...]
  <<...>>;                           <<...>>;

  at Entity_2[...]                   at Entity_2[...]
  <<...>>;                           <<...>>;

  at Rel[...]                        at Rel[...]
  <<...>>                            <<...>>
   traverse[...];                     traverse[...];

}                                  }
```

Figure 2.8: Traversal/Visitor Specifications

required translation (these are elided inside of the inline code, which is denoted as <<...>>). In Figure 2.8, the first two steps in the model translation are shown by two arrows. The remaining traversal/visitor sequence would follow similarly.

Composition Filters

An earlier effort at isolating crosscutting concerns is the composition filters approach. With this technique, explicit message-level filters are added to objects and the messages that they receive [Aksit et al., 1992], [Bergmans and Aksit, 2001]. The motivation for composition filters came from the recognition that conventional object models lack the required support for separating functionality from message coordination code. As objects send messages to each other, the messages must pass through a layer of filters. Each filter has the possibility of transparently redirecting a message to other objects. Different types of filters have been found to be effective at isolating constraints and error checking [Aksit et al., 1994]. The CF approach can be very useful in executing actions before and after the interception of a method call. A related technique, proposed in [Filman et al., 2002], intercepts communication among functional components and injects behavior to support various additional capabilities (e.g., reliability, security). CORBA interceptors [Narasimhan et al., 1999] have some similarities with composition filters because they also can intercept messages. There are many exciting things on the horizon for research in aspect-oriented software development. The remainder of this section surveys some of these other research areas.

Weaver Development and Tool Support

Some of the earliest aspect languages and weavers were focused on specific concerns like synchronization and distribution. Examples of these particular aspect languages include COOL and RIDL, as defined in the dissertation of Cristina Lopes [Lopes, 1997]. More recent work, like AspectJ, has focused on generic aspect languages. Aside from Java and AspectJ, other languages are being explored with respect to AOP. The use of AspectC was cited earlier in the discussion of prefetching [Coady et al., 2001]. Although there are many difficulties in writing a C++ parser, initial efforts at providing an AspectC++ weaver (in support of real-time systems) are reported in [Gal et al., 2002], [Mahrenholz, 2002]. AspectS is an approach to general-purpose AOP in the Squeak environment [Hirschfield, 2001]. Apostle is an aspect weaver for Smalltalk [de Alwis, 2001]. A simple weaver even exists for Ruby [Bryant and Feldt, 2001]. Additionally, there has been work on making the CORBA IDL aspect-oriented [Hunleth et al., 2001], as well as efforts for bringing AOP into the realm of Microsoft .NET [Shukla et al., 2002], [Lam, 2002]. All of the weavers mentioned above are typically much more immature than the capabilities offered in AspectJ, yet they provide the major impetus for taking the ideas of AOP to other languages. In addition to weaver development, there are several other development tools that are being created to support AOP. A debugger for AspectJ, with GUI support, is available. There also has been effort to support AspectJ within several Integrated Development Environments (IDEs). Another related interesting research area is the application of AOP to compilers. As observed in [Tsay et al., 2000], "The code to do one coherent operation is spread over all node classes, making the code difficult to maintain and debug." The advantages of using AOP techniques for a weaver can be found in [de Moor et al., 1999]. In their work, the descriptions of the effects on attribute grammars are separated from the grammar productions. The benefit of this was also recognized in [VanWyk, 2000].

Debugging Aspect Code

Many aspect weavers are preprocessors that target their output code in another traditional programming language. Given the obfuscation created by the mangled names, and

the numerous indirections present in the generated code, it seems that there is a mismatch between the implementation space and the execution space. That is to say, how does a programmer write code using a particular conceptualization, and then debug the generated code that is void of that conceptualization? This question is not peculiar to AOP – the problem can be found in almost any implementation of a domain-specific language [Faith et al., 1997], [van Deursen and Knit, 1997]. To answer the question concerning the debugging of aspect code, it should be recognized that AOP is still in its early infancy. Although tool support is being developed, such as an aspect debugger, the technology is still immature. Yet, it is reasonable to expect future tools will be developed that will make the underlying execution transparent to the paradigm. In fact, the path that AspectJ is taking is not unlike the development of the earliest C++ compilers. The initial C++ compilers were merely preprocessors that generated C code. The resulting C code was void of any semblance of true object-oriented concepts – the C++ representation was merely simulated in a language that had more mature compilers. The same can be said of AspectJ and other languages concerning the incubation period needed for growth and stabilization. Perhaps a future solution to this problem will be found in an adaptation to the work in [Faith, 1997], which describes a tracking engine that interacts with a debugger and maps nodes from syntax trees.

Analysis and Design with Aspects

A study of the history of software development paradigms reveals that a new paradigm often has its genesis in programming languages and then moves out to design and analysis, or even other research areas (see [Rashid and Pulvermueller, 2000] for a description of aspects applied to databases). This same pattern also can be observed with respect to aspect-orientation. Most of the existing work on advanced separation of concerns has been heavily concentrated on issues at the coding phase of the software lifecycle. There have been, however, efforts that have focused on applying advanced separation of concerns in earlier phases of the software lifecycle. One of the first examples of this type of work can be found in [Clarke et al., 1999], where the principles of SOP were applied at the design level. Similarly, [Herrero et al., 2000] have investigated the benefits of aspects at the design level. Extensions to the UML have been proposed in order to support composition patterns as a facility for handling crosscutting requirements [Clarke and Walker, 2001], [Clarke, 2002]. A set of generic design principles for aspect-oriented software development is the focus of [Chavez and de Lucena, 2001]. An analysis of design patterns, and the aspect oriented techniques that can improve their specification and implementation, are the subject of [Nordberg, 2001]. There has been an increased interest in the need for formal verification of systems designed with support for crosscutting concerns. The most mature effort in this area can be found in [Nelson et al., 2001], where two formal languages are presented that assist in the verification of concerns focused on concurrent processes.

Aspect Mining

There is an overwhelming amount of legacy code that has been written in languages that do not support the clean separation of crosscutting concerns. To convert legacy code into languages that support AOSD, it is necessary to refactor the original program. A correct refactoring into a cleaner separation of concerns requires the examination of the original code with an eye toward aspect mining (i.e., the identification and isolation of aspects). An aspect mining tool offers assistance in this process. The Aspect Browser tool, presented in [Griswold et al., 2001], is such an example. The tool has been applied to a case study that

contained 500,000 lines of source code in FORTRAN and C. Another tool for aspect mining is described in [Hannemann and Kiczales, 2001].

AOP Validation Research

Case studies that transform legacy applications into AspectJ, like [Lippert and Lopes, 2000] and [Kersten and Murphy, 1999], provide practitioners with heuristics for adopting AOP. Both a case study and an experimental method were used in [Walker et al., 1999] to assess AOP. In an experiment that studied the ease of debugging, three synchronization errors were introduced into a Java program. A separate program that duplicated the errors was also written in AspectJ. Several teams of programmers were given the task of tracking down the errors in each of the implementations. The results of this experiment show that AspectJ provided a clear benefit to increasing localized reasoning, but no benefit when the solution required non-localized reasoning. Here, localized reasoning refers to whether or not a programmer needs to leave the context of the module (in this study, the file) that contains the error. Overall, the program teams that used AspectJ isolated and fixed the errors quicker than those who used pure Java. There are case studies that have compared the various different mechanisms for supporting advanced separation of concerns [Murphy et al., 2001]. Obviously, as AOP matures, additional studies will be needed to determine the benefits of these new approaches.

Aspect Reuse

As a large collection of different types of aspects is assembled, the idea of aspect reuse will become an interesting research topic. AOP presents new issues for reuse researchers [Grundy, 2000]. In order to be successful at aspect reuse, developers will need to begin writing their aspects in a more generic style than is currently prevalent. To see why this is so, consider the code fragments that are provided. The pointcuts of these aspects are concretized and bound specifically to the methods called DisplayError and Handle. This assumption is too strong. It may often be the case that others will want to reuse this aspect, but their code does not conform to these concrete names. To remedy this problem, a style of pointcut designation is needed such that the pointcuts of the reusable aspects are abstract. In this case, those who would wish to use and extend an abstract aspect must concretize it. In fact, AspectJ permits such designations, but its use is very infrequent in the current aspect code that is being developed. Some of the issues in support of aspect reuse and composition have been initially explored in the work on aspectual components [Lieberherr et al., 1999].

Another research issue occurs in the reuse of orthogonal aspects that apply to the same join point. This issue is important because the ordering of the generated code may be essential. For example, given the two previous aspects of locking and logging, it is often the case that, when applied to the same join point, the mutex code should appear before the logging instructions. AspectJ provides the dominates construct to allow the specification of priority between two different aspects. It is unclear, however, whether this construct alone is able to allay all of the possible problems in composing several aspects within the same join point.

Generative Programming

The first FORTRAN compiler took 18 programmer-years to complete [Backus et al., 1957]. One could argue that the time that it would take today to write an equivalent compiler would be on the order of programmer-months, not programmer-years. Of course, much of the

decreased development time would be related to the experience that has been collected on the topic of compiler construction. Most would agree, however, that the principal reason for the decreased development time would be that we have moved beyond the manual handcrafting of "one-of-a-kind" solutions to an approach that resembles an automated assembly line. To be specific, in the case of implementing a simplistic version of a FORTRAN compiler, a programmer today would use parser generators, specialized components, and perhaps even object-oriented frameworks. In implementing a compiler using modern techniques, the reduction in development time is the result of a paradigm shift toward the engineering of families of systems, as proposed in [Parnas, 1976]. The idea of a family of systems is best categorized as a domain-specific product-line architecture, where a set of different products can be created from adaptations that are made from a set of varying features [Clements and Northrop, 2001]. An excellent example of this idea is found in [Delisle and Garlan, 1990], which describes development at Tektronix on a family of oscilloscopes. An additional contributing factor to the relative ease in constructing a modern-day FORTRAN compiler is in the recognition that many of the arduous implementation details of software construction can be handed off to a generator. This paradigm shift has led toward a research area that has been dubbed Generative Programming (GP). Generative programming is accomplished by transforming higher-level representations of programs into a lower-level equivalent representation. This section surveys several of the promising research areas that are being associated with the GP movement. More detailed coverage of GP can be found in [Czarnecki and Eisenecker, 2000].

Domain-Specific Languages

A Domain-Specific Language (DSL) is a, "programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" [van Deursen et al., 2000]. DSLs assist in the creation of programs that are more concise than an equivalent program written in a traditional programming language. An upward shift in abstraction often leads to a boost in productivity. It has been observed that a few lines of code written in a DSL can generate a hundred lines of code in a traditional programming language [Herndon and Berzins, 1988]. A key advantage is that a DSL is perspicuous to the domain expert using the language. A DSL is typically more concise because much of the intentionality of the domain is built into the generator. To use a connotation borrowed from Polya, the intent of a DSL is "pregnant with meaning" [Polya, 1957]. A DSL can assist in isolating programmers from lower-level details, such as making the decisions about specific data structures to be used in an implementation. Instead, a programmer uses idioms that are closer to the abstractions found in the problem domain. This has several advantages:

- The tedious and mundane parts of writing a program are automated in the translation from the DSL to a traditional programming language.
- Repetitive code sequences are generated automatically instead of the error-prone manual cut-and-paste method. The generation of error-prone code also has advantages during the maintenance phase of a project's lifecycle. Programs written in a DSL are usually easier to understand and modify because the intention of the program is closer to the domain.
- Solutions can be constructed quickly because the programmer can more easily focus on the key abstractions.

The size and scope of a DSL is much smaller than that of a traditional programming language. In fact, DSLs are often called "little languages" [Bentley, 1986], [van Deursen and

Knit, 1997], [Aycock, 1998]. Another common characteristic is the declarative nature of these languages. In some cases, a DSL can be viewed as a type of specification language in addition to a general purpose programming language. A DSL can be declarative because the domain provides a particular underlying interpretation. The notations and abstractions of the domain are built into the generator that synthesizes a program written in a DSL. A DSL translator can be implemented using the standard approaches for constructing a compiler or interpreter [Aho et al., 1986]. However, the majority of the literature implements DSLs with a preprocessor. Although this approach can be simpler than writing a complete compiler, it has several disadvantages. The main disadvantage is that the generated code is converted to a base programming language. This means that type checking and other compile-time tests are done outside of the domain. It also means that feedback from run-time errors are couched in terms of the base language, not the domain. A solution to this problem (previously cited in the section on "Debugging Aspect Code") is suggested in [Faith, 1997]. There are other disadvantages in using a DSL that often arise later in the development cycle. As observed in [van Deursen and Knit, 1997], the use of a DSL introduces new maintenance issues. For instance, the generators that process the programs in a DSL may often need maintenance.

Example Domains

There are numerous domains where DSLs have been applied. Some of the example domains are telecommunications [Bonachea et al., 1999], operating systems [Puet al., 1997], typesetting and drawing [Bentley, 1986], web services [Fernández et al., 1999], caching policies [Barnes and Pandey, 1999], [Gulwani et al., 2001], and databases [Horowitz et al., 1985]. The concept of a domain-specific metalangauge has also been put forth as a technique for assisting in the domain of language translators [Van Wyk, 2000]. An extensive annotated bibliography of research in this area can be found in [van Deursen et al., 2000]. Domain-specific modeling has been successfully applied in several different domains, including automotive manufacturing [Long et al., 1998], digital signal processing [Sztipanovits et al., 1998], and electrical utilities [Moore et al., 2000].

Compilers for DSLs have often been called *application generators* [Horowitz et al., 1985], [Cleaveland, 1988], [Smaragdakis and Batory, 2000]. A *generator* is a tool – a type of translator or compiler – that takes as input a domain-specific language and produces as output source code that can be compiled as a traditional programming language. The internal architecture of a generator is very similar to a compiler. A generator requires: a front-end to parse a source language into an intermediate representation, a translation engine to perform transformations and optimizations, and a back-end to produce the target code. In [Hunt and Thomas, 2000], a distinction is made between passive code generators and active code generators. In a *passive code generator*, the generator is executed just once to produce a result. After the output of a passive generator is obtained, the result becomes freestanding. The origin of the file is forgotten. An example of this type of generator would be a design wizard, like that described in [Batory et al., 2000]. With a wizard, a user enters various configuration data as a response to interacting with a dialog window. Based upon this configuration information, the wizard can then generate code that would have been tedious to create by hand. The code produced from an *active code generator*, though, frequently hanges such that it is advantageous to invoke the generator on variations of the input. There is some evidence that generators improve productivity and reliability. A comparative experiment for a Command, Control, Communication, and Information (C^3I) system is described in [Kieburtz et al., 1996]. This experiment compared the use of generators with a previously developed Ada template-based approach for implementing message translation and validation. The results of this experiment show that the teams that used the generator approach were three

times more productive than those who performed the same task using templates. The generator approach also realized improvements in reliability, with under half as many test run failures.

GenVoca

GenVoca permits hierarchical construction of software through the assembly of interchangeable/reusable components [Batory and Geraci, 1997]. The GenVoca model is based upon stacked layers of abstraction that can be composed. A *realm* is a library of plug-compatible components. It can be thought of as a catalog of problem solutions that are represented as pluggable components that can be used to build applications in the catalog domain. Each realm exposes a common interface that all components in that realm must satisfy. This provides the ability to have many alternative implementations for the same interface. The layered decomposition of implementations offers component composition that is similar to the stacking of layers in a hierarchical system. Each realm in the hierarchy is denoted by a GenVoca grammar. This grammar describes all of the legal compositions that may occur within the realm. The composition of components in GenVoca is performed by writing parameterized type expressions. These expressions are checked against the grammar to preserve validity.

A comparison between GenVoca and AOP is made in [Cardone, 1999]. Both aspect languages and GenVoca type equations guide the transformation of programs. The AOP weaver and the GenVoca generator are the preprocessors that implement such transformations. GenVoca has the capability of validating the correctness of component compositions. This is an issue that has not received much focus within the AOP research community. As mentioned in an earlier section, control over the order in which a weaver applies multiple aspects on the same join point is very limited. GenVoca, though, provides control over the ordering of component composition.

Intentional Programming

Intentional programming (IP) provides a software development environment that is not tied to a specific programming language. The power of IP is the ability to create new abstractions for languages. It allows the tailorability of a specific language to a new domain. As Charles Simonyi states, "Under IP, domain experts write models/specs/programs in domain terms" [Simonyi, 2001]. The IP system provides the functionality for defining the manner in which these new abstractions interact with the environment's text editor, as well as syntactic and semantic constructs for translating these extensions to the abstractions already supported in the IP system [Simonyi, 1996]. Thus, IP allows a programmer to write ordinary programs and domain transformations. The nodes of an Abstract Syntax Tree (AST) typically represent the semantic constructs of a language (e.g., a *while*-loop or *if*-statement). In IP, these nodes are called *intentions*. Many intentions are common across a wide variety of programming languages. The IP environment provides the capability to modify the semantics of an intention for a particular language, as well as introduce new intentions peculiar to that language. New intentions introduce their own syntax in addition to prescribing the effects of interactions with the programmer through an editor. The IP concept of an *enzyme* represents a transformation that is performed on an AST. An enzyme assists in the creation of new intentions that are built on top of existing intentions.

Parser Generators, Language Extenders, and Analysis Tools

Parser generators, like the Purdue Compiler Construction Tool (PCCTS) and YACC (Yet Another Compiler-Compiler), are programs that help in the creation of other programs that perform transformations on source code [Parr, 1993]. In the area of parser generators, an example of an extensible framework for building compilers in Python is described in [Aycock, 1998]. A framework that creates ASTs and associated tree-walker classes, based on the Visitor pattern [Gamma et al., 1995], is described in [Gagnon, 1998]. Other compiler frameworks, like Zephyr [Wang et al., 1997] and SUIF [SUIF2, 2000], provide an extensible framework to support collaborative experimental research. A primary goal of these efforts is to provide an infrastructure to benchmark different techniques that are used in compilers.

The Jakarta Tool Suite (JTS) contains the basic tools to support the addition of new programming features to the Java language [Batory et al., 1998]. It assists in the construction of new preprocessors for DSLs that are transformed into a host language. The supported host language in JTS is called Jak. Jak is described as a superset of Java that supports metaprogramming. It seems likely that JTS could be used to create a weaver for new aspect languages to support Java. The JTS environment builds upon the ideas of GenVoca. Each new extension to Java represents a new realm. Within the context of the Ptolemy project, a code generator for transforming Java programs is available [Tsay et al., 2000]. This generator is situated within an infrastructure that can parse Java programs and perform transformations on the AST using the Visitor pattern [Gamma et al., 1995].

Frameworks

A *framework* can be defined as a skeleton of an application that can be extended to produce a customized program [Fayad et al., 1999]. This type of framework is usually defined as a collection of classes that together help support a domain-specific architecture. A framework architecture must define the objects that are to participate in the framework as well as the interaction patterns among all objects. In this architecture, there is a distinction between those who create the framework and core objects (the framework developer) and the programmer who extends the framework by plugging in their own application objects (the application programmer). Frameworks typically cost more to develop than a single application, although their cost can be amortized over each instantiation [Johnson, 1997].

Adaptability in frameworks is provided by factoring out component objects that implement the core functionality in the application domain from those objects that vary with each instantiation of the framework. A framework instantiation is defined as the insertion of instance-specific classes into the framework architecture. The locations of variability within a framework are referred to as the *hot spots* of the framework [Lewis, 1995]. The instance-specific classes must conform to a predefined interface in order to properly interact with the core objects. The specification of the hot spots is needed for users of the framework because frameworks exhibit the property of inversion of control. In typical software development, the components that are written contain the locus of control in the application and selectively pass control onto other library components or lower-level calls to an Application Program Interface (API). In a framework, however, the locus of control resides in the framework, rather than the application objects. The flow of control traverses through the objects of the framework until a hot spot is reached, at which time the application object is dispatched.

Event-based infrastructures also demonstrate the principle of inversion control [Gianpaolo et al., 1998]. In an event-based approach, there is a distinction in the architecture between suppliers, consumers, and the event dispatcher. Suppliers submit events to a mediating dispatcher that forwards events to all consumer objects that have subscribed to the

event (suppliers may also be consumers of other events). The asynchronous nature of the consumers suggests a type of control inversion that provides a high degree of dynamic reconfigurability within distributed object computing. A popular example of this architecture is present in the CORBA event service [Harrison et al., 1997].

Frameworks have been developed in practically every domain that supports variability among a family of products [Fayad et al., 1999], [Fayad, 2000]. One particular interesting research area combines the topic of a previous section (AOP) with a framework for a concurrent object system [Constantinides et al., 2000].

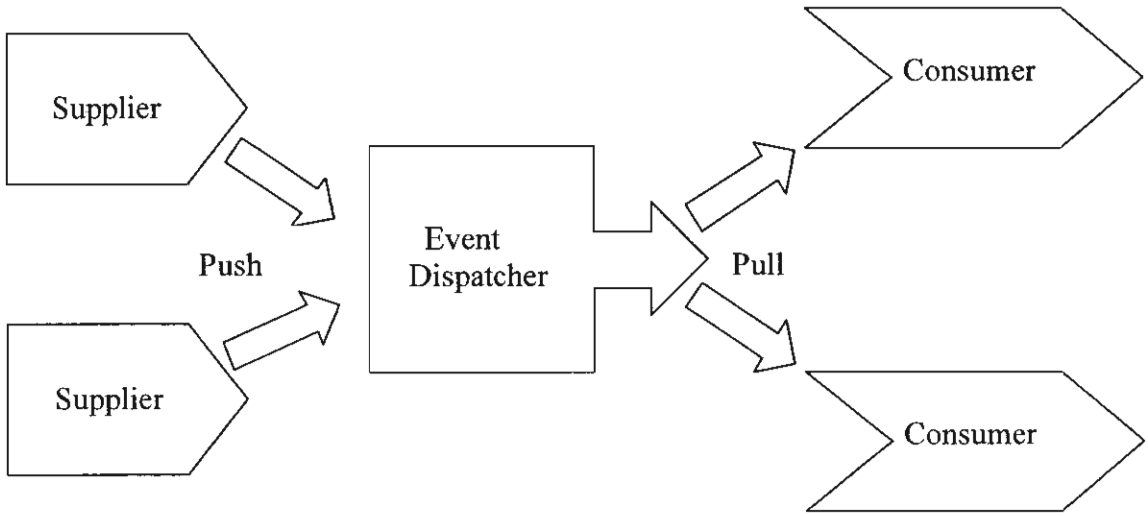


Figure 2.9: Architecture for Event-based Dispatching

Summary

This chapter provided a synopsis of the techniques that are useful in the development of software that must adapt to changing requirements. The first half of the chapter presented an overview of the literature on reflection, metaprogramming, and AOSD. The research in these areas has produced new ideas and methods for improving adaptability, and extensibility for separating crosscutting concerns. This separation provides an advantage for realizing the three objectives presented by Parnas (see "Criteria for Decomposition" in the Chapter 1). The second half of the chapter surveyed research that can be classified under the general area of Generative Programming. A generative approach captures the intent of the problem space at a higher level of abstraction. Generators map the higher abstractions to the lower-level details in the solution space. In the next two chapters, these techniques (e.g., reflection and metamodeling, advanced separation of concerns, and generative programming) will be extended to support aspect-oriented domain-specific modeling.

CHAPTER III

THE FRAMEWORK

In Aspect-Oriented Programming we decompose a problem into a number of functional components as well as a number of aspects and then we compose these components and aspects to obtain system implementations. The goal is to achieve an improved separation of concerns in both design, and implementation. Our work concentrates on the aspectual decomposition of concurrent object-oriented systems. Following the component hierarchy within the object-oriented programming paradigm we categorized aspects as intra-method, intra-object and intra-package according to their hierarchical level of cross-cutting. We achieve composition of concerns; through the use of an object we call the moderator that coordinates the interaction of components and aspects while preserving the semantics of the overall system. Since aspects can crosscut components at every level, we view the moderator is a recurring pattern from intra-method to intra-package. Our design framework provides an adaptable model and a component hierarchy using a design pattern. The moderator pattern is an architecture that allows for an open language where new aspects (specifications) can be added and their semantics can be delivered to the compiler through the moderator. In essence the moderator is a program that extends the language itself. Our goal is to achieve separation of concerns and retain this separation without having to produce an intermingled source code.

Regarding how aspects are defined and merged to provide the overall system, we believe that neither the use of aspect languages nor a weaver tool provides a necessity in order to achieve separation of concerns. We shift the weavers responsibility to a class, which we call the moderator class that would coordinate aspects and components together (figure 1). The moderator class should be extensible in order to make the overall system adaptable to addition of new aspects. We also believe that the use of a moderator class provides the flexibility, adaptability, and extensibility to the programmer to retain the definition of aspects by current programming languages. It also provides the basis for a design framework that would make use of patterns. The importance of design patterns within the AO technology was addressed in [Lorenz 98]. The moderator class defines the semantic interaction between the components and the aspects. Further, the semantics of the model define the order of activation of the aspects. We view a concurrent (shared) object as being decomposed into a set of abstractions that form a cluster of cooperating objects: a functional behavior, synchronization, and scheduling. The behavior of a concurrent object can be reused, or extended. There are other issues that might also be involved, such as security and fault tolerance. We focus on the relationships between these abstractions within the cluster. We propose an aspect-oriented design pattern that we call the aspect moderator pattern. This pattern makes use of a class, which acts as a proxy to the functional component, and would moderate the functional behavior together with different aspects of concern, by handling their interdependencies. We stress the fact that the activation order of the aspects is the most important part in order to verify the semantics of the system. Synchronization has to be verified before scheduling. A possible reverse in the order of activation may violate the semantics. If security is introduced to a shared object, we first need to verify the identity of the caller and therefore we first have to handle security before synchronization.

Architecture of the moderator pattern

A sequential object is comprised by functionality control and shared data. Access to this shared data is controlled by synchronization and scheduling abstractions.

Synchronization controls enable or disable method invocations for selection. The synchronization abstraction is composed of guards and post-actions. During the precondition phase, guards will validate the synchronization conditions. In the post-condition phase, post-actions will update the synchronization variables. The scheduling abstraction allows the specification of scheduling restrictions and terminate actions. At the pre-condition phase, scheduling restrictions use scheduling counters to form the scheduling condition for each method. At the post-condition phase, terminate actions update the scheduling counters. The moderator class is derived from the functionality class. During the pre-condition phase, the synchronization constraints of the invoked method are evaluated. If the current synchronization condition evaluates to TRUE, a RESUME value is returned to the caller, and the scheduling constraints are evaluated; otherwise a BLOCKED value is returned. The evaluation of the scheduling restrictions will also return RESUME or BLOCKED. After executing the precondition phase, the moderator will activate the method in the sequential object. During post-condition, synchronization variables and scheduling counters are updated upon method completion. This section addresses four issues: 1) non-orthogonality of aspects, 2) the provision of an adaptable model, 3) the provision of a design and implementation hierarchy and 4) composition of aspects.

Non-orthogonal aspects

The moderator can handle the issue of non-orthogonal aspects by expressing the semantics of the dependencies between two non-orthogonal aspects. For example, during the pre-condition phase of the security aspect, the moderator can include variables from any non-orthogonal aspect to security.

Extensibility and Adaptability

System software undergoes two types of evolution: functional evolution, when the problem domain changes, and adaptation, when the characteristics of the solution change. The latter is also called non-functional evolution, and it is often related to the technological changes in the applications environment. The object-oriented approach was originally developed to simplify software evolution. Unfortunately, objects are only concerned with functional evolution; they have serious problems coping with the majority of non-functional concerns, which are usually scattered in many classes, in obscure ways. Experience shows that extensibility is not a directly addressed by object-orientation: using objects does not guarantee that the software will be easily modifiable. Objects are not, therefore, the composition units we are seeking for an extensible architecture. Currently, new paradigms have emerged to deal with the intrinsic problems of objects. In particular, we have aspect-oriented programming (AOP) and component models. In aspect oriented programming, an application is built as the integration of aspects which are different solutions to different concerns. Each concern represents Aspects can be replaced, or extended. One of the advantages of this approach is that if a new aspect of concern would have to be added to the system, we do not need to modify the moderator. We can simply create a new class to inherit and re-define it, and reuse it for a new behavior. The inherited class can handle all previous aspects, together with the newly added aspect. Much like one can weave aspects on demand, such as tracing aspects [Böllert, 1998], our framework provides this option by easily adding or ignoring an aspect of a component within a cluster.

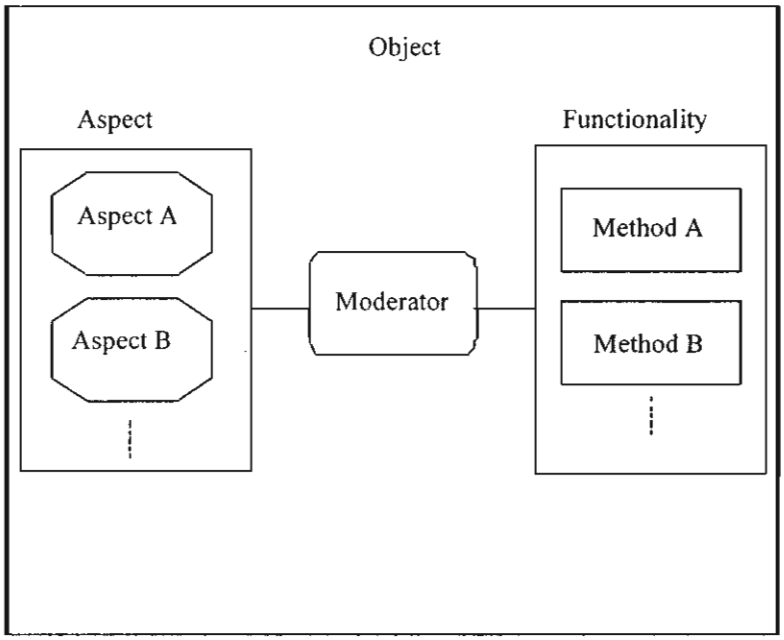


Figure 3.1. The Aspect Moderator.

Adaptability is also applied to components. This design framework addresses the complexity issue in the case where new aspects are introduced and would have to be added. The aspect-moderator pattern does not require some new syntactic structure for the representation of new aspects, but simply a new class for the new aspect. Adaptability includes cases where an existing aspect will have to be modified, or even removed from the overall system. The only composition mechanism is the functional connection, which permits to substitute different implementations of the same functionality, but is not sufficient to support unexpected evolution of the problem domain. Therefore, using components as evolution units is not completely satisfactory. We want to build applications by composition of high-level elements. Those elements are neither objects nor components, and the extension mechanism is not the simply the connection of well-defined interfaces. In itself, this goal is not new, and in the recent years interesting work has been performed to reach this objective, through different means. The following presents, in a general way, how we have reached that extensibility goal.

Design Hierarchy

Aspect Moderator seems natural to choose classes (objects) as components in the OOP paradigm. We take this argument further and propose a hierarchy of components according to the component hierarchy within the OOP paradigm. At the lowest level we have a method. Methods are combined into objects where each object belongs to a class, and several classes can belong to a package. We can apply the moderator pattern to all levels of this hierarchy since aspects can cut across every member of this component hierarchy. One or more aspects can cut across invocations within a single method. We call these aspects, intra-method (or inter-invocation). Aspects can also cut across methods within a single object. We refer to these as intra-object aspects (or inter-method). Aspects can also cut across objects within the same package. We refer to these as intra-package aspects (or inter-object). The programmer has to identify the aspects at each level and address them independently. Since

aspects can cut across components at every level, the moderator is a recurring pattern from intra-method to intra-package. Our design framework will be based on this hierarchy since we believe that it provides a better aspectual analysis and design of a system. Our approach follows a component design and implementation hierarchy.

According to our classification of aspects, the moderator at the lowest level can therefore be referred to as intra-method. At the next level the moderator is referred to as intra-object, and at the highest level it is referred to as intra-package moderator.

Composition of Aspects

At each level of the hierarchy we can maintain an aspect bank, where the moderator of a cluster may initially need to collect all the required aspects from. The aspect bank is a hierarchical 3-Dimensional composition of the system in terms of aspects and components. The moderator will initially consult the aspect bank in order to collect the required aspects.

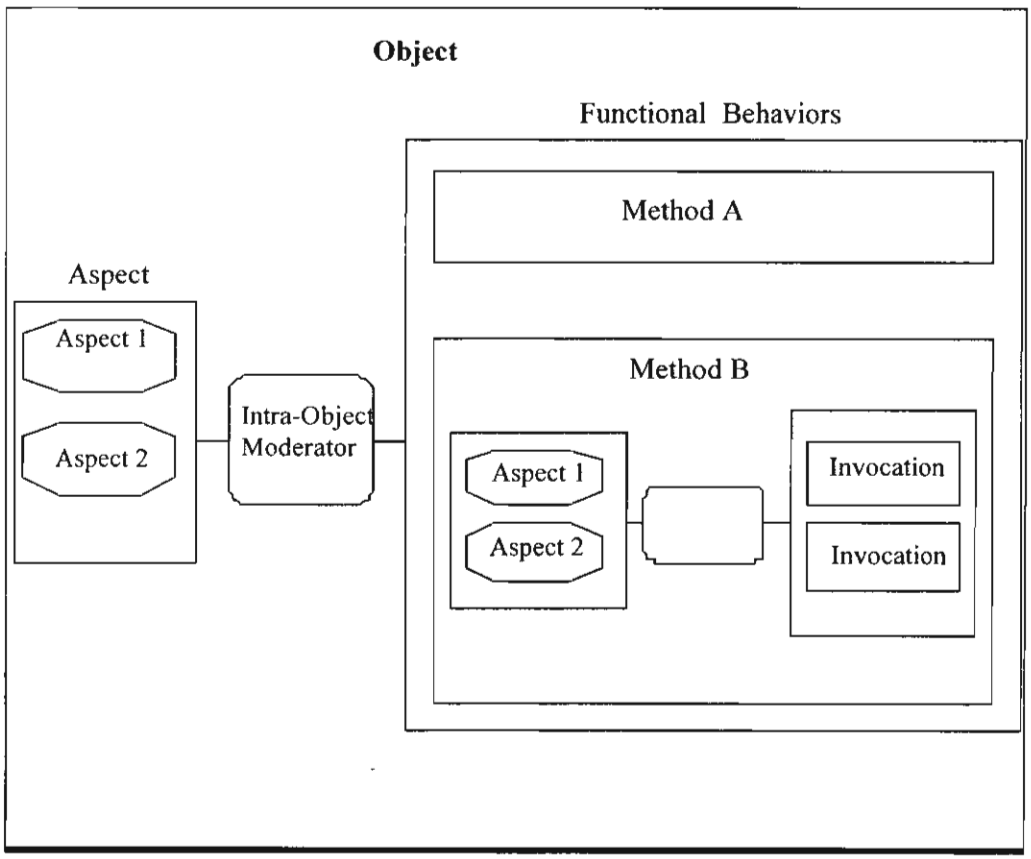


Figure 3.2. Design Hierarchy.

Example: The Conference Room Reservation System

To illustrate the rational behind the design principles of the Moderator, we present the Conference Room reservation system; an extended version of the room reservation system that presented in [Vogel and Duddy, 1998]. In this system we have components that represent rooms and employees. If a meeting organizer is interested in reserving a conference room for a meeting on a certain date and time, then the meeting organizer must check the availability of the conference room on that date and time (Figure 3.1-3.4). A new requirement states that a conference room is reserved based on the security requirements that only employee at the level of technical managers or above may reserve conference rooms. To codify this requirement, we only need to add the security aspect to the aspect bank and extend the moderator to evaluate the security aspects without the need to modify the functionality of the participating components (Figure 3.5). It is the moderator that evaluates the security code during the pre-activation phase. Therefore the moderator must be extended in order to register this new aspect for evaluation.

```
public interface  ModeratorIF {

 synchronized public int preactivation(int MethodID, Object object);
 synchronized public int postactivation(int MethodID, Object object);
 public int RegisterAspect(int MethodID, int AspectKind, AspectObject aspectObject);

}
```

Figure 3.3. The moderator interface.

```
public class AspectBank
AspectFactoryIF {
// Each method has its own aspect objects.
// The moderator evaluates the cross-cutting aspects for the involved components.
public AspectObject create (MethodID  id, AspectKind  aspect, Object Component) {
 if (id == RESERVEROOM) {
 if (aspect == SYNC)
   return new ReserveRoomSync (Component);

 }

return(0);
 }
}
```

Figure 3.4. Implementation of the aspect bank.

```
public class ConferenceRoomReservation {
// Constructor
ConferenceRoomReservation(Moderator  moderator,  AspectBank aspectBank) {
// register all aspects for each method with the moderator
moderator.RegisterAspect (SYNC, ReserveRoom,
aspectBank.create(ReserveRoom,  SYNC, this));
}

public void  ReserveRoom(int RoomId, int Date, int StartTime, int TimeWindow,
object MeetingOrganizer) {

//  PREACTIVATION PHASE : call  preactivation
// Evaluate the aspects for this method
if( moderator.preactivation(ReserveRoom, this)) == ABORT)
return ABORT;

// ACTIVATION PHASE : execute the guarded code
room[RoomId].reserver(Date,startTime,TimeWindow);
MeetingOrganizer.Update(PersonalCalendar, Date, StartTime, TimeWindow);

//  POSTACTIVATION PHASE : call  postactivation
moderator.postactivation (ReserveRoom, this);
}
}
```

Figure 3.5. Implementation of the room reservation system class.

```
synchronized public State preactivation(int MethodID, Object object) {
int AspectIndex, ComponentIndex;

// evaluate each aspect for each of the participants.
for(AspectIndex=0; AspectIndex <NoOfAspects; AspectIndex ++)
for(ComponentIndex =0;  ComponentIndex < NoOfComponents; ComponentIndex++) {
if (EvaluateAspect(AspectIndex, MethodID, ComponentIndex) == ABORT)
return ABORT;
}

// All aspects evaluated to true for all participating components.
return(RESUME);
}
```

Figure 3.6. Implementation of pre-activation.

Relation between moderator and open implementation

The moderator pattern is an architecture that allows for an open language where new aspects (specifications) can be added and their semantics can be delivered to the compiler through the moderator. In essence the moderator is a program that extends the language itself.

```
public class AspectBank2 extends AspectBank {

public AspectObject create(MethodID  id, AspectKind  aspect, Object Component){
      if (id == RESERVEROOM) {
        if (aspect == SECURITY)
        return new ReserveRoomSecurity(Component);
      }
      // the aspect may be defined in the base class
      return (super. create(id, aspect, Component));
  }
}
```

Figure 3.7. Extensibility aspect bank.

Comparison with other work

This section compares our proposal for a design framework using the moderator pattern, and current approaches that rely on the use of a weaver. Both the weaver and the moderator approaches provide the elegance of the original clean code during the analysis and design of the system. The differences between using a weaver and using the moderator pattern are summarized by the following table:

Weaver	Moderator
Combines two kinds of code (aspect and component code) into one intermingled source code. Output of the weaver is the equivalent of traditional approach.	Coordinates two kinds of code, retaining the separation of concerns (aspect and component code). Avoid having to produce an intermingled source code.
One weaver combines all aspects and components together. There is no design hierarchy.	Moderator is a recurring design pattern. It provides an overall system hierarchy, addressing intra-method, intra-object, and intra-package aspects in a systematic way.
Adding new aspect(s) will require either new aspect language(s) or new construct(s) within current aspect languages	Adding new aspect(s) is done by inheritance, and by adding new pre-condition and post-condition.
Must gather contact points of emerging entities.	A design pattern hooks aspects and components: the moderator class defines the semantic interaction between components and aspects
Two phases of compilation (weaving, compiling).	One compilation phase.

Figure 3.8. Comparison of the Framework.

Summary

In AOP, the weaver combines components (functional behavior) and aspects into one unit, which is the overall behavior of the system. In our design framework the overall behavior is made up of 1) the functional behavior, 2) the aspects, and 3) a moderator class that coordinates the interaction between aspects and components while observing the overall semantics. The Moderator approach partitions systems into a collection of cooperating classes. The collaboration among the participants may have few aspects. Addressing these aspects that cut-across the participating objects may produce tightly coupled classes which may reduce reusability. The moderator approach attempts to separate these aspects from the functional components in order to promote code reusability and make it easier to validate the design and correctness of these systems. This framework can address non-orthogonal aspects, and provide for an adaptable model with ease of modification. It further provides a component hierarchy, where the moderator is a recurring pattern. This design principle manages to achieve separation of concerns. There is no difference in the way we separate the concerns. We still have to think about them from the early stages of the software life cycle.

CHAPTER IV

REVISITED FRAMEWORK

System aspectual properties are, for instances, mutual exclusion, scheduling, synchronization, fault tolerance, logging, tracing, security, load balancing, performance measurement, testing, verifications and etc. They are all expressed in such a way that tends to crosscut groups of functional components or services. This tangling design and implementation code of system aspectual properties results increasing of code dependencies between functional components and aspectual properties of the system. It makes their source code difficult to understand, reuse, adapt, and maintain. One current attempt to resolve this issue is the Aspect-Oriented System (AOS). AOS aims at language and architecture independence, where functional components and aspectual properties are separately decomposed in both design and implementation. These properties can be captured in the design and implementation, reused, and adapted in the application software later. Finally, functional components and system aspectual properties are combined together at run-time. We distinguish between functional components and aspects in the design of systems. System aspectual properties are defined as properties of the system that do not necessarily align with functional components or services but tend to crosscut groups of functional components, increasing either *inter-dependency* or *intra-dependency*, and thus affecting the quality of the software. Intra-dependency defines as a system aspectual property that crosscuts between many services (functionalities or methods) in the same components, as illustrated in Figure 1. Inter-dependency defines as a system aspectual property that crosscuts between many components or services, as illustrated in the below figure.

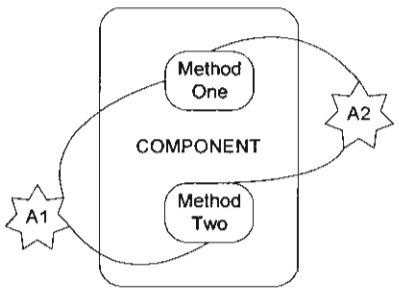


Figure 4.1. Intra-Dependency

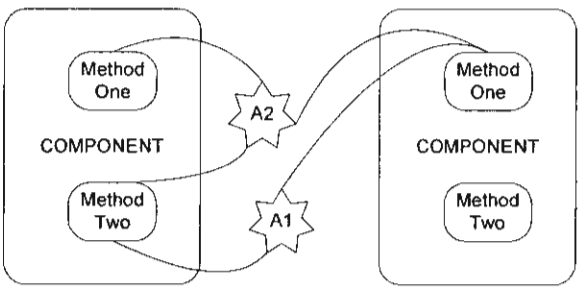


Figure 4.2. Inter-Dependency

Although not bound to OOP, Aspect-Oriented Software Development (AOSD) is a paradigm proposal that retains the advantages of OOP and aims at achieving a better separation of concerns. AOSD suggests that from the early stages of the software life cycle aspects should be addressed relatively separately from the components. As a result, aspectual decomposition manages to achieve a better design and implementation for both operating system and application. At the implementation phase, aspectual properties and functional components are combined together, forming the overall system.

In this research we have shown the system design and implementation based on system aspectual decomposition in the context of the aspectual decomposition for the design and implementation of operating systems. Our approach is an aspect-oriented framework. Compared with what has so far been able to be supported by traditional approaches, our goals are to provide a better modularity for the design and implementation of operating systems,

better flexibility, higher reusability, extensibility and adaptability, as well as to provide a technique that would be practical.

An Aspect-Oriented Framework for Operating Systems

Our observation suggests that an Aspect-Oriented Systems (AOS) that uses Aspect-Oriented Framework could support designers and programmers in cleanly separating components and system aspectual properties from each other. Our framework is based on Aspect-Oriented techniques and layered approach. We argue that system aspectual properties of the operating system should be excluded from the system components or services if there is a possibility to often change it, and it should not be treated as a single monolithic aspect.

One way of structuring system software is to decompose it into layers. Each layer is decomposed into its components. This decomposition of the system design horizontally and vertically helps to deal with the complexity and reusability of system software. The layered architectural design decomposes a system into a set of horizontal layers where each layer provides an additional level of abstraction over it's the next lower layer and provides an interface for using the abstraction it represents to a higher-level layer. Every layer is decomposed into system components and system aspectual properties. System components and system aspectual properties are separated from each other.

Changing either system components or system aspectual properties does not affect the other. The advantage of this decomposition is that system software tends to be easy to understand and maintain. Each layer can be understood and maintained individually without affecting other layers. However, it may be bad for traceability because of using lower layer components.

The framework expresses a fundamental paradigm for structuring system software, a vertical composition of each layer where system components and system aspectual properties are composed into an abstraction of the layer. The framework uses a client-server model in which the server components (Functional Components and System Aspectual Components) are composed by the Aspect Moderator and make their services available to clients. Clients access the server component services by sending requests to the Proxy component. The Proxy component intercepts a requesting message from clients and forwards the message to Aspect Moderator component. The Aspect Moderator component locates and instantiates the composition rules defined by pointcut(s) – where consist of join points between functional components and system aspectual components.

The aspect-oriented framework supports both vertical and horizontal compositions. Functional and aspectual property components in the framework can be composed vertically or horizontally. In vertical composition, the upper layer can use the lower functional or aspectual property components from the lower layer. In horizontal composition, functional and aspectual property components in the particular layer only use to be composed.
The framework is based on system aspectual decomposition of crosscutting concerns in operating system design and implementation.

The framework consists of two frameworks: The Based Layer and The Application Layer Framework. A system aspectual property is implemented in the SystemAspect class, while a component of the system is implemented as a Component class. Alike AspectJ, our framework uses *PointCut, Precondition,* and *Advice.* The framework uses *PointCut, Precondition,* and *Advice.* The AspectModerator class, where the point cut is defined, combines both system aspectual properties and components together at runtime. Pointcuts are defined collections of join points, where system aspectual properties will be altered and executed in the program flow. Every aspectual property can identify and implement preconditions. A precondition is defined a set of conditions or requirements that must hold in

order that an aspect may be executed. Advice is a defined collection of methods for each aspectual property that should be executed at join points. Advice can be either *before* or *after* advice. Before advice can be implemented as *blocking* or *non-blocking*. Before advice is executed when the join point is reached, before the component is executed, if the precondition holds. After advice is executed after the component at the join point is executed. Every aspectual property will define advice methods. Figure 4.3 and 4.4 illustrated the execution model of a pointcut in the framework based on inter-dependency and intra-dependency.

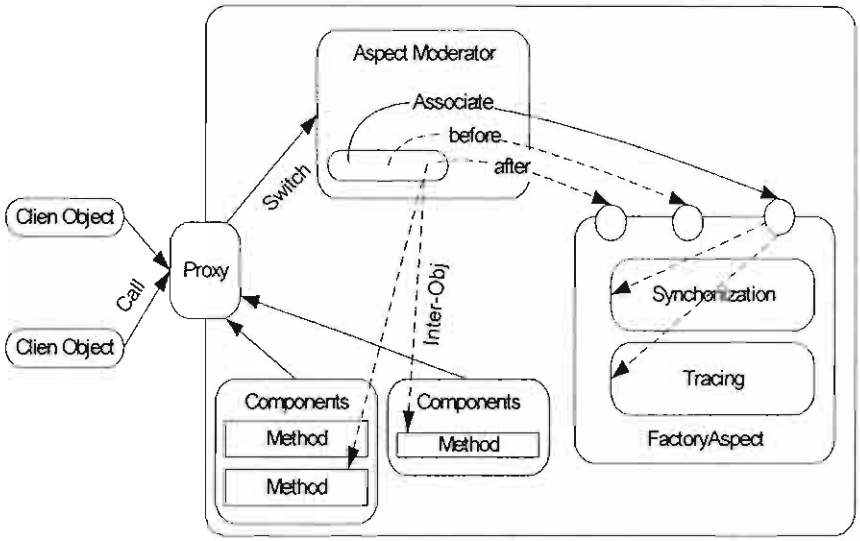


Figure 4.3. PointCut Defines Inter-dependency

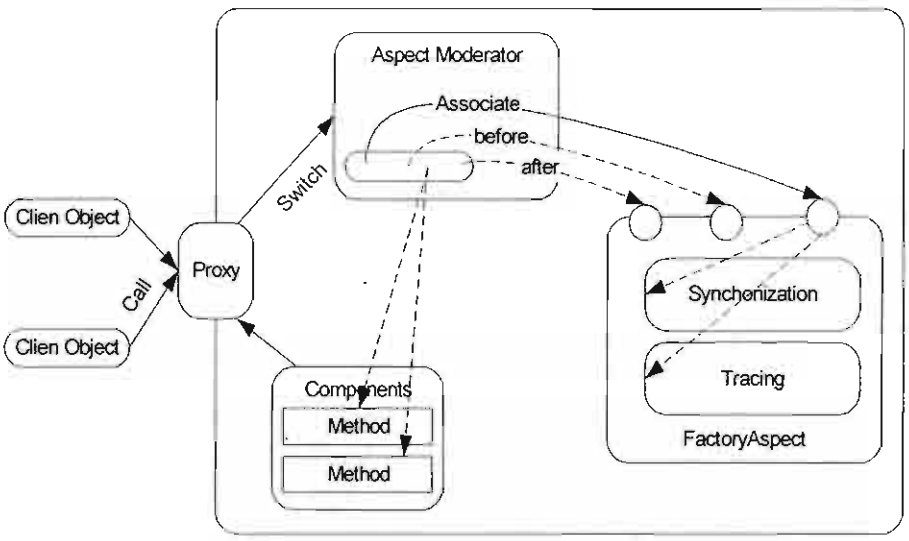


Figure 4.4. PointCut Defines Intra-dependency

Our proposed framework (CAL) is based on system aspectual decomposition of crosscutting concerns in operating system design and implementation. ACL framework consists of two frameworks: Based Layer and Application Layer Framework. In this paper, we show how producers/consumers problem can be implement in the based layer framework. A system aspectual property is implemented in SystemAspect class, while a component of the system is implemented as Component class. AspectModerator object, where the point cut is defined, combines both system aspectual properties and components together at run-time. A Pointcut is defined collections of join points, where system aspectual properties will be altered and executed in the program flow. Every aspectual property could identify and implement precondition. Precondition is defined a set of conditions or requirements that must be hold in order to be executed an aspect. Advice is defined collections of methods for each aspectual property that should be executed at join points. Advice could be either before or after. Before advice could be implemented as blocking or non-blocking. Before advice executes when join point is reached, before the component executed, and if the precondition is hold. After advice executes after the component at the join point executes.

Implementing Aspect-Oriented Framework

The framework consists of four components comprising the architecture of the framework.

- Each functional object (component) provides its services (methods) stripped of any aspectual properties (for example, no synchronization is included in Buffer objects).
- A proxy object intercepts called methods and transfers the calls to the AspectModerator.
- An AspectModerator object consists of the rules and strategies needed to bind aspects at runtime. Aspects are selected from the AspectBank. The AspectModerator orders the execution of aspects. The order of execution can be static or dynamic. Then, each precondition will be checked whether it is satisfied or not.
- An AspectBank object consists of aspect objects that implement different policies of a variety of aspects.

This section presents the design and development of aspect-oriented framework. The model is presented to demonstrate horizontal composition of the framework. The system service must be implemented as a Component class. The system aspectual property (SystemAspect class) must be derived from the SystemAbstractAspect interface to implement the required behavior of a system aspectual property. A SystemAspectFactory consists of many system aspectual properties such as synchronization, tracing, logging, and reliability. The SystemAspectFactory, derived from the SystemAbstract.

AspectFactory interface, is known as an aspect bank. During runtime, each SystemAspectFactory will be associated with one SystemAspect. The AspectModerator class must be derived from the AspectModerator interface to implement the required behavior.

The following points are important about the aspect-oriented framework:

- A base layer framework is an implementation of an underlying system.
- An application layer framework is an implementation of application software over the system software represented by a base layer framework.
- A client object requests a service through a ProxyObject object of a framework.
- A functional component is implemented as a Component class without any aspectual property.
- A SystemAspectFactory object consists of various SystemAspect objects. A SystemAspect object is controlled by a SystemAspectFactory object.

- Each system aspectual property must be implemented as a SystemAspect object.
- Each crosscutting between Component object and an SystemAspect object must be defined in AspectModerator object as joinpoints in a Pointcut method.
- A client requests a service by sending a message to a ProxyObject object. The ProxyObject object changes the request to a specific pointcut method, and forwards it to the AspectModerator object.

The Proxy class is responsible for intercepting and forwarding the message sent from client object to request a service. The Proxy class must implement the behavior of intercepting a service request. A client object of an aspect-oriented framework must request a service by calling the call() method. A call() method consists of at least two parameters: object name provided a service and a service requested to serve. The first parameter is of type string, and the second is type of string as well. The ProxyObject class will forward a request to the AspectModerator object by calling a PointCut() method. A PointCut() method must have the same number parameters and the same parameter type as the call() method The SystemAspectFactor class must be derived from the SystemAspectFactoryAbstract interface to implement the required behavior.

The AspectModerator class is responsible for composing the functional components and the system aspectual property into a service request. The AspectModerator class acts like a coordinator between functional components and system aspectual properties, when and where system aspectual properties will be composed into a functional component. The composition of system aspectual properties and functional components must be guided and defined as PointCut() method. Each PointCut() method must have at least two parameters: component name and service name (methods of the component) that will be composed. The first parameter is of type string, and the second is type of string as well.

The SystemAspectFactor class must be derived from the SystemAspectFactoryAbstract interface to implement the required behavior. The SystemAspectFactory class provides a dynamic binding of variety system aspectual properties. It focuses on the interface of the system aspectual property. Each system aspectual property must be derived from the SystemAspectAbstract interface to implement the required behavior. Implementation of a system aspectual property is implemented in the SystemAspect class. Each system aspectual property can define before(), after(), and precondition() methods depending on its needs. Figure 10 demonstrates the system aspectual property (SystemAspect class) declaration determined from the base class SystemAspectAbstract.

The AspectModerator class operates composition between system aspectual properties and functional components using a composition rule defined by join points of a pointcut. The AspectModerator class performs composition rules by sending AspectFactory messages. Messages sending causes polymorphism. The implementation of AspectFactory uses bridge patterns. A message finds the correct member object of the AspectFactory, and invokes that object. With polymorphism calls, AspectModerator requires less information about each SystemAspect, so the AspectModerator only needs to have the right SystemAspect interface.

The abstract aspectual class defines a SystemAbstractAspect interface that controls the implementation of an aspectual property class. This class is implemented using the concrete classes of aspectual properties, which implement the virtual functions before() and after(). The AspectModerator creates instances of an aspectual property, which requires composing a requested service. If an aspectual property crosscuts more than one method in the same component, it must have a parameter ServiceName identifying what it should be done for each method. If an aspectual property crosscuts more than one component, it must have two parameters: ServiceName and ComponentName identifies what it should be done for each method of each component.

Summary

In this research, we stressed the importance of the better separation of concerns within the context of an Aspect-Oriented Framework. We discussed how this technique provide an alternative to operating system design and implementation, and show how our approach can be achieved separation of crosscutting concerns in the design and implementation of operating systems. Our work concentrates on the decomposition of system aspectual properties crosscutting functional components in the system and our goal is to achieve a better design and implementation of operating systems while supporting separation the crosscutting concerns in every layer. Our design framework provides an adaptable model that allows for open languages and architectures where new aspects and components can be easily manageable and added without invasive changes or modifications. In application, system aspectual properties could be reused and redefined from the system layer preventing the re-engineering of all aspects and components. The framework approach is promising, as it seems to be able to address a large number of system and application aspects and components. The advantage of decomposing of functional components and aspects makes the design and implementation of operating systems better modularity as well as is to promote comprehension, reusability, adaptability, manageability, and extensibility of both components and aspects in the system.

CHAPTER V

CONCLUSION

The object-oriented approach was originally developed to simplify software volution. Unfortunately, objects are only concerned with functional evolution; they have nous problems coping with the majority of non-functional concerns, which are usually ttered in many classes, in obscure ways. Experience shows that extensibility is not a ectly addressed by object-orientation: using objects does not guarantee that the software ll be easily modifiable. Objects are not, therefore, the composition units we are seeking for extensible architecture. Currently, new paradigms have emerged to deal with the intrinsic oblems of objects. In particular, we have aspect-oriented programming (AOP) and mponent models. Each concern represents a problem facet. The basic idea is to define, parately, on the one hand the application logic and on the other hand the different concerns, d to weave them all later on.

In component-oriented programming, the basic structure of the architecture is a graph here nodes are (black box) components described by means of their functional capacities, d the links represent the provide/require relationship. The only missing decision is the ncrete implementation of each one of those boxes. Thanks to this decoupling, local olution inside one component is well supported, but global evolution (a change of the aph structure) or adaptability is hardly handled. The only composition mechanism is the nctional connection, which permits to substitute different implementations of the same nctionality, but is not sufficient to support unexpected evolution of the problem domain. herefore, using components as evolution units is not completely satisfactory.

AOP approach has several advantages but it poorly supports evolution because the eaving is performed directly on the language structures that can evolve. The application and aspects are too closely related. The underlying problem is that in the AOP architecture here are no composition elements, but only a mechanism for code weaving. For this reason, e do not consider that AOP proposes an extensible architecture.

We have identified important issues in the design of adaptable and extensible erating systems, the complexity of system comprehension, development, reusability, tensibility and adaptability. Functional components and system aspectual properties, such mutual exclusion, synchronization, fault tolerance, and tracing aspects, are not well parated using current operating system design. This prevents the designer and developer m understanding, modifying, extending, adapting, and reusing the components of the stem.

To solve these issues, we developed an aspect-oriented framework for the design of tensible and adaptable operating systems. The framework is designed based on the concept Aspect-Oriented Software Development. It allows designers and programmers to separate nctional components and system aspectual properties from each other in every component.

We have shown implementation of classical problems using an aspect-oriented mework. An aspect-oriented design framework simplifies system design by expressing it at higher level of abstraction.

trength of This Research

As with the architecture of a building, the excellence of a software structure or design not easy to measure. Many researchers and developers use the attribute comprehension,

Comprehensibility

Comprehension is a measure of how easy it is for a designer and a programmer to understand the design and implementation of the system. System aspectual properties crosscut basic functional components of the system. With consistency of the design and implementation, system aspectual properties can be captured in both the system design and implementation. We believe that an aspect-oriented framework supports the designers and programmers in cleanly separating functional components and system aspectual components from each other, by providing a mechanism that makes it possible to abstract and compose both functional components and system aspectual components to produce the overall system. Both functional components and system aspectual components can be easily understood.

We believe that the framework provides a better separation of concerns in the design of operating systems. The framework promotes better modularity and quality in the design of the system. The design of operating systems should not be seen as a two-dimensional model with a single monolithic aspectual property. In this research we stress the importance of the complete separation of concerns as proposed by Aspect-Oriented Software Development and we discuss how this methodology can provide an alternative approach to operating system design. Our approach simplifies system design by expressing it at a higher level of abstraction using a three-dimensional model. It further supports the designers and programmers in cleanly separating functional components and system aspectual components from each other in different layers.

Adaptability

Adaptability is a measure of how flexible, modifiable, and easily extensible it is for a designer and a programmer to adapt the existing system. With better separation of concerns, adaptability or refinement of either functional components or system aspectual components of the system can further be achieved easily.

Adding or changing functional components does not affect system aspectual components at all. On the other hand, adding or changing system aspectual components does not affect functional components either. Only Pointcuts, defined in the AspectModerator component, are modified; thus, system aspectual properties that crosscut functional components will not be affected.

Applicability

Applicability refers to the utility of the framework for its intended use. The framework is primarily designed to be an alternative for the design of the adaptable operating systems with better separation of concerns, reuse, and adaptability. Indeed, the design that is good for one software package or application may be poor for others, and conversely. The framework solves complexity of both adaptability of functional components and system aspectual components.

Scalability and Expansibility

Expansibility is a measure of how easy it is for a designer or a programmer to increase or scale the capability of functional components and system aspectual components. The framework supports horizontal and vertical scalability and expansibility. From experimental