

**Unifying Aspect- and  
Object-Oriented  
Program Design**

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy

Computer Science

by

**Hridesh Rajan**

August 2005

© Copyright August 2005

Hridesh Rajan

All rights reserved

To my parents, Dibya & Prabhat Sharma:

Without your efforts and sacrifices I would have never reached this point in life.

## Approvals

This dissertation is submitted in partial fulfillment of the requirements for the degree of  
Doctor of Philosophy  
Computer Science

---

Hridesh Rajan

Approved:

---

Kevin J. Sullivan (Advisor)

---

John C. Knight (Chair)

---

Mary Lou Soffa

---

William G. Griswold

---

Jack W. Davidson

---

David Evans

Accepted by the School of Engineering and Applied Science:

---

James H. Aylor (Dean)

August 2005

## Abstract

---

The dominant family of aspect-oriented programming (AOP) languages, namely the family of languages based on the AspectJ model, provides aspects as a new abstraction mechanism separate from classes. Aspects are promoted as means to modularize certain traditionally non-modularized concerns. The design decision to separate aspects and classes, however, reduces the conceptual integrity of the language model. The resulting language model is non-orthogonal and asymmetric with respect to the key capabilities of aspects and classes. Furthermore, these asymmetries impair our ability to easily separate important classes of concerns namely, integration and higher-order concerns, and distorts our conceptual model of how to structure software systems with AOP.

In this dissertation, I show that aspects as a separate abstraction mechanism are not essential to aspect-oriented programming. Instead a new construct, quantified binding, similar to the event-handler binding in the implicit invocation style, is at the core of aspect-oriented programming. I present a language model in which aspects and classes are unified. Key technical contributions include: support for aspect instantiation under program control, instance-level advising, advising as a *general* alternative to object-oriented method invocation and overriding, and the provision of a separate join-point-method binding construct. The new language model is at once simpler and more expressive. The simplification is manifested by the reduction of specialized constructs in favor of uniform orthogonal constructs. The expressiveness is manifested by the improvement in the modularization of integration and higher-order concerns.

## Acknowledgments

---

A craftsman starts with a mold and with skills, patience, and perseverance shapes a pot out of the mold. The output to a large extent is dependent upon the efforts put into the process. I believe that an advisor similarly starts with a mold and throughout the graduate program shapes a researcher out of that mold. Like shaping a pot, it is a delicate process and the skills, patience, and perseverance of the advisor determines the output to a large extent. That being said, it would not be an understatement to say that my advisor Kevin Sullivan is a great craftsman. His ability to balance a student's research and personal freedom is incredible. His careful efforts towards managing a student's visibility is unmatched. Throughout the graduate program I have never felt that he could have handled a situation any better, he always had the best solution in mind. Thank you Kevin! You are great.

I would also like to thank my committee members, Jack Davidson, David Evans, William Griswold, John Knight, and Mary Lou Soffa for being so patient and for offering insightful comments during my research program. I agree that scheduling has always been a challenge, and all of you have responded patiently to my numerous e-mails about it.

It helps a lot sometimes to get more than one view on an issue. Whenever I got to that point, a door next to my graduate student's office was always open for me. That door led to the office of David Evans and to a comfortable chair where I have spent numerous hours discussing various issues. You were so patient and always offered very helpful advice. Thank you Dave!

Let me make a confession here to John Knight. John it took me a little while to understand your sense of humor and to be comfortable in your presence, but once I was at that level it was so much fun. I want to thank you for agreeing to be my committee chair, for agreeing to give me a letter of reference, and for your constant support and comments during my research program.

Dr. Soffa, before you came to our department, I heard a lot of great things about you from my friend, Naveen Kumar, at the University of Pittsburgh. After you arrived as our chair, I experienced it. It was wonderful to know you and to have your input in my research program. I have learned a lot about methodical presentation of research results from you. Above all, your affectionate behavior always made me feel at home far away from home. Thank you so much.

Bill, during our research-group meeting, you were a constant learning source. It also helped to see a different point of view on certain things from Kevin, for example on research result presentation style. Your criticism and comments during my research program were so helpful.

Jack, besides being on my committee I had the chance to interact with you about several issues pertaining to the graduate program in general. You were always very helpful and offered great advice. I particularly want to thank you for helping me prepare for the interviews and offering insightful comments. It was very effective. Thank you.

My acknowledgment would not be complete without mentioning Tarek Abdelzaher, Worthy Martin, Jörg Liebeherr, and Jack Stankovic. I have always turned to them for advice on various issues and they have always been very patient and helpful. Thank you.

I think my friends from undergraduate program are the most considerate people on the face of the planet. I always had their support even after I forgot birthdays, anniversaries, missed many engagements & weddings, and sometimes lost contact for months. They have always forgiven me and made me feel like I never lost touch with them. My sincere thanks to Rohit Ranjan, Dharmesh Parikh, Munindra Nath Das, Nitin Kumar Singh, Shrish Ranjan, Abhishek Pandit, Anshumani, Arvind Sachdeva, Akhilesh Mohan Saxena, Naveen Kumar, Mukesh Kumar Singh, Tarun Matta, Vinay Paradkar, Shashank Vinchurkar, Nikhil Pal Singh, Amit Srivastava, Reetesh Rathi, Lalit Narayan Sharma, and Ankur Agarwal.

My colleagues and friends here at University of Virginia care no less. I had so much support from Pavel Sorokin, Anshu Bhowmik, Anthony Wood, Chengdu Huang, Bei Zhang, Jinlin Yang, Lin Gu, Jeremy Sheaffer, Prashant Nagaraddi, Vivek Sharma, Shiva Behra, Dhiraj Parashar, Amit Naidu, Abi, Kellie-Ann Smith, Yuanfang Cai, Yuanyuan Song, Qiuhua Cao, Joe Carnahan, Qin

Chen, Thao Doan, Pascal Vicaire, Tibor Horvath, Wei Le, Karthik Sankaranarayanan, Ana Sovarel, Radu Stoleru and Elisabeth Strunk and from the rest of the happy hour crowd.

Last but not the least, I would like to thank my family for bearing with me, while I seem to be in school forever (in their words). My thanks to my parents, Dibya and Prabhat Sharma, and my grand parents, Vimla and Surendra Sharma, for not disowning me even after the number of phone calls dropped down to abysmally low. Thanks to my siblings, Smita and Sudhesh, for accepting that I am too busy to talk to them, and still writing to me and finally thanks to my significant other, Ferzana, for living through it with me. I know, it was very hard folks, but thanks for the constant support and encouragement to keep me on track.

The research described in this dissertation was supported by NSF grants ITR-0086003 and FCA-0429947. This dissertation has my name on the front cover, but it is a combined effort of so many other people. The errors and omissions here are just mine. I have tried to acknowledge everyone here, but if I missed your name here that doesn't mean I am any less grateful to you. Please accept my earnest thanks.

Hridesh Rajan

August 2005

Charlottesville, VA



## Contents

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Thesis and Goal . . . . .	3
1.3	Scope . . . . .	5
1.4	Outline . . . . .	6
<b>2</b>	<b>Advanced Separation of Concerns</b>	<b>7</b>
2.1	Separation of Concerns . . . . .	7
2.2	Crosscutting Concerns . . . . .	9
2.3	Separation of Crosscutting Concerns . . . . .	10
2.4	The AspectJ Model . . . . .	11
2.5	The HyperJ Model . . . . .	17
2.6	Adaptive Methods . . . . .	19
2.7	The Composition Filter Model . . . . .	20
<b>3</b>	<b>The Unified Language Design</b>	<b>22</b>
3.1	Non-Orthogonality and Asymmetries in Language Design . . . . .	22
3.2	Unifying Aspects and Classes . . . . .	28
3.3	Crosscut Specification . . . . .	29
3.4	Additional Power of Overriding . . . . .	33
3.5	Static vs. Non-Static Binding . . . . .	33
3.6	Weaving of Static and Non-Static Bindings . . . . .	37

<i>Contents</i>	x
3.7 Performance Analysis of Static vs. Non-Static Bindings . . . . .	41
3.8 Related Work . . . . .	43
3.9 Summary . . . . .	48
<b>4 Challenge: Separation of Integration Concerns</b>	<b>49</b>
4.1 A Running Example . . . . .	50
4.2 Simple Object-Oriented Integration . . . . .	51
4.3 Mediator-Based Design . . . . .	54
4.4 Mediators as Aspects . . . . .	58
4.5 Work-Arounds and Their Costs . . . . .	59
4.6 The Conceptual Gap . . . . .	65
4.7 Mediator as Classpects . . . . .	66
4.8 Summary . . . . .	71
<b>5 Challenge: Separation of Higher-Order Concerns</b>	<b>72</b>
5.1 Delayed Propagation Concern . . . . .	73
5.2 Simple Realization of the Delayed Propagation Concern . . . . .	74
5.3 Modularizing The Delayed Propagation Concern . . . . .	75
5.4 Delayed Propagation as Classpect . . . . .	78
5.5 Summary . . . . .	80
<b>6 On The Expressive Power of Classpects</b>	<b>82</b>
6.1 On the Expressive Power of Programming Languages . . . . .	83
6.2 Expressive Power of the Unified Model . . . . .	86
6.3 Representation of the Language Models . . . . .	88
6.4 Translation from $L_{Combined}$ to $L_{Eos}$ . . . . .	90
6.5 Translation from $L_{Combined}$ to $L_{AJ}$ . . . . .	93
6.6 Discussion . . . . .	96
6.7 Summary . . . . .	98

<b>7</b>	<b>Case Studies</b>	<b>100</b>
7.1	Design Patterns . . . . .	101
7.2	Complex Mediator-Based Design Idioms . . . . .	108
7.3	Modularizing Crosscutting Concerns in Eos . . . . .	116
7.4	Integrating ConcernCov and NUnit . . . . .	119
7.5	Limitations . . . . .	122
<b>8</b>	<b>Conclusions</b>	<b>127</b>
8.1	Claims and Contributions . . . . .	127
8.2	Limitations of Evaluation . . . . .	129
8.3	Benefits: An Empirical Question . . . . .	130
8.4	Future Work . . . . .	132
8.5	Summary . . . . .	135
<b>A</b>	<b>Eos Language Manual</b>	<b>136</b>
A.1	Join Points . . . . .	138
A.2	Pointcuts . . . . .	142
A.3	Bindings . . . . .	144
<b>B</b>	<b>Eos Grammar</b>	<b>147</b>
B.1	Tokens . . . . .	147
B.2	Production Rules . . . . .	148
<b>C</b>	<b>Eos Implementation</b>	<b>155</b>
C.1	The Eos Compiler . . . . .	155
C.2	Compilation Passes in Eos . . . . .	157
C.3	Limitations and Future Extensions . . . . .	160
<b>D</b>	<b>Additional Source Listings</b>	<b>162</b>
D.1	Concern Coverage Tool- NUnit Integration Code . . . . .	162

*Contents*

xii

**Bibliography**

**166**

## List of Figures

---

2.1	A Fragmented Implementation of a Design Dimension . . . . .	9
2.2	A Modularized Implementation of a Design Dimension . . . . .	11
2.3	Organization of AspectJ Programs . . . . .	12
2.4	The Tracing Aspect in AspectJ . . . . .	12
2.5	Some Example Join Points . . . . .	14
2.6	Anatomy of a Pointcut . . . . .	14
2.7	Some Example Pointcuts . . . . .	14
2.8	Anatomy of an Advice . . . . .	15
2.9	Three Types of After Advice . . . . .	16
2.10	An Example Around Advice . . . . .	16
2.11	An Example Inter-Type Declaration . . . . .	16
2.12	Concern Composition Using Equate in HyperJ . . . . .	17
2.13	Results of Using Equate in HyperJ . . . . .	17
2.14	Concern Composition Using Equate Operation . . . . .	17
2.15	Results of Using Equate Operation . . . . .	18
2.16	Tracing in HyperJ . . . . .	18
3.1	Non-Orthogonal and Asymmetric Aspects and Classes . . . . .	23
3.2	Underlying Representation of an Aspect in AspectJ 1.0 . . . . .	25
3.3	Underlying Representation of an After Advice in AspectJ 1.0 . . . . .	25
3.4	Non-Orthogonal and Asymmetric Advice and Methods . . . . .	26
3.5	A World of Two-Story Buildings . . . . .	27

3.6	Classpect: A Unified Model . . . . .	28
3.7	Re-factoring Advice into Method and Binding . . . . .	29
3.8	Syntax of the Eos Binding Declaration . . . . .	29
3.9	An AspectJ Advice and an Equivalent Eos Binding Declaration . . . . .	30
3.10	A Method Bound Around . . . . .	31
3.11	Masking Sensor's Selectively . . . . .	34
3.12	Static Trace Using Binding . . . . .	36
3.13	Selective Trace Using Binding . . . . .	37
3.14	Underlying Implementation of Static Trace . . . . .	38
3.15	Underlying Implementation of Selective Trace . . . . .	39
4.1	Sensor-Camera-Display System . . . . .	50
4.2	Class Diagram of the Sensor-Camera-Display System . . . . .	52
4.3	Scattered and Tangled Integration Concerns in OO Implementation . . . . .	53
4.4	Class Diagram of Mediator-based Implementation . . . . .	54
4.5	Scattered Event Code in Mediator-Based Implementation . . . . .	56
4.6	Class Diagram of Aspect-Oriented Implementation of Mediators . . . . .	58
4.7	First Work-Around for Sensor-Camera Integration . . . . .	60
4.8	First Work-Around for Sensor-Display Integration . . . . .	61
4.9	Object Diagram for Aspect-Oriented Implementation of Mediators . . . . .	61
4.10	Performance Curve for AspectJ, HyperJ, and Mediator-Based Design . . . . .	62
4.11	Second Work-Around for Sensor-Camera Integration . . . . .	63
4.12	Second Work-Around for Sensor-Display Integration . . . . .	64
4.13	Ideal Object Diagram of the Sensor-Camera-Display System . . . . .	65
4.14	Actual Aspect-Oriented Object Diagram . . . . .	65
4.15	Classpect-Based Implementation of the SensorCamera-Display System . . . . .	67
4.16	Modular Composition of Components and Connectors . . . . .	68
4.17	Object Diagram for Classpect Based Implementation . . . . .	69

4.18	Performance Curve for Type-Level Aspects vs Classpects . . . . .	70
4.19	A Comparative View of Implementations . . . . .	71
5.1	Adding a Layer on Top of Aspects . . . . .	72
5.2	Delayed Propagation Concern . . . . .	74
5.3	Scattered Implementation of Delay Propagation Concern . . . . .	75
5.4	Problems With Modularization of Delayed Propagation Concern . . . . .	76
5.5	Work-Around Applied to Around Advice . . . . .	78
5.6	The Delayed Propagation Concern as Classpects . . . . .	79
6.1	Pointcut and Join Points . . . . .	88
6.2	Aspect and Classpect Members . . . . .	89
6.3	Translation of a Class . . . . .	91
6.4	Translation of an Aspect . . . . .	91
6.5	Translation of an Advice . . . . .	92
6.6	The Retry Policy Binding and Method . . . . .	96
6.7	The Overhead Computation Classpect . . . . .	96
7.1	The Observer Protocol: Eos Code 78% Smaller . . . . .	102
7.2	A Simple Graphical Figure Element System . . . . .	104
7.3	The Color Observer in AspectJ and Eos . . . . .	105
7.4	The Mediator Protocol: Eos Code 66% Smaller. . . . .	107
7.5	Mediator-Based Model View System [95] . . . . .	109
7.6	The Set Implementation . . . . .	109
7.7	The Element Implementation . . . . .	110
7.8	The Set Bijection Mediator's Implementation . . . . .	111
7.9	The Element Ordered Pair Mediator's Implementation . . . . .	112
7.10	The Set Cross Product Mediator's Implementation . . . . .	124
7.11	The Parser Introduction Collector Mediator . . . . .	125
7.12	The Parser Declare Collector Mediator . . . . .	125

7.13 A Screen-shot of <i>ConcernCov</i> in Action . . . . .	126
A.1 Tracing in Eos . . . . .	136
A.2 Inter-Type Declaration in Eos . . . . .	137
B.1 Keywords in C# . . . . .	147
B.2 Additional Keywords in Eos . . . . .	148
B.3 Operators and Punctuations . . . . .	148
C.1 Components of Eos . . . . .	155
C.2 Directory Structure of Eos . . . . .	156
C.3 The Architecture of the Eos System . . . . .	156
C.4 AST Event Announcer . . . . .	158



## List of Tables

---

1.1	A Mapping of Claims to Evaluation Mechanism . . . . .	4
2.1	Join Points in AspectJ . . . . .	13
2.2	Pointcuts in AspectJ . . . . .	21
3.1	A Characterization of a Subset of Aspect-Oriented Languages and Approaches . .	46
6.1	Difference in Constructs . . . . .	87
A.1	The Eos.Runtime.IJoinPoint Interface . . . . .	138
A.2	The Eos.Runtime.ISourceLocation Interface . . . . .	138
A.3	The Eos.Runtime.IStaticPart Interface . . . . .	139
A.4	The Eos.Runtime.Signature.ISignature Interface . . . . .	139
A.5	The Eos.Runtime.Signature.ICatchClauseSignature Interface . . . . .	139
A.6	The Eos.Runtime.Signature.IMethodSignature Interface . . . . .	140
A.7	The Eos.Runtime.Signature.IFieldSignature Interface . . . . .	141
A.8	The Eos.Runtime.Signature.IConstructorSignature Interface . . . . .	142
A.9	Return Types of Join Points . . . . .	145
C.1	Line of Code Distribution for Eos . . . . .	157

# Chapter 1

## Introduction

---

### 1.1 Motivation

In the original work on the concept of separation of concerns [22], Dijkstra wrote:

Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects.

We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day ... But nothing is gained—on the contrary—by tackling these various aspects simultaneously. It is what I sometimes have called *the separation of concerns*.

Today, finding an appropriate *separation of concerns* is understood to be perhaps the fundamental challenge in software design. It promotes studying different interests or concerns of a complex problem separately, with none or very little knowledge of the other concerns. Separation of concerns is the key to maintaining overall intellectual control in the face of complex problem and design solutions.

Over the years, the quest for better separation of concerns has led to different technologies, among which are the well-established modularization techniques such as structured programming

[20, 21, 58, 108], abstract data types [68], and object-orientation [19, 42, 43, 61, 74, 90, 104]. A new synthesis that goes under the rubric of aspect-oriented programming (AOP) [30, 53] is now emerging for advanced separation of concerns. AOP is generating much interest in the software engineering and languages communities, and in many other areas. Among others, IBM has reaped significant benefits by applying AOP to its WebSphere project [87] and to its other middleware product-line [15].

Aspect-oriented programming (AOP) has shown the potential to improve the ability of software architects to devise more effective modularizations for some traditionally non-modular concerns. The dominant family of languages in this domain is based on the AspectJ model [52]. Other languages in this family are AspectC++ [89], AspectR [10], AspectWerkz [6], AspectS [48], Caesar [75], etc. In this dissertation, I study this model. The designs of AspectJ-like languages are unique in having shown that AOP can succeed in practice. Their success is due in part to the careful balancing of competing pressures in the design of the language. A major goal was industrial adoption. The pursuit of this goal led the designers of AspectJ to make important early design decisions that in some cases traded against orthogonality, uniformity and generality in the language design in order to make the new constructs more acceptable to potential early adopters. In particular, the designers strongly differentiated aspects from classes, and advice bodies within aspects from methods. Now that the model has shown promise, it makes sense to ask whether we can improve upon it by the application of more traditional principles of programming language design [70] [109].

One specific design decision that is the subject of this dissertation is to introduce aspects as a new concept and as a separate abstraction mechanism. The most successful aspect-oriented (AO) languages today—the AspectJ-like languages [52]—supports two related but distinct constructs for modular design: classes and aspects. Aspects are promoted as means to modularize traditionally non-modular concerns. Generally, AspectJ programs are designed in a two-layered style: a *base* layer, typically of object-oriented code, is advised and extended by an aspect layer. In this dissertation, I show that some of the properties of aspects in the AspectJ style make it hard to go beyond such a two-layered style. Although aspects can advise classes, classes cannot advise aspects, and aspects cannot advise other aspects in full generality. On the one hand, introduction of the aspect

as an analytic and syntactic category has caused people to struggle with the question, “what is an aspect?” just as they struggled with the question, “what is an object?” when object-oriented languages and methods were introduced. On the other hand, this design decision has consequences that are felt in the form of unnecessary constraints on designers, surprising non-compositionality properties, undue design complexity in resulting programs, and performance problems for which it does not appear optimizations are readily available. All of these problems are known. The open question was how to address them effectively with language mechanisms.

The problem that I address in this dissertation is that the design decision to provide classes and aspects (and methods and advice bodies) as separate constructs:

- Reduces the conceptual integrity of the programming language design.
- Promotes an artificial and harmful dichotomy between aspects and objects in the programming model.
- and at the minimum:
  - Impairs the programmer’s ability to separate integration and higher-order concerns.
  - Leads to programs in which implementation of integration and higher-order concerns are unnecessarily complex and incur unnecessary performance costs for which optimizations doesn’t appear to be readily available today.

Here by conceptual integrity I refer to the notion promoted by Fred Brooks [8]. Brooks argued that *conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas.*

## 1.2 Thesis and Goal

My thesis is that unification of aspects and classes is possible, and leads to a language design that is both simpler and more expressive. By simpler, I mean a language design that complies

Table 1.1: A Mapping of Claims to Evaluation Mechanism

Claims	Evaluation Mechanism
A unified language design improves the conceptual integrity of the programming model of AspectJ-like languages.	Design the proof of concept language design and verify whether it has fewer, orthogonal, and regular constructs compared to the AspectJ language model.
The programming model further improves the modularization of integration and higher-order concerns, eliminating the design complexity and scalability problems observed in the modularizations of these concerns using AspectJ-like languages.	Use separation of integration and higher-order concern scenarios as challenge problems for the unified and the AspectJ language model and compare the performance of the two models and verify whether the unified language model improves modularity.
The unified language design further improves the compositionality of AspectJ-like languages, eliminating the dichotomy from the programming model.	Compare the compositionality of language constructs in the AspectJ and the unified model and verify whether the new language constructs are more compositional.
The unified language model is more expressive than AspectJ-like languages.	Compare the unified language model with the AspectJ language model using the Felleisen's notion of expressive power of programming languages [31] and verify whether the unified language model is more expressive.

with MacLennan's *simplicity principle* [70], which suggests that *a language should be as simple as possible and there should be a minimum number of concepts with simple rules for their combination*.

I further claim that the unified language design has the following properties.

1. Such a language design improves the conceptual integrity of the programming model.
2. The programming model further improves the modularization of integration and higher-order concerns, eliminating the design complexity and scalability problems observed in AspectJ-based modularizations of these concerns [82, 92].
3. The unified language design further improves the compositionality of AspectJ-like languages eliminating the dichotomy from the programming model.

4. The resulting programming model is more expressive [31] than that of AspectJ-like languages.

My goal is to show that a significant simplification and unification is possible, not only without compromising the capabilities of current AspectJ-like languages, but while also enhancing their expressiveness and making them simpler and more uniform and orthogonal. To verify that my research satisfies the goal that I have set forth, I have developed a unified language design and built a robust compiler that provides a proof of concept of the proposed language design and verifies that a real unification in both design and practical implementation is achievable under the given constraints. To validate the claims about the unified language design, I have used the evaluation mechanisms documented in Table 1.1.

### 1.3 Scope

The thesis of my dissertation is based on the supposition that aspect-oriented programming is a promising approach. Although the jury is still out, aspect-oriented views of modularity, the new and in some cases radical capabilities of aspect languages, and trends toward industrial adoption combine to warrant research on the design, implications, and uses of such languages and methods.

On one hand aspect-oriented techniques have shown promise to enable modularization of previously non-modular concerns, on the other, early benefit projections were unrealistic. Modular reasoning in the presence of aspects is an open problem. An aspect's ability to observe and modify states in other modules in some sense violates the traditional notions of encapsulation. The conceptual gap between the static structure and the dynamic structure of aspect-oriented programs attract criticisms like *AOP considered Harmful* [17]. As of this writing, there is no single generally accepted formal model of aspect-oriented programs. There is, however, a significant and growing body of research addressing these problems including work on modular reasoning [55, 13, 16, 69, 2, 94], modular verification [60, 26, 50] and, formal semantics of aspect-oriented programs [3, 14, 9, 49, 107, 106, 12].

Stipulating that aspect-oriented programming is worth exploring, my goal is to understand if there are significantly better ways to realize the promise of aspect-oriented programming. I believe that, to the extent that aspect-oriented methods turn out to be beneficial, a unified model is likely to be even more so.

## 1.4 Outline

The next chapter provides background on the new techniques for advanced separation of concerns including the most prominent AspectJ Language model [53], the HyperJ model [77], the adaptive programming model [64] and event-based aspect-oriented programming approaches. Chapter 3 describes the unification of the aspect- and object-oriented programming model.

Chapters 4 to 6 validate the claims made in this chapter. In particular, Chapter 4 validates the claim that the unified language design improves the separation of integration concerns. It considers separation of integration concern as a challenge problem for the AspectJ and the unified model. Chapter 5 validates the claim whether the unified language design improves the separation of *higher-order concerns*. It discusses difficulties with the separation of *higher-order concerns*. Chapter 6 compares the relative expressiveness of the AspectJ model and the Unified Model using Felleisen's notion of expressive power of programming languages [31].

Chapter 7 describes the result of some case studies where unified model was applied to real world systems and canonical examples. I used the unified model and its implementation to modularize crosscutting concerns in small to medium scale systems. Chapter 8 discusses future work and concludes. Appendix A provides an overview of Eos, an extension of C#, Eos implements the proposed unified model. Appendix B describes the grammar for Eos.

# Chapter 2

## Advanced Separation of Concerns

---

In this chapter, I briefly describe separation of concerns principle and give an overview of various advanced separation of concerns techniques. I review the four language models for advanced separation of concerns namely, the asymmetric model of AspectJ [52], symmetric model of HyperJ [77], adaptive methods [64], and the composition filter model [1]. Harrison et al. argue that an aspect language model is asymmetric if *aspects are composed (woven) into components that implement a base model; aspects and components are different, and component-component, aspect-aspect, and class-class composition are not supported. A symmetric paradigm, on the other hand, makes no distinction between components and aspects, and does not mandate a distinguished base model* [46].

### 2.1 Separation of Concerns

Increasing software complexity demands means to cope with it so that we can build intellectually manageable systems [21, 23]. Dijkstra in his work (On the role of scientific thought) [22] proposed *separation of concerns* as one such method. He argued that to solve a complex problem one should study different interests or concerns of the problem *in isolation for the sake of its own consistency*. Adhering to, and invention of, new separation of concerns techniques is becoming more important with the growing complexity of software systems.

Dijkstra also argued that the *discovery of which aspects can be meaningfully “studied in isolation for the sake of their own consistency”* is also important. He used the phrase *discovery of various*



*aspects* to essentially mean decomposition of the system into different concerns. A decomposition of the system such that different concerns in it can be studied independently without requiring detailed knowledge of the other parts is called modular decomposition or modularization [78]. A concern in this context is defined as a dimension in which a design decision is made [63]. For example, the design decision to trace the execution of all methods in the program is a concern. The design decision to use a common thread pool to optimize the use of kernel level threads is another concern. Some other examples of concerns are security policy enforcement, caching, transaction management, etc.

In practice, a good separation of concern is achieved by finding the right decomposition or modularization of a problem. The ideal modularization partitions a system into modules such that each module hides a design dimension that is likely to change from other modules [78]. Parnas et al. [79] argue that:

The primary goal of the decomposition into modules is reduction of overall software cost by allowing modules to be designed and revised independently.

If different concerns are implemented in separate modules, and the interfaces between these modules are thin, each module can be implemented and reasoned about separately. This independence brings evolvability, variability in key dimensions and scalability in development processes. The development process becomes scalable because each concern can be developed and evolved independently. Being relatively independent from each other allows different developers to work in parallel without incurring significant communication overhead.

New modularization techniques emerge to cope with increase in software complexity. The emergence of rich GUI interfaces [51] drove the invention, refinement, and adoption of object-oriented (OO) languages and design methods [41]. Now aspect-orientation and multi-dimensional separation of concern principles are emerging to modularize so called *crosscutting concerns*.



Figure 2.1: A Fragmented Implementation of a Design Dimension

## 2.2 Crosscutting Concerns

The prevalent paradigm for achieving modular software design advocates decomposing software systems into modules with procedural interfaces. Object-oriented programming [19], component-based software development [98, 99] and web-services are all examples of such paradigm. As it turns out, these techniques for separation of concerns are inadequate for modularizing important dimensions of design. These dimensions do not fit into any one partition; instead, they end up being fragmented across several partitions. Execution tracing policy, cross-module optimization, use of common thread pool, security policy enforcement, component integration, etc., are some example design dimensions that exhibit such behavior when decomposed using the prevalent approaches.

The implementation of these concerns is often hard to factor out in separate modules using classical decomposition techniques. For example, the implementation of the fault-tolerant behavior is not well separated in one module, but cuts across the entire implementation of the system. Similarly, the implementation of method execution tracing is spread across all the methods in the system. Tarr, Ossher, Harrison, and Sutton [101] attribute this problem to the *tyranny of dominant*

*decomposition:*

Our ability to achieve the goals of software engineering depends fundamentally on our ability to keep separate all concerns of importance in software systems. All modern software formalisms support separation of concerns to some extent, through mechanisms for decomposition and composition. However, existing formalisms at all lifecycle phases provide only small, restricted sets of decomposition and composition mechanisms, and these typically support only a single, "dominant" dimension of separation at a time. We call this "tyranny of the dominant decomposition."

Figure 2.1 shows this phenomenon at the implementation level. It shows a SeeSoft-like view [29] of a system in which the code implementing a particular design dimension (grey lines) is fragmented across the system. The code for this design dimension (grey lines) is also mixed with the code for other design dimensions (black lines). This phenomenon is called *scattering and tangling* in the aspect-oriented programming terminology and these concerns are called *crosscutting concerns*. A scattered implementation of a design dimension is not local but spread across the implementation of the system. A tangled implementation of a design dimension is one that is mixed with the implementation of other design dimensions. Several new advanced separation of concerns techniques, such as aspect-orientation and multi-dimensional separation of concern are emerging from the need to modularize these scattered and tangled concerns.

## 2.3 Separation of Crosscutting Concerns

Aspect-oriented programming [54] and multi-dimensional separation of concern [101] provides opportunities to modularize crosscutting concerns. The idea is to represent these concerns as separate modules and then allow them to affect the behavior of the rest of the system.

Figure 2.2 shows the modularized representation of the concern that was previously crosscutting (Figure 2.1). The grey lines representing the crosscutting concern code are now isolated into a separate module. This module is composed with the rest of the system such that at run-time it affects the behavior of other concerns to achieve the desired system-level behavior.



Figure 2.2: A Modularized Implementation of a Design Dimension

Different techniques for advanced separation of concerns differ in the mechanisms provided to specify the composition of this new module with the rest of the system. In the rest of this chapter, I discuss four different AOP approaches that provide distinct composition strategies.

## 2.4 The AspectJ Model

In this section, I will review basic concepts of the dominant aspect-oriented model, namely the AspectJ model. AspectJ [52] is an extension to Java [43]. The rest of this thesis starts with this model. Masuhara and Kiczales also call this model *the Pointcut-Advice model* and show that other aspect language models can be mapped to this model [72]. Other languages in this class include AspectC++ [89], AspectR [10], AspectWerkz [6], AspectS [48], and Caesar [75]. The central goal of AspectJ-like languages is to enable the modular representation of crosscutting concerns.

AspectJ-like languages generally organize programs into a two-layered hierarchy as shown in Figure 2.3. Adding a third layer of aspects on top of the second layer is possible but awkward to construct. The concerns that can be modularized using the traditional OO modularization tech-

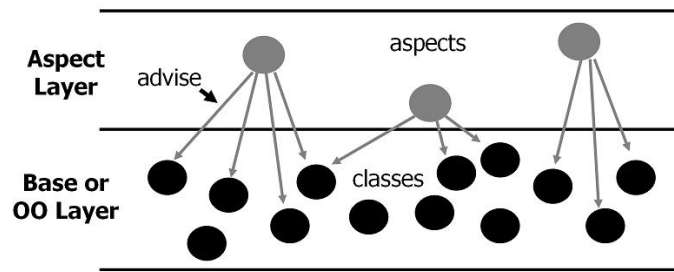


Figure 2.3: Organization of AspectJ Programs

niques, classes, are in the first layer. The first layer is also often called the base layer [63]. The modularized representations of the crosscutting concerns, aspects, are in the second layer. These aspects affect the behavior of the classes in the base layer.

These languages add five key constructs to the object-oriented model: join points, pointcuts, advice, inter-type declarations, and aspects. Figure 2.4 shows a simple example to make the points concrete.

```

1 aspect Tracing {
2   pointcut tracedMethod():
3     execution(* *(..));
4   before(): tracedMethod() {
5     System.out.println(thisJoinPoint);
6   }
7 }

```

Figure 2.4: The Tracing Aspect in AspectJ

An aspect (lines 1-7), modifies the behavior of a program at certain selected execution events exposed to such modification by the semantics of the programming language. These events are called join points. The execution of a method in the program in which the Tracing aspect appears is an example of a join point. A pointcut (lines 2-3) is a predicate that selects a subset of join points for such modification – here, any method execution. An advice (see lines 4-6) is a special *method-like* construct that effect such a modification at each join point selected by a pointcut. Here the *before* advice is executed after every method in the program. The advice prints out the string representation of the *thisJoinPoint*. This special variable contains the reflective information at the join point, e.g. name of the join point, argument at the join point, the value of *this* variable at the

join point, etc. The *thisJoinPoint* variable is implicitly available to the advice.

### 2.4.1 Join Points

Table 2.1: Join Points in AspectJ

Join Points	Semantics
Method call	When a method is called, not including super calls of non-static methods.
Method execution	When the body of code for an actual method executes.
Constructor call	When an object is built and that object's initial constructor is called.
Constructor execution	When the body of code for an actual constructor executes, after its this or super constructor call.
Static initializer execution	When the static initializer for a class executes.
Object pre-initialization	Before the object initialization code for a particular class runs. This encompasses the time between the start of its first called constructor and the start of its parent's constructor. Thus, the execution of these join points encompass the join points of the evaluation of the arguments of <i>this()</i> and <i>super()</i> constructor calls.
Object initialization	When the object initialization code for a particular class runs. This encompasses the time between the return of its parent's constructor and the return of its first called constructor. It includes all the dynamic initializers and constructors used to create the object.
Field reference	When a non-constant field is referenced.
Field set	When a field is assigned to.
Handler execution	When an exception handler executes.
Advice execution	When the body of code for a piece of advice executes.

A join point is a point in the execution of the program defined by the language semantics. In the listing shown in Figure 2.5, there is a *method call join point* corresponding to the *Detect* method being called on the object *s1* of *Sensor* type (Line 4), a *method execution join point* at which the

```

1 Sensor s1 = new Sensor();
2 Camera c1 = new Camera();
3 s1.Detect();
4 ...
5 ...
6 class Sensor {
7     bool isDetected;
8     void Detect(){
9         isDetected = true;
10    }
11 }

```

Figure 2.5: Some Example Join Points

*Detect* method defined in the class *Sensor* begins executing (Line 8), a *field set join point* where the *isDetected* field of *Sensor* class is referenced, etc.

The join point is the point where the advices in the aspect code are composed with the base code. The types of join points exposed by the language semantics constitute the join point model of the language. The join point model of an aspect language, largely, determines the kind of crosscutting concerns it can help modularize. AspectJ has a rich join point model. A complete description of join points supported by AspectJ as of this writing is shown in Table 2.1 on Page 13 <sup>1</sup>.

## 2.4.2 Pointcuts

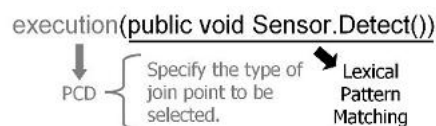


Figure 2.6: Anatomy of a Pointcut

```

1 /* The execution of the Sensor.Detect Method */
2 pointcut DetectExecution(): execution(public void Sensor.Detect());
3 /* All method call join points */
4 pointcut AllCalls(): call(* *(..));
5 /* All non recursive call join points */
6 pointcut NonRecursiveCalls(): AllCalls() & !cflowbelow(AllCalls());
7 /* All outside calls to the methods and constructors in package Foo */
8 pointcut FooUsage():
9     (call(* Foo.*(..)) || call(Foo.*.new(..)))    && !within(Foo.*);

```

Figure 2.7: Some Example Pointcuts

<sup>1</sup>From the AspectJ language manual at <http://www.eclipse.org/aspectj>.

A pointcut is an expression that evaluates to a subset of join points. A pointcut can be either named or anonymous. They can also be composed together to yield complex pointcuts using and, or, and not operators. AspectJ was not the first to support pattern matching on program execution events. In the Field environment [85] for example, one could specify a pattern to select a subset of events that are of interest. This pattern was matched with the ASCII strings that were part of the event notification.

As shown in Figure 2.6, a simple pointcut has two parts. The first part called Pointcut Designator (PCD) defines the type of join points selected by the pointcut expression. A complete description of PCDs supported by AspectJ as of this writing is shown in Table 2.2 on Page 21 <sup>2</sup>. The second part specifies a lexical pattern to select a subset of join points. Figure 2.7 shows some interesting example pointcut expressions.

### 2.4.3 Advice

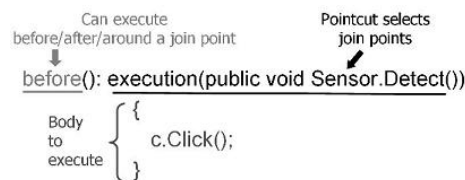


Figure 2.8: Anatomy of an Advice

An advice is a special method-like construct. Crosscutting concerns are specified using advice. As shown in Figure 2.8, an advice specification has three parts: advice kind, pointcut, and body of the advice. AspectJ defines three kinds of advice. The before advice runs before its join points, after advice runs after its join points, and around advice runs in place of its join point. The advice runs at the join points selected by pointcuts. In Figure 2.8 the before advice runs at the join point *execution of method Sensor.Detect*.

Figure 2.9 shows three special cases of after advice: normal completion of the execution of the join point, after the join point execution throws an exception, or after it does either one. Figure 2.10

<sup>2</sup>From the AspectJ language manual at <http://www.eclipse.org/aspectj>.



```

1  /* After normal completion of the execution of the join point */
2      after() returning(): execution(public void Sensor.Detect()) {
3      ...
4      }
5
6  /* After the join point execution throws an exception */
7      after() throwing (Exception e): execution(public void Sensor.Detect()) {
8      ...
9      }
10
11 /* After both normal and exceptional return */
12     after():execution(public void Sensor.Detect()) {
13         ...
14     }

```

Figure 2.9: Three Types of After Advice

```

1 void around(): execution(public void Sensor.Detect()) {
2     proceed(); /* Execute the original join point */
3 }

```

Figure 2.10: An Example Around Advice

shows an example around advice. An around advice executes instead of the join point. It can return a value. It can also execute the original join point by calling a special method *proceed*.

#### 2.4.4 Inter-Type Declarations

```

1 public class Bit{
2     bool value;
3     public Bit(){value = false;}
4     public void Set(){value = true;}
5     public bool Get(){return value;}
6     public void Clear(){value = false;}
7 }
8 public aspect MakeCloneable{
9     declare parents: Bit implements IsCloneable;
10    public object Bit.clone(){
11        ...
12    }
13 }

```

Figure 2.11: An Example Inter-Type Declaration

Figure 2.11 provides a simple example of an inter-type declaration. It allows a third party type to add additional members and interfaces to a type without involvement of the type itself. In the

example shown in Figure 2.11 the aspect `MakeCloneable` (Lines 8-13) makes two compile-time modifications to the class `Bit` (Lines 1-7). First, using the `declare parent` construct (Line 9) it adds the interface `IsCloneable` to the class `Bit`. Second, using the `inter-type` declaration (Lines 10-14), it adds the method `clone` (Lines 11-13) to the class `Bit`.

## 2.5 The HyperJ Model

```

1 class A {
2     void a() {
3
4     }
5 }
6 class B {
7     void b() {
8
9     }
10 }
11 // Compose A and B to yield C
12 equate class A,B into C;
```

Figure 2.12: Concern Composition Using `Equate` in HyperJ

```

1 // Result of the composition
2 class C {
3     void a() {
4         ...
5     }
6     void b() {
7         ...
8     }
9 }
```

Figure 2.13: Results of Using `Equate` in HyperJ

```

1 // Compose methods a and b to yield c
2 equate class A,B into C;
3 equate operation a,b into c;
```

Figure 2.14: Concern Composition Using `Equate` Operation

HyperJ [77, 100] supports multi-dimensional separation of concerns (MDSOC). MDSOC is an approach to decompose software into modules, encapsulating a specific area of interest or concerns in individual module. The fundamental capability of HyperJ is class composition, yielding

```

1  // Result of the composition
2  class C {
3      void c() {
4          a();
5          b();
6      }
7      void a() {
8          ...
9      }
10     void b() {
11         ...
12     }
13 }

```

Figure 2.15: Results of Using Equate Operation

```

1  public class Trace{
2  public void _Invoke( String cName, String mName){
3      System.out.println("After: Class-" + cName + " Method-" + mName );
4      };
5  }
6  // Concerns File
7  class Trace: Feature.Trace
8  // HyperModule File
9  hypermodule Trace
10     hyperslices:
11         /* Other Features */,
12         Feature.Trace;
13     relationships:
14         mergeByName;
15 bracket "*"."{~,~<}*"
16     after Feature.Trace.Trace._Invoke( $ClassName, $OperationName);
17 end hypermodule;

```

Figure 2.16: Tracing in HyperJ

combined classes that can be instantiated. Figure 2.12 and 2.14 demonstrates two example compositions. In the example, we have two classes, A (lines 1–5) and B (lines 6–10). These two classes are composed using the equate construct (line 12) to produce class C (Figure 2.13 lines 1–9). The resulting class contains methods from both class A and class B. All instantiations of A and B will now be replaced by an instantiation of an object of type C. The methods in these classes can also be merged. For example, in an alternate composition rule (lines 1–3 in Figure 2.14), the methods a and b in classes A and B are merged into method c (lines 3–6 in Figure 2.15) in class C (lines 1–13 in Figure 2.15).

Implementation of tracing in HyperJ is very similar to the implementation in AspectJ. Here the tracing logic is written in the `Trace` class (lines 1–5), that provides a method `_invoke` (lines 2–4) to be invoked after the join points of interest. The join points are picked using the bracket constructs (lines 15–16). The bracket construct in this case selects (line 15) all methods that do not begin with an underscore and a less than operator to avoid recursive calls inside the trace class. The bracket construct then specifies (line 16) that method `_invoke` is executed after these methods. The join points that can be selected are limited to method calls and returns.

## 2.6 Adaptive Methods

Adaptive programming and the Demeter project [64, 71] aim to separate class hierarchies and structures from concerns implemented in operations. Lieberherr et al. [65] observe that an operation in an object-oriented program often involve several different collaborating classes. For example, performing static analysis on an abstract syntax tree (AST) involves traversing the entire AST. Such an operation is implemented either by several small methods spread across the class hierarchy or by a monolithic method that knows about the entire class hierarchy. In the case of static analysis, either each node type in AST has its own method for performing the analysis or there is a single method that knows about the complete AST structure.

The problem with the former approach is that the operation is scattered across the entire class structure, making it difficult to adapt when the operation changes. The problem with the latter approach is that too much information about the structure of the classes (is-a and has-a relationships) is tangled into the monolithic method, making it difficult to adapt to changes in the class structure.

The adaptive programming project solves this problem by introducing the notion of an adaptive method also known as a propagation pattern [67]. An adaptive method avoids the scattering problem by encapsulating the behavior of an operation into one place. It avoids the tangling problem by abstracting over the class structure. The abstraction over the class structure is achieved by giving a traversal strategy [66] as well as a specification of the intended behavior when each participating node is reached. By specifying very little information, the traversal strategy abstracts out the

concrete class structure.

## **2.7 The Composition Filter Model**

The composition filters model is an extension of the object-oriented model [1]. There are two parts of a composition filter model: an implementation part and an interface part. The implementation part is the kernel of a composition filters object and adheres to the conventional object-oriented model. The interface part is a layer that encapsulates the implementation part. This part contains the extensions made by the composition filters model, namely input filters and output filters.

The interface part receives incoming messages, processes them, and optionally hands them over to the implementation part. The interface part additionally processes messages that are sent by the implementation part, and sends them to their respective targets. Input filters process received messages, while output filters process sent messages. Functionalities in filters are orthogonal. Filters are specified in a declarative fashion and can be composed in arbitrary combinations. The combined functionalities of these filters are thus added to the implementation part. Acceptance of a message by a filter is dependent upon message itself and some conditions usually specified in the implementation part.

The next few chapters describe the non-orthogonality and asymmetries in the current aspect-oriented language models. These properties are demonstrated by applying the language models to separate important crosscutting concerns. For most of these chapters, the focus is on the dominant asymmetric model of AspectJ-like languages and asymmetries are demonstrated in this context. Whenever required, I also discuss other language models.

Table 2.2: Pointcuts in AspectJ

Pointcuts	Description
Call	call(MethodPattern or ConstructorPattern): selects method call join points matching MethodPattern or constructor call join points matching ConstructorPattern.
Execution	execution(MethodPattern or ConstructorPattern): selects method execution join points matching MethodPattern or constructor execution join points matching ConstructorPattern.
Get/Set	get/set(FieldPattern): selects field reference/field set join points matching FieldPattern.
Initialization	initialization(ConstructorPattern): selects object initialization join points matching ConstructorPattern.
Preinitialization	preinitialization(ConstructorPattern): selects object pre-initialization join points matching ConstructorPattern.
Staticinitialization	staticinitialization(TypePattern): selects static initializer execution join points matching TypePattern.
Handler	handler(TypePattern): selects exception handler join points matching TypePattern.
Adviceexecution	selects all advice execution join points.
Within/Withincode	within(TypePattern)/ withincode(MethodPattern): selects join points where the executing code is defined in a type matched by TypePattern/ in a method matched by the MethodPattern.
Cflow/Cflowbelow	cflow(pointcut)/cflowbelow(pointcut): selects join points in the control flow of any join point P picked out by pointcut, including P itself. In case of cflowbelow P is not selected
This/Target/Args	This/Target/Args(Type or Id): selects join points where the currently executing object/target object/Argument is an instance of Type, or of the type of the identifier Id.
PointcutId	PointcutId(TypePattern or Id, ...): selects join points picked out by the user-defined pointcut designator named by PointcutId.
If	if(BooleanExpression): selects join points where the boolean expression evaluates to true.

# Chapter 3

## The Unified Language Design

---

In Chapter 1, I claimed that unification of aspects and classes in a language design is possible. I also claimed that this unification enhances the conceptual integrity of the programming model. To support these claims, in this chapter<sup>1</sup>, I describe the design and implementation of a programming language that unifies classes and aspects. This new module construct is informally called the *classpect*. I will present the design and implementation of Eos, an AspectJ-like language based on C# that supports the new module constructs, classpects, as the basic unit of modularity. The underpinnings of the language design include support for instantiation under program control, instance-level advising, advising as a general alternative to object-oriented method invocation and overriding, and the provision of a separate join-point-method binding construct.

In the next section, I reexamine one of the most fundamental decisions made early in the design of AspectJ: to support separate but closely related *class* and aspect module constructs. I also study the commitment that this decision entailed to a two-level program design style, with systems organized as object-oriented base layers advised by superimposed aspects.

### 3.1 Non-Orthogonality and Asymmetries in Language Design

The motivation for the unification of aspects and classes rests on two observations. First, separating classes and aspects reduces the conceptual integrity of the programming model, arguably making it

---

<sup>1</sup>This chapter presents a revised and extended form of some contents that appeared previously in [84].

harder in the long run for programmers to understand and use aspect-oriented programming. Second, the asymmetry of classes and aspects complicates system composition and ultimately harms modularity. Asymmetries occur in two areas. First, while aspects can advise classes, classes cannot advise aspects, and aspects cannot advise aspects in full generality. Second, aspect instances cannot be created or manipulated under program control in the same ways as class-based objects [81, 82]. In practice, these asymmetries constrain designers to the two-layer architectural style mentioned before. Hierarchical layering of aspects is difficult at best.

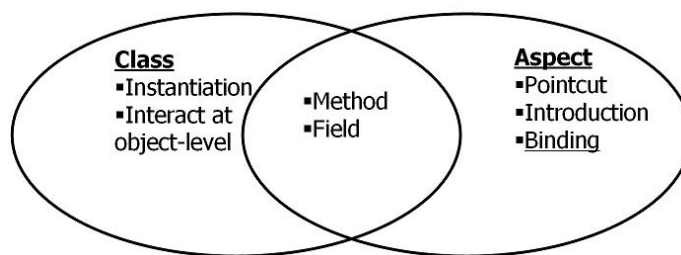


Figure 3.1: Non-Orthogonal and Asymmetric Aspects and Classes

The root of the asymmetries lies in non-orthogonal constructs in the language design. By non-orthogonality I mean that the language design does not comply with the MacLennan’s orthogonality principle [70], which suggests that independent functions should be controlled by independent mechanisms.

These languages introduce the aspect as a separate abstraction mechanism and advice as a separate procedural abstraction mechanism. The new aspect construct and old *class* construct are non-orthogonal. As shown in Figure 3.1, in some dimensions aspects in the current language models are same as classes, they can also support the data abstraction and inheritance abilities of classes. In others more expressive than classes, they can use pointcuts, advice, and inter-type declarations. In yet other dimensions less expressive than classes, they are not first-class. Rather, current language designs in general embrace a module-based view of what an aspect is and how it interacts with the rest of a system.

An aspect in the AspectJ language model is a module-like construct in the sense that it is (in most cases) treated as a single global module instance. In no case is aspect instantiation explicitly



under program control. There is no general-purpose mechanism such as *new* for aspect modules. Instead in the default case, a single global instance is created by the language runtime on the first reference to an aspect. An aspect's interaction with the rest of the system is static in the sense that an aspect modifies the behaviors of the classes that it advises, and thus all instances of a given *class* in a given system.

There are a few ad hoc mechanisms for associating aspect instances with object instances. For example, if a *perthis* modifier is used on aspect, an instance of the aspect is created for every object that is the executing object; if a *pertarget* modifier is used on aspect, an instance of the aspect is created for every object that is the target object of the join points e.g. target object of a call. Similarly, when *percflow* and *percflowbelow* modifiers are used on aspect, an instance of the aspect is created corresponding to every control flow. These ad hoc mechanisms show that first, there are use cases that need support for aspect instantiation, and second, instead of creating a general purpose mechanism such as *new* the language design evolved to become irregular by including new ad hoc constructs as they were needed. This is contrary to MacLennan's regularity principle [70], which suggests that *regular rules, without exceptions are easier to learn, use, describe, and implement*.

In a nutshell, aspects are thus not first-class in two different ways: they cannot in general be instantiated under program control; rather the language runtime manages aspect instantiation; nor can they advise individual object instances; rather selective instance-level advising has to be emulated by class-level advice, which leads to the performance penalties.

Many problems stem from this commitment. At the implementation level, for example, every object of a given *class* in a system built using a language such as AspectJ has to pay a performance penalty for any individual object of that *class* to be subject to advising, and the cost imposed on each instance increases with the number of aspects that advise its *class*. Such a structure is not scalable in general. In particular, for generic classes such as collection classes, instances of which are used in many different ways throughout a system, such a situation is likely to be unacceptable.

To understand the source of the performance penalty and to explore possible optimization techniques that can be applied, let us look at the underlying implementation of advising relationships

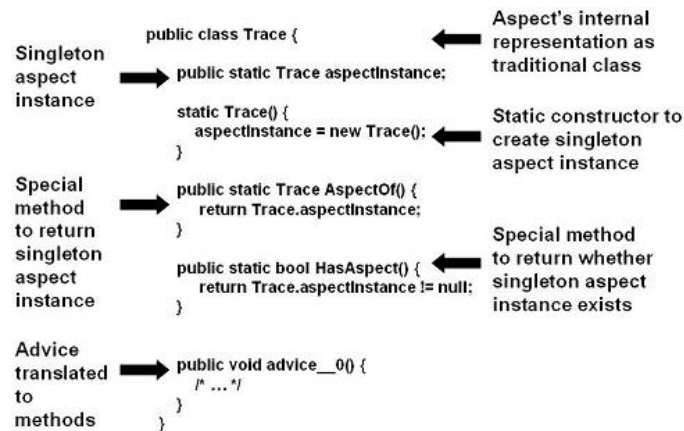


Figure 3.2: Underlying Representation of an Aspect in AspectJ 1.0

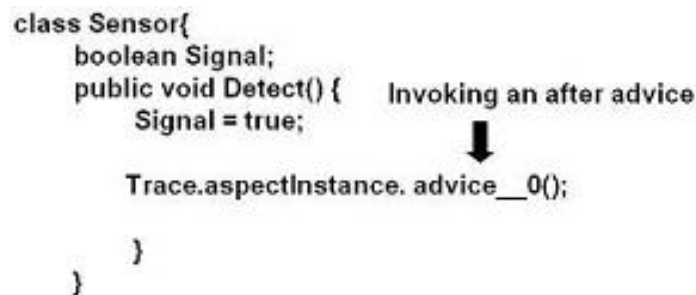


Figure 3.3: Underlying Representation of an After Advice in AspectJ 1.0

in the AspectJ (version 1.0). Figure 3.2 and Figure 3.3 shows the underlying implementation of an aspect and advice invocation at a join point. The aspect is translated to an object-oriented class. The advice in the aspect is translated to a method with an automatically generated name. The Figure shows the trivial case for advice invocation, where no reflective information is needed by the advice. As shown in Figure 3.3, advice invocation is implemented by statically inserting a method call at the join point. For non-trivial advice, reflective information at the join point is constructed before advice invocation. This and the method invocation itself are the sources of the performance penalty for individual objects. For each object of a class that is subject to advising, reflective information creation cost and advice invocation cost is incurred at the join point whether or not that particular object need to be advised. A further lookup cost to determine whether an object is subject to advising is generally incurred on the advice side.

A clever compiler implementation might be able to optimize away some of these overheads. For example, a compiler implementation that switches implementation strategies based on the nature of advising relationships in the system by looking at its execution profile may be able to provide the right implementation for the right advising scenario. Two possible optimization strategies seem possible: first, a compiler may collect the execution trace of the system beforehand and optimize the advising relationships according to the trace. Second, a compiler may add additional infrastructure to dynamically monitor and switch the implementation as needed at run-time. The first optimization strategy will incur a one-time cost of profiling the system. In this case, however, if the actual system behavior deviates from the profile some optimizations might have to be revisited in the light of the new execution profile. The second optimization strategy will incur a time and space overhead for including the monitoring infrastructure, for monitoring the program execution, and for switching the implementation strategy. I speculate that in most cases these optimizations might be able to offer reasonable performance guarantees. However, at the minimum, these commitments made at the language design level make the underlying language implementation complex.

Note that the design decision made by initial AspectJ language designers to commit to a *non-object-oriented*, namely *static module-based*, view of aspects were justified at the time. However, abandoning the key idea in object-oriented programming that running systems are composed of object instances, has real opportunity costs.

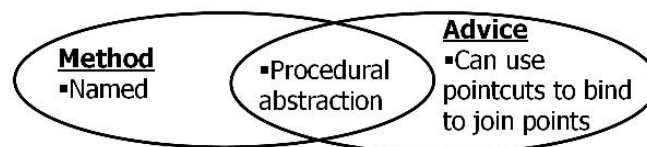


Figure 3.4: Non-Orthogonal and Asymmetric Advice and Methods

The advice and method constructs are also non-orthogonal and asymmetric. As shown in Figure 3.4, advice and method both support procedural abstraction. Advice is more expressive than method. It can use pointcuts to bind to join points. Advice in other dimensions is less expressive than method. They are anonymous therefore it is not possible to distinguish between two advice constructs in a pointcut expression, instead the pointcut descriptor *adviceexecution* selects all ad-

vice constructs in a program. Note that pointcuts use lexical pattern matching on names to select join points. As a result, although an advice can advise methods with fine selectivity, they can select advice bodies to advise only in coarse-grained ways.

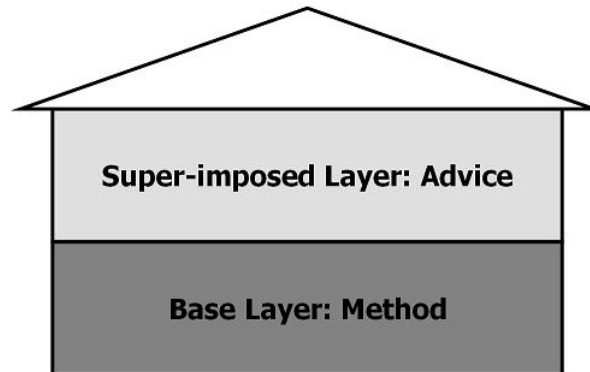


Figure 3.5: A World of Two-Story Buildings

This restriction constrains application of advising as an invocation mechanism to two-layered structures (analogous to the world of two-story buildings shown in Figure 3.5) where methods at the bottom level are being advised by advices at top level. It also results in the lack of full aspect-aspect compositionality in the language model.

The lack of full aspect-aspect compositionality precludes use of advising as an invocation mechanism in a range of architectural styles including layered, hierarchical, and networked [40]. Rajan and Sullivan [84] provide a case study involving hierarchical architectures in which connectors integrate two or more components, and another level of connector treats two or more connectors as components. Another case study involving layered architectural style is discussed later in Chapter 5. Both case studies demonstrate the restrictive compositionality. Sullivan and Notkin showed that such requirements are useful in designing systems for ease of evolution [96, 97]. The inability to support such styles without workarounds—and the tacit constraints to two-layered designs, restricts natural use of aspect technology for separating concerns in a fully compositional style.

The next section describes the unified language design in which advising emerges as a general alternative to overriding or method invocation. There are two basic requirements for a unified, more compositional model. First, a new model should preserve the expressive power of AspectJ.

This constraint rules out the use of languages with much more limited join point and pointcut models. Second, a unified design should be based on a single, first-class, class-like module construct (whereas AspectJ-like languages have both aspects and classes), and a single method construct for procedural abstraction (whereas AspectJ has both methods and advice).

## 3.2 Unifying Aspects and Classes

Eos unifies aspects and objects in three ways as shown in Figure 3.6. First, it unifies aspects and classes. A *class* in Eos supports the full *classpect* notion: all C# *class* constructs, all of the essential capabilities of AspectJ aspects, and the Eos extensions to aspects needed to make them first-class objects. Second, Eos eliminates advice in favor of using methods only, with a separate and explicit join-point-method binding construct. Third, Eos supports a generalized advising model. To the usual object-oriented mechanisms of explicit or implicit method call and overriding based on inheritance we add implicit invocation using *before* and *after* bindings, and overriding using *around* bindings, respectively.

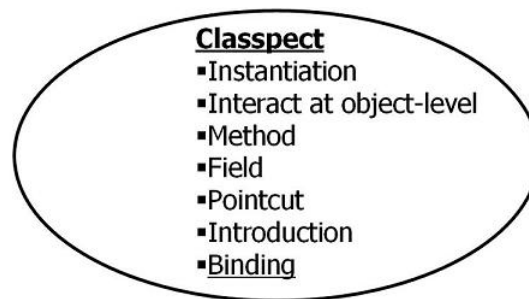


Figure 3.6: Classpect: A Unified Model

The unification proposed in this research also breaks the commitment to the *static, module-based view* of aspects. It provides a more general instantiation model in which classpects can be instantiated just like OO classes using the operator *new*. Classpects may provide constructors like classes.

### 3.3 Crosscut Specification

Eos eliminates *anonymous advice* in favor of *named-methods*. It also makes join-point-method bindings separate and abstract, in a style similar to the event-method binding constructs of implicit invocation systems [39, 96, 97]. Eos separates what I call crosscut specifications from advice bodies, which are now just methods, as shown in Figure 3.7. A crosscut specification defines both a pointcut and when given advice should run: before, after or around. This separation allows one to reason separately about binding issues and to change them independently; and it supports advice abstraction, overloading, and inheritance based on the existing rules for methods.

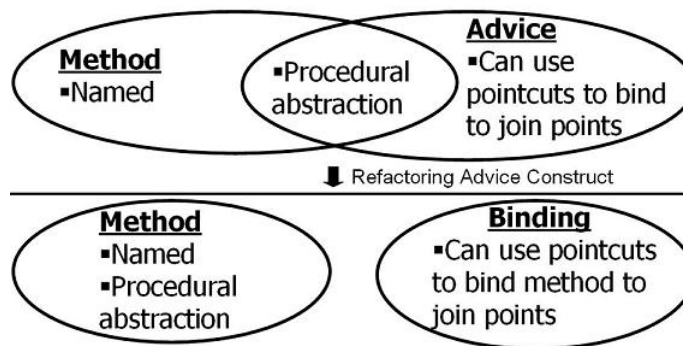


Figure 3.7: Re-factoring Advice into Method and Binding

```
binding_declaration
: opt_static AFTER pointcuts COLON method_bindings SEMICOLON
| opt_static BEFORE pointcuts COLON method_bindings SEMICOLON
| opt_static type AROUND pointcuts COLON method_bindings SEMICOLON
;

method_bindings
: method_bindings COMMA method_binding
| method_binding
;

method_binding
: IDENTIFIER OPEN_PAREN opt_formal_parameter_list CLOSE_PAREN
;

opt_static
: EMPTY
| STATIC
;
```

Figure 3.8: Syntax of the Eos Binding Declaration

<pre> /* Advice */ before(): execution(Sensor.Detect()){ } </pre>	<pre> /* Binding */ before():execution(Sensor.Detect()): beforeDetect();  /* Method Equivalent to Advice Body*/ void beforeDetect(){ } </pre>
---	---

Figure 3.9: An AspectJ Advice and an Equivalent Eos Binding Declaration

The grammar production, binding-declaration (Figure 3.8) presents the crosscut specification construct. A binding-declaration has four parts. The first, *opt-static*, specifies whether a binding is static. Non-static bindings result in instance-level advising [81] – selective advising of the join points of individual object instances. A static binding affects all instances of advised classes. The second part of a binding (*after/before/around*) states when the advising method executes: *after*, *before*, or *around*. The third part, *pointcuts*, selects the join points at which an advising method executes. These join points are called *subjects* of the binding. The final part specifies the advising methods. These advising methods are called *handlers* of the binding.

A binding provides a list of methods to execute at a join point, in the specified order. A binding can also pass reflective information about the join point to the methods invoked, by binding method parameters to reflective information using the AspectJ pointcut designators such as *this*, *target*, *args*, etc. As with around advice in AspectJ, an Eos method bound around is allowed to return a value, so around bindings must be declared with return types.

Methods subject to binding have to follow certain rules. First, a method must be accessible in the *class* declaring a binding. Second, a method bound before or after a join point can have only void as a return type. Third, a method bound around a join point must have a return type that is a subtype of the return type at the join point. For example, if method Foo is bound around the execution join point `execution(public int *.Bar())`, then it must return `int`.

The listing in Figure 3.9 presents an advice construct as it would appear in current aspect languages and the equivalent method binding in Eos. The advice executes before the join point `execution(Sensor.Detect())`. The binding separates the advice body from the crosscut specification. The advice body becomes the body of the method `beforeDetect`. The crosscut specification becomes part of the binding (on the top right hand side).

### 3.3.1 Around Bindings

An *around* advice in AspectJ is executed instead of a join point, and can invoke the join point using *proceed*. In essence, the around advice overrides the join point method, with calls to *proceed* being analogous to delegating calls to *super*. In Eos, a method bound around is also executed instead of the join point. If the method might need to call the overridden method, the first method takes an argument of type *Eos.Runtime.AroundADP*. This *class* represents a delegate chain including the original join point and other around method bindings, and it provides a method called *InnerInvoke* to invoke the next element in the delegate chain. The argument to the method is bound to the delegate chain at the join point using the pointcut designator *aroundptr* (line 6 in Figure 3.10).

```

1 void Cache (Eos.Runtime.AroundADP d){
2     if( /* need to invoke inner join point */ )
3         d.InnerInvoke();
4 }
5 static void around execution(public void SomeClass.SomeMethod())
6     && aroundptr(d): Cache(Eos.Runtime.AroundADP d);

```

Figure 3.10: A Method Bound Around

The binding (lines 5-6) binds the method *Cache* around the execution of *SomeClass.SomeMethod* and exposes the around delegate chain at the join point using the pointcut expression *aroundptr(d)*, which binds the reference to the delegate chain to the argument *d* of the method *Cache* (lines 1-4). The method *Cache* can invoke the inner delegate in the chain by invoking the method *InnerInvoke* on *d* (line 3).

Unifying around advice and methods poses a question: whether to allow *proceed* in all methods. Allowing *proceed* in methods that are bound around but not in other methods introduces a special case. Making inner join point invocations explicit in an object-oriented style eliminates this special case. The Eos *InnerInvoke* method removes any such special cases from the language design.

A current limitation of our language implementation is that the return type of the method *InnerInvoke* is *object*, precluding static type checking. Method return values could be statically typed to be the same as the surrounding around method using generics. As of this writing, Eos is build upon .NET Framework 1.1. Generics are not supported in version 1.1. Full support for generics is



expected in .NET Framework 2.0.

### 3.3.2 New Pointcut Designators

To pass reflective information at a join point to a bound method, a binding uses AspectJ-like pointcut designators *args*, *target* and *this*. In AspectJ-like languages, three special variables are visible within the bodies of advice: *thisJoinPoint*, *thisJoinPointStaticPart*, and *thisEnclosingJoinPointStaticPart*. These variables can be used to explicitly marshal reflective information at a join point. For example, to access the return value at a join point, one calls the method *getReturnValue* on the variable *thisJoinPoint*.

Unifying advice and methods poses another question: whether to allow these special variables in all methods. Allowing these variables in methods that are bound *before*, *after* or *around*, but not in other methods introduces a special case. Eos removes this special case by binding method arguments to the required reflective information in the crosscut specification construct using pointcut designators.

The pointcut designators in the original Eos are incomplete for this purpose, in that not all the information available at join points is exposed. Other information, marshaled earlier from the three special variables, might be needed. For example, to access the return value at a join point, one calls the method *getReturnValue* on the implicit argument. Eos adds new pointcut designators to fill the gap. For example, the pointcut designator *return* exposes the return value at the join point. The pointcut designator *joinpoint* exposes all information about the join point by exposing an object of type *Eos.Runtime.JoinPoint*. These designators allow previously implicit arguments to advice to be passed as explicit arguments to the method bound at the join points.

Eos fulfills the requirements laid out for a unified model. There is one unit of modularity, *class*, and one mechanism for procedural abstraction, *method*. All of the essential expressiveness of AspectJ-like languages is present in Eos, along with the extensions needed for aspects to work as first-class objects, as they must in a unified model. In addition, join-point-to-method bindings are separate, orthogonal, abstract interface elements in Eos. Eos thus does appear to achieve a novel unity of design in the programming model w.r.t. the family of AspectJ like languages.

### 3.4 Additional Power of Overriding

In AspectJ-like languages, there are two different ways to override a method: by object-oriented inheritance and by aspect-oriented around advice. A consequence of replacing advice bodies with methods is that methods that serve as advice can be overridden in either of these ways. These mechanisms differ fundamentally, and in a way consistent with the nature of aspect-oriented programming: not in their effect on runtime behavior, but rather on the design structure.

Consider two analogies. In object-oriented systems that support implicit invocation [39], there are two ways for an invoker to invoke an invokee: explicit call or implicit invocation. The runtime result is the same, but the design-time structures are different. Having both mechanisms gives the designer the flexibility to shape the static structure independently of the runtime invocation structure. Inter-type declarations in AspectJ-like languages provide a similar capability for *class* state and behavior. They allow a third party aspect to change the members of a *class* without the involvement of the *class* itself. The runtime effects are again the same, but the resulting architectural properties are different. Supporting inheritance and around advising as two mechanisms for overriding methods that serve as advice bodies provides just such architectural flexibility with respect to advice overriding. Object-oriented overriding demands an inheritance relation; aspect-oriented around advising does not [84].

A binding can be static or non-static. A non-static binding results in instance-level advising. The next section describes this generalization in detail <sup>2</sup>.

### 3.5 Static vs. Non-Static Binding

The (static binding, method) pair is equivalent to AspectJ advice. The non-static binding allows selective instance-level advising. The non-static binding is illustrated in Figure 3.11. This example system has three components of type *Sensor*, *s1*, *s2*, and *s3*. A sensor is a battery-operated device. To increase the lifetime of the battery in the device, when it is not being used it is put in the power-save mode. When a sensor is in power saving mode it is not affected by the changes in the

---

<sup>2</sup>Some of the contents of this chapter appeared in [81].

environment. At a given time, a subset of sensors can be in power saving mode. To keep the power saving mode concern separate, this feature is modeled as a *classpect Mask*. The *Mask classpect* masks a subset of sensors, so that they are not affected by the environmental changes.

```

1  public class Sensor{
2      boolean Signal;
3      public void Detect(){
4          System.Console.WriteLine("`Detected Signal'");
5          Signal = true;
6      }
7  }
8  public class Mask {
9      public Mask(Sensor s){
10         this.s = s; addObject(s);
11     }
12     Sensor s;
13     void around():execution(public void Sensor.Detect()): MaskIt();
14     public void MaskIt(){ /* Mask detection by not calling proceed */
15         System.Console.WriteLine("`Sensor Masked'");
16     }
17 }
18 public class SensorMain{
19     public static void Main(string[] arguments){
20         Sensor s1 = new Sensor(); /* Make 3 Sensors */
21         Sensor s2 = new Sensor();
22         Sensor s3 = new Sensor();
23         Mask m1 = new Mask(s2); /* Mask Sensor 2 */
24         System.Console.WriteLine("`Setting s1'");
25         s1.Detect();
26         System.Console.WriteLine("`Setting s2'");
27         s2.Detect();
28         System.Console.WriteLine("`Setting s3'");
29         s3.Detect();
30     }
31 }
32 Output:
33 Setting s1
34 Detected Signal
35 Setting s2
36 Sensor Masked
37 Setting s3
38 Detected Signal

```

Figure 3.11: Masking Sensor's Selectively

The sensor component type is modeled as the *Sensor classpect* (lines 1–7). The instances of the sensor are modeled as the instances of the *classpect* (lines 20–23). These sensors are tested by the

main program one-by-one (See lines 24–29). Before testing each sensor, the main program prints its intent (lines 24, 26, and 28). When the *Detect* method (line 3–6) in the *Sensor* class is called it prints a message *Detected Signal*, sets the value of the field *Signal* (line 2) to true, and returns.

The feature *Mask* (lines 8–17) is also represented as a *classpect*. A *classpect* provides the ability to selectively advise object instances via non-static bindings and *new* for instantiation. An *aspect*, on the other hand advises a *class* and thus all objects of that *class* and does not provide a general mechanism for instantiation. Here the *classpect* *Mask* bounds the method *MaskIt* (lines 14–16) around the execution of the *Sensor.Detect* method using a binding (line 13). This is a non-static binding. The Eos compiler implementation reads the non-static modifier as a hint to allow instance-level weaving. The Eos compiler delays binding of join point to methods until runtime. At compile time, it attaches event stubs at the matched join points, and generates implicit methods *addObject* and *removeObject* methods in *Mask* to enable run-time registration and deregistration. When *addObject* is called inside a *classpect* with a reference to an instance, it registers all bound methods to be implicitly invoked at all matched join points for that object. When *removeObject* is called inside a *classpect* with a reference to an object of a *class*, the bindings in the *classpect* deregisters all bound methods from all matched join points for that object. I will describe the weaving process in detail in the next section.

The *classpect* *Mask*'s constructor (lines 9–11) takes a reference to a sensor instance as argument, stores the reference in the member field *s* and calls the implicit method *addObject* with the reference as an argument. When *addObject* is called with *s* as an argument (line 10), the method *MaskIt* (lines 14–16) is bound around the join point *execution of method Detect* of the object *s*.

Similar to an object-oriented class, an instance of the classpect *Mask* is created (line 23) using *new*. A reference to the second sensor *s2* is also passed as argument to the constructor, at which point the *classpect* starts advising the sensor instance. The result of selective advising is that the execution of *Detect* on *s2* is overridden by the method *MaskIt* (lines 14–16). When the sensors are tested *s1* and *s3* behave normally printing *Detected Signal* (lines 33–34 and 37–38); however, when the *s2* is tested the method prints the message *Sensor Masked* (lines 35–36) and skips the execution of the *Detect* method.

```

1 class Trace{
2     static before execution(public void Sensor.Detect()):trace();
3     public void trace(){
4         Console.WriteLine("`Before Sensor.Detect`");
5     }
6     public static void Main(string[] arguments){
7         Sensor s1 = new Sensor();
8         Sensor s2 = new Sensor();
9         Sensor s3 = new Sensor();
10        Console.WriteLine("`Setting Sensor 1`");
11        s1.Detect();
12        Console.WriteLine("`Setting Sensor 2`");
13        s2.Detect();
14        Console.WriteLine("`Setting Sensor 3`");
15        s3.Detect();
16    }
17 }
18

```

Output

```

Setting Sensor 1
Before Sensor Detect
Setting Sensor 2
Before Sensor Detect
Setting Sensor 3
Before Sensor Detect

```

Figure 3.12: Static Trace Using Binding

The second example shown in Figure 3.12 and Figure 3.13, illustrates the difference between static and non-static bindings. Figure 3.12 shows an example static binding (Line 2 in Figure 3.12) and the Figure 3.13 shows an example non-static or instance-level binding (Line 2 in Figure 3.13). The effect of static binding is to execute the method *trace* (Lines 3-5 in both Figures) after the execution of the method *Detect* for all *Sensor* instances (Lines 10, 12 and 14 in Figure 3.12). In the non-static binding case, however, the *classpect SelectiveTrace* selectively advises the *Sensor* instance *s1* by calling the implicit method *addObject* on that instance (Line 11 in Figure 3.13). The implicit method *addObject* registers the method *trace* (Lines 3-5 in Figure 3.13) with the *Sensor* instance. As a result the method *trace* (Line 3-5 in Figure 3.13) is only executed after the execution of the method *Detect* on *s1* (Line 17).

```

1 class SelectiveTrace{
2     before execution(public void Sensor.Detect()):trace();
3     public void trace(){
4         Console.WriteLine("`Before Sensor Detect`");
5     }
6     public static void Main(string[] arguments){
7         Sensor s1 = new Sensor();
8         Sensor s2 = new Sensor();
9         Sensor s3 = new Sensor();
10        SelectiveTrace t = new SelectiveTrace();
11        t.addObject(s1);
12        Console.WriteLine("`Setting Sensor 1`");
13        s1.Detect();
14        Console.WriteLine("`Setting Sensor 2`");
15        s2.Detect();
16        Console.WriteLine("`Setting Sensor 3`");
17        s3.Detect();
18    }
19 }

```

Output

```

Setting Sensor 1
Before Sensor Detect
Setting Sensor 2
Setting Sensor 3

```

Figure 3.13: Selective Trace Using Binding

### 3.6 Weaving of Static and Non-Static Bindings

The Eos compiler performs weaving on the source code to process the bindings and to generate appropriate stubs. In the rest of this section, I will discuss the weaving process for both static and non-static bindings. I will use the example described in the previous section to illustrate the process. Note that this is just one realization of the unified model. Other implementations are also possible.

Let us first look at the output of the weaving process in case of static bindings. Figure shows the output *Sensor* and *Trace* classes. The code inserted by the weaving process is in bold face font. In case of classpects containing static bindings, like AspectJ, Eos compiler inserts a static instance *staticInstance* (Line 16), a static constructor to initialize this instance (Lines 17–19), a method *StaticInstance* (Lines 20–22) to retrieve this static instance and a method *HasBindings* (Lines 23–25) to check if the classpect contains any bindings. The compiler also inserts a line (Line 4) in the method *Detect* (Lines 3–5) in the classpect *Sensor* (Lines 1–7) to invoke the method *trace* in the

```

1  class Sensor{
2      boolean Signal;
3      public void Detect(){
4          Trace.staticInstance.trace();
5          Signal = true;
6      }
7  }
8  using Eos.Runtime;
9  class Trace{
10     public void trace(){
11         Console.WriteLine("`Before Sensor.Detect`");
12     }
13     public static void Main(System.String[] arguments){
14         ...
15     }
16     public static Trace staticInstance;
17     static Trace(){
18         staticInstance = new Trace();
19     }
20     public static Trace StaticInstance(){
21         return Trace.staticInstance;
22     }
23     public static bool HasBindings(){
24         return Trace.staticInstance != null;
25     }
26 }

```

Figure 3.14: Underlying Implementation of Static Trace

classpect *Trace*. The handler method *trace* is invoked on the static instance *staticInstance* (Line 16) of the classpect *Trace*. This implementation technique is the same as that used by the AspectJ compiler. As a result, Eos implementation does not incur any additional space and time overhead due to the weaving process when compared to AspectJ implementation.

Let us now look at the output of the selective tracing using non-static bindings. Figure 3.15 shows the output *Sensor* and *SelectiveTrace* classes. The code inserted by the weaving process is in bold face font. In case of classpects containing non-static bindings, the Eos compiler generates and inserts two methods, *addObject* and *removeObject*, in the classpect. Here these methods are added to the classpect *SelectiveTrace* (Lines 11–34).

For each join point that is the *subject* of any non-static binding in the system, a delegate of type *Eos.Runtime.ADP* (Line 8) is inserted in the classpect. The type *Eos.Runtime.ADP* is part of the Eos runtime API and it provides operations to manipulate delegates. The delegate is named to avoid

```

1  class Sensor{
2      boolean Signal;
3      public void Detect(){
4          if(ADP_Detect!=null)
5              ADP_Detect.Invoke();
6          Signal = true;
7      }
8      public Eos.Runtime.ADP ADP_Detect;
9  }
10 using Eos.Runtime;
11 class SelectiveTrace{
12     public void trace(){
13         Console.WriteLine("`Before Sensor.Detect`");
14     }
15     public static void Main(System.String[] arguments){
16         ...
17     }
18     public void addObject(object obj){
19         if(obj == null)return;
20         if(obj is Sensor){
21             Sensor cobj = ((Sensor)(obj));
22             cobj.ADP_Detect = ADP.Combine(
23                 cobj.ADP_Detect, ADP.Create( this,`trace`));
24         }
25     }
26     public void removeObject(object obj) {
27         if(obj == null)return;
28         if(obj is Sensor){
29             Sensor cobj = ((Sensor)(obj));
30             cobj.ADP_Detect = ADP.Remove(
31                 cobj.ADP_Detect, ADP.Create( this,`trace`));
32         }
33     }
34 }

```

Figure 3.15: Underlying Implementation of Selective Trace

naming conflicts. Here, a delegate *ADP\_Detect* is inserted in the *Sensor* class corresponding to the point *before the execution of the detect method*. For presentation purposes the original name of the delegate *ADP\_Eos\_before\_execution\_Detect* is shortened to *ADP\_Detect*. The location of the join point is instrumented to check if this delegate is *null* and if not call the method *Invoke* on it to run all the delegates that are listening to that join points. In the case of selective tracing, the method *Detect* of the classpect *Sensor* is instrumented (Lines 4–5) to check the value of the delegate *ADP\_Detect* and if it is not *null* call the method *Invoke* on it.

The generated method *addObject* (Lines 18–25) takes an instance of type *object* as argument



(Line 18). The selective weaving process is applied to this instance. The *addObject* method first checks if the supplied instance is *null* (Line 19). If it is *null* the method returns immediately (Line 19). An *if* statement block is generated in the method *addObject* for each type that contain any subject join points for a non-static binding in the classpect. The *if* statement block checks if the argument *obj* of the method *addObject* is of the type of interest. If the argument *obj* is of the type it is casted into an object *cobj* of that type.

For each (subject join point, method handler) binding pair such that the join point is contained in that type, a delegatee corresponding to the method handler is added to the delegate corresponding to the join point. Here, the only subject join point is *execution of the method Detect* and it is contained in the type *Sensor*, hence only one *if* statement block (Lines 20–24) is generated that checks whether *obj* is an instance of *Sensor*. The instance *obj* is then cast to an object *cobj* of the *Sensor* type. Here, the only method handler is *trace*, so after casting, a delegatee for the method *trace* is created by calling the operation *Create* of the runtime type *Eos.Runtime.ADP* (Line 23). The operation *Create* takes an instance and a method name as argument and creates a delegate to call the named method on the supplied instance. The delegate is then combined with the delegate *ADP\_Detect* of the *Sensor* instance *cobj* using the operation *Combine* of the runtime type *Eos.Runtime.ADP*. The operation *Combine* takes two delegate instances and combines their delegate chains such that the delegates in the chain of the second delegate instance are appended at the end of the delegate chain of the first delegate instance.

The method *removeObject* works similarly. It checks the argument object instance for being null and determines if the type of the supplied instance contains any subject join points. If so, a delegatee for all the handler methods are created and removed from corresponding join point delegates using the operation *Remove* of the runtime type *Eos.Runtime.ADP*. The operation *Remove* takes two delegate instances and removes the first occurrence of the delegate chain of the second instance from the delegate chain of the first delegate instance. For the classpect *SelectiveTrace*, the method *removeObject* (Line 26–33) first checks whether the argument instance is null (Line 27). It then verifies whether it is of type *Sensor*. If it is, a delegatee for the method *trace* is created by calling the operation *Create* of the runtime type *Eos.Runtime.ADP* (Line 31). The delegate is then

removed from the delegate *ADP\_Detect* of the *Sensor* instance *cobj* using the operation *Remove* of the runtime type *Eos.Runtime.ADP* (Line 30).

In case of non-static bindings, only time overhead incurred by all instances of a classpect is a check to determine if the delegate corresponding to a join point is null. Only those instances that are actually advised by calling *addObject* invoke the delegate chain to run the handler methods. The space overhead for each instance of a classpect is an additional field of type *Eos.Runtime.ADP* corresponding to each subject join point in the classpect. The space overhead for classpect that also have bindings are two additional methods *addObject* and *removeObject*. The implementation strategy illustrates that in use cases where there is a need to emulate AspectJ like aspects, a non-static binding can be used without incurring any additional overhead with respect to the AspectJ 1.0 implementation. In use cases where there is a need to selectively advise object instances, non-static bindings can be used. The only additional overhead for every advised instance is a *null check*. These overheads are analyzed in detail in the next section.

### 3.7 Performance Analysis of Static vs. Non-Static Bindings

The provision of static and non-static bindings naturally leads to the question: *When should a static binding be used as opposed to non-static binding?*. This section analyzes the system structures and their fit, performance-wise, with either of these two levels of granularity of bindings, and offers guidelines for organizing bindings. In this analysis, relevant factors are total number of instances of a *class* and the subset of these instances that are affected by a binding. Let us assume a set of instances of a *classpect*  $C_i$  is  $N_{C_i}$  and a set of instances of classpects  $C_i$  that are affected by a binding  $B_j$  is  $M_{C_i B_j}$ . The fraction of the instances of *class*  $C_i$  being affected by binding  $B_j$  is  $\frac{|M_{C_i B_j}|}{|N_{C_i}|}$ , where  $|N_{C_i}|$  denotes the cardinality of the set  $N_i$  and  $|M_{C_i B_j}|$  denotes the cardinality of the set  $M_{C_i B_j}$ . Depending on the value of this fraction, this binding can be classified into the following five partitions.

1.  $\frac{|M_{C_i B_j}|}{|N_{C_i}|} = 1,$
2.  $\frac{|M_{C_i B_j}|}{|N_{C_i}|} \rightarrow 1,$

3.  $1 - \delta > \frac{|M_{C_i B_j}|}{|N_{C_i}|} > \delta$ , where  $\delta \rightarrow 0$ ,
4.  $\frac{|M_{C_i B_j}|}{|N_{C_i}|} \rightarrow 0$ ,
5.  $\frac{|M_{C_i B_j}|}{|N_{C_i}|} = 0$ .

For the bindings in the first partition, the fraction is one (i.e., the total number of instances of the *class* is equal to the number of instances that should be advised). These bindings should always be static. For the bindings in the fifth partition, the fraction is zero (i.e., no instance is being affected). For the bindings in the second, third, and fourth fifth partition, the fraction is between 0 and 1 (i.e., some instances are being affected). In the second partition, the fraction tends to 1, in other words nearly all instances are being affected but not all. In the fourth partition, the fraction tends to 0, which means that a very small fraction is being affected. The rest of the bindings are in the fourth partition.

Ideally, bindings in the second, third, and fourth partitions need to be non-static or in other words binding should be selective instance-level. If the language implementation (i.e. compiler) does not support selective instance-level bindings, it can be implemented in at least two different ways. In the first implementation strategy, the method bound by  $B_j$  keeps a list of instances ( $M_{C_i B_j}$ ) being advised and then looks up the invoking instance in that list. This implementation strategy will incur lookup overhead and method invocation overhead. Let us assume the method invocation overhead to be  $O_{invoke}$  and the constant lookup overhead of looking up an instance in the list of instance ( $M_{ij}$ ) to be  $O_{lookup}$  (assuming an  $O(1)$  lookup algorithm). Total overhead of this implementation technique is:

$$\sum_j \sum_{i \text{ (where } |M_{C_i B_j}| > 0)} ((N_{C_i} - M_{C_i B_j}) * O_{invoke} + N_{C_i} * O_{lookup}).$$

This overhead has two parts: the overhead incurred by each instance that should not be affected by the binding and the overhead of looking up an instance to determine whether it should be affected by the binding. The second part of the overhead is constant. For bindings in the second partition, the first part of this overhead is not significant, whereas for bindings in the fourth partition, the first part of the overhead is significant. For bindings in the third partition, as the fraction of instances

affected decrease, this overhead will increase. It might be possible, however, for a compiler to employ static or dynamic analysis techniques such as profiling to optimize these overheads.

In the second implementation technique, an object instance will keep a list of bindings affecting it and invoke all bound methods in the list one by one or depending on some precedence. There is no lookup overhead in this case; however, a zero check will be necessary to determine whether the list is empty. This condition check will be necessary for only the join points that are potentially affected by the binding. Let us assume constant overhead of this condition check is  $O_{zero}$ . The total overhead in this case will be:

$$\sum_{\forall i \exists j (|M_{C_i A_j}| > 0)} N_{C_i} * O_{zero}.$$

Given that  $O_{zero} \ll O_{lookup}$  the overhead in first case will be significantly larger than second case. The Eos compiler implements the second technique (to achieve minimal overhead) without introducing design dependence. It eliminates the need to provide the implicit invocation mechanism sufficient to meet the need of the bindings by abstracting the implicit invocation structure behind join point/pointcut mechanism.

## 3.8 Related Work

### 3.8.1 Unification of Aspects and Classes

AspectJ [52], AspectWerkz [6], and Caesar [75] are all related to our work. Kiczales reports [13] that in at least one early version of AspectJ, there was no separate aspect construct. Rather, the *class* was extended to support advice. No evidence indicates, however, that those early designs achieved the synthesis of OO and AO techniques of Eos. Advice bodies and methods were still separate; it is unclear to what extent advice could be advised at all; and there was no support for flexible aspect instantiation.

AspectWerkz [6] is the design most closely related to our work. The aim of this project was to provide the expressiveness of AspectJ [52] without sacrificing pure Java and the supporting tool infrastructure. The solution is to use normal Java classes to represent both classes and AspectJ-like aspects, with advice represented in normal methods, and to separate all join-point-advice bindings

either into annotations in the form of comments, or into separate XML binding files. AspectWerkz provides a proven solution to the problem of AspectJ-like programming in pure Java, but it does not achieve the unification that we have pursued.

First, and crucially, the system does not support the concept of aspects as objects under program control; rather it is really an implementation of the AspectJ model. Instead, the use of Java classes as aspects is highly constrained so that the runtime system can maintain control. A *class* representing an aspect must have either no constructor or one with one of two predefined signatures, and a method representing an advice body has one argument of type `JoinPoint`. AspectWerkz uses this interface to manage aspect creation and advice invocation. AspectWerkz also lacks a single-language design, in that it uses both Java and XML binding files. Third, AspectWerkz lacks static type checking of advice parameters. Rather, reflective information is marshaled from the `JoinPoint` arguments to advice methods.

The design of Caesar [75] is also closely related to our approach. The aim of Caesar was to decouple aspect implementation and the aspect binding with a new feature called an aspect collaboration interface (ACI). By separating these concepts from aspect abstraction, Caesar enables reuse and componentization of aspects. This approach is similar to ours and to AspectWerkz in that it uses plain Java to represent both classes and aspects; however, it represents advice using AspectJ like syntax. Methods and advices are still separate constructs, and advice constructs couples crosscut specifications with advice bodies. Consequently, as in AspectJ, advice bodies are still not addressable as individual entities. They can be advised as a group using an advice-execution pointcut. In Caesar, as in Eos, advice can be bound statically or dynamically; however, aspects in Caesar cannot directly advise individual objects on a selective basis. In the next chapter, I will show that both first-class aspect instances and instance-level advising are essential for expressing integration concerns as aspects [82,92].

Aspect languages such as HyperJ [101,77] have one unit of modularity, classes, with a separate notation for expressing bindings. However, they do not support program control over aspects as first-class objects, and to date the join point models that they have implemented have been limited mainly to methods [46].

Tucker and Krisnamurthi [103] consider making both pointcuts and advice first-class entities in the functional language context. Their approach builds upon a base language that already provides functions as first-class entities expressing pointcuts and advices as a function easy. On the other hand, in languages such as Java and C# functions are not first-class entities making the design of aspect languages difficult.

### **3.8.2 Affecting Behaviors of Instances Selectively**

The idea to affect behaviors of instances selectively is not new. It has appeared in many forms including, the observer–pattern based design style in which observers selectively registers with objects to observe and affect their behavior. Many aspect–oriented approaches, e.g. AspectS [47], Composition Filters [1], the Aspect Moderator Framework [18], the Object Infrastructure Framework [33], and EAOP [25], that take a wrapper-based approach, in which messages sent to and from components are intercepted for processing by aspect wrapper objects, also have the ability to selectively affect behaviors of instances. The wrapper-based approach has appeared in many forms over the decades: from Common Lisp, to tool integration frameworks. It has been picked up and given an AOSD interpretation by efforts such as Sina/st [59] and ility–insertion [33].

Many such languages are called aspect–oriented, however, by taking the wrapper–based approaches they generally give up on the richness of the join point model and limit themselves to what has been called operational–level composition [46]. They provide a limited or non–existent pointcut language. Other AspectJ-like languages, such as AspectC++ [36] and AspectR [4], on the other hand, do not support first-class aspect instances and instance–level advising. Languages such as HyperJ [101] and DJ [71], in which the pointcut concept does not apply, also lack the instance–level capabilities.

As part of this research, a survey of a range of major aspect-oriented programming languages was conducted. The survey showed that none except Eos support first-class aspects and generalized weaving model. The required language should have support for rich join point models, expressive pointcut languages, instance-level weaving, and first-class aspect instances. These terms are defined below.

Table 3.1: A Characterization of a Subset of Aspect-Oriented Languages and Approaches

*Abbreviations for the table*

IL: Instance-level aspect weaving

CL: Call, execution and return join point

E: Exception handling join points

M: Messages as join points

RP: Rich pointcut sub language

FS/FG: Field set and get join points

OI: Object initialization join points

Languages	IL	RP	CL	FS/FG	E	OI	M
AspectJ [52]		X	X	X	X	X	
HyperJ [100]			X				
EAOP [25]	X		X				
Composition filter model [5]	X		X				X
Aspect Moderator Framework [18]	X		X				
Object Infrastructure Framework [33]	X		X				X
DJ [71]			X				
AspectC# [56]			X				
Claw			X				
AOP#			X				
AspectR [10]			X	X	X	X	
AspectC++ [36]		X	X	X			
AspectS [47]	X		X				X
AspectWerkz [6]		X	X	X	X		
Caesar [75]			X				
Eos	X	X	X	X	X	X	

- By *rich join point model* we mean that a language defines join points well beyond function call and return. Important join points include field get and set, exception and message passing, and object creation and deletion.
- By *expressive pointcut language*, we mean that a language permits declarative expression of join points to be advised by an aspect. The ability to *quantify* [35] join points across a program is a key enabler of modular representation of far-reaching crosscutting concerns, such as call tracing.
- By *instance-level weaving* we mean that advice can be woven with selected object instances.

Instance level weaving is different from run-time aspect weaving. The former allows the developers to say, *Weave this aspect to only these instances of that class*, whereas the later allows them to say either explicitly or implicitly *When the control reaches this point of execution weave this aspect and when it reaches that point unweave it*. An implementation of instance level aspect weaver may provide additional abilities to dynamically attach and detach aspects from base object at run-time or the ability to instantiate an aspect but by definition it is not required to do so.

- By *first-class aspect instances* we mean first-class objects of given aspect types, typically used to maintain separate aspect state for separately advised objects. The ability to create an instance of an aspect using a general-purpose mechanism such as *new* naturally stems from this.

Table 3.1 presents the state of the art in aspect language design as reflected in a broad survey of extant aspect languages. Rows name languages; columns, features. For languages such as HyperJ, the table reflects current implementation status, rather than hypothesized capabilities. The compiler for other languages, such as Sina/St is not available for experimentation. In these cases, the table reflects the language designs as described in published works. The detailed survey of aspect-languages in Table 3.1 shows that the combination of features is indeed non-existent in current aspect-oriented programming languages and approaches.

AspectJ supported instance-level advising, but without first-class aspect instances through version 0.5. The AspectJ mailing list records discussions on the need for `addObject`, with examples presented by De Luca, van Gurp, and others. The current form provides constructs like `per this` and `per target` for associating aspect-instances and object-instances. These constructs are inadequate. First, `per this` and `per target` aspect instances are not under program control but are associated automatically with all instances of advised types, and each instance is associated with just one other object. AspectJ, to the best of author's knowledge, never supported both instance-level advising and first-class aspect instances. This is the combination needed for an adequate generalization to the instance level.



### 3.9 Summary

In this chapter, I showed that the unification of aspects and classes in a language design is possible. The unification brings conceptual unity to the programming model. In the new language design, aspect-like constructs support all of the capabilities of classes-notably *new*. Classes support aspect-oriented advising as a generalized alternative to traditional invocation and overriding. Supporting *new* eliminates the need for the irregular *per\** constructs from the language design. The anonymous, asymmetric, and non-orthogonal advice was replaced in favor of methods as the sole mechanism for procedural abstraction. The asymmetric and non-orthogonal aspects were eliminated and quantified binding surfaced as the central notion of aspect-oriented programming.

I presented the design and implementation of Eos, an extension of C# that embodies the unification. In Eos classpects are the basic unit of modularity. The classpects are first class in all respects: they can be instantiated, passed as parameters, returned as values, etc. Besides OO method invocation, classpects also offer AO method-join point binding as a generalized invocation mechanism.

## Chapter 4

### Challenge: Separation of Integration Concerns

---

Component integration creates value by automating the costly and error-prone task of imposing desired behavioral relationships on components manually. A problem is that straightforward software design techniques map integration requirements to scattered and tangled code, compromising modularity in ways that dramatically increase development and maintenance costs. This chapter presents the first data point in the overall evaluation of my approach. It validates the claim that the unified model improves the separation of integration concerns (SOIC). It compares the implementation of a simple but representative integration scenario using AspectJ and Eos.

This chapter is organized as follows. First, a representative example system and its simple object-oriented implementation are presented. The OO implementation demonstrates that component integration is indeed fragmented, scattered, and tangled with the component code. Secondly, I briefly describe mediator-based design style proposed by Sullivan and Notkin [97] to separate integration concerns and present the implementation of our representative system using this technique. This design style largely modularizes integration concerns, but leaves event declaration, announcements, and registration scattered and tangled, which suggests that aspect-oriented languages could be a good candidate to fully modularize this concern. Finally, I discuss and analyze aspect-oriented implementations of our example system using AspectJ and the unified model of Eos and compare these two versions to verify the claim that the unified model improves modularization of integration concerns.

## 4.1 A Running Example

This section presents a simple but representative example Sensor-Camera-Display system that will be used for the rest of this chapter. Figure 4.1 shows the example system. This system has components of three types: *sensors* (black boxes), *cameras* (grey boxes), and *displays* (white boxes). There are three sensors *s1*, *s2*, & *s3*, two cameras *c1* & *c2* and a display *d* in the system. These components are required to behave together such that whenever sensor *s1* detects a signal, the cameras *c1* & *c2* takes a picture and the display *d* is updated. These relationships coordinate the control, actions, and states of subsets of system components to satisfy overall system requirements. They are also called *behavioral relationships* [97].

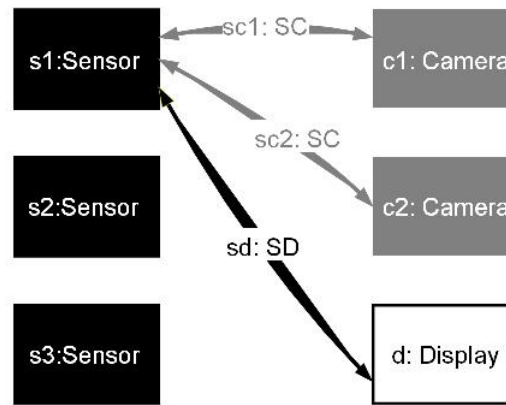


Figure 4.1: Sensor-Camera-Display System

In the rest of this work, I will use the terms *behavioral relationship* and *integration concern* interchangeably. The behavioral relationship *whenever a sensor detects a signal, the camera takes a picture* is also called the sensor-camera integration concern and the relationship *whenever sensor detects a signal, the display is updated* is also called the sensor-display integration concern.

Figure 4.1 shows the behavioral relationships between components as double-headed arrows. There are two types of relationships in the system. First, among sensors and cameras, and second, among sensors and displays. The relationships among sensors and cameras is labeled *SC* and shown as grey arrows. The relationships among sensors and displays is labeled *SD* and shown as black arrows. There are two instances of the first type of relationship *SC*, *sc1* between sensor *s1* and

camera *c1* and *sc2* between sensor *s1* and camera *c2*. There is only one instance of the second type of the relationship *SD*, *sd* between sensor *s1* and display *d*.

This example system is a representative of a broad class of systems called integrated systems. An integrated system is one in which logically unrelated objects, such as compilers, editors, debuggers, or any other kinds of discrete components must interact dynamically to meet system-level requirements (e.g., that the editor must automatically open the right file and scroll to the right line when the debugger encounters a breakpoint). The key ideas that this example demonstrates are, first, a component type may be integrated using more than one kind of integration relationships, and second, an instance of a component might participate in more than one instance of any given kind of relationship. The next section looks at a simple implementation of this system using object-oriented programming techniques.

## 4.2 Simple Object-Oriented Integration

In this implementation strategy, components are represented as instances of object-oriented classes. Figure 4.2 shows classes implementing *sensor*, *camera*, and *display* components. Realizing desired behavioral relationships between components requires *sensors*, *cameras*, and *display* to keep track of the other component instances with which they are integrated. One way to do so is to keep a list of other component instances with which a given component is integrated. As shown in Figure 4.2, *sensor* class keeps a list of *cameras* and *display* with which it is integrated and vice-versa. By keeping a list it will refer to other components, here *camera* and *display*.

Components to observe the desired behavior will need to invoke each other, which will be achieved by calling each other and thus there will be a name dependence between these components resulting in coupling and preventing their separate compilation, link, test, use, etc. For example, in Figure 4.2, *sensor s1* will invoke the *camera c1* and *camera c2* to take pictures. Similarly, *sensor s1* will invoke the *display d* to update itself.

Moreover, the integration code is also scattered and tangled across the component classes resulting in code complexity and non-modularity in design. Figure 4.3 shows the implementation of

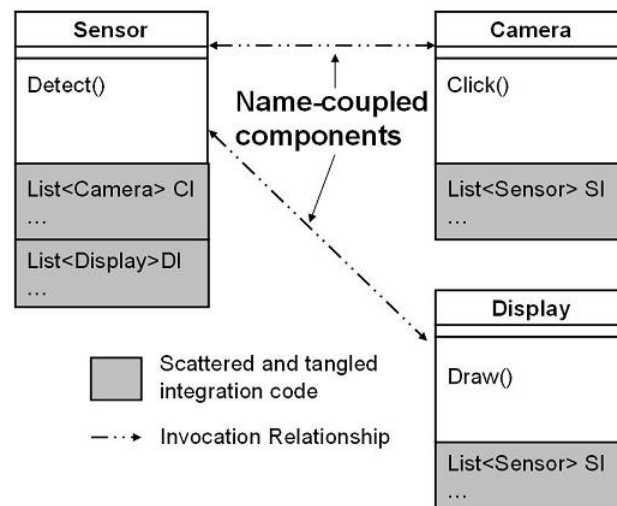


Figure 4.2: Class Diagram of the Sensor-Camera-Display System

the sensor system. The left column shows the *sensor* implementation as *Sensor* class. The right column shows the *camera* and *display* implementations as *Camera* and *Display* classes.

To implement the sensor–camera integration concern, the *Sensor* class is modified to keep a list *CameraList* to store the reference to the *camera* instances with which it is integrated. It is also modified to provide a method *AddCamera* to add a camera to this list and a method *InvokeCameraClick* that iterates through the list to invoke *Click* method on each stored *camera* reference. The original *Detect* method is modified to call *InvokeCameraClick* to simulate the desired relationship.

Similarly, to implement the sensor–display integration concern, the *Sensor* class is modified to keep a list *DisplayList* to store the reference to the *display* instances with which it is integrated. It is also modified to provide a method *AddDisplay* to add a display to this list and a method *InvokeDisplayUpdate* that iterates through the list to invoke *Update* method on each stored *display* reference. The original *Detect* method is modified again to call *InvokeDisplayUpdate* to simulate the desired behavioral relationship.

The *Camera* and *Display* classes are similarly modified to keep a list of sensors, provide a method to add a sensor to the list of references and to invoke *Detect* on each stored sensor instances, whenever the method *Click* and *Update* are called.

This implementation has three problems. First, the code that implements the sensor–camera

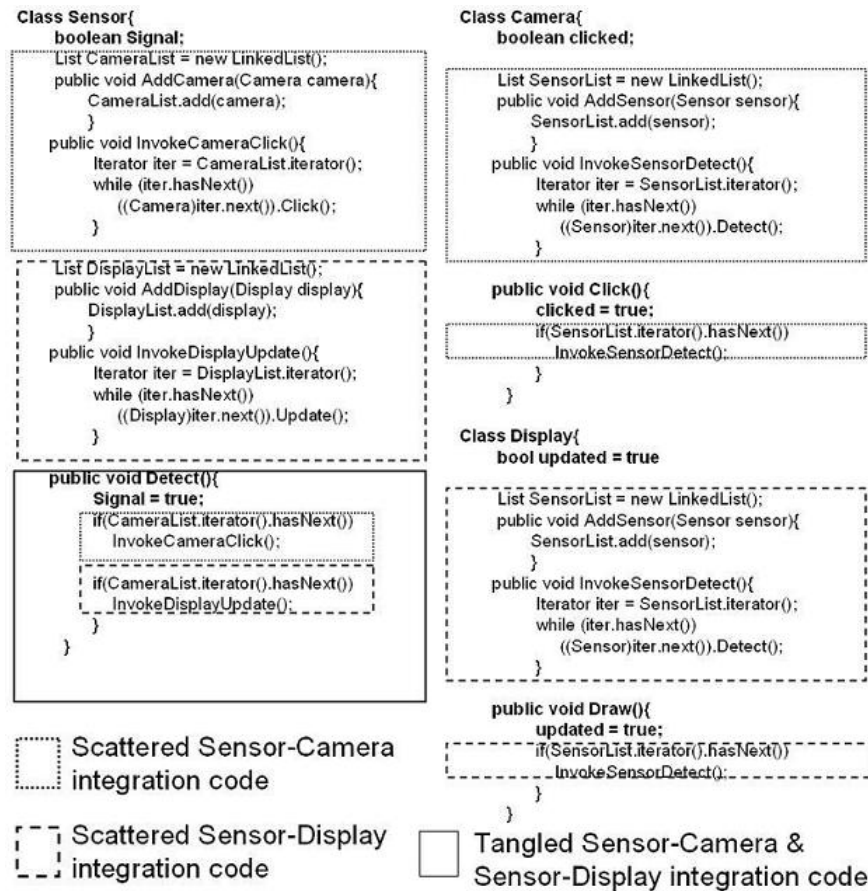


Figure 4.3: Scattered and Tangled Integration Concerns in OO Implementation

integration concern is scattered across the *sensor* and *camera* implementation. Second, the code that implements the sensor–display integration concern is scattered across the *sensor* and *display* implementation. Third, the code that implements both these concerns are tangled with each other and with the sensor concern in the method *Detect* of *Sensor* class. The fragmented, scattered, and tangled integration code increases the probability of faulty software due to inconsistencies and leads to costly time-consuming development making software evolution hard.

Sullivan and Notkin [97] proposed the Mediator–based design style to modularize fragmented, scattered, and tangled integration concerns. An implementation of the sensor–camera–display system using this design style is described and analyzed in the next section.

### 4.3 Mediator-Based Design

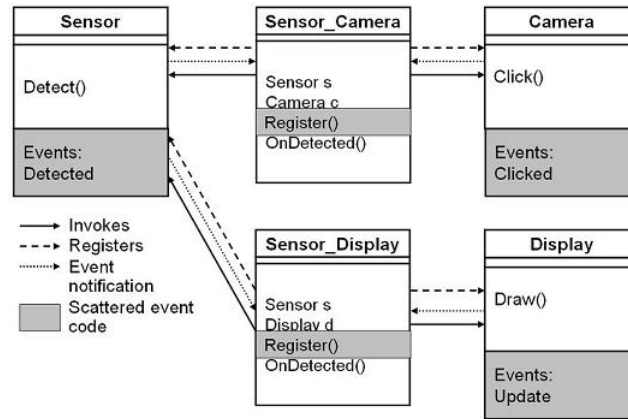


Figure 4.4: Class Diagram of Mediator-based Implementation

Mediator-based design style [96] [97] was introduced to ease the design and evolution of integrated systems. As previously described, the integrated systems are a very broad class of systems in which objects have to work together to achieve system objectives. The design style advocates structuring designs as behavioral entity relationships (ER) models and preserving the modular structure of these designs in programs. The key idea is to modularize behavioral relationships or integration concerns as separate mediator modules. The programming solution using the mediator approach involves two ideas.

First, the entities and relationships are now represented as abstract behavioral types (ABT's) instead of abstract data types (ADT's) [68], as in OO languages. The abstract behavioral type is an extension of the abstract data type (ADT). ADT's define abstractions in terms of method interfaces; ABT, in contrast, also include explicitly exported events. An ADT defines a class of objects in terms of operations that can be applied to an object of that class. An ABT defines a class of objects in terms of the operations that can be applied to an object of that class and in terms of events that such an object can announce. Announcing an event invokes (meta-level) operations implemented by other objects that have registered to receive such events from a given object. For example consider a simple ABT *Sensor*, which provides operation to convey detection of a signal. Apart from this operation, *Sensor* ABT also defines an event *Detected* that is announced whenever any

Sensor instance detects a signal.

Second, entities and relationships are mapped to corresponding ABT-based objects in a way that would avoid crosscutting implementations of behavioral relationships. For example, the behavioral integration relationship Sensor–Camera, will be modeled as a separate ABT, *SenCamMediator*. An object, *sc1*, of this ABT *SenCamMediator* will register with the events announced by sensor *s1* and camera *c1* and *c2*. When the sensor *s1* announces event *Detected*, the mediator *sc1* will invoke the *Click* method on *c1* and vice-versa. Similarly, the behavioral relationship *Sensor–Display* will be modeled as a separate ABT *SenDispMediator*. An object, *sd*, of this ABT *SenDispMediator* will register with the events announced by the sensor *s1* and display *d*. When the sensor *s1* announces event *Detected*, the mediator *sd* will invoke the *Update* method on *d* and vice-versa. The implementation of the integration concerns are thus largely modularized in the *SenCamMediator* and *SenDispMediator* ABTs.

At the source code level, in a mediator-based implementation of the sensor–camera–display system (see Figure 4.5), components are represented as instances of object-oriented classes *Sensor*, *Camera*, and *Display*. The classes representing components expose events as well as methods in their interfaces. Objects announce events to notify registered mediators of occurrences.

The behavioral relationships among component instances is represented as instances of separate classes, called mediators. Mediator instances function as observers that effect component integration upon notification. Here the *Sensor–Camera* integration concern is represented as the *SenCamMediator* mediator, and the *Sensor–Display* integration concern as the *SenDispMediator* mediator. A *Sensor–Camera* object would, maintain references, *s1* and *c*, to *Sensor* and *Camera* objects to be integrated; implement method, *SenCamMediator.OnDetect* that explicitly invokes *c.Click*; and, implement a function that registers *SenCamMediator.OnDetect* to be implicitly invoked by *s.Detected*. Similarly, a *SenDispMediator* object would, maintain references, *s* and *d*, to *Sensor* and *Display* objects to be integrated; implement method, *SenDispMediator.OnDetect* that explicitly invokes *d.Update*; and, implement a function that registers *SenDispMediator.OnDetect* to be implicitly invoked by *s.Detected*.

Now when the *Sensor* detects a signal, the mediators respond by clicking the *Camera* and up-



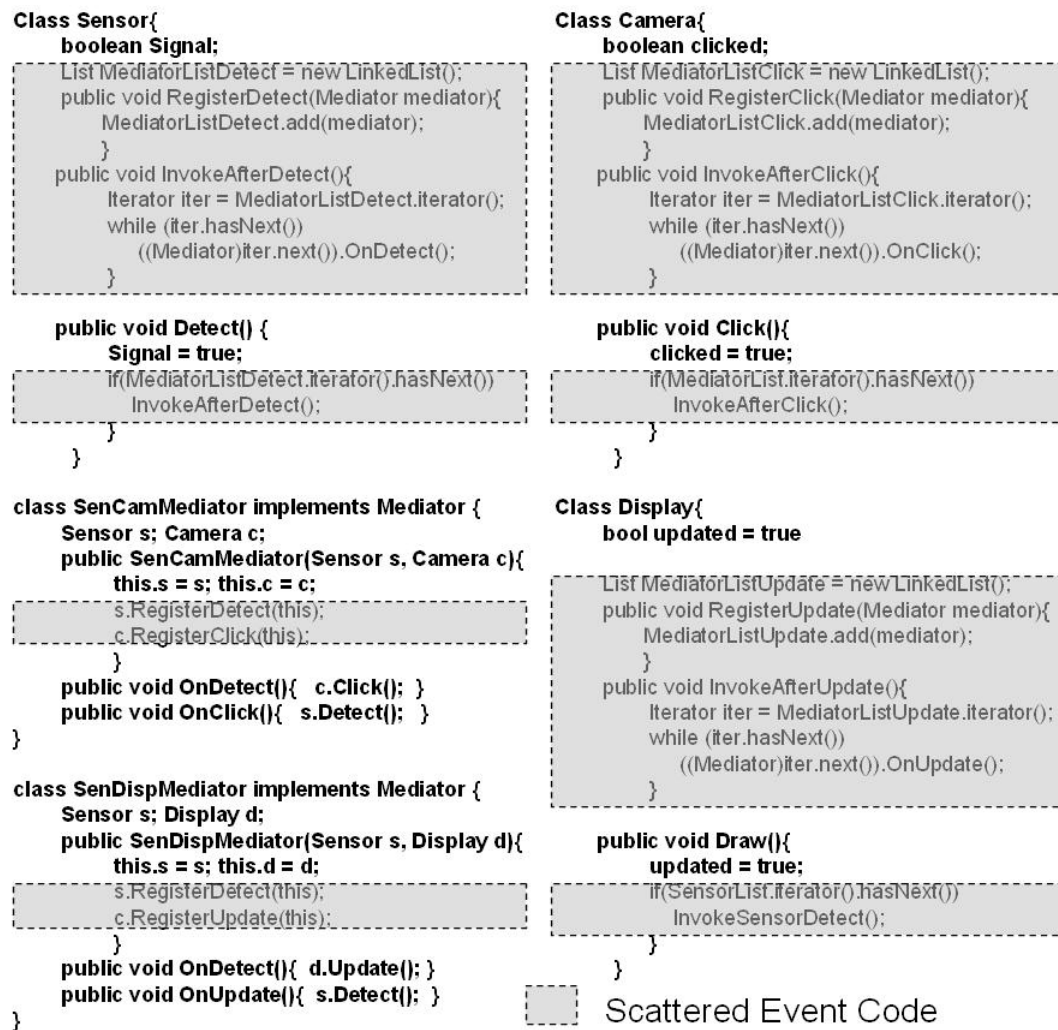


Figure 4.5: Scattered Event Code in Mediator-Based Implementation

dating the *Display* and vice versa. Yet the representations of the *Sensor*, *Camera*, and the *Display* are not statically tied to each other or to the mediator. The *Sensor*, *Camera*, and the *Display* components also remain uncomplicated by integration code and the behavioral relationships are largely modularized. Behavioral integration is thus reconciled with component independence, resulting in a modular structure that supports the independent implementation, testing, use, and evolution of the *Sensor*, *Camera*, and *Display* objects, and easier evolution of the now separately abstracted integrating behavioral relationship.

However, at least one problem remains. Components to be integrated have to declare and an-

nounce events sufficient to meet the needs of observing mediators. There is thus design dependence between components and mediators, even though there is no naming dependence: Component designers have to be aware of requirements for event notifications imposed by extant or admitted mediator types; and unanticipated changes in that set can require changes in mediated component types. Filman and Friedmann would argue that the components are not oblivious to mediators [35]. These changes may not even be possible to controlled source code or to components in a third party supplied library. Kiczales would argue that the behavioral relationships are not entirely modularized, insofar as the required event code has to be produced and maintained by the component developers [53]. Moreover, the event declaration, announcement and registration code is fragmented, scattered, and tangled with the component code as shown in the Figure 4.5. Thus, adding a new behavioral relationship and corresponding mediator class can require changes to multiple component classes—in the worst case, across a whole system.

Aspect-oriented programming (AOP) has emerged to address precisely the problem of scattered and tangled concerns. A crucial difference between the mediator-based design style and AOP is in the underlying event models. The mediator approach assumes explicit event declaration, announcement, and registration. On the other hand, in the aspect-oriented programming model, language semantics makes a subset of events in program execution available as implicit declared join points. Pointcut expressions are provided to register with a set of these implicit join points.

Aspects lead to an idea for improving mediator-based design: implement mediators as aspects, and use join points and pointcuts in place of explicit events. The solution for our example (See Figure 4.6) simultaneously preserves the name independence of the components being integrated by providing a mechanism that enables one component to invoke another without naming it and eliminates the need for fragmented, scattered, and tangled event concern. Sullivan et al. [92] investigated the mapping of mediators to aspects in AspectJ-like languages. They showed that mediators can be implemented as aspects in current AspectJ-like languages, with one caveat. Most current, major aspect languages, including AspectJ and HyperJ, suffer two shortcomings with respect to the mediator style. First, aspects are essentially global modules, rather than class-like constructs supporting first-class instances under program control. Second, aspects advise entire classes, not

object instances. The next section summarizes and significantly extends the results presented there.

## 4.4 Mediators as Aspects

In the Mediator-based design style, behavioral relationships among component instances are represented as instances of separate classes, informally called mediators. These mediator instances then register *selectively* with component instances to receive event notifications. From the description of the design style, two feature requirements emerge for any abstraction that is used to represent mediators. First, it should have instantiation capabilities. Second, it should be able to create associations selectively with component instances. These two requirements translate to two key features in the aspect-oriented world – the ability to arbitrarily instantiate aspects and the ability to selectively advise object instances.

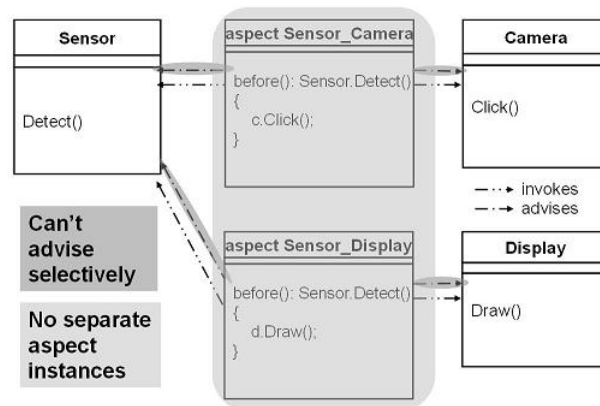


Figure 4.6: Class Diagram of Aspect-Oriented Implementation of Mediators

Unfortunately, most major current aspect-languages lack these features. A survey of a subset of aspect-oriented languages and approaches was presented in the previous chapter. The survey showed that the combination of features required for mediator-based design style is not present in any of the extant languages and approaches.

The mediator approach requires instance-level weaving and first-class aspect instances. There are languages having these features. The problem is that choosing them requires giving up rich join point models and expressive pointcut languages. Early versions of AspectJ had instance-level

weaving, and the current version has limited aspect instances but no support for instance-level weaving.)

Even major current aspect languages, including AspectJ and HyperJ, suffer from these two shortcomings with respect to the separation of integration concerns. First, aspects are essentially global modules, rather than class-like constructs supporting first-class instances under program control. Second, aspects advise entire classes, not object instances. Thus, there is, for most practical purposes, effectively one instance of each aspect type per system, and it essentially registers with events in all instances of each advised class.

By contrast, the mediator style requires that each type of behavioral relationship be represented as a mediator class, with class instances representing relationship instances. Moreover, the class instances register with the events of the instances to be integrated. Recall, for example, that in our sensor-camera-display system we have two instances of the `SenDispMediator`, one connecting the sensor to the first instance of the camera class, and the other, to the second instance. Each `SenDispMediator` instance registers with the detected event of the shared sensor instance, and with the clicked event of its particular camera instance.

Mediators cannot be mapped directly to aspect instances in AspectJ-like languages because aspects cannot be instantiated in a general way, nor can they selectively advise instances of other classes. Some aspect languages do support aspect instances or instance-level advising, but they generally have limited join point models (just calls and returns) and limited or no pointcut constructs. Most are in the decades-long tradition of message interception mechanisms. Work-arounds are possible in AspectJ, but even the best ones known incur unnecessary, non-negligible costs in performance or design complexity. The next section looks at two work-arounds.


## **4.5 Work-Arounds and Their Costs**

There are at least two basic work-arounds consistent with the use of aspects as behavioral relationship modules. In both cases, behavioral relationship types are mapped to aspect modules programmed to simulate first-class aspect instances and instance-level advising.

```

class Sensor{
    boolean Signal;
    public void Detect() {
        Signal = true;
    }
}
Class Camera{
    boolean clicked;
    public void Click(){
        clicked = true;
    }
}
class SenCamMediator implements Mediator {
    Sensor s; Camera c;
    public SenCamMediator(Sensor s, Camera c){
        this.s = s; this.c = c;
    }
    public void OnDetect(){ c.Click(); }
    public void OnClick(){ s.Detect(); }
}

```

 Emulation Code

```

aspect SenCamMediatorModule {
    static WeakHashMap Map;
    static {
        Map = new WeakHashMap();
    }
    public void Connect(Sensor s, Camera c){
        SenCamMediator sc =
            new SenCamMediator(s, c);
        Map.put(sc, s); Map.put(sc, c);
    }
    before():execution(void Sensor.Detect()){
        Sensor s = (Sensor)
            thisJoinPoint.getThis();
        SenCamMediator sc =
            (SenCamMediator)Map.get(s);
        if(sc!= null) sc.OnDetect();
    }
    before():execution(void Camera.Click()){
        Camera c = (Camera)
            thisJoinPoint.getThis();
        SenCamMediator sc =
            (SenCamMediator)Map.get(c);
        if(sc!= null) sc.OnClick();
    }
}

```

Figure 4.7: First Work-Around for Sensor–Camera Integration

To simulate instances, the aspect provides methods to create, delete, and manipulate instances implemented as records. The difference is in the simulation of instance-level advising. In the first approach shown in Figures 4.7 and 4.8, the aspect advises relevant join points of the classes whose instances are to be integrated. All instances invoke the aspect at each such join point. The aspect maintains tables recording the identities of objects to be treated as advised instances. When the aspect is invoked, it looks up the invoker see if it is such an instance. If so, the aspect delegates control to a simulated advice method, if not it returns immediately (See Figure 4.9).

This work-around works in the sense that it both modularizes the behavioral relationship code, data, and invokes relations, and relieves the developer of having to work with explicit events. However, the approach is less than ideal for several reasons.

First, it is awkward to have to simulate an object-oriented style in an otherwise object-oriented language. Second, such programs are actually harder to understand: the aspects read as advising classes, when, in fact the intent is to advise instances. Third, the approach adds unnecessary design complexity and thus cost and undependability with simulation implementations. Fourth, the

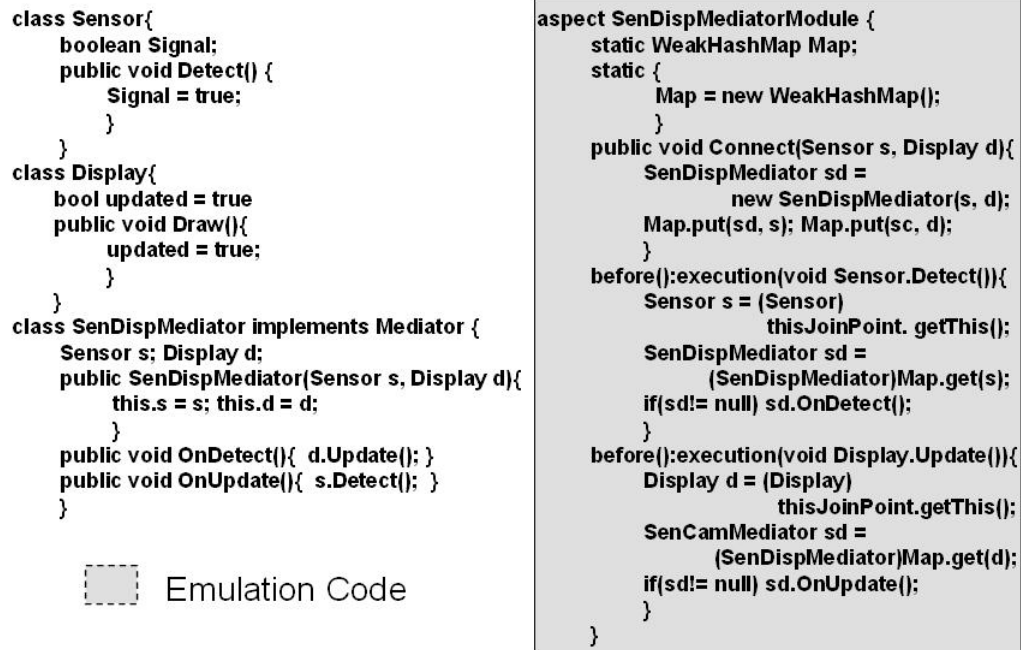


Figure 4.8: First Work-Around for Sensor–Display Integration

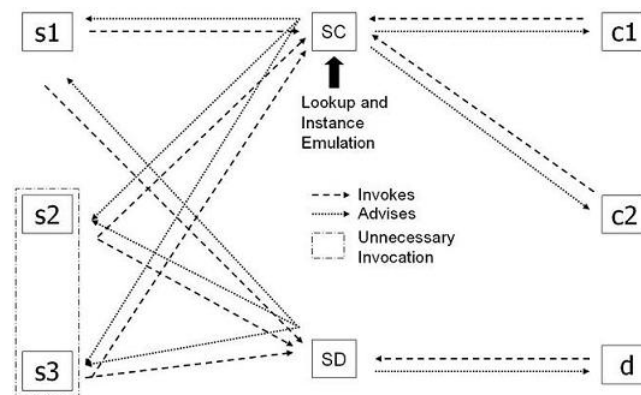


Figure 4.9: Object Diagram for Aspect-Oriented Implementation of Mediators

workaround adds runtime overhead in two dimensions. In particular, at each join point, each instance of an advised class has to invoke each advising aspect, if only to have it return upon failing the check for a simulated advised instance.

To understand the performance impact of the work-around, I did a study of the penalties associated with implementing mediators using AspectJ and HyperJ. Figure 4.10 presents the results. The X-axis shows the number of different mediators or aspects that advise the *Detected* event of

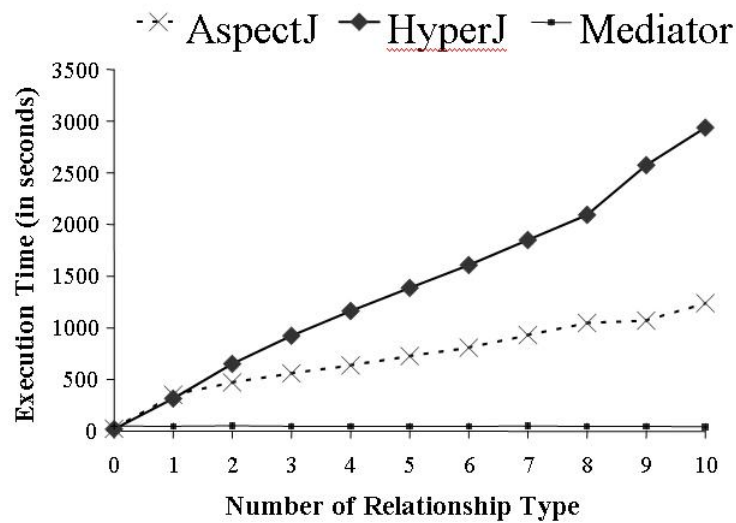


Figure 4.10: Performance Curve for AspectJ, HyperJ, and Mediator-Based Design

the simple *Sensor* type. The Y-axis shows the cost of invoking the *Detect* member function a fixed, large number of times. The cost per invocation clearly rises, as expected, with the number of aspects advising the type—and will do so for every instance of the type. The advice code in this case did nothing but return immediately. Having additional code in the advice will affect the study in two ways. First, the fraction (overhead-time / advice-execution-time) will decrease if the advice-execution time increases. Second, additional lookup code will have to be added to execute the advice only when required. Note that the results presented here represent the state of the implementation of AspectJ and HyperJ compilers and not of the language model. Also note that it might be possible for the compiler implementation to optimize some of these overheads but no compiler implementation as of this writing appears to provide these optimizations, nor is it entirely clear that effective optimization is even technically feasible.

By contrast, as the chart shows, a mediator style of integration using plain Java imposes a very small, constant overhead. Event code needs to be present in the *Sensor* object to notify registered mediators, but if none are registered, the cost is constant (a single *zero*) to check to see if any mediators are registered to be invoked. A slightly larger cost is paid only for mediators registered with specific object instances. It does not matter how many mediator types *might* advise a given object, but only how many mediator instances actually do so. It does not matter how many join

```

class Sensor{
    boolean Signal;
    public void Detect() {
        Signal = true;
    }
}
class Camera{
    boolean clicked;
    public void Click(){
        clicked = true;
    }
}
class SenCamMediator implements Mediator {
    Sensor s; Camera c;
    public SenCamMediator(Sensor s, Camera c){
        this.s = s; this.c = c;
        s.RegisterDetect(this);
        c.RegisterClick(this);
    }
    public void OnDetect(){ c.Click(); }
    public void OnClick(){ s.Detect(); }
}

```

Event Introduction Code

```

aspect SensorExtension{
    introduce in Sensor {
        List MediatorListDetect = new LinkedList();
        public void RegisterDetect(Mediator mediator){
            MediatorListDetect.add(mediator);
        }
        public void InvokeAfterDetect(){
            Iterator iter = MediatorListDetect.iterator();
            while (iter.hasNext())
                ((Mediator)iter.next()).OnDetect();
        }
    }
    after():execution(public void Sensor.Detect()){
        m.InvokeAfterDetect();
    }
}
aspect CameraExtension{
    introduce in Camera {
        List MediatorListClick = new LinkedList();
        public void RegisterClick(Mediator mediator){
            MediatorListClick.add(mediator);
        }
        public void InvokeAfterClick(){
            Iterator iter = MediatorListClick.iterator();
            while (iter.hasNext())
                ((Mediator)iter.next()).OnClick();
        }
    }
    after():execution(public void Sensor.Detect()){
        m.InvokeAfterClick();
    }
}

```

Figure 4.11: Second Work-Around for Sensor–Camera Integration

points are advisable.

There are situations in which the cost of this work-around might be unacceptable. An example would be the case of a mediator implemented as an aspect that has to respond to insertions on just one instance of a widely used, basic List class. It would be unreasonable, and is unscalable, for all clients of all instances of List to have to pay a price for one, isolated client.

The second work-around uses AspectJ-like introduction to extend the classes to be integrated with explicit event interfaces and code (See Figure 4.11 and Figure 4.12). The aspect also advises the join point at which the event is to be announced. The advice announces the event. The objects registered are not themselves aspects but ordinary mediators. The effect is to implement a traditional mediator design, but with explicit events modularized with the mediators that need them.

This work-around works, too, achieving integration without loss of modularity. It also avoids



```

class Sensor{
    boolean Signal;
    public void Detect() {
        Signal = true;
    }
}
class Display{
    bool updated = true
    public void Draw(){
        updated = true;
    }
}
class SenDispMediator implements Mediator {
    Sensor s; Display d;
    public SenDispMediator(Sensor s, Display d){
        this.s = s; this.d = d;
        s.RegisterDetect(this);
        d.RegisterUpdate(this);
    }
    public void OnDetect(){ d.Update(); }
    public void OnUpdate(){ s.Detect(); }
}

```

Event Introduction Code

```

aspect SensorExtension{
    introduce in Sensor {
        List MediatorListDetect = new LinkedList();
        public void RegisterDetect(Mediator mediator){
            MediatorListDetect.add(mediator);
        }
        public void InvokeAfterDetect(){
            Iterator iter = MediatorListDetect.iterator();
            while (iter.hasNext())
                ((Mediator)iter.next()).OnDetect();
        }
    }
    after():execution(public void Sensor.Detect()){
        m.InvokeAfterDetect();
    }
}
aspect DisplayExtension{
    introduce in Camera {
        List MediatorListUpdate = new LinkedList();
        public void RegisterUpdate(Mediator mediator){
            MediatorListUpdate.add(mediator);
        }
        public void InvokeAfterUpdate(){
            Iterator iter = MediatorListUpdate.iterator();
            while (iter.hasNext())
                ((Mediator)iter.next()).OnUpdate();
        }
    }
    after():execution(public void Sensor.Detect()){
        m.InvokeAfterUpdate();
    }
}

```

Figure 4.12: Second Work-Around for Sensor–Display Integration

the performance overhead of the first work-around by using essentially the same underlying event mechanisms that mediators use. Finally, it relieves the component developer of having to anticipate the events that mediators might need. In that sense, it arguably improves on the original mediator style.

Yet the approach has some problems. First, it does not really implement mediators as aspects at all, but only modularizes the explicit events that mediators need. It misses the point: we want to use join points rather than explicit events to invoke mediators. Having join points invoke advice that announces events that invoke mediators is at best a complex, relatively costly approach, using redundant mechanisms (events, join points). Second, if several mediators need to respond to the same event, each introduces its own event code and interface—bloating the code—rather than using the same event or join point.

## 4.6 The Conceptual Gap

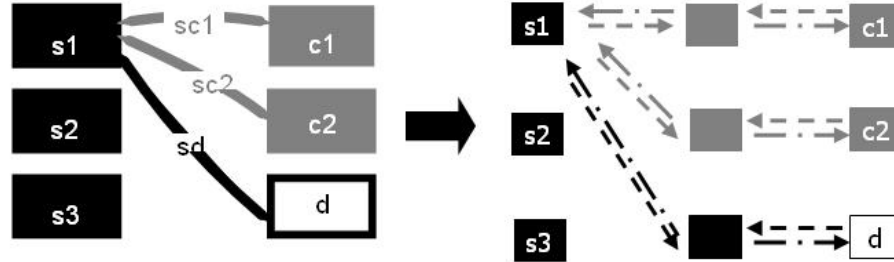


Figure 4.13: Ideal Object Diagram of the Sensor-Camera-Display System

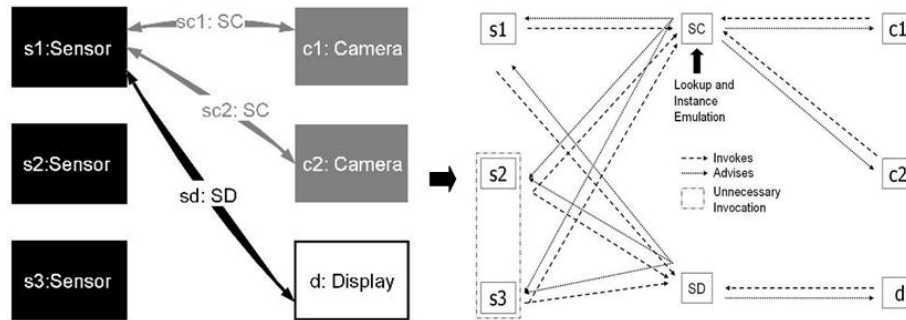


Figure 4.14: Actual Aspect-Oriented Object Diagram

In his famous 1968 letter to the editors of the *Communications of the ACM*, *Go To Statement Considered Harmful* [20], Edsger Dijkstra wrote:

We should do ... our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in test space) and the process (spread out in time) as trivial as possible.

Dijkstra's argument in the context of conceptual gap between code and runtime structure is equally applicable to the mapping between specification and run-time structure. In essence, Dijkstra argues that the run-time conceptual model of the system should in fact be very close to any static model of the system including its specification. MacLennan's *structure principle* [70] of *programming language design* similarly suggests that *the static structure of the program should correspond in a simple way to the dynamic structure of the corresponding computations*.

Figure 4.13 shows the ideal mapping and Figure 4.14 shows the actual AO-mapping from specification of the *Sensor-Camera-Display* system to its run-time structure. The conceptual model of the system at run-time is far removed from that at specification time in two significant ways: first, instances of the behavioral relationships in the specification are represented by one global aspect-instance at run-time, and second, associations between component instances and behavioral relationship instances are mangled in the run-time structure. The gap between these two models leads to unnecessarily hard to understand programs, as well as design time complexity. This conceptual gap can be traced back to the static module-based view of aspects.

To recall, Eos unifies aspects and classes in favor of a more compositional block of program design, namely classpects. Like classes, classpects are first class (i.e. they can be instantiated). Similar to aspects, classpects can advise using bindings. In addition to advising at the type-level, classpects can also selectively advise instances. Eos has a rich join point model and expressive pointcut language. These features promise to fill the gap in the realization of the mediator-based design style using aspect-oriented programming techniques. The next section shows the implementation of the *Sensor-Camera-Display* system in Eos.

## 4.7 Mediator as Classpects

Figure 4.15 shows the implementation of the *Sensor-Camera-Display* system using classpects. The implementations of *Sensor* (lines 1–6), *Camera* (lines 7–12), and, *Display* (lines 13–18) components are elided for presentation. The implementations of these concerns, however, remain separate from the implementation of the integration concerns and from each other. A modularization of component concerns is thus achieved in this implementation.

The lines 19–29 and lines 30–40 shows the implementation of *Sensor-Camera* and *Sensor-Display* integration concerns respectively. Line 19 and 30 declares the mediators as classpects using the *class* keyword. The *SenCamMediator* classpect stores references to a sensor and a camera (line 20) and the *SenDispMediator* classpect stores references to a sensor and a display (line 31). The *SenCamMediator* classpect provides a constructor (lines 21–24) that takes as argument a reference

```

1  class Sensor{
    ...
6  }
7  class Camera{
    ...
12 }
13 class Display{
    ...
18 }
19 class SenCamMediator{
20     Sensor s; Camera c;
21     public SenCamMediator(Sensor s, Camera c){
22         this.s = s; this.c = c;
23         addObject(s); addObject(c);
24     }
25     after():execution(public void Sensor.Detect()):Click();
26     public void Click(){c.Click();}
27     after():execution(public void Camera.Click()):Detect();
28     public void Detect(){s.Detect();}
29 }
30 class SenDispMediator{
31     Sensor s; Display d;
32     public SenDispMediator(Sensor s, Display d){
33         this.s = s; this.d = d;
34         addObject(s); addObject(d);
35     }
36     after():execution(public void Sensor.Detect()):Update();
37     public void Update(){d.Update();}
38     after():execution(public void Display.Update()):Detect();
39     public void Detect(){s.Detect();}
40 }

```

Figure 4.15: Classpect-Based Implementation of the SensorCamera–Display System

to a sensor and a reference to a camera that are to be mediated. The implicit method *addObject* is used on line 23 to selectively advise objects *s* and *c*. Similarly, the *SenDispMediator* classpect provides a constructor (lines 32–35) that takes as argument references to a sensor and a display to be mediated and selectively advises them using the implicit method *addObject* (line 34).

The classpect *SenCamMediator* declares two *non-static* bindings (lines 25 and 27). To recall, non-static bindings allow selective advising of object instances. The first binding binds the method *Click* (line 26) to execute after the execution of the method *Detect* in *Sensor*. The second binding binds the method *Detect* (line 28) to execute after the method *Click* in *Camera*. Similarly, the classpect *SenDispMediator* declares two bindings (lines 36 and 38). The first binding construct

```

1  public static void Main(string[] arg){
2
3      Sensor s1 = new Sensor();
4      Sensor s2 = new Sensor();
5      Sensor s3 = new Sensor();
6      Camera c1 = new Camera();
7      Camera c2 = new Camera();
8      Display d1 = new Display();
9
10     SenCamMediator sc1 = new SenCamMediator(s1,c1);
11     SenCamMediator sc2 = new SenCamMediator(s1, c2);
12     SenDispMediator sd1 = new SenDispMediator(s1,d1);
13
14     }

```

Figure 4.16: Modular Composition of Components and Connectors

binds the method *Update* (line 37) to execute after the execution of the method *Detect* in *Sensor*. The second binding construct binds the method *Detect* (line 39) to execute after the execution of the method *Update* in *Display*. The methods encapsulate the integration logic. The solution therefore achieves modularization of integration logic. The join point/pointcut model of aspect-oriented languages is used instead of explicit event declaration, announcement, and registration code modularizing the scattered and tangled event concern as well. This solution thus achieves a complete modularization of the integration concern.

Due to selective instance-level advising, the method *SenCamMediator.Click* is only invoked when the method *Detect* is called on instance *s* of *Sensor*. As a result there is no need to maintain a hash-table in the classpect to emulate selective instance-weaving resulting in a significant decrease in code complexity compared to the implementations using type-level aspects shown in Figure 4.7 and Figure 4.11. There are no unnecessary invocations so the implementation does not exhibit performance overheads such as those shown in Figure 4.10. On invocation the method *SenCamMediator.Click* calls the *Click* method on *Camera* instance *c* to complete the integration requirement. For the sake of brevity, the additional code to prevent recursive invocation is not shown in the figure.

The second method *SenCamMediator.Detect* similarly is only invoked when the method *Click* is called on instance *c*. On invocation it calls the method *Detect* on *Sensor* instance *s*. The classpect *SenDispMediator* also declares two bindings (lines 36 and 38). The first binding binds the method

*SenDispMediator.Update* to execute after the method *Sensor.Detect*. The method *SenDispMediator.Update* calls the *Update* method on *Display* instance *d*; the second, binds the method *SenDispMediator.Detect* to execute after *Update* in class *Display*. The method *SenDispMediator.Detect* calls the *Detect* method on *Sensor* instance *s*.

The *Sensor–Camera–Display* system is now composed naturally with component and connector instances as shown in Figure 4.16. The main routine constructs components instances and connector instances and connects component instances using connector instances. It creates three instances of *Sensor* component (lines 3–5), two instances of *Camera* component (lines 6–7), and an instance of *Display* component (line 8). It also creates two instances of the connector *SenCamMediator* (lines 10–11) and an instance of the connector *SenDispMediator* (line 12). To connect component instances, connector instances are supplied reference to them. For example, in Figure 4.16 line 10, connector *sc1* is supplied references to components *s1* and *c1*.

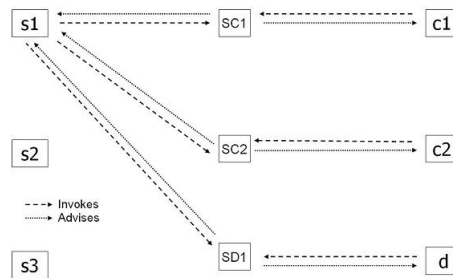


Figure 4.17: Object Diagram for Classpect Based Implementation

The run-time structure of the system shown in Figure 4.17 now mirrors the specification of the system, eliminating the conceptual gap between the static program structure and dynamic program structure. The behavioral relationships that were hidden until run-time by the instance-emulation and instance-level-weaving emulation code are now explicit in the design and implementation. In summary, the integration using classpects achieves a natural mapping from specification and design to implementation without resorting to unnecessary design complexity and performance overhead.

I measured the performance of selective instance-level bindings using the same benchmarks as used above to evaluate AspectJ, HyperJ and mediator-based designs. The comparison is complicated by the differences in host languages (Java, C#). I substituted type-level Eos bindings for

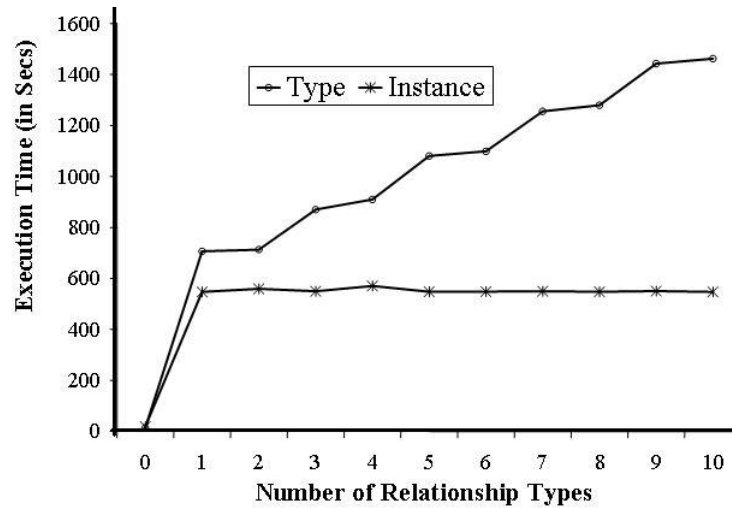


Figure 4.18: Performance Curve for Type-Level Aspects vs Classpects

AspectJ aspects for this comparison. Figure 4.18 shows that Eos type-level bindings replicate the degrading performance of AspectJ aspects, while Eos selective instance-level bindings indeed exhibit the constant overhead of mediator-based designs.

Figure 4.19 shows the side-by-side SeeSoft view of all implementations of the Sensor-Display integration concern discussed so far. The first, second, third, and fourth columns shows the mediator based implementation, the AspectJ first work-around, the AspectJ second work-around, and the Eos implementation respectively. The first box in all four columns shows the implementation of the Sensor concern. The second box shows the implementation of camera concern, and the third box shows the implementation of the sensor-camera integration concern. As can be observed, in the mediator-based style sensor and camera concerns are complicated by the event code. The first work-around in AspectJ replaces explicit events with implicit join points, but the additional design complexity to emulate aspect instantiation and selective weaving complicates the integration concern. The second work-around emulates mediator-based design, but without modifying the actual components, bringing back the event code. The Eos version, however, is free of both explicit event-related code and additional design complexity. The length of the columns also suggests that the Eos implementation is shorter compared to other implementations. Eos clearly achieved a significant improvement in the separation of integration concerns.

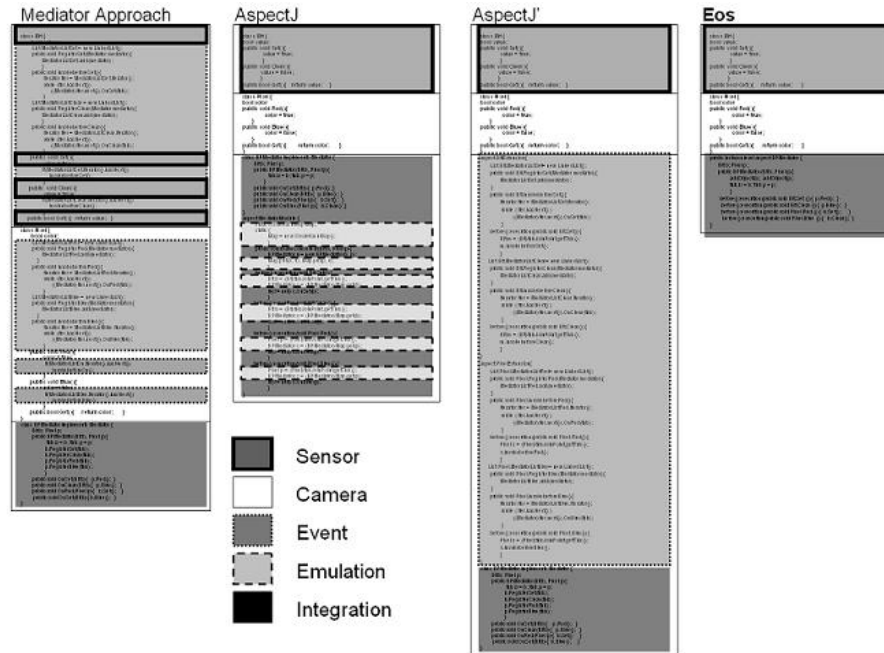


Figure 4.19: A Comparative View of Implementations

## 4.8 Summary

In this chapter, I validated the claim that the unification will improve the modularization of integration concern. I compared the implementation of a simple but representative system using aspects and classpects. The aspects based solution required work-arounds adding unnecessary design-time and run-time complexity. The run-time complexity makes the solution non-scalable in general. In addition, the aspects based solution introduces a conceptual gap between the specification, the implementation and the run-time structure of the system. The need for these work-arounds arises from the inability of aspects to selectively advise instances and the lack of general-purpose aspect instantiation. These shortcomings in the language design are due to the non-orthogonalities and asymmetries among aspects and classes. The classpects based solution on the other hand did not require any work-arounds. The implementation and the run-time structure now mirror the specification of the system clearly demonstrating an improvement in the separation of integration concerns.



## Chapter 5

### Challenge: Separation of Higher-Order Concerns

---

Aspect-oriented programming is a relatively new paradigm. The adoptability and interest in the current language models and approaches shows promise. The interest of the developer community makes it important for researchers to investigate the new technology by applying it to new application areas. In the last chapter, I described separation of integration concerns as a challenge problem for AspectJ-like languages. The experiments revealed an important shortcoming largely due to the commitment to have aspects as separate abstraction mechanism different from classes. The unified model of Eos significantly improved the modularization of integration concerns. This chapter provides the second data point of the overall evaluation of my approach. It supports the claim that the unified model improves the modularization of *higher-order crosscutting concerns*.

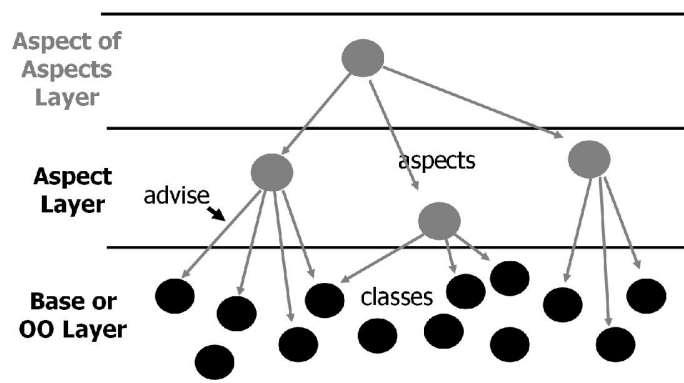


Figure 5.1: Adding a Layer on Top of Aspects

To recap, a concern is a dimension in which a design decision is made and it is crosscutting

if its realization using prevailing decomposition techniques leads to scattered and tangled code. A crosscutting concern is *higher-order* if its realization would be scattered and tangled across the implementations of other crosscutting concerns. For example, in Figure 5.1 the aspect in the top layer modularizes concerns that were scattered and tangled across the second layer of aspects. A simple *higher-order concern*, *Delayed Propagation* is introduced in the next section in the context of the *Sensor-Camera-Display* system discussed in Chapter 4. The rest of the chapter analyzes the attempts to modularize this simple *higher-order concern* using prevalent AO programming techniques <sup>1</sup>.

## 5.1 Delayed Propagation Concern

Let us consider an evolution scenario for the *Sensor-Display-Camera* system introduced in Chapter 4 that introduces a new feature described below. This feature is labeled *Delayed Propagation* in the Figure 5.2.

The system should provide means to switch on or switch off the delayed propagation. This feature when applied to behavioral relationships in the system enforces the following behavior:

- Turning on the delayed propagation causes caching of all updates required by the behavioral relationships, to coordinate the control, actions, and states of subsets of system components to satisfy overall system requirements, to be cached.
- Turning off the delayed propagation causes all cached updates to be flushed to restore the system state and for the subsequent updates to be handled immediately.

This feature is representative of caching feature commonly found in software systems. A common use-case is to avoid updating other views of a model, while the user is editing in one view to avoid screen flickering. For example, in Galileo [93] updates of the Microsoft Visio view could be avoided, when the fault-tree model is being edited in the Microsoft Word view. The next section de-

---

<sup>1</sup>Some of the contents of this chapter appeared in [84]

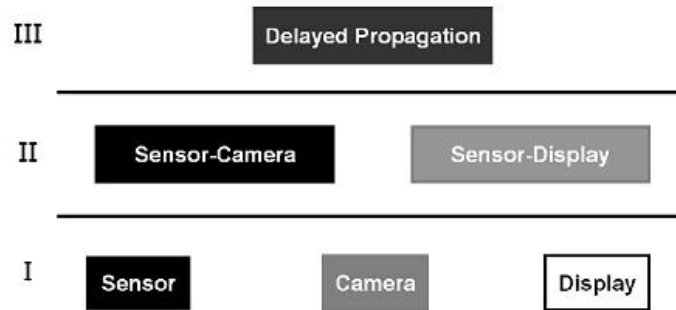


Figure 5.2: Delayed Propagation Concern

scribes a simple implementation of the delayed propagation concern in the *Sensor-Display-Camera* system.

## 5.2 Simple Realization of the Delayed Propagation Concern

Figure 5.3 shows a straightforward implementation of the *delayed propagation* concern in the *Sensor-Display-Camera* system in the AspectJ language model. The realization of this concern crosscuts the aspects *SenCamMediatorModule* and *SenDispMediatorModule* that in turn are themselves representations of crosscutting integration concerns. The delayed propagation is therefore a *higher-order* crosscutting concern.

To implement delayed propagation, the following three changes are made to each aspect representing a mediator. First, a Boolean flag (line 2), and a property (lines 5-15) to get/set the flag is added to each aspect to store and change the state of delayed propagation. Second, each advice is modified to handle the delayed propagation (lines 28-30, 39-41). If the delayed propagation flag is *false*, the advice works normally. If it is *true*, the advice caches the update and skips the required update.

Even though this implementation strategy is simple, it results in the scattered and tangled implementation of the delayed propagation concern. The integration logic that was well modularized before in advice constructs is now coupled with delayed propagation concern. The implementation also does not provide a single interface to switch delay propagation on/off; instead, delayed

<pre> 1 aspect SenCamMediatorModule { 2     bool delay; 3     int detectCount = 0; 4     int clickCount = 0; 5     public bool Delay{ 6         get{ return delay;} 7         set{ 8             delay = value; 9             if(value==false){ 10                 for(int i=detectCount; i&gt;0; i++) 11                     // Flush cached detects; 12                 for(int j=clickCount; j&gt;0; j++) 13                     //Flush cached clicks; 14             } 15         } 16     } 17 18     ... 19 20     after():execution(void Sensor.Detect()){ 21         if(delay) 22             detectCount++; 23         else { 24 25         } 26     } 27 28     after():execution(void Camera.Click()){ 29         if(delay) 30             clickCount++; 31         else { 32 33         } 34     } 35 36     ... 37 38     after():execution(void Display.Update()){ 39         if(delay) 40             updateCount++; 41         else { 42 43         } 44     } 45 46     ... 47 48     } 49 } </pre>	<pre> 1 aspect SenDispMediatorModule { 2     bool delay; 3     int detectCount = 0; 4     int updateCount = 0; 5     public bool Delay{ 6         get{ return delay;} 7         set{ 8             delay = value; 9             if(value==false){ 10                 for(int i=detectCount; i&gt;0; i++) 11                     // Flush cached detects; 12                 for(int j=updateCount; j&gt;0; j++) 13                     // Flush cached updates; 14             } 15         } 16     } 17 18     ... 19 20     after():execution(void Sensor.Detect()){ 21         if(delay) 22             detectCount++; 23         else { 24 25         } 26     } 27 28     after():execution(void Display.Update()){ 29         if(delay) 30             updateCount++; 31         else { 32 33         } 34     } 35 36     ... 37 38     after():execution(void Display.Update()){ 39         if(delay) 40             updateCount++; 41         else { 42 43         } 44     } 45 46     ... 47 48     } 49 } </pre>
---	---

### Scattered Implementation of Delayed Propagation

Figure 5.3: Scattered Implementation of Delay Propagation Concern

propagation has to be toggled for each mediator. Aspect-oriented programming aims to solve precisely this problem. The next section analyzes how delayed propagation can be modularized using aspects.

## 5.3 Modularizing The Delayed Propagation Concern

An alternative strategy is to implement delayed propagation as an aspect. The aspect will provide a single interface to turn delayed propagation on and off. It will advise the advice constructs in the *SenCamMediatorModule* and *SenDispMediatorModule*. It will use around advice to override these advice constructs. If we recall, an around advice executes instead of the join point it is advising. It can trigger the execution of the original join point by calling a special form *proceed*. In this case,

```

1 aspect SenCamMediatorModule {
2     static WeakHashMap Map;
3
4     ...
5
6     11 before():execution(void Sensor.Detect()){
7         Sensor s = (Sensor)
8         thisJoinPoint.getThis();
9         SenCamMediator sc =
10         (SenCamMediator)Map.get(s);
11         if(sc != null) sc.OnDetect();
12     }
13
14     18 before():execution(void Camera.Click()){
15         Camera c = (Camera)
16         thisJoinPoint.getThis();
17         SenCamMediator sc =
18         (SenCamMediator)Map.get(c);
19         if(sc != null) sc.OnClick();
20     }
21
22     26 aspect SenDispMediatorModule {
23         static WeakHashMap Map;
24
25         ...
26
27         36 before():execution(void Sensor.Detect()){
28             Sensor s = (Sensor)
29             thisJoinPoint.getThis();
30             SenDispMediator sd =
31             (SenDispMediator)Map.get(s);
32             if(sd != null) sd.OnDetect();
33         }
34
35         43 before():execution(void Display.Update()){
36             Display d = (Display)
37             thisJoinPoint.getThis();
38             SenDispMediator sd =
39             (SenDispMediator)Map.get(d);
40             if(sd != null) sd.OnUpdate();
41         }
42     }
43
44     51 aspect Delay {
45         52
46         53 void around(SenCamMediator sc): ??? & this(sc){
47             54
48             55 if(delay) /* cache updates */; else proceed();
49             56 }
50
51         58 void around(SenCamMediator sc): ??? & this(sc){
52             59
53             60 if(delay) /* cache updates */; else proceed();
54             61 }
55
56         62 void around(SenDispMediator sd): ??? & this(sd){
57             63
58             64 if(delay) /* cache updates */; else proceed();
59             65 }
60
61         66 void around(SenDispMediator sd): ??? & this(sd){
62             67
63             68 if(delay) /* cache updates */; else proceed();
64             69 }
65
66         70 bool delay;
67         71 public bool Get(){ return delay;}
68         72 public void Set(bool value){
69             73 delay = value;
70             74 if(value == false) /* Propagate updates */
71             75 }
72     }
73
74     76 }
75
76     77 }
78
79     79 }

```

Figure 5.4: Problems With Modularization of Delayed Propagation Concern

our join points of interest are the advice constructs in the aspects *SenCamMediatorModule* and *SenDispMediatorModule*. The solution then is to call *proceed* if delay is off, so that the updates in the mediators proceed as usual and to cache the updates and omit original join point execution if delay is on. This solution approach promises to modularize the scattered and tangled delayed concern. It also satisfies the requirement to provide a single interface to turn delayed propagation on and off.

The alternative approach solves the problems of the simple solution. The code for *delayed propagation* is now a separate, modularized, and reusable aspect. To add or remove this feature from the system, we just need to add or remove the aspect. The component code is now independent of the integration code. The alternative solution looks promising but unfortunately; it cannot always be easily realized using the AspectJ-like language model. In this language model this structure is

not achievable owing to a key restriction—while aspects can advise classes in many ways, they can advise other aspects only in restricted ways.

Figure 5.4 shows attempts to realize this solution approach. An aspect *Delay* (lines 51–79) represents the delayed propagation concern. The aspect provides interface to turn delayed propagation on or off (lines 73–78). It also provides four around advice constructs to override the advice constructs in the aspect *SemCamMediatorModule* and *SenDispMediatorModule*. The around advice constructs on lines 53–56, 58–61, 63–66, and 68–71 will override the advice constructs on lines 11–17, 18–24, 36–42, and 23–49 respectively. The intended overriding behavior is shown by dotted lines in Figure 5.4.

The problem with the solution strategy is that aspects can advise methods in other aspects, but they can advise *advice* in other aspects in only limited ways. In the current model, individual advice bodies are anonymous, so pointcut expressions cannot select a subset of them based on their names. The pointcut designator *adviceexecution* selects all advice-execution join points in the program. One can narrow down this selection by composing the *adviceexecution* pointcut with *within* pointcut. For example, the pointcut expression *adviceexecution() & within(SenCamMediatorModule)* selects execution of every advice in the aspect *SenCamMediatorModule* (Figure 5.4). To implement delayed propagation for *Sensor–Camera* and *Sensor–Display* integration concerns, addressing each advice (lines 11–17, 18–24, 36–42, and 23–49) in the aspect *SenCamMediatorModule* and *SenDispMediatorModule* independently is necessary. In the current model, it is not possible to make such fine-grained selection.

A workaround that springs to mind is to have advice delegate to corresponding aspect methods and to advise these methods. The need for such a hack is evidence that there is something wrong in the current design; and the work-around is unsatisfactory, in general.

First, it requires either ubiquitous up-front use of the delegation pattern, or—contrary to the central purpose of aspect-orientation—that scattered changes be made to aspect modules whenever any of their advice bodies become subject to advising. Both approaches require source code, which is not always available. Second, delegating is not entirely straightforward. Advice bodies have to be analyzed to determine whether or not they use implicitly declared reflective information, such as

*thisJoinPoint* or implicit methods, namely *proceed*.

```

1 // Original advice
2     void around(): <pointcut> {
3         if(ShallProceed)proceed();
4     }
5 // Workaround applied to advice above
6     void around():<pointcut> {
7         OriginalAdviceCodeInMethod();
8     }
9     void OriginalAdviceCodeInMethod(){
10        if(ShallProceed) proceed();
11    }

```

Figure 5.5: Work-Around Applied to Around Advice

Passing all such parameters to the delegate methods incurs additional design—and run-time costs and the risks of error. The situation is even more complicated in cases of around advice bodies, which execute instead of the original join point and which can call the original join point using *proceed*. Figure 5.5 presents an example (lines 1–4): if *ShallProceed* is true, the original join point is invoked. Applying the workaround results in the *proceed* call being moved to a delegatee (lines 9–11). *Proceed* is allowed only in advice bodies, not in methods, in the current languages. *Proceed* will thus have to be passed from the advice body to the delegatee as a closure, perhaps using the worker object pattern of Laddad [62]. The work-around is both complicated and incurs the need for scattered changes, undermining the purpose of aspects.

The unified model replaces non-orthogonal and asymmetric aspect & class and advice & method by symmetric classpects, bindings, and methods. I claimed that this reorganization of language constructs improves the compositionality of the resulting language model under advising as an invocation mechanism and improves the modularization of higher-order concerns. To validate the claims, the next section presents the classpect-based implementation of the delayed propagation concern in the Sensor-Camera-Display system.

## 5.4 Delayed Propagation as Classpect

Earlier in the chapter, I showed the scattered implementation of the delayed propagation concern (Figure 5.3) and ideal AO solution (Figure 5.4) using AspectJ-like languages that required work-

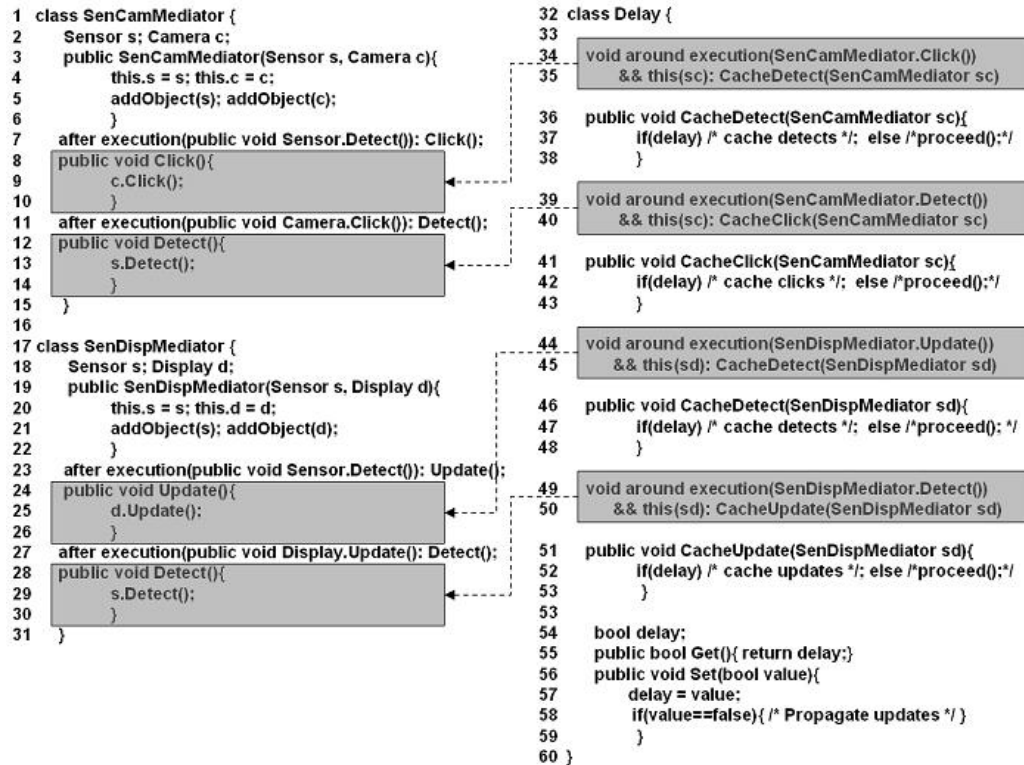


Figure 5.6: The Delayed Propagation Concern as Classpects

around to modularize this concern completely. These work-arounds in general are non-trivial and require crosscutting changes in the code base compromising the basic philosophy of AOP. In the first solution, the resulting implementation was scattered and tangled with the implementation of *Sensor–Camera* and *Sensor–Display* mediators. The second solution offered a complete modularization of the *delayed propagation* concern, but the inability to select appropriate *advice–execution* join points rendered it unusable without scattered use of the delegation pattern.

The problem with the second solution was that the advice constructs in the *SenCamMediatorModule* (lines 11–17, and 18–24,) and *SenDispMediatorModule* (lines 36–42, and 23–49) are not named. As a result, writing the around advice constructs in the aspect *Delay* required crosscutting usage of the delegation pattern. I showed previously that the use of the delegation pattern requires analysis of the advice and unnecessary and scattered changes to the base code. The implementation of a simple concern on top of the aspect–layer turns out to be more difficult than one would expect.

A key advance that the classpect-based implementation of *SenCamMediator* and *SenDispMedi-*



ator makes over aspect-based implementation is that the integration logic is now modularized in a named construct, method. Figure 5.6 shows the representation of integration logic as normal methods. These methods can be selected individually using existing pointcut designators and patterns. The re-factoring of advice construct into binding and method thus solves the problems encountered by the second solution, making the complete modularization of the *delayed propagation* concern feasible without resorting to work-arounds.

Figure 5.6 shows the *delayed propagation* concern as a classpect. The *Delay* classpect declares four methods (lines 36–38, 41–42, 46–48, and 51–53) to cache the sensor detects, camera clicks, and display updates. It binds these methods to the execution of methods in the *SenCamMediator* and *SenDispMediator*. For example, the first binding (lines 34–35) binds the method *CacheDetect* to execute around the execution of method *SenCamMediator.Click*. The effect of this binding is that whenever *SenCamMediator.Click* is called, instead of executing its body control is transferred to the method *CacheDetect*. The method *CacheDetect* either caches the call or allows it to proceed as usual.

Note that the method *SenCamMediator.Click* is called whenever a *Sensor* instance detects a signal, to propagate the change to the *Camera* instance. So caching its execution is equivalent to caching the propagation of the Propagation of *sensor–camera* integration concern, precisely the requirement of the *delayed propagation* concern. The solution shows that classpects were able to improve the modularization of the higher-order concern without the need for workarounds. Moreover, the classpect *Delay* remains amenable to be advised by another layer of classpects, demonstrating the full compositionality of the unified language model.

## 5.5 Summary

In this chapter, I validate the claim that the unified model improves the modularization of higher-order concerns. I demonstrated and compared the implementation of a simple but representative concern using aspects and classpects. The aspect based implementation required crosscutting use of delegation pattern to expose the right set of join points. The need for such work-around

also demonstrated that in the current language model aspect-aspect compositionality is restrictive. The classpect-based implementation was able to modularize the delayed propagation concern completely, showing improvement in the modularization of higher-order concerns and the compositionality of the language model. Getting rid of aspect and advice as a separate abstraction mechanism and including bindings thus opens up new architectural possibilities using advising as an invocation mechanism.

## Chapter 6

### On The Expressive Power of Classpects

---

Until now I have shown that the design decision to support two related but distinct constructs: aspects & classes has consequences that are felt in the form of unnecessary constraints on designers, surprising non-compositionality properties, undue design complexity in resulting programs, and unavoidable performance penalties. I also proposed a unified model, embodied by the Eos language, which replaces non-orthogonal and asymmetric abstractions namely, aspect & class and advice & method, with orthogonal classpect, binding, and method. Based on Eos programming model's ability to improve the modularization of integration and higher-order concerns, I made informal claims about its expressiveness.

This chapter presents the third data point in the overall evaluation of my approach by validating the claim that the unified model is more expressive than the AspectJ language model. The validation is provided by a sound formal argument based on Felleisen's notion of macro-eliminable programming language extensions [31]. Many languages and approaches have been compared using Felleisen's notion of expressive power of programming languages. For example, Brogi et al. compare the expressive power of three classes of coordination models based on shared dataspaces [7]. In particular, I prove that the constructs in AspectJ language model are macro-eliminable with respect to the Eos language model, but the vice-versa is not true. Informally, I prove that the unified language model retains the expressive power of AspectJ language model in all dimensions. In some dimensions, the unified language model is able to express some concerns using just local

transformations that require global transformation in the AspectJ language model.

## 6.1 On the Expressive Power of Programming Languages

In response to the lack of formal framework for specifying and verifying statements about the expressiveness of programming language, Felleisen in his work, *On the expressive power of programming languages* [31] proposed a formal notion of expressiveness and showed that his formal framework captures many informal ideas on expressiveness. His approach adopts the ideas from the formal systems to programming languages. In particular, he adopts the notion of expressible or eliminable systems proposed by Kleene [57] and extensions of Kleene's notion by Troelstra [102].

In order to understand the adaptation in the programming language context, let us first look into the notion of expressible systems in the logic/formal systems paradigm. The next subsection presents essential definitions. These definitions are originally from Troelstra's work. Felleisen adopted them to the programming language context.

### 6.1.1 Formal Systems

A formal system  $L$  is a 3-tuple of sets  $\{\text{Expressions}(L), \text{Formula}(L), \text{Theorems}(L)\}$  such that  $\text{Expressions}(L) \supseteq \text{Formula}(L) \supseteq \text{Theorems}(L)$ . Expressions either are terms, or generated by applying logical and non-logical operators over other expressions. A formula is a recursive subset of the set of expressions and satisfies well-formedness criteria. Theorems are formula defined to be true in the formal system.

*Conservative Extension:* A conservative extension of a formal system  $L'$  is a formal system  $L$  if the following relationships hold between the expressions, formula and theorems in these two formal systems.

1.  $\text{Expressions}(L) \supseteq \text{Expressions}(L')$
2.  $\text{Formula}(L) \cap \text{Expressions}(L') = \text{Formula}(L')$
3.  $\text{Theorems}(L) \cap \text{Expressions}(L') = \text{Theorems}(L')$

The first relationship denotes that the conservative extension  $L$  contains all the expressions in  $L'$  and possibly more. The richer sets of operations in the extension generate these additional expressions. The next two relationships express the fact that formula and theorems in the original formal system  $L'$  can be expressed in the extension  $L$ .

*Definitional Extension:* A definitional extension  $L$  of  $L'$ , is a conservative extension for which there exists a relation  $R : Expressions(L) \rightarrow Expressions(L')$  such that:

1.  $\forall f \in Formula(L), R(f) \in Formula(L')$
2.  $\forall f \in Formula(L'), R(f) = f$
3.  $R$  is homomorphic in all logical operators <sup>1</sup>.
4.  $L \vdash t \iff L' \vdash R(t)$
5.  $L \vdash t \leftrightarrow R(t)$

The existence of the relation  $R$  that satisfies the properties described above makes the symbols that generate additional expressions in the extension  $L$  eliminable. Felleisen adopted this notion from the formal systems context to the programming language world to assess their relative expressiveness. The next subsection presents the definition of programming languages and their definitional and conservative extension from his work.

### 6.1.2 Programming Languages

*Programming Language:* A programming language is modeled as a 3-tuple of sets  $\{Phrases(L), Programs(L), Semantics(L)\}$  where these sets are defined as follows:

- *Phrases(L)* is a set of abstract syntax trees freely generated from function symbols  $F, F_1, \dots$  with arities  $a, a_1, \dots$
- *Programs(L)* is non-empty recursive subset of the *Phrases(L)*.
- *Semantics, eval<sub>L</sub>*, is a recursively enumerable predicate on *Programs(L)*. The program terminates iff *eval<sub>L</sub>* holds for that program.

---

<sup>1</sup>A mapping  $R$  from formal system  $L$  to  $L'$  is homomorphic with respect to some hypothetical binary logical operator  $\circ$  if  $a \in Expressions(L), b \in Expressions(L), R(a \circ b) = o \iff R(a) \circ R(b)$ .

*Conservative Extension (Restriction) of a Programming Language:* Let us assume a programming language  $L'$  is extended with additional symbols  $\{F_1, \dots, F_n, \dots\}$  to yield an extension  $L$ . This extension is represented as  $L = L' \setminus \{F_1, \dots, F_n, \dots\}$ , and is a conservative extension if:

1. constructors of  $L$  are constructors of  $L'$  plus additional  $\{F_1, \dots, F_n, \dots\}$ .
2.  $Phrases(L') \subset Phrases(L)$  with no constructs in  $\{F_1, \dots, F_n, \dots\}$ .
3.  $Programs(L') \subset Programs(L)$  with no constructs in  $\{F_1, \dots, F_n, \dots\}$ .
4.  $Semantics(L')$  is a restriction of  $Semantics(L)$ .

*Eliminable Programming Constructs:* Let  $L = L' \setminus \{F_1, \dots, F_n, \dots\}$  be a conservative extension of  $L'$ . The new constructs  $\{F_1, \dots, F_n, \dots\}$  in the language  $L$  are *eliminable* if there exists a mapping  $R : Phrases(L) \rightarrow Phrases(L')$  such that:

1.  $\forall P \in Programs(L), R(P) \in Programs(L')$
2.  $R$  is homomorphic in all constructs of  $L'$ , i.e.  $R(F(e_1, \dots, e_a)) = F(R(e_1), \dots, R(e_a))$  for all constructs  $F$  of  $L'$ .
3.  $\forall P \in Programs(L), eval_L(e)$  is true  $\leftrightarrow eval_{L'}(R(P))$  is true.
4.  $Programs(L') \subset Programs(L)$  with no constructs in  $\{F_1, \dots, F_n, \dots\}$ .
5.  $\forall P \in Programs(L'), eval_{L'}(P)$  is true  $\leftrightarrow eval_L(P)$  is true.

*Macro Eliminable Extension of a programming language:* Let  $L = L' \setminus \{F_1, \dots, F_n, \dots\}$  be a conservative extension of  $L'$ , i.e.  $L'$  is conservative restriction of  $L$ . The constructs  $\{F_1, \dots, F_n, \dots\}$  are *macro eliminable* if they are eliminable and if eliminating mapping  $R$  satisfies the following condition as well.

- $\forall F \in \{F_1, \dots, F_n, \dots\} \exists \alpha - \text{ary syntactic abstraction, } A, \text{ over } L' \text{ such that } R(F(e_1, \dots, e_\alpha)) = A(R(e_1), \dots, R(e_\alpha)).$

In other words, the original language can locally express each additional construct in the conservative extension. If such an abstraction does not exist the language extension is not macro eliminable, therefore it is more expressive. Felleisen described this expressiveness relationship as follows:

Given two universal programming languages that only differ by a set of programming constructs,  $c_1, c_2, \dots, c_n$ , the relation holds if the additional constructs make the larger language more expressive than the smaller one. Here *more expressive* means that the translation of a program with occurrences of one of the constructs  $c_i$  to the smaller language requires a global reorganization of the entire program.

In the next section, I apply this notion of expressive power to compare the unified model proposed in this dissertation and the model of the family of the AspectJ-like languages [52]. For the rest of this chapter, a *translation* from a language  $L$  to another language  $L'$  is defined as a meaning-preserving syntactically defined map that transforms programs from  $L$  to  $L'$  [86]. A translation  $T$  is *textually local* if a syntactic abstraction  $A$  exists such that  $\forall C_i, T(C_i(e_1, e_2, \dots, e_m)) = A(T(e_1), T(e_2), \dots, T(e_m))$ . A translation  $T$  is *global* if it is not textually local.

## 6.2 Expressive Power of the Unified Model

The unified model ( $L_{Eos}$ ) proposed in this work differs from the AspectJ model ( $L_{AJ}$ ) by the constructs  $\{advice, aspect, binding, classpect, class\}$  as shown in the Table 6.1. The symbol  $X$  denotes the presence of a construct in the model. For brevity, the table only shows the constructs of interest. The row *Other* denotes other constructs like *events*, *indexers*, *inter-type declaration* etc. that are present in both language model.

As depicted in Table 6.1,  $L_{AJ}$  contains advice, aspect, and class that are eliminated in  $L_{Eos}$  in favor of classpect and binding. The rest of the constructs are present in both models, so they will not be discussed in the rest of this section, unless relevant. Let us construct a third model  $L_{Combined}$ , where constructs in  $L_{Combined} = \text{constructs in } L_{Eos} \cup \text{constructs in } L_{AJ}$ . The language  $L_{Combined}$  can be represented as  $L_{Combined} = L_{Eos} \setminus \{aspect, advice, class\}$  or  $L_{Combined} = L_{AJ} \setminus \{classpect, binding\}$ ,

Table 6.1: Difference in Constructs

AspectJ Model:  $L_{AJ}$ Unified Model:  $L_{Eos}$ Combined Model:  $L_{Combined}$ 

Construct	$L_{AJ}$	$L_{Eos}$	$L_{Combined}$
Advice	X		X
Aspect	X		X
Binding		X	X
Classpect		X	X
Class	X		X
Constructor	X	X	X
Field	X	X	X
Method	X	X	X
Pointcut	X	X	X
Others	X	X	X

i.e. as conservative extensions of  $L_{Eos}$  and  $L_{AJ}$ . To support the informal claims about expressiveness, it will be sufficient to prove the Theorem 6.1.

**Theorem 6.1.**  $L_{Eos}$  is more expressive than  $L_{AJ}$ .

**Lemma 6.1.** The constructs  $\{aspect, advice, class\}$  are macro-eliminable from the language  $L_{Combined} = L_{Eos} \setminus \{aspect, advice, class\}$ .

**Lemma 6.2.** The constructs  $\{classpect, binding\}$  are not macro-eliminable from the language  $L_{Combined} = L_{AJ} \setminus \{classpect, binding\}$ .

Theorem 6.1 is self-explanatory. Informally, the Lemma 6.1 means that the extra constructs aspect, advice and class construct can be eliminated because they do not add additional expressiveness to the combined model relative to the unified model. The Lemma 6.5 means that the constructs classpect and binding add additional expressiveness to the combined model relative to the AspectJ model, so they cannot be eliminated without sacrificing expressiveness. To prove Theorem 6.1, it suffices to prove the Lemma 6.1 and 6.5 and to compose the two results. The next three sections will sketch the proof. The next section provides a representation of the language models. The rest of this chapter uses that representation.



$\varepsilon \in E$	(Expression)
$\zeta \in E$	(Method call expression)
$\xi \in E$	(Advice or method body)
$\sigma \in E$	(Argument)
$r \in E$	(Method return expressions)
$proceed \in E$	(Proceed expression)
$j \in J \subset (E \times E \times E)$	(Join points)
$\wp(J)$	(Power set of J)
$ej \in EJ$	(Method execution join points)
$aj \in AJ$	(Advice execution join points)
$id \in E$	(Identifier expression)
$reflect \in E$	(Reflective information at the join point)
$\rho \in \wp$	(Pointcut expression)
$methodexecution \in \wp_m(J) \subset \wp(J)$	(Method execution pointcut designators)
$adviceexecution \in \wp_a(J) \subset \wp(J)$	(Advice execution pointcut designators)
$pattern \in E$	(Pattern expression)
<hr/>	
$j ::= id_j reflect_j \varepsilon_j$	(Join point)
$\xi ::= \varepsilon_\xi \mid ej_\xi \mid aj_\xi$	{body}
$methodexecution: (pattern \times J) \rightarrow$ $\{ ej : \forall ej \in J, id_{ej} \mapsto pattern \}$	(Execution point cut matching.)
$adviceexecution: J \rightarrow \{ aj : \forall aj \in J \}$	( Advice execution point cut matching.)
$\wp(J) = \wp_m(J) \cup \wp_a(J)$	

Figure 6.1: Pointcut and Join Points

### 6.3 Representation of the Language Models

Figure 6.1 shows a representation of the relevant expressions, pointcuts, and join point matching process. A join point ( $j$ ) is an expression that evaluates to an identifier expression (tag), an expression that evaluates to the reflective information, and an expression that evaluates to the body that constitutes the join point. A method-execution join point ( $ej$ ) is a join point that evaluates to a tag equal to the name of the method, reflective information expression, and an expression equivalent to the method body. An advice-execution join point ( $aj$ ) evaluates to a tag equal to null (advice is anonymous), reflective information expression, and an expression equivalent to the advice body.

A pointcut expression ( $\rho$ ) can be an execution pointcut expression or an advice execution pointcut expression. To keep the formalism brief and relevant to the current discussion, I have elided other pointcut expressions except method-execution and advice-execution join points in both models. The formalization only considers simple pointcut expressions. Extending it to complex pointcut

$\alpha \in AS$	(Aspects)
$c \in C$	(Classes)
$\hat{c} \in \hat{C}$	(Classpects)
$ctor \in (E \times E)$	(Constructors)
$f \in E$	(Fields)
$m \in M \subset (E \times E)$	(Methods)
$a \in A \subset (\tau \times E \times E)$	(Advice)
$\tau ::= before \mid after \mid around$	(Temporal specification)
$b \in (\tau \times E \times E \times E)$	(Bindings)
<hr/>	
$a ::= \tau_a \sigma_a \rho_a \xi_a$	Advice form
Where $\exists \sigma_a \mapsto reflect_{\rho_a}$ .	
$b ::= \tau_b \rho_b \zeta_b \sigma_b$	Binding form
Where $\exists \sigma_b \mapsto reflect_{\rho_b} \wedge \exists \sigma_b \mapsto \sigma_{\zeta_b}$ .	
$m ::= id_m \sigma_m \xi_m$	Method form
$\alpha ::= [a_\alpha]^* [f_\alpha]^* [m_\alpha]^* [\rho_\alpha]^*$	Aspects
$c ::= [ctor_c]^* [f_c]^* [m_c]^* [\rho_c]^*$	Classes
$\hat{c} ::= [b_{\hat{c}}]^* [ctor_{\hat{c}}]^* [f_{\hat{c}}]^* [m_{\hat{c}}]^* [\rho_{\hat{c}}]^*$	Classpects

Figure 6.2: Aspect and Classpect Members

expressions by adding operators is trivial and not relevant to the current discussion as both AspectJ model and Eos model support exactly the same pointcut sub-language. An execution pointcut expression evaluates to a subset of execution join points such that the pattern of the pointcut expression matches the tag of the execution join point. An advice-execution pointcut expression evaluates to all advice execution join points.

There are other expressions such as method call expression  $\zeta$ , method and advice body as expression  $\xi$ , method and advice arguments as expression  $\sigma$ , method return expression  $r$ , and proceed expression  $p$ . The *proceed* expression can only be a sub-expression of advice body expression.

Figure 6.2 presents representations of aspect, class, advice, classpect, and binding. The notation  $X_Y$  means that  $X$  is associated with  $Y$  and the notation  $[X_Y]^*$  means that one or more  $X$ 's are associated with or contained in  $Y$ .

- An *aspect*  $\alpha$  is a collection of zero or more advice constructs, fields, methods, and, pointcuts. Note that according to the AspectJ programming guide [4], *aspects are not directly instantiated with a new expression, with cloning, or with serialization. Aspects may have one constructor definition, but if so it must be of a constructor taking no arguments and throwing*

*no checked exceptions.*

- A *class*  $c$  is a collection of zero or more constructors, fields, methods, and pointcuts. For simplicity, inner classes are not considered.
- A *classpect*  $\hat{c}$  is a collection of zero or more bindings, constructors, fields, methods, and pointcuts.
- An *advice* evaluates to a 4-tuple: temporal specification, advice argument expression, pointcut expression, and body expression, such that there exists a translation between advice arguments and the reflective information exposed by the pointcut expression.
- A *method* evaluates to a 3-tuple: identifier expression, method argument expression, and body expression.
- A *binding* evaluates to a 3-tuple: temporal specification, pointcut expression, and method call expression, such that there exists a translation between the binding arguments and the reflective information exposed by the pointcut expression, and between the binding argument and the argument to the method call expression.

To prove the Lemma 6.1, it is sufficient to provide a textually local translation of the constructs  $\{aspect, advice, class\}$  from  $L_{Combined}$  to  $L_{Eos}$ . To prove the Lemma 6.5, it is sufficient to provide a translation of the constructs  $\{classpect, binding\}$  from  $L_{Combined}$  to  $L_{AJ}$ , to show that it is not textually local, and to prove that any other translation will preserve this property. The next section presents a translation of constructs  $\{aspect, advice, class\}$  from  $L_{Combined}$  to  $L_{Eos}$ .

## 6.4 Translation from $L_{Combined}$ to $L_{Eos}$

**Lemma 6.3.** *A class is macro-eliminable in  $L_{Combined} = L_{Eos} \setminus \{aspect, advice, class\}$ .*

The translation  $\Gamma : C \mapsto \hat{C}$ , of a class to a classpect proves the Lemma 6.3. This translation is trivial as shown in Figure 6.3. The translation from the members of a class to equivalent members in the classpect is also trivial. In practice, this translation is not even necessary in Eos as classpects

$$\begin{aligned}
\Gamma : C &\mapsto \hat{C} \iff \\
&\forall c \in C \exists \hat{c} \in \hat{C} \\
&\quad [[ctor_c]^* \equiv [ctor_{\hat{c}}]^* \\
&\quad \wedge [f_c]^* \equiv [f_{\hat{c}}]^* \\
&\quad \wedge [m_c]^* \equiv [m_{\hat{c}}]^* \\
&\quad \wedge [\rho_c]^* \equiv [\rho_{\hat{c}}]^*]
\end{aligned}$$

Figure 6.3: Translation of a Class

are also represented using the keyword *class*. As can be observed, this translation is textually local and therefore classes are macro-eliminable in the conservative extension  $L_{Combined}$ .

**Lemma 6.4.** *An aspect is macro-eliminable in  $L_{Combined} = L_{Eos} \setminus \{aspect, advice, class\}$ .*

$$\begin{aligned}
\ddot{\Gamma} : \alpha &\mapsto \hat{c} \iff \\
&\forall \alpha \in AS \exists \hat{c} \in \hat{C} \\
&\quad [[f_\alpha]^* \equiv [f_{\hat{c}}]^* \\
&\quad \wedge [m_\alpha]^* \equiv [m_{\hat{c}}]^* \\
&\quad \wedge [\rho_\alpha]^* \equiv [\rho_{\hat{c}}]^* \\
&\quad \wedge [\Pi(a_\alpha)]^* \equiv [(b \times m)_{\hat{c}}]^*]
\end{aligned}$$

Figure 6.4: Translation of an Aspect

The translation  $\ddot{\Gamma}$  from an aspect in the combined language model to a classpect in the unified model (See Figure 6.4) has two parts: translation  $\Pi$  of an advice to its equivalent in the classpect and translation of other members to their equivalent. The second part of the translation is trivial, as it just requires straightforward translation (in practice textual copy) of the other members to the classpect. The first part of the translation is non-trivial, and complete aspect to classpect translation exists if and only if a valid translation from advice exists. This translation is textually local, so the construct aspect is also macro-eliminable if and only if advice is macro-eliminable.

**Lemma 6.5.** *An advice is macro-eliminable in  $L_{Combined} = L_{Eos} \setminus \{aspect, advice, class\}$ .*

The translation  $\Pi$  of an advice to an equivalent method and a binding is shown in Figure 6.5. This translation exists if and only if:

- The temporal specification of the advice is equivalent to the temporal specification of the binding.

$$\begin{aligned}
\Pi : A \mapsto (B \times M) &\iff \\
\forall a \in A \exists (b, m) \in (B \times M) & \\
& [\tau_a = \tau_b \\
& \wedge \rho_a \equiv \rho_b \\
& \wedge \xi_a \equiv \xi_m \\
& \wedge \zeta_b \equiv id_m \\
& \wedge \sigma_a \equiv \sigma_b \\
& \wedge (\xi_a \text{ depends on } reflect_{\rho_a} \rightarrow \\
& \quad reflect_{\rho_b} \mapsto \sigma_b \wedge reflect_{\rho_b} \mapsto \sigma_{\zeta_b}) \\
& \wedge (\xi_a \text{ depends on } proceed_{\rho_a} \rightarrow \\
& \quad proceed_{\rho_a} \mapsto \sigma_b \wedge proceed_{\rho_a} \mapsto \sigma_{\zeta_b})]
\end{aligned}$$

Figure 6.5: Translation of an Advice

- There exists a translation from the pointcut in the advice to the pointcut in the binding.
- There exists a translation from the advice body to the method body.
- There exists a translation from the binding call expression to the method's identifier expression.
- There exists a translation from advice parameters to the binding parameters.
- In addition, if the advice body expression depends on reflective information expression or *proceed* expression, there exists a translation from them to binding argument expression and binding call expression.

All these sub-translations, except the sub-translation from advice body to method body, are trivial. For example the temporal specification of the advice *before/after/around* will be textually translated to the temporal specification of the binding *before/after/around*. Only the translation from the advice body to the method body is slightly involved, because AspectJ advice can use implicit reflective variables like *thisJoinPoint* and special forms like *proceed* to invoke the inner join points. If the advice body does not use any of these special construct, the translation is equivalent to textual copy. If the advice body uses the implicit variables, the translation from advice body to method body has three steps. First, a unique argument is added to the method. Second, all occurrences of the implicit variable in the advice body are replaced by the new argument, and third, the argument is bound to the reflective information in the binding.

If the advice body uses the special form *proceed*, the translation has three steps. First, a unique argument to represent delegate chain is added to the method. Second, all occurrences of the special form in the advice body are replaced by the call on the argument to invoke the next delegate in the chain, and third, the argument is bound to the delegate chain in the binding. For advice body with both implicit variable and binding, a combination of the two translations described above can be used. All these translations are textually local, so the construct advice is macro-eliminable, and therefore construct aspect is also macro-eliminable proving Lemma 6.4 and Lemma 6.5.

The Lemma's 6.3, 6.4, and 6.5 together complete the proof of Lemma 6.1. If we recall, Lemma 6.1 shows that  $\{class, advice, and aspect\}$  can be eliminated from the combined language model without sacrificing expressiveness. In other words, if a language model contains  $\{classpect, binding\}$ ; addition of  $\{class, aspect, advice\}$  to it doesn't enhance its expressiveness. All new programs generated by  $\{class, aspect, advice\}$  can be translated to an equivalent program using  $\{classpect, binding\}$ . The next section provides a proof for the Lemma 6.5.

## 6.5 Translation from $L_{Combined}$ to $L_{AJ}$

In this section, I show that the constructs  $\{classpect, binding\}$  are not macro-eliminable in  $L_{Combined} = L_{AJ} \setminus \{classpect, binding\}$ . To construct the proof, I start with the assumption that these constructs are macro-eliminable and then show that the existence of such translation depends on the existence of a one-to-one mapping between advice execution join point and pointcut pattern. In  $L_{AJ}$  such a mapping is not present due to anonymous advice constructs contradicting the original assumption. The rest of this section sketches the proof.

Let us assume that the constructs  $\{classpect, binding\}$  are macro-eliminable. From the definition of macro-eliminability in Subsection 6.1.2 (Definition 3.11 in [32]), it follows that in order for these constructs to be macro-eliminable there must exist a translation that maps  $\{classpect, binding\}$  to abstractions in  $L_{AJ}$ . The translation of classpect with no bindings is trivial and equivalent to the textual copy of the classpect to a class. For the classpects with bindings, we already know that the abstraction *class* in  $L_{AJ}$  may not advise and translation of bindings in

classpects require advising capabilities. Therefore, for classpects in  $L_{Combined}$  that contain bindings, the abstraction in  $L_{AJ}$  must be an *aspect*. We also know that the abstraction *method* may not advise, therefore a binding and corresponding handler method must be translated to an advice. Let us assume that  $\Omega : L_{Combined} \mapsto L_{AJ}$ , is such a translation. For  $\{classpect, binding\}$  to be macro-eliminable the following property must be true for  $\Omega$ .

$$\Omega(\hat{C}(e_1, e_2, \dots, e_n)) = AS(\Omega(e_1), \Omega(e_2), \dots, \Omega(e_n)),$$

Here  $e_1, e_2, \dots, e_n$  are sub-constructs of  $\hat{C}$ . In this case, they can be bindings, fields, methods, pointcuts, etc. In other words, a classpect  $\hat{C}$  and the constructs  $e_1, e_2, \dots, e_n$  in it are translated to an aspect and translations of the constructs  $e_1, e_2, \dots, e_n$ . For fields, methods, and other members, except bindings and pointcuts, these translations are trivial and are equivalent to the textual copy of the members to the aspect. For the  $\{classpect, binding\}$  to be macro-eliminable  $\Omega$  must provide a mapping for the constructs field, method, etc to the equivalent constructs in  $L_{AJ}$ . For the binding and corresponding handler method the translation must provide a mapping to the corresponding advice in  $L_{AJ}$ . Let us represent these required sub-translations as follows. Here an aspect  $\alpha \in AS$  is the mapping of the classpect  $\hat{c} \in \hat{C}$ .

1.  $\Omega(\hat{C}(\dots, f_{\hat{c}}, \dots)) = AS(\dots, f_{\alpha}, \dots)$
2.  $\Omega(\hat{C}(\dots, m_{\hat{c}}, \dots)) = AS(\dots, m_{\alpha}, \dots)$
3.  $\Omega(\hat{C}(\dots, \rho_{\hat{c}}, \dots)) = AS(\dots, \rho_{\alpha}, \dots)$
4.  $\Omega(\hat{C}(\dots, b_{\hat{c}}, \dots, m_{\hat{c}})) = AS(\dots, a_{\alpha}, \dots)$

For the fourth part of the translation,  $b_{\hat{c}}$ ,  $m_{\hat{c}}$  and  $a_{\alpha}$  must be such that  $\tau_{b_{\hat{c}}} = \tau_{a_{\alpha}} \wedge \rho_{b_{\hat{c}}} \equiv \rho_{a_{\alpha}}$   $\wedge \xi_{m_{\hat{c}}} \equiv \xi_{a_{\alpha}} \wedge \sigma_b \equiv \sigma_{a_{\alpha}}$  (i.e. the temporal specification of the advice must be the same as the temporal specification of the binding, the pointcut expression that is part of the advice must be equivalent to the pointcut expression that is part of the binding, the method body must be equivalent to the advice body, and arguments of the binding must be the same as the argument to the advice). All other equivalence relationships except that of pointcuts are straightforward.

Let us assume the set of join points in the programs in  $L_{combined}$  and  $L_{AJ}$  are  $J_{combined}$  and  $J_{AJ}$  respectively. For  $\rho_{b_{\varepsilon}} \equiv \rho_{a_{\alpha}}$  to be true,  $\rho_{b_{\varepsilon}}(J_{combined}) \equiv \rho_{a_{\alpha}}(J_{AJ})$  must be true (i.e. the pointcuts in both language models select equivalent join points). There are three type of join points in  $J_{combined}$ , method execution join points, advice execution join points, and method execution join points that are handlers of other bindings. There are two type of join points in  $J_{AJ}$ , method and advice execution join points. For the equivalence  $\rho_{b_{\varepsilon}}(J_{combined}) \equiv \rho_{a_{\alpha}}(J_{AJ})$  to be true,  $\forall j \in J_{combined}, id_j \mapsto pattern_{\rho_{b_{\varepsilon}}} \wedge \exists j' \in J_{AJ}, id_{j'} \mapsto pattern_{\rho_{a_{\alpha}}}$  must be true. That means for every join point in the program in  $L_{Combined}$  that is matched by the pattern of the pointcut in the binding there exists an equivalent join point in the program in  $L_{AJ}$  that is matched by the pattern of the pointcut in the equivalent advice.

Recall that some classpects may contain higher-order bindings. A binding is a *higher-order binding* if the *subject join points* of the binding are *handlers* of other bindings or contained in *handlers* of other bindings. A subject join point of a binding is matched by the pattern of the pointcut in the binding. If the *subject join points* of the binding are *handlers* of other bindings or contained in *handlers* of other bindings these *handlers* are mapped to corresponding advice constructs in  $L_{AJ}$ . That means for a higher-order binding  $b'$ ,  $\forall j \in J_{combined}, id_j \mapsto pattern_{\rho_{b'_{\varepsilon}}} \wedge \exists j' \in J_{AJ}, j \mapsto j' \wedge j' \in AJ$ . Here  $AJ$  is the set of advice-execution join points.

An advice is anonymous, so an advice-execution join point does not have an identifier to match with the pattern of the pointcut expression, which means that for *classpects with higher-order bindings*  $\forall j \in J_{combined}, id_j \mapsto pattern_{\rho_{b_{\varepsilon}}} \wedge \exists j' \in J_{AJ}, id_{j'} \mapsto pattern_{\rho_{a_{\alpha}}}$  is false. This further means that  $\rho_{b_{\varepsilon}}(J_{combined}) \equiv \rho_{a_{\alpha}}(J_{AJ})$  is false, and that in turn means that  $\rho_{b_{\varepsilon}} \equiv \rho_{a_{\alpha}}$  is false. As mentioned before the equivalence relationship  $\rho_{b_{\varepsilon}} \equiv \rho_{a_{\alpha}}$  was required for an arbitrary macro-eliminable translation to exist. Therefore, the assumption that the arbitrary translation  $\Omega$  is a macro-eliminable translation is false for classpects that contain *higher-order bindings*, contradicting the initial assumption that classpect, binding are macro-eliminable proving Lemma . If we recall, Lemma shows that  $\{classpect, binding\}$  cannot be eliminated from the combined language model without sacrificing expressiveness. In other words, the addition of  $\{classpect, binding\}$  enhances the expressiveness of  $L_{AJ}$ .



In this chapter I have shown that if a language model contains  $\{classpect, binding\}$ ; addition of  $\{class, aspect, advice\}$  to it doesn't enhance its expressiveness. All new programs generated by  $\{class, aspect, advice\}$  can be translated to an equivalent program using  $\{classpect, binding\}$ . However, if a language model just has  $\{class, aspect, advice\}$  then the addition of  $\{classpect, binding\}$  enhances its expressiveness. The programs generated by  $\{classpect, binding\}$  cannot be generated in a textually local way using  $\{class, aspect, advice\}$ . In terms of language models, I have proved that the combined model is more expressive than the AspectJ model and the combined model has the same expressive power as the Eos programming model. From the two results it follows that Eos is more expressive than AspectJ with respect to higher-order advising.

## 6.6 Discussion

$$\begin{aligned}
 b_{P_i} &= \tau_{b_{P_i}} \rho_{b_{P_i}} \zeta_{b_{P_i}} \sigma_{b_{P_i}} \\
 m_{P_i} &= id_{m_{P_i}} \sigma_{m_{P_i}} \xi_{m_{P_i}} \\
 \tau_{b_{P_i}} &= \text{around} \\
 \rho_{b_{P_i}} &= \text{pointcut to select execution of method } m_{\hat{c}_i} \\
 \zeta_{b_{P_i}} &= \text{call expression, target method is } m_{P_i} \\
 \sigma_{b_{P_i}} &\mapsto \text{reflect}_{\rho_{b_{P_i}}} /* \text{ Binding arguments map to the reflective} \\
 &\quad \text{variables exposed by pointcuts.*/} \\
 \sigma_{b_{P_i}} &\mapsto \sigma_{\zeta_{b_{P_i}}} /* \text{ Binding arguments map to the arguments} \\
 &\quad \text{of the method call expression.*/}
 \end{aligned}$$

Figure 6.6: The Retry Policy Binding and Method

$$\begin{aligned}
 \hat{c}_O &= b_i \text{ ctor } f \ m_i \\
 b_i &= \tau_{b_i} \rho_{b_i} \zeta_{b_i} \sigma_{b_i} \\
 m_i &::= id_{m_i} \sigma_{m_i} \xi_{m_i} \\
 \tau_{b_i} &= \text{after} \\
 \rho_{b_i} &= \text{pointcut to select execution of method } m_{P_i} \\
 \zeta_{b_i} &= \text{call expression, target method is } m_i \\
 \sigma_{b_i} &\mapsto \text{reflect}_{\rho_{b_i}} /* \text{ Binding arguments map to the reflective} \\
 &\quad \text{variables exposed by pointcuts.*/} \\
 \sigma_{b_i} &\mapsto \sigma_{\zeta_{b_i}} /* \text{ Binding arguments map to the arguments} \\
 &\quad \text{of the method call expression.*/}
 \end{aligned}$$

Figure 6.7: The Overhead Computation Classpect

In this section, I present an informal discussion of the formalism described in the previous section. For the purpose of this discussion, I use a simple but representative example. I use the notation described in Figure 6.1 and 6.2. Our example system models error-prone resources and fault-tolerance policies. There are  $n$  different types of resources in the system,  $R_1, R_2, \dots, R_n$ . A requirement is to construct a fault-tolerant system with these error-prone resources. To construct a fault-tolerant system, a retry policy type  $P_i$  is defined for each resource type  $R_i$ .

To implement this system using  $L_{Eos}$ , the resource types will be modeled as classpects  $\hat{c}_i \in \hat{C}$ . For simplicity, let us assume that each basic resource type  $R_i$  provides exactly one operation  $op_i$ . We can always model a resource type that provides more than one operation as a combination of basic resource types. These operations are modeled as methods,  $m_{\hat{c}_i} \in M$ . The retry policy types are also modeled as classpects  $\hat{c}_{P_i} \in \hat{C}$ . The retry policies are implemented using an around binding and a method. Each retry policy classpect  $\hat{c}_{P_i}$  declares a binding  $b_{P_i}$  and a method  $m_{P_i}$  as shown in Figure 6.6. The binding  $b_{P_i}$  selects the execution of the method  $m_{\hat{c}_i}$  and binds the method  $m_{P_i}$  to execute around it. When the method  $m_{\hat{c}_i}$  is called to perform operation  $op_i$  on resource type  $R_i$  the method  $m_{P_i}$  bound around is invoked instead. The around method retries the original method specified number of times.

To translate this system to  $L_{AJ}$ , the resource types will be modeled to equivalent classes  $c_i \in C$ . The retry policies will be mapped to equivalent aspects  $\alpha_i \in AS$  such that the binding  $b_{P_i}$  and a method  $m_{P_i}$  is mapped to an around advice  $a_{P_i}$ .

Let us consider an evolution scenario in which the system needs to compute the overhead of each retry. For each retry policy type  $P_i$  there is an associated overhead  $O_i$ . In  $L_{Eos}$ , to compute the overhead for retry policy, an overhead computation module is implemented as a classpect  $\hat{c}_O \in \hat{C}$  as shown in Figure 6.7. The classpect provides a field to store the overhead, a constructor to initialize it to zero, a binding  $b_i$  and a method  $m_i$  corresponding to each retry policy  $P_i$ . The binding binds the method  $m_i$  to execute after the method  $m_{P_i}$  to record the overhead.

To translate the overhead requirement to  $L_{AJ}$ , one would think of translating the overhead classpect  $\hat{c}_O$  to an aspect  $\alpha_O$  with each binding  $b_i$  and method  $m_i$  pair translated to an after advice  $a_i$ . To pursue this straightforward implementation technique, selecting individual advice-execution

join points  $a_{P_i}$  is required. We cannot select individual advice-execution join points because advice is not a named construct. An alternative is to perform a translation from  $a_{P_i}$  to another around advice  $a'_{P_i}$  and a method  $m'_{P_i}$  such that the around advice  $a'_{P_i}$  calls the method  $m'_{P_i}$ . Another alternative is the upfront translation from the binding  $b_{P_i}$  and the method  $m_{P_i}$  to the around advice  $a'_{P_i}$  and the method  $m'_{P_i}$ . Both these translations require global changes showing that  $L_{Eos}$  is more expressive compared to  $L_{AJ}$  in this dimension.

## 6.7 Summary

In this chapter, I validated my claim that the unified model is more expressive than the programming model of AspectJ-like languages. In particular, I showed that the translation of certain programs from a language based on classpect and binding to a language based on class, aspect, and advice requires non-local transformations in the program. The set of programs that require non-local transformations contain classpects with one or more higher-order binding. One way to try to account for these improvements is by appeal to the idea of conceptual integrity in design. Brooks wrote,

...that conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas. Simplicity and straightforwardness proceed from conceptual integrity. Every part must reflect the same philosophies and the same balancing of desiderata. Every part must even use the same techniques in syntax and analogous notions in semantics. Ease of use, then, dictates unity of design, conceptual integrity.” [8](pp 42-44).

The additional expressive and compositional power of classpects emerged when this kind of design unity is enforced. It forced aspect-like constructs to support all of the capabilities of classes—notably new. It forced classes to support aspect-oriented advising as a generalized alternative to traditional invocation and overriding. By driving out anonymous advice in favor of methods as the sole mechanism for procedural abstraction, it also pushed a previously submerged but important

abstraction to the fore: the join–point–method binding. A binding turned to be at the core of the aspect–oriented advising replacing asymmetric and non–orthogonal aspect and advice.

# Chapter 7

## Case Studies

---

I used Eos to modularize crosscutting concerns in several case studies. This chapter presents a subset of those case studies. The point of these studies is three-fold. First, they show that the supporting infrastructure is robust enough for real world systems. Second, they show that the unified model is scalable to the real world systems and third, they show that the unified model shows benefits in real world systems. Using Eos in real world systems also showed some practical problems with a new language infrastructure. At the end of this chapter, I describe these problems.

I used some canonical examples and two real systems as case studies. The canonical examples included in the study are, (1) the Gang-Of-Four object-oriented design patterns, and (2) the mediator based design idioms used in PRISM (A Radiation Treatment Planning System at University of Washington) [97]. These canonical examples represent widely used design styles; therefore, an improvement noticed in these cases is likely to manifest itself in the real world. Section 7.1 describes the implementation of some of the GOF design patterns.

The two real systems that I used as case study were the Eos compiler and a ConcernCov a tool for Concern Coverage Processing [83]. I applied the new language model extensively towards the experimental re-design of the Eos compiler. The compiler presents multiple opportunities where the new language model results in improved modularization. Section 7.3 describes some of the scenarios.

## 7.1 Design Patterns

The Gang-of-Four (GoF) design patterns [37] identify common design structures in software systems; and provide mechanism and sample implementation strategies for efficiently tackling them. Initial implementation strategies are oriented towards object-oriented paradigm, however, it has been shown that the implementation language affects the use and implementation strategies [11] [105] [76] [91].

Inspired by these studies, Hannemann and Kiczales [45] presented implementation of design patterns in AspectJ and showed that their implementation is better with respect to a set of criteria. Sakurai et al. [88] briefly observed that the type-level aspect-oriented implementation of the design patterns described by Hannemann et al. exhibit the design and performance overhead of the form described by Rajan and Sullivan [81]. This section looks at the implementation of some of the design patterns. I present a realization of design patterns using the unified language model. I also compare the new implementation with that of Hannemann et al.'s. Note that Hannemann et al.'s implementation uses the current language model of AspectJ-like languages.

Chapter 6 showed that translation of constructs from the AspectJ language model to their equivalent in the proposed unified model is possible. This translation does not require any non-local changes. The constructs in the unified model can essentially express constructs in the current model and much more. Based on that observation one would expect to be able to translate the implementation of design patterns from AspectJ to Eos, which is indeed the case. The pattern implementations provided by Hannemann and Kiczales can be translated to Eos without any non-modular changes. The translation of pattern protocols is straightforward. The examples provided with the pattern implementation, however, are heavily dependent upon the platform. As a result, their translation is not quite direct owing to the host framework differences. AspectJ operates on Java Virtual Machine (JVM), whereas Eos operates on .Net Framework.

The translation rules from AspectJ model to Eos model do not require any non-modular changes, preserving the modularity of AspectJ based solution, and implying that the results in the Hannemann and Kiczales [45] as well as those by Garcia et al. [38] apply to the Eos implemen-

tation as well. For some patterns, however, expressiveness of the unified language model allows realizations of even better implementation strategy. The next subsections describe these patterns. The next subsection 7.1.1 analyzes the observer pattern. Hannemann and Kiczales's work discusses this pattern in most detail.

### 7.1.1 Observer Pattern

AspectJ	Eos
<pre> 1 public abstract aspect ObserverProtocol { 2   protected interface Subject {} 3   protected interface Observer {} 4   private WeakHashMap perSubjectObservers; 5   protected List getObservers(Subject s) { 6     if (perSubjectObservers == null) { 7       perSubjectObservers = new WeakHashMap(); 8     } 9     List observers = 10      (List)perSubjectObservers.get(s); 11     if (observers == null) { 12       observers = new LinkedList(); 13       perSubjectObservers.put(s, observers); 14     } 15     return observers; 16   } 17   public void addObserver(Subject s, Observer o) { 18     getObservers(s).add(o); 19   } 20   public void removeObserver(Subject s, Observer o) { 21     getObservers(s).remove(o); 22   } 23   protected abstract pointcut 24     subjectChange(Subject s); 24   protected abstract void 25     update (Subject s, Observer o); 26   after(Subject s): subjectChange(s) { 27     Iterator iter = getObservers(s).iterator(); 28     while ( iter.hasNext() ) { 29       update (s, ((Observer)iter.next())); 30     } 31   } 32 } </pre>	<pre> 1 public abstract class ObserverProtocol { 2   protected abstract pointcut 3     subjectChange(); 4   protected abstract void 5     update (); 6   after subjectChange(s): update(Subject s); 7 } </pre>

Figure 7.1: The Observer Protocol: Eos Code 78% Smaller

The intent of the observer pattern is to define a one-to-many dependency between objects so that on an object's state change, all dependents are notified and updated automatically [37]. The AspectJ implementation divides the pattern implementation into two parts: parts that are common to all instantiations of the pattern and parts specific to an instantiation. The implementation abstracts

the common part as a reusable ObserverProtocol aspect as shown in the left half of Figure 7.1. It provides an abstract pointcut `subjectChange` (lines 23–24) to represent observable state change of the subject. A concrete observer implementation defines this pointcut. The implementation also provides an abstract method; `update` (lines 24–25) to be redefined in concrete observers to implement the observer’s logic.

The AspectJ language model does not support aspect instantiation and selective advising of object-instances. In the observer pattern, an instance of observer needs to selectively advise instances of subject. To emulate instance-level advising using type-level aspects, Hannemann and Kiczales’s implementation of the observer protocol needs to manipulate instances of participants. To be able to do so without coupling the ObserverProtocol with participants, it defines two new interfaces that are introduced by the concrete observers into participants so that ObserverProtocol can manipulate them. The pattern implementation therefore unnecessarily superimposes two roles, subjects (line 2) and observers (line 3), on participants.

It also keeps a `HashMap` (line 4) of observers corresponding to an instance of the subject. It provides methods to add (lines 17–19) and remove (lines 20–22) observers corresponding to a subject. It also provides methods to retrieve observers for a subject (lines 5–16). The observer protocol logic is implemented by the advice (line 26–31). This advice is invoked by each instance of the class being advised, even if any observer is not observing the instance. On being invoked, the advice looks up the invoking instance and retrieves the list of observers. It then iterates through the list to invoke each observer. In summary, the AspectJ implementation tangles the instance-level advising and instance-emulation concern with the observer pattern concern. The need for roles and for maintaining a `HashMap` are examples of design-time overheads incurred due to the asymmetry of the language model.

The AspectJ implementation of the observer pattern is localized, reusable, compositionally transparent, and (un) pluggable. The Eos implementation mimics the implementation strategy by similarly partitioning the pattern implementation into abstract classpect `ObserverProtocol` and concrete realization of observers inheriting from this classpect. The abstracted pattern is shown on the right hand side of Figure 7.1.



The Eos implementation improves the modularity of the pattern. It does not tangle emulation concern with observer protocol concern. It clearly abstracts the behavior of the pattern in a much nicer way. It clearly (and only) conveys the intent of the pattern, which is to update an observer when a subject changes. The binding states that after the join points selected by the abstract pointcut SubjectChange, the method Update should be called. All the interfaces and additional code required to emulate instance-level behavior is gone.

With respect to the metrics used by Garcia et al [38], it achieves nearly 78 percent reduction in LOC (Line of Code) of the observer protocol concern without increasing the complexity of the remaining code. Each line in Eos implementation corresponds to a line in AspectJ implementation. The Eos implementation, however, eliminates all the emulation code from the pattern implementation. This implementation is also localized, reusable, compositionally transparent, and (un) pluggable. It also decreases the Number of Attributes (NOA) [38] of the observer protocol concern to zero from one.

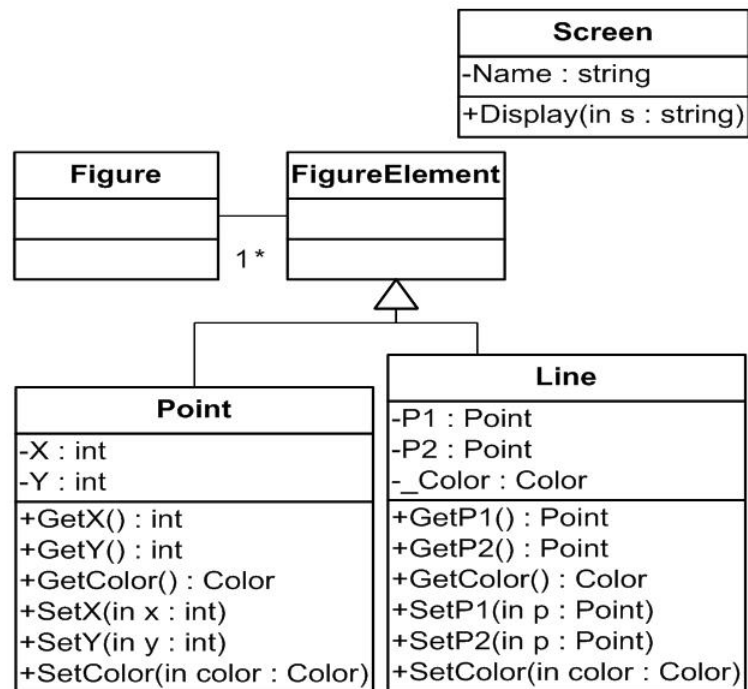


Figure 7.2: A Simple Graphical Figure Element System

AspectJ	Eos
<pre> 1 public aspect ColorObserver 2   extends ObserverProtocol{  3   declare parents: Point implements Subject; 4   declare parents: Screen implements Observer;  5   protected pointcut subjectChange(Subject s): 6     call(void Point.setColor(Color)) &amp;&amp; target(s);  7   protected void update (Subject s, Observer o) { 8     ((Screen)o).display("Color -&gt; Screen."); 9   } 10 }</pre>	<pre> 1 public class ColorObserver 2   : ObserverProtocol { 3   Point p; Screen s; 4   public ColorObserver (Point p, Screen s) { 5     addObject(p); this.p = p; this.s = s; 6   } 7   override pointcut subjectChange(): 8     execution(void Point.setColor(Color));  9   public override void update(){ 10    s.display("Color -&gt; Screen update."); 11  } 12 }</pre>

Figure 7.3: The Color Observer in AspectJ and Eos

Moreover, the composition of the participants into the observing relationships becomes more intuitive in the Eos implementation. To illustrate let us consider the example system presented by Hannemann et al. shown in Figure 7.2. In this example figure, element system we have two potential subjects a Point, a Line, and an observer Screen. Instances of the class Screen observe change in the color and co-ordinates of instances of Point. A subject-observer relationship between Point and Screen in which Screen instance observes change in color of the Point instance is shown on the left hand side of Figure 7.3. The ColorObserver relationship implementation does not clearly communicate the specification that it involves two object instances, an observer instance and an observed instance. This part of the specification is hidden in the parent class, ObserverProtocol. Understanding the behavior of the parent class is necessary to deduce how to put two objects instances, a Screen and a Point into a color observing relationship. As a result, even though the pattern protocol achieves a physical separation of code, separation of concern between parent ObserverProtocol and the relationship ColorObserver is not achieved.

The implementation of the same ColorObserver relationship is shown on the right hand side of Figure 7.3. The implementation clearly represents the intent of the pattern. By declaring a constructor that takes a point and a screen as an argument, it depicts the observing relationship between these two entities. Compared to the AspectJ implementation where relationship instances are emulated implicitly using hash tables, in the Eos implementation one explicit instance of ColorObserver

exists for each point and class instance that participate in the observing relationship.

The Eos implementation does not require code for instance-level weaving emulation. It represents the ColorObserver as a class containing an instance variable screen to store reference to the observer Screen instance and the subject Point instance (line 3), a constructor (lines 4–6), definition of what it means for a subject to change (lines 7–8) and method to update the observer (Line 9–11). For comparison, the listing below shows the key parts of the client code. AspectJ code is preceded by *AspectJ:* and Eos code is preceded by *Eos:*.

```
/* Construct Point p and Screens s1, s2 here */
AspectJ: ColorObserver.aspectOf().addObserver(p, s1);
AspectJ: ColorObserver.aspectOf().addObserver(p, s2);
Eos: ColorObserver cobs1 = new ColorObserver(p, s1);
Eos: ColorObserver cobs2 = new ColorObserver(p, s2);
```

The client code shows that Eos achieves modular component composition. As opposed to calling a special aspectOf method on ColorObserver module and then calling addObserver on that module, now subjects and observers are composed by creating new instances of observing relationship. In summary, Eos significantly improves upon the AO implementation of the Observer pattern. No additional design time overhead is needed to emulate instantiation and instance-level weaving, which results in significant reduction in the size and complexity of the implementation.

### 7.1.2 Mediator Pattern

The intent of the Mediator pattern:

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. pp 273 [37]

Similar to the observer pattern's implementation, the AspectJ implementation provided by Hannemann and Kiczales divides the pattern implementation into two parts: parts that are common to

AspectJ	Eos
1 public abstract aspect MediatorProtocol {	1 public abstract class MediatorProtocol {
2   protected interface Colleague { }	
3   protected interface Mediator { }	
4   WeakHashMap ColToMed = new WeakHashMap();	
5   Mediator getMediator(Colleague colleague) {	
6       Mediator mediator = (Mediator)	
7           ColToMed.get(colleague);	
8       return mediator;	
9   }	
10   public void setMediator(Colleague colleague,	
11       Mediator mediator) {	
12       ColToMed.put(colleague, mediator);	
13   }	
14   protected abstract pointcut	2   protected abstract pointcut
15       change(Colleague colleague);	3       change();
16   after(Colleague c): change(c) {	4   after change() : notify();
17       notify (c, getMediator(c));	
18   }	
19   protected abstract void	5   protected abstract void
20       notify (Colleague c, Mediator m);	6       notify();
21 }	7 }

Figure 7.4: The Mediator Protocol: Eos Code 66% Smaller.

all instantiations of the pattern and parts specific to an instantiation. The implementation abstracts the common part as a reusable MediatorProtocol aspect as shown in the left half of Figure 7.4. It provides an abstract pointcut change (lines 14–15) to represent state change of the colleagues. A concrete mediator implementation defines this pointcut. The implementation also provides an abstract method; notify (lines 19–20) to be redefined in concrete mediators to implement the notification logic.

The MediatorProtocol aspect also keeps a HashMap (line 4) to keep track of the colleague instances that are being mediated by a mediator instance. It provides methods to set (lines 10–13) and get (lines 5–9) mediator corresponding to a colleague. *This implementation does not work in cases where a colleague instance is participating in more than one mediating relationship.* Let us assume a scenario where a colleague instance *c* is involved in two mediating relationships, *m1*

and `m2`. To put the colleague in the mediating relationships the method `setMediator` will call with parameters `(c, m1)` and `(c, m2)` in any order. The method `setMediator` in turn will call the method `put` on `WeakHashMap ColToMed` with `Colleague c` as the key. When these calls are completed, the last mapping from colleague to mediator remains in the `WeakHashMap` as it replaces the value supplied in the old mapping with the new one.

Like observer pattern the Eos implementation shown on the right hand side of Figure 7.4 is does not tangles emulation concern with mediator protocol concern resulting in a modular implementation of the mediator protocol. The implementation clearly represents the behavior of the pattern, decreasing the conceptual gap between the specification and implementation of the pattern. It clearly (and only) conveys the intent of the pattern. The intent of the pattern is that after change in a colleague mediator notifies the changes to other colleagues. The binding states that after the join points selected by the abstract pointcut change, the method `notify` should be called. No interfaces and additional code required to emulate instance-level advising is required.

The implementation of design patterns in Eos showed that the unified language model eliminates the need for emulation strategies making the resulting implementation much simpler. The simplification is significant. The composition of participants and patterns also becomes much more intuitive in the new implementation. The next section describes realization of idioms, used to evaluate mediator-based design style [95], in Eos.

## 7.2 Complex Mediator-Based Design Idioms

The next two sections show the implementation of two key mediator structures. These mediator structures were used in the design of Prism [97] [95], an integrated environment for radiation treatment planning itself a major test of the mediator approach <sup>1</sup>. The first mediator, described in subsection 7.2.1, maintains a bijection between two sets; the second mediator, described in subsection 7.2.2, a cross-product between two sets. These fragments played important roles in Prism, and are representative of the kinds of mediators that arise when the approach is used to design real

---

<sup>1</sup>Some contents of this section appeared previously in [81]

systems. I selected these idioms for case study to show that Eos supports the design of realistic integrated systems using classpects as mediators.

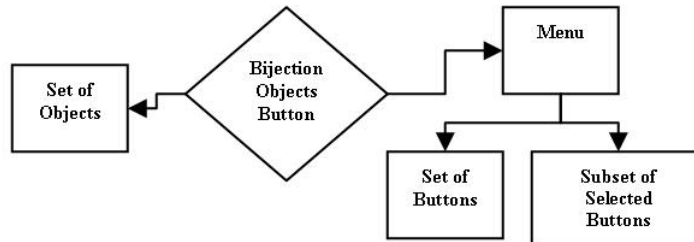


Figure 7.5: Mediator-Based Model View System [95]

```

1 public class Set : CollectionBase {
2   public bool Insert(Element element) {
3     // check duplication
4     System.Collections.IEnumerator itor = this.GetEnumerator();
5     while ( itor.MoveNext() )
6       if(((Element)itor.Current).Name == element.Name)
7         return false; /// Duplicate element
8     List.Add(element);
9     return true;
10  }
11  public bool Remove(Element element){
12    int index = 0;
13    // Find element
14    System.Collections.IEnumerator itor = this.GetEnumerator();
15    while ( itor.MoveNext() ){
16      if(((Element)itor.Current).Name == element.Name){ // Got it
17        if (index <= Count - 1 && index >= 0){
18          List.RemoveAt(index);
19          return true;
20        }
21        else return false;
22      }
23      index ++;
24    }
25    return false;
26  }
27 }

```

Figure 7.6: The Set Implementation

```

1 public class Element {
2     public string Name;
3     public Element(string Name){
4         this.Name = Name;
5     }
6 }

```

Figure 7.7: The Element Implementation

### 7.2.1 Bijection Mediator

A common requirement is to maintain consistency between sets of GUI objects (buttons, menu items, and shapes) and sets of model objects (in Prism, models of internal organs of cancer patients). In Prism, one or more menu items can be selected to open panels for editing the designated model objects. This system is represented as a set of model objects, a set of menu items, a selected subset of menu items, a relationship that keeps the set of model objects and the set of menu items in one-to-one correspondence, and a relationship that keeps the subset of selected menu items consistent with a set of individual model editing panels.

Figure 7.5 presents a schematic showing the first relationship. Sets and menus are implemented as instances of Set and Menu types with operations to insert, delete and iterate over elements. In the original mediator implementation, these classes had explicit events. Eos eliminates this requirement. The key parts of the Eos Set and Bijection mediator implementation are shown in Figure 7.8.

The Set class (shown in Figure 7.6) exposes no explicit events. The Bijection mediator, implemented as another class, provides a constructor to relate two instances of the Set type. It uses the implicit addObject method (line 4) to *register* the advice with these instances.

The binding (lines 6–8) uses the pointcut execution(`public bool Set.Insert(Element)`) to identify the join point method execution of `Set.Insert` and invokes the method `AfterInsert` (lines 12–20) at that join point. It uses the *return*, *this*, and *args* pointcuts to associate arguments *r*, *s*, and *e* of the `AfterInsert` to the return value of the `Set.Insert` method, the instance on which the method is called, and the argument `Element` instance supplied to the method. In other words, the binding says that after the method `Set.Insert` is called on a Set instance *s1* with argument `Element` instance *e1* and it

```

1  public class Bijection {
2      bool busy = false ; Set s1, s2;
3      public Bijection(Set s1, Set s2){
4          addObject(s1); addObject(s2); this.s1 = s1; this.s2 = s2;
5      }

6      after execution(public bool SET.Set.Insert(Element))
7          && return(r) && this(s) && args(e):
8          AfterInsert(bool r, Set s, Element s);

9      after execution(public bool SET.Set.Remove(Element))
10         && return(r) && this(s) && args(e):
11         AfterRemove(bool r, Set s, Element s);

12     public void AfterInsert(bool r, Set s, Element e){
13         if(r && !busy) {
14             busy = true;
15             Element new_elem = new Element(e.Name);
16             if(s == this.s1) s2.Insert(new_elem);
17             else s1.Insert(new_elem);
18             busy = false;
19         }
20     }

21     public void AfterRemove(bool r, Set s, Element e){
22         if(r && !busy) {
23             busy = true;
24             Element new_elem = new Element(e.Name);
25             if(s == this.s1) s2.Remove(new_elem);
26             else s1.Remove(new_elem);
27             busy = false;
28         }
29     }
30 }

```

Figure 7.8: The Set Bijection Mediator's Implementation

returns  $r1$  (where  $r1$  can be true or false) the method `AfterInsert` is called on the `Bijection` instance with arguments  $r1$ ,  $s1$ , and  $e1$ . The method `AfterInsert` then inserts the corresponding element in the other set that is in `Bijection` relationship.

Similarly, to identify the join point method execution of `Set.Remove`, the binding (line 9–11) uses the pointcut `execution(public bool Set.Remove(Element))` and invokes the method `AfterRemove` (lines 21–29) at that join point. It also uses the *return*, *this*, and *args* pointcuts to associate arguments  $r$ ,  $s$ , and  $e$  of the `AfterRemove` to the return value of the `Set.Remove` method, the `Set`



```

1  public class OrderedPair {
2      bool busy; Element a; Element b;
3      public OrderedPair(Element a, Element b) {
4          addObject(a); addObject(b); this.a = a; this.b = b;
5      }

6      after fset(string Element.Name) && this(e): AfterNameChange(Element e);

7      public void RemoveOrder() { removeObject(a); removeObject(b); }

8      public bool isPair(Element a, Element b) {
9          if(a.Name == this.a.Name && b.Name == this.b.Name) return true;
10         return false;
11     }

12     public void AfterNameChange(Element e) {
13         if(!busy) {
14             busy = true;
15             if(e == this.a) b.Name = e.Name;
16             else a.Name = e.Name;
17             busy = false;
18         }
19     }
20     public override bool Equals(object obj) {
21         if(obj is OrderedPair) {
22             OrderedPair op = (OrderedPair)obj;
23             if(op.a == this.a && op.b == this.b) return true;
24         }
25         return false;
26     }
27 }

```

Figure 7.9: The Element Ordered Pair Mediator's Implementation

instance on which the method is called, and the Element instance being removed. The method `AfterRemove` then removes the corresponding element from the other set that is in Bijection relationship. The join points are instrumented with the required events. When the events occur, the methods are invoked to keep the sets consistent.

### 7.2.2 Cross Product Mediator

The cross product mediator is similar to the bijection mediator but it maintains a cross product of two sets. Prism uses such a mediator to ensure that each model in a set of models is depicted in each

view in a set of views. As views or objects are added and deleted, the cross product is maintained in a consistent state. Moreover, Prism keeps corresponding pairs of objects consistent by creating a sub-mediator for each pair. The code shown in Figure 7.10 is an Eos implementation of the cross product mediator, with the `Set` type as defined in Figure 7.6 and the `Element` type as defined in Figure 7.7.

The `CrossProduct` mediator keeps references to the `Set` instances being mediated and a boolean variable `busy` to prevent recursive invocations (line 2). It declares a constructor (line 3–4) to store these references and to register itself with these `Set` instances using the implicit method `addObject`. It declares two bindings (lines 6–8 and 9–11). Similar to the bijection mediator, the first binding uses the pointcut execution(`public bool Set.Insert(Element)`) to identify the join point method execution of `Set.Insert` and invokes the method `AfterInsert` (lines 12–21) at that join point. It uses the *return*, *this*, and *args* pointcuts to associate arguments `r`, `s`, and `e` of the `AfterInsert` to the return value of the `Set.Insert` method, the instance on which the method is called, and the argument `Element` instance supplied to the method.

The method `AfterInsert` first inserts the corresponding element in the other set that is in cross product relationship. It then dispatches a new ordered pair sub-mediator (line 18) to keep the newly inserted elements consistent and then adds the sub-mediator to a list that it maintains.

Similar to the bijection mediator, the second binding (lines 9–11) uses the pointcut execution(`public bool Set.Remove(Element)`) to identify the join point method execution of `Set.Remove`, and invokes the method `AfterRemove` (lines 22–30) at that join point. It also uses the *return*, *this*, and *args* pointcuts to associate arguments `r`, `s`, and `e` of the `AfterRemove` to the return value of the `Set.Remove` method, the `Set` instance on which the method is called, and the `Element` instance being removed. The `AfterRemove` method in this case first removes the ordered pair sub-mediator from its list and then removes the corresponding element from the other `Set` instance in the cross product relationship.

The code shown in Figure 7.9 is an Eos implementation of the ordered pair sub-mediator. The mediator keeps references to the elements in the relationship, and a Boolean variable to prevent recursion (line 2). The constructor of this mediator (lines 3–5) stores the references to the elements

for later use and registers itself with the element instances to keep them consistent. The mediator also provides a method `isPair` (lines 8–11) to check whether this instance of the mediator is integrating the supplied elements, and overrides the default `Equals` method (lines 20–26) to check equality of two ordered pairs.

The mediator declares a binding (line 6) that uses the pointcut `fset(string Element.Name)` to identify the join point *when the field Name of Element class is being modified*, and invokes the method `AfterNameChange` (lines 12–19) at that join point. It uses the *this* pointcuts to associate argument `e` of the `AfterInsert` to the `Element` instance that is subject to the name change.

The cross product mediator implementation demonstrates that a complex system can be constructed using the proposed unified model. The language model does provide the ability to use mediators in an arbitrary fashion. In particular, I showed the ability to use nested mediators.

### 7.2.3 Advance Over Mediator-Based Design Style

The bijection and cross product implementation in Eos shows an advance over mediator-based design style in the following ways:

1. **Event Declaration:** Explicit event declaration, announcement, and registration requirements are removed. Instead, mediators implicitly select the events of interest using pointcuts and associate methods with these events using bindings. The unified language model implemented by Eos thus enables the application of mediator-based design style without requiring the components to change. Thus, source code control for components is not necessary. Even for components where the source code is not available, for example external library components, IL level weaver proposed in future work could potentially enable integration using mediator-based design style.

The downside of using implicit join point-pointcut mechanism is that the events exposed by the language semantics restrict the possibility to use mediator-based design style. However, in cases where the event of interest is not made available by the language semantics, and the source code of the component is available for modification as it would be for traditional

mediator-based design style to work, one could always re-factor the components to expose the right join point. The programming efforts of explicit event declaration, announcement, and registration are still saved. The limitation of the language model only precludes applying mediator-based design style in cases where source code is not available. In those cases traditional mediator-based design style is not applicable either.

2. **Reflective Information:** The ability to access the reflective information at the join point also gives more flexibility in the implementation of the integration logic in the mediator. In the traditional design style, one could only access the information associated with the event supplied explicitly. This information is typically supplied via an interface that all mediators implement, so that the components announcing events are only coupled with the mediator interface and not with the concrete mediators. This interface could potentially be expanded to include all the reflective information that is available at a join point in Eos, however, this would make event announcement code complex as it would need to explicitly pass this information. In favor of simpler event announcement code, the mediator interface is typically kept simpler reducing the information that a concrete mediator could access and modify.

The downside of using reflective information is that the mediators are now limited again by the language semantics. For example, the local variable at a join point is not available as part of the reflective information. Explicit event announcements can always pass this information. Once again, if the source code is available, re-factoring to expose the right reflective information comes to the rescue and the designer is no worse off than the traditional design style.

3. **Overriding Mediators:** In traditional mediator-based design style, the methods in mediators could only add behavior to the execution of original events. In the unified language model implemented by Eos, additive as well as overriding behavior is possible. An overriding method in mediators will be associated with the join points using around binding. This opens up the design space for mediators that can potentially control the behavior of all the colleagues.

For example, consider a mediator that integrates a file module and a GUI component drop

down numbered list. The integration requirement now is to control the file-read retry policy such that the number of retries is determined by the number selected in the drop down list. Writing such mediator becomes possible now without changing the file module or coupling it with the drop down list. One would just use an around binding to associate a method in the mediator with the join point execution of file.read. The method in the mediator would then implement the retry policy optionally invoking the original join point as many times as specified by the drop down list.

The downside of this new ability is that it makes reasoning about mediators potentially more difficult. This is an open question for aspect-oriented programming in general, orthogonal to current research and some efforts in this direction are being made.

### 7.3 Modularizing Crosscutting Concerns in Eos

To facilitate the discussions of the redesign efforts of the Eos compiler, it will be worthwhile to give a brief overview of the organization of the Eos compiler's source code, components and architecture. Section C.1 in Appendix, describes the implementation of the Eos compiler in details. The current implementation of Eos has seven components: Abstract Syntax Tree, Code Compiler, Code Weaver, Main Module, Parser, Runtime, and Utilities. All components create or manipulate programs stored as an abstract syntax tree. The Code Weaver component performs ten different transformations on the abstract syntax tree. Each transformation requires traversing the abstract syntax tree, called a compilation pass. Some of these traversals can be done in parallel, so in total five passes are required.

More of these passes can ideally be performed in parallel resulting in a further reduced compile-time. For example, the pass in which introductions and declare statements are collected can be merged with the source code parsing pass so that while parsing the parser also collects the introductions and declare statements. An inter-type declaration is also called an introduction. This merge, however, would either require that the introduction collection concern and declare statement collection concern is scattered inside the parsing concern or the parser allows introduction collector and

declare collector to observe it. These components register with the parser to get event notification. The parser invokes these concerns implicitly whenever it completes parsing an introduction and a declare statement. First design alternative is clearly non-modular. The second design alternative decouples parser from introduction and declare collector. It, however, leaves introduction and declare statement collector name-coupled with parser. As a result, the component Code Weaver cannot be independently compiled and tested.

### 7.3.1 Modularizing Introduction and Declare Statement Collection

A third design alternative is to use the mediator-based design structure to integrate the parser with the introduction collector and declare collector. Using this design alternative, the parser will declare and announce the event *CodeMemberIntroductionParsed* and *CodeMemberDeclareParsed*. Separate mediators will register with the *CodeMemberIntroductionParsed* and *CodeMemberDeclareParsed* events announced by the parser instance and on invocation; invoke the Collect method on the introduction collector and declare collector instance respectively. This design alternative achieves a complete decoupling of the components. It, however, leaves the event declaration and announcement concern scattered in the parser component. Using aspect-oriented programming to modularize this integration concern requires instantiation and instance-level advising emulation.

Using the unified model, these integration concern can be nicely modularized as shown in Figure 7.11 and 7.12 as classpects *ParIntMediator* and *ParDeclMediator*. These two mediators when introduced in the system *eliminate the second pass* of the compiler by merging inter type declaration and declare statement collection in between the parsing pass without introducing any name dependencies between the parser and the weaver component.

The *ParIntMediator* classpect (lines 1–14 in Figure 7.11) declares a constructor (lines 2–6) to integrate a *ASTParser* and an *IntroductionCollector* instance. The constructor stores references to both instances and binds itself to the parser instance using the implicit method *addObject*. The binding (lines 8–10) in the mediator selects the execution of the *ASTParser.CreateIntroduction* method and binds method *AfterIntroductionParsed* (line 11–13) to execute after it. It also binds the parameter *o* to the *AfterIntroductionParsed* method with the return value of the *AST-*

Parser.CreateIntroduction method. The AfterIntroductionParsed method calls the Collect method (line 12) on the stored IntroductionCollector instance passing it the introduction to collect.

Similarly, the ParDeclMediator classpect (lines 1–14 in Figure 7.12) declares a constructor (lines 2–6) to integrate an ASTParser and a DeclareCollector instance. The constructor stores references to both instances and binds itself to the parser instance using the implicit method addObject. The binding (lines 8–10) in the mediator selects the execution of all methods in the ASTParser that begin with CreateDeclare\*. The wildcard is used to collect creation of declare statement concern that is spread over three different methods that create declare parents, declare error, and declare warning constructs. The binding binds the method AfterDeclareParsed (line 11–13) to execute after the execution of these methods. It also binds the parameter o to the AfterDeclareParsed method with the return value of these methods. The AfterDeclareParsed method calls the Collect method (line 12) on the stored DeclareCollector instance passing it the declare statement to collect.

This redesign effort thus led to the saving a short pass resulting in shortened compile time. For the rest of the discussion, the numbering of passes will remain the same to facilitate ease of reference, so even though the introduction collection and declare collection passes are merged with the first pass and there is no second compilation pass now, we will continue calling introduction planner and successive passes third pass and so on.

### 7.3.2 Reordering Third and Fourth Full Pass

Encouraged by elimination of the second pass by merging it with the first parsing pass in a modular way, I explored the possibilities of merging the fourth pass with the first pass. Note that the third and fourth pass cannot be reordered without accounting for their interaction due to the data dependence between these passes; however, benefits of saving a full pass of the compiler was worth the efforts to give it a try.

To understand and identify the data dependence between the third pass and the fourth pass, understanding the nature of an introduction is necessary. An introduction identifies type in the programs by a pattern, and introduces a set of type members such as methods, fields, bindings, actions, etc. into them at compile-time. An implicit effect of introducing new type members into

a type is to also introduce new join point shadows in the type, that in turn need to be collected by the join point shadow collector. The join point collector in the fourth pass thus depends upon the output of the introduction planner in the third pass.

An optimization strategy might be to first collect the existing join point shadows and then in the introduction planner pass additively insert the join point shadows as new type members are added into type declarations. Two new classpects were introduced to implement this strategy. The first classpect, `ParJpCMediator` acts as mediator between the Parser and the Join Point Shadow collector module. Like the mediator between the Parser and the Introduction Collector, this classpect keeps references to the Parser and the Join Point Shadow Collector instances, register a method with the join points in the Parser class to be invoked when Parser finishes parsing join point shadows and calls the Join Point Shadow Collector instance to collect shadows.

The second classpect, `JpCIPMediator` acts as a mediator between the introduction planner and join point shadow collector. This classpect invokes the join point collector to collect join point shadows whenever the introduction planner introduces a new type member into a type declaration. I introduced other classpects between binding collector and parser, action collector and parser, specification checker and parser and action collector and parser to complete the merge of the fourth full pass with the first parsing pass.

## 7.4 Integrating ConcernCov and NUnit

Rajan and Sullivan [83] discuss a new approach for code coverage analysis, namely concern coverage, and a supporting tool for concern coverage. The supporting tool was build by integrating tool implementation with the NUnit GUI implementation. One of the goals of that project was to keep the tool implementation name-independent from GUI implementation and vice-versa so that no modification in the original code of NUnit is required to fit the new tool. The unified model of Eos was able to achieve the objectives successfully. In the rest of this section, I discuss the challenges and the solution presented by Eos.



### 7.4.1 Adding GUI Elements Transparently

The coverage tool uses the GUI interface of NUnit as the front-end. As shown in Figure 7.13 on Page 126, it was necessary to add two GUI elements, a new content tab and a button, to the NUnit interface. I added these GUI elements transparently to the NUnit interface using inter-type declarations as shown in the classpect in Subsection D.1.1 in Appendix D.

The classpect `GUIMixin` adds three fields, a button, a tab page and a text box, to the main GUI form of NUnit using inter-type declarations. It also adds two methods, `CInit` and `CInitComponents`, to perform additional initializations required due to these new controls. A .NET Framework form initializes its GUI components in a standard function called `InitializeComponent`. The `GUIMixin` classpect uses two bindings to make sure that the execution of `CInit` occurs after the constructor of the form, and execution of `CInitComponents` occurs after the execution of `InitializeComponent`. The `GUIMixin` classpect was able to add the required GUI elements to the NUnit interface without having to modify the NUnit source code.

### 7.4.2 Collecting Coverage Information

The assembly version 2.1.4 of NUnit uses static member `outWriter` and `errWriter` of the class `NUnit.UiKit.AppUI` throughout the execution as standard output and standard error respectively. The `AppUI.Init` function initializes these members. The NUnit form calls these functions to redirect the output and error from test cases to specific content tabs designed in the form. In the initial versions of `ConcernCov`, the tool instrumented the executable under test such that it would output information for coverage analysis at each join point on the standard output. The coverage tool then filters the standard output for coverage information, collects it, and passes the rest to the NUnit form. The classpect `IntroduceFilter` in Subsection D.1.2 in Appendix D does this filtering without modifying the NUnit code by.

The classpect `IntroduceFilter` binds a replace method around the execution of `AppUI.Init` function. The replace function, constructs a coverage information collector using the text writer supplied as argument to the `AppUI.Init` function, and passes the coverage information collector as the argu-

ment to the original function, thereby introducing a filter between the AppUI text writers and the form text writers to collect coverage information.

### 7.4.3 Displaying Coverage Information

The next integration requirement was to display the coverage information when the display coverage button is pushed. The involved components in this requirement are coverage tab, coverage collector, and display button. The coverage tab already implements the text writer interface, so I implemented coverage collector to take a text writer interface as input to emit coverage information. If the coverage collector was to be integrated after-the-fact transforming the coverage output form to the display form might have been necessary.

Another classpect `FormCovColMediator` shown in Subsection D.1.3 in Appendix D fulfils this requirement. This classpect provides a constructor that takes a form reference and a coverage collector reference as argument. It stores the references and registers itself with the form using `addObject` method. It also provides a binding that selects the execution of the method `displayResultButton_Click`. When a user clicks the display result button, the .NET environment run-time automatically calls the method `displayResultButton_Click`. The binding binds a method `Display` to the method `displayResultButton_Click`. The method `Display` calls the `Display` method on coverage collector instance, passing it the text box writer constructed from the coverage tab.

Finally, to put everything together when NUnit is invoked, the control and supplied arguments must be passed to the coverage tool to allow it to filter out its argument, compile and instrument the source code according to the `ConcernCov` directives. This integration is also performed using a classpect. This classpect, binds a method, `InitCodeCoverage`, before the execution of the entry method (`Main`) of NUnit. This method constructs an instance of the coverage tool, and passes the arguments to the `Main` method to the tool. After returning from `InitCodeCoverage`, the `Main` method of NUnit proceeds normally.

## 7.5 Limitations

### 7.5.1 Source Level Weaving: A Constraint

Currently Eos performs source level aspect weaving. It requires source code of a component or class to advise it. This requirement severely affects separate compilation. The production versions of Eos, essentially the bootstrapped version build using plain C#, are organized into a directory hierarchy corresponding to different components in the system. The directory structure is shown in Figure C.2 and the component structure is shown in Figure C.1. These components are compiled, tested, and debugged separately.

The new version of Eos built for this evaluation, retained the directory structure shown in Figure C.2, however, keeping the separation at source code level between component boundaries was difficult when they were integrated using classpects. To implement a connector, access to the source code of all involved components was required.

To address this concern, weaving approaches at the .NET's intermediate language (IL) level will be explored in future. The intermediate language called MSIL is interpreted by the Common Language Runtime (CLR) [44], a language and platform-neutral infrastructure. The CLI supports multiple programming languages and is controlled by a vendor neutral Common Language Infrastructure (CLI) [27] specification. Enabling weaving support at the CLR level thus also opens up opportunities for cross-language aspect-oriented programming.

### 7.5.2 Tool Support

Building large and complex systems without significant tool support is very difficult. Currently Eos project does not provide any tool support, such as integrated development environments for developing and debugging programs in Eos. The lack of debugging support is a significant problem. Aspect-oriented programming features are not widely understood i.e. chances of error are even greater. Lack of support for detecting precisely where the error is may make it very difficult to write programs. Writing debugging support for Eos is difficult due to the proprietary nature of the PDB format of Microsoft.

There is however, hope stemming from the community involvement. A student from University of Szeged in Hungary, Somkutas Pter, recently developed an add-in for Eos [80]. This add-in allows Visual Studio to compile Eos programs using the compiler developed at the University of Virginia. The add-in is still in preliminary stage, however, it shows signs of the impact of the Eos project internationally and the hopes of supporting tool development for the language model.

```

1  public class CrossProduct: CollectionBase {
2      bool busy = false ; Set s1, s2;
3      public CrossProduct(Set s1, Set s2){
4          addObject(s1); addObject(s2); this.s1 = s1; this.s2 = s2;
5      }

6      after execution(public bool SET.Set.Insert(Element))
7          && return(r) && this(s) && args(e):
8          AfterInsert(bool r, Set s, Element s);

9      after execution(public bool SET.Set.Remove(Element))
10         && return(r) && this(s) && args(e):
11         AfterRemove(bool r, Set s, Element s);

12     public void AfterInsert(bool r, Set s, Element e){
13         if(r && !busy) {
14             busy = true;
15             Element new_elem = new Element(e.Name);
16             if(s == this.s1) s2.Insert(new_elem);
17             else s1.Insert(new_elem);
18             List.Add( new OrderedPair(e, new_elem));
19             busy = false;
20         }
21     }

22     public void AfterRemove(bool r, Set s, Element e){
23         if(r && !busy) {
24             busy = true;
25             Element new_elem = new Element(e.Name);
26             List.Remove( new OrderedPair(e, new_elem));
27             if(s == this.s1) s2.Remove(new_elem);
28             else s1.Remove(new_elem);
29             busy = false;
30         }
31     }
32 }

```

Figure 7.10: The Set Cross Product Mediator's Implementation

```

1  public class ParIntMediator {
2      public ParIntMediator(ASParser parser, IntroductionCollector IntC) {
3          this.parser = parser;
4          this.IntC = IntC;
5          addObject(parser);
6      }
7      ASParser parser; IntroductionCollector IntC;
8      after execution(public virtual CodeMemberIntroduction
9          ASTParser.CreateIntroduction(..) && return(o):
10         AfterIntroductionParsed(CodeMemberIntroduction o);
11     public void AfterIntroductionParsed(CodeMemberIntroduction o) {
12         IntC.Collect(o);
13     }
14 }

```

Figure 7.11: The Parser Introduction Collector Mediator

```

1  public class ParDeclMediator {
2      public ParDeclMediator(ASParser parser, DeclareCollector DeclC) {
3          this.parser = parser;
4          this.DeclC = DeclC;
5          addObject(parser);
6      }
7      ASParser parser; DeclareCollector DeclC;
8      after execution(public virtual CodeMemberDeclare
9          ASTParser.CreateDeclare* (..) && return(o):
10         AfterDeclareParsed(CodeMemberDeclare o);
11     public void AfterDeclareParsed(CodeMemberDeclare o) {
12         DeclC.Collect(o);
13     }
14 }

```

Figure 7.12: The Parser Declare Collector Mediator

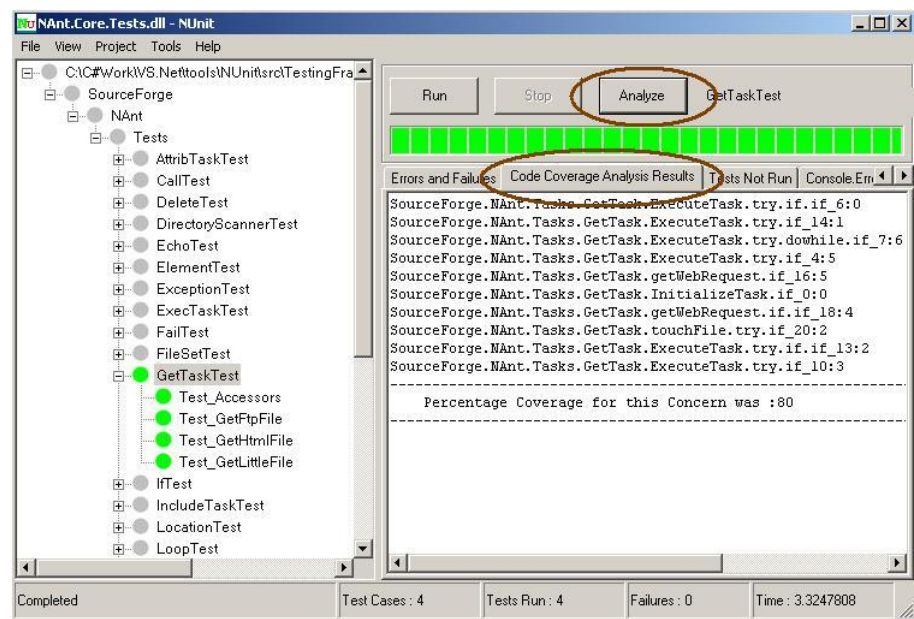


Figure 7.13: A Screen-shot of *ConcernCov* in Action

# Chapter 8

## Conclusions

---

*This is the end*

*My only friend, the end ...*

*– In The End by Nirvana.*

This chapter concludes this dissertation with a summary of contributions and a discussion of future work.

### 8.1 Claims and Contributions

At the beginning of this dissertation, I made some claims. In this section, I revisit and evaluate the support for each claim.

- *The distinctions between aspect and class and between advice and method are unnecessary in an AspectJ-like language model:* This claim was tested and found supported in Chapter 5 and 6. The proof presented in Chapter 6 showed that the distinction between class and aspects does not add any expressive power to the language design. The challenge problem discussed in Chapter 5 showed that this distinction in fact leads to a programming model in which use of advising as an invocation mechanism is limited to restrictive architectural styles.
- *A unification of object- and aspect-oriented program design is possible in an AspectJ-like language model:* This claim was supported by the unified model and proof of concept lan-



guage design, Eos, presented in Chapter 3. The unified language design eliminates non-orthogonal and asymmetric aspect and advice constructs. Instead, quantified binding emerged as the central concept instead of advice-like constructs in AspectJ-like languages.

- *Such a language design improves the conceptual integrity of the programming model of AspectJ-like languages:* The resulting language model presented in 3 showed the design unity that emerged because of the unification. The unification leads to a language design with fewer, more orthogonal and more regular language constructs.
- *In terms of higher-order crosscutting concerns the resulting programming model is more expressive than that of AspectJ-like languages with respect to Felleisen's notion of expressive power:* In Chapter 6, I proved that the unified model is more expressive than AspectJ-like languages. In particular, I showed that representation of certain concerns using AspectJ model requires global transformation of programs, whereas with unified model they do not.
- *The programming model further improves the modularization of integration and higher-order concerns over the implementations provided by AspectJ-like languages:* The simplicity achieved and the improved ability to modularize integration and higher-order concerns was evident repeatedly in Chapters 4 and 5. In these chapters, I evaluated the AspectJ model's and the unified model's capabilities to address two challenge problems: Separation of integration concerns and separation of higher-order concerns. The AspectJ model required work-arounds to represent both types of concerns. These work-arounds incur design overhead and either incur runtime overhead or require a yet undeveloped optimization techniques. The unified model was able to represent both types of concerns with ease without the need for work-arounds showing a clear improvement in the modularization of these concerns.
- *The unified language design further improves the compositionality of AspectJ-like languages:* In Chapter 5, I showed that due to the anonymity of advice, building multi-layered and other architectural styles using advising as an invocation mechanism required work-arounds. Further, the comparative analysis of the new language model showed its ability to preserve, at

the code level, the modular structure of a specification that featured multi-level integration concerns.

## 8.2 Limitations of Evaluation

The research that I described in this dissertation presents evidence that, at the programming language level, the separation of aspects and classes is harmful. It also raises the question: *Are aspects valuable as a separate conceptual category? Or should we instead think in terms of a single conceptual building block for program design?* The research also showed that unification of aspects and classes in an aspect-oriented language design is potentially valuable. To evaluate the research, I used separation of integration concerns and separation of higher-order concerns as challenge problems and compared the expressive power of the unified model with that of aspect-oriented languages. I also conducted some case studies where the unified language model showed benefits.

The evaluation provided in this dissertation, however, does not provide direct evidence that the unified model is beneficial beyond the challenge problems and case studies. The concerns that I have applied the unified model are largely co-ordination concerns, in that they co-ordinate the behavior of two or more components. Co-ordination concerns are extremely important class of concerns; however, they are not the only crosscutting concerns aspect-oriented programming addresses. Aspects are used for modularizations of other concerns like transaction management, thread pooling, security policy enforcement, etc. From the evaluation presented in this dissertation, it is not evident whether the unified model is beneficial in these cases.

I speculate, but have not yet systematically tested, that the benefits of the unified model will be perceived in the modularization of other concerns. For example, consider a system where a variety of security policies exist and a policy is applied to an object instance depending upon how it interacts with the outside world, e.g. local security policy for local object instances, domain security policy for object instances that only interact with the other entities on the local area network (LAN) and global security policy for the rest of the object instances. In such system, implementing security policy as an aspect with no support for instance-level advising will at the minimum complicate the

concern code with the code required to emulate aspect instances and instance-level advising. The unified model will not incur this design-time overhead.

### 8.3 Benefits: An Empirical Question

Another limitation of the evaluation is that, as of today, I designed and implemented all significant systems in which Eos was used. From an empirical standpoint, such case studies have limited value relative to the experiences that practicing engineers might have with the approach. Until and unless a significant user community is developed for Eos, answering empirical questions about benefits will be difficult. The robust language compiler and effective support will help in building a strong community is in progress. Eos is gaining visibility both nationally and internationally. It has an active forum where developers send support requests, but the community is still not able to support empirical broad studies. In addition, Eos is currently not open source. I believe that making it open source will help in generating further interests in the technology and in developing a strong user base for empirical studies.

The benefit of aspect-oriented programming techniques itself is open to question. On one hand, aspect-oriented techniques promise improved modularity. On the other, similar to the implicit invocation architectural style [39,96], AOP techniques make it hard to analyze software systems due to the loose coupling between components that increases the size of the state space. Aspect-oriented techniques go a step beyond implicit invocation architectural style by eliminating the need to explicitly declare, announce and register with events replacing them with language-defined events. Without a large data set to support empirical evaluation it is hard to judge whether it is better to have coupling (which complicates intellectual control) but explicit representations of interactions among components (which arguable eases it), or improved modularity (which arguably eases intellectual control) but at the cost of implicit relationships (that arguable make it much harder to manage complexity)?

My hypothesis is that to the extent that aspect-oriented methods turn out to be beneficial, a unified model is likely to be even more so. First, it will improve the ease-of-learning and ease-

of-use of aspect-oriented methods. From an understanding point of view, most of the code in the unified language model, looks and works like a traditional object-oriented program, for example. This symmetry could well make the transition from object-oriented to aspect-oriented design easier ultimately favoring the adoptability that led AspectJ designers to sacrifice a symmetric design in the first place.

Second, a unified model can further improve our ability to modularize systems, with benefits in evolvability, understandability, communication overheads in development, and parallel development. In particular, abstracting from the example of Chapter 5 it is evident that the unified design supports, as a practical possibility, modularization of higher-order crosscutting concerns. Whereas the first-level mediators modularized integration concerns that cut across the base objects, the Delayed Evaluation mediator modularized a concern that cut across the first-level integration concerns. The AspectJ de facto commitment to a two-level structure with one base level and one aspect level makes it hard to cleanly modularize such towers. In effect, higher-order concerns are all mapped to—and consequently scattered across and tangled into—the single available aspect layer.

The AspectJ rationale for separating classes and aspects was adoptability. Users asked for the separation because they wanted to be able to see and control a risky new construct in their systems. The non-integration of classes and aspects was thus an early tradeoff against unity for a better chance at adoption. Current AspectJ-like language designs thus still reflect that evolutionary path. Now that these languages are achieving significant adoption, with concomitant reductions in perceived risk, designers should revisit deep tradeoffs made to surmount early adoptability barriers. Ideally, reconsideration would occur before strong adoption creates irreversible lock-in. This research thus presents a timely analysis of a potentially important alternative language design and program structuring philosophy.

In terms of basic knowledge, this research has revealed deeper connections between AOP and the implicit invocation model [39] [24], suggesting that existing methods for reasoning about implicit invocation systems might have some potential to help us reason about aspect-oriented programs [110], [111]. In terms of practical impact, the work promises to bring to developers greater ease of understanding of the programming model through improved conceptual integrity in lan-

language design; to simplify learning and using aspects for system integration; and to enhance runtime performance when aspects are used in this way. Eos has also already shown itself to provide a good foundation for experimental research on future advances.

## 8.4 Future Work

There are many avenues of future work resulting from this research. They can be broadly divided into two categories: new language designs and new operating environments.

### 8.4.1 Language Design

The unification achieved by Eos shows that the reorganization of language constructs brings symmetry to the language model making the conceptual leap from current programming paradigm smaller. The resulting language model contains fewer aspect-oriented constructs: bindings, pointcuts, and inter-type declarations. The language design, however, leave these constructs second-class objects of the language in that they cannot be assigned to variables, passed as arguments, returned as results etc.

The key questions that remain to be answered are: What is the potential design space that remains unexplored due to bindings, pointcuts, and advices being second-class objects, in other words, is it even beneficial to make them first-class and if yes in what context? What are the benefits? Is it significant? What are the costs? What is the syntax and semantics of a language that makes bindings, pointcuts, and inter-type declaration first-class? What are the impacts of this new language on modular reasoning about aspect-oriented programming?

Another question that this work did not address was the interaction of different bindings in a program. Currently, the binding semantics guarantees the execution of all methods bound in an unspecified order. This model and the advising model of AspectJ have two problems with respect to real-time constraints. First, advising may violate the real-time constraints of the methods enclosing the advised join points. Second, execution in the unspecified order does not work in general in systems where based on the schedule optionally one or more tasks are performed. Using advising

as an invocation mechanism thus breaks real-time guarantees. In future, I will explore solution to these problems. The solution to the first problem will be based on the concepts of static analysis to determine if interaction of bindings with each other and with the join points is consistent with real-time requirements. The solution to the second problem will be based on language features to provide a mechanism to re-institute real-time guarantees for advising.

My work showed that generalization of the aspect-oriented language model at once made it simpler and expressive in some dimensions. One might ask whether there are other dimensions where such generalizations are possible. Separate from my dissertation research, I have explored one such dimension—separation of pointcut matching and compile-time actions at the matched join points. This generalization, called generalized concern processing, is described in my research work published in Aspect-Oriented Software Development Conference 2005 [83]. An application of the generalized concern processing towards code coverage analysis is also described. In future, I hope to develop the approach further and explore other applications.

#### **8.4.2 New Operating Environments**

The requirements for embedded systems and ad hoc wireless and sensor networks are becoming exceedingly complicated. The need for these systems is being felt in applications such as biological and chemical detection systems for homeland security, home health care for the elderly and disabled, learning environments, etc. These systems are poised to become crucial parts of the critical infrastructure of the society and I believe that they should be designed and developed with utmost care and should meet demanding criteria for dependability, evolvability, and practical utility.

The software development for this new domain needs to accommodate changes in the environment, e.g. constantly changing network topology as opposed to fixed network topology, as well as limited battery power as opposed to unlimited power, mobile nodes as opposed to static nodes, and additional requirements for real-time operation and dependability. The implementations of most of these concerns using traditional programming models end up being scattered across and tangled into system structures, making their design and evolution a difficult problem, and increases the likelihood of inconsistencies and errors thus compromising their dependability. Aspect-oriented

programming can help in the modular representation of these new concerns, with tangible benefits in the key dimensions: ease of design, understanding, evolution, and improved dependability.

The current research shows benefits of the language model via a concrete language implementation realized for C# and .NET framework. An open question is whether the same language implementation can be realized in other operating environments as well. Most current aspect-languages, including Eos, are not designed with requirements of embedded systems and wireless networks in mind. Their requirements affect both language semantics and the implementation strategies that can be used by underlying tools. For example, implementations need to account for memory and power constraints, semantics (e.g. advice invocation order), and other characteristics of the underlying embedded infrastructure. Language design and implementation thus needs to be reconsidered for this domain. One possible direction to explore seems based on building on top of an event infrastructure for mobile ad hoc and sensor networks.

A join point in the aspect-oriented terminology is similar to an event in the implicit invocation paradigm. In fact, in Eos, advising is largely implemented as join points represented as events and advices as handlers that may at run-time register with join points to add additional behavior. This implementation strategy makes Eos naturally scalable to distributed settings, dependent upon the availability of an appropriate infrastructure to provide event abstractions.

To enable aspect language support for mobile ad hoc and sensor networks, the first step is to build an event infrastructure for this environment. The topology of an ad hoc network changes over time as nodes leave, join, and move around the network. Such networks do not satisfy the design assumptions of many existing distributed event infrastructures, such as known connectivity, known upper bound on the number of participating nodes, and known resource requirements [73]. Designing an event infrastructure in absence of these assumptions will be a challenge and the lessons learned from it will be directly applicable towards providing real-time infrastructure for such networks.

## **8.5 Summary**

In summary, the main contribution of this work is a novel synthesis of object- and aspect-oriented programming language constructs and design methods, including the Eos language, a robust compiler, and evidence that suggests that this synthesis has potentially significant benefits in aspect-oriented program design. In particular, I showed that separate aspect and advice are not essential; instead, quantified binding is at the core of aspect-oriented programming. This work creates a timely opportunity to rethink the two-level ontology for aspect-oriented programs that the original separation of aspects and objects entailed and to reconcile the differences between asymmetric and symmetric language models.



# Appendix A

## Eos Language Manual

---

```
1 class Tracing {  
2   pointcut tracedCalls():execution(* *(..))&&!within(Tracing);  
3   static after tracedCall(): Trace();  
4   void Trace() { System.Console.WriteLine(Method called); }  
5}
```

Figure A.1: Tracing in Eos

Eos extends C# [28] to support modularization of crosscutting concerns. A concern is a dimension in which some design decision is made [78]. Some examples of concerns are execution trace policy, use of common thread pool, security policy enforcement, caching etc. A concern is crosscutting if it cannot be realized in traditional object-oriented designs in a modular way – that is, if any implementation of the concern involves scattered and tangled code. Scattered means not localized in a module but fragmented across a system. Tangled means mingled with code for other concerns. Eos is based on a unified model for aspect-languages.

Let us look at a simple example. I would like to trace the execution of every method in a program. In traditional object-oriented languages like C#, I would have to add a print statement in every method. This is an example of scattered and tangled concern. Here the execution trace concern is scattered across all methods of the program and tangled with the concerns implemented by these methods. Next, I will look at how this concern is modularized using Eos.

Similar to AspectJ, an aspect-oriented extension to Java, Eos adds a new concept, join point,

to the language semantics of C#. A join point is a point in program execution exposed by the semantics of the language to possible modifications. The execution of a method is an example of a join point. Eos also adds three new constructs to C#: pointcuts, bindings and inter-type declarations. A pointcut is a predicate that selects a subset of join points in the program. In the listing shown in Figure A.1, a named pointcut `tracedCalls` (lines 2-3) selects a subset of join points.

The pointcut expression has two parts. The first part `execution(* *(..))` selects all method execution join points. The second part `within(Tracing)` selects all join points inside the class `Tracing`. These two parts are composed using the operators `&&!`. The resulting set of join points consists of all join points that are in the first set but not in the second set.

I have thus identified all points in the program where I would like to add a print statement to trace. I add the print statement to the method `Trace` (Line 4). The remaining work now is to ensure that this method is called at all points I selected before. The binding construct (Line 3) makes sure that the method `Trace` is called after each join point in the selected set is executed. This is an improved realization of the tracing concern in that it is now modularized in the *Tracing* module and not scattered and tangled across the system.

```

1 public class Bit{
2     bool value;
3     public Bit(){value = false;}
4     public void Set(){value = true;}
5     public bool Get(){return value;}
6     public void Clear(){value= false;}
7 }
8 public class MakeCloneable{
9     declare parents: Bit : System.ICloneable;
10    introduce in Bit{
11        public virtual object Clone(){
12            return this.MemberwiseClone();
13        }
14    }
15 }
```

Figure A.2: Inter-Type Declaration in Eos

Like AspectJ, the inter-type declaration allows a third party type to add additional members and interfaces to a type without involvement of the type itself. Figure A.2 presents a simple example.

The class `MakeCloneable`(Lines 8-15) makes two compile-time modifications to the class `Bit`(Lines 1-7). First, using the declare parent construct (Line 9) it adds the interface `System.ICloneable` to the class `Bit`. Second, using the inter-type declaration (Lines 10-14), it adds the method `Clone` (Lines 11-13) to the class. The rest of this chapter describes parts of the Eos language in more details. The next section describes the join points in Eos.

## A.1 Join Points

Table A.1: The `Eos.Runtime.IJoinPoint` Interface

Property	Description
This	Returns the value of <i>this</i> at the join point. Returns null in the case of static join points.
Target	Returns the target of the join point. Returns null in cases where there is no target.
ReturnValue	Returns the return value at the join point if it is a valid return value otherwise null.
Args	Returns the list of arguments of the join points. Returns an array of length zero if there are no arguments.
Kind	Returns the string representation of the join point kind.
Signature	Returns the signature of the join point. Please see the interface <code>Eos.Runtime.Signature.ISignature</code> for more details.
StaticPart	Returns the static representation at the join point. Please see <code>Eos.Runtime.IStaticPart</code> for more details.
Location	Returns the Location of the join point. Please see <code>Eos.Runtime.ISourceLocation</code> for more details.

Table A.2: The `Eos.Runtime.ISourceLocation` Interface

Property	Description
EnclosingType	The <code>System.Type</code> representation of the type that encloses the join point.
FileName	The name of the file that contains the join point.
Line	The line number at which the join point occurs.

A join point in Eos allows access to a set of reflective information. This information is accessed using a set of API's. In particular, the reflective information is exposed via an the interface `Eos.Runtime.IJoinPoint` shown in Table A.1. This interface provides properties to access

Table A.3: The `Eos.Runtime.IStaticPart` Interface

Property	Description
Signature	The signature of the join point.
Location	The source location of the join point.
Kind	The string representation of the join point kind.

Table A.4: The `Eos.Runtime.Signature.ISignature` Interface

Property	Description
Name	Name of the signature.
Modifiers	Modifiers of this signature.
DeclaringType	Type of the enclosing classpect.
DeclaringTypeName	Name of the enclosing class.
ToString method	Returns a string representation of this signature.

executing object (This), the target object (Target), the return value (ReturnValue), the arguments (Args), the join point kind (Kind), the signature at the join point (Signature), the static representation of the join point (StaticPart) and the location of the join point (Location). The signature property returns an object that implements the interface *Eos.Runtime.Signature.ISignature* shown in Table A.4. The StaticPart property returns an object that implements the interface *Eos.Runtime.IStaticPart* shown in Table A.3. The location property returns an object that implements the interface *Eos.Runtime.ISourceLocation* shown in Table A.2.

Eos language definition exposes exception handling, method execution, field access, object initialization, property access, and static initialization as join points for extension. I will discuss each of them briefly in the rest of this section.

### A.1.1 Exception Join Point

Table A.5: The `Eos.Runtime.Signature.ICatchClauseSignature` Interface

Property	Description
ParameterType	Returns the type of the exception caught.
ParameterName	Returns the name of the exception caught.

When exception handling code (catch clauses) executes in a program. The argument of the catch clause (the exception being caught) is considered as the argument available at the join

point and it can be accessed using the property *Args* of the interface *Eos.Runtime.IJoinPoint*. No value is returned from a exception handler join point, so its return type is considered to be void. The property *Eos.Runtime.ReturnValue* is thus *null* for this join point. The signature object returned by *Eos.Runtime.IJoinPoint.Signature* property for this join point also implements the *Eos.Runtime.IJoinPoint.Signature.ICatchClauseSignature* interface shown in Table A.5.

### A.1.2 Execution Join Point

Table A.6: The *Eos.Runtime.Signature.IMethodSignature* Interface

Property	Description
<i>ReturnType</i>	The return type of the method.
<i>ParameterTypes</i>	The list of the types of the parameters at the join point.
<i>ParameterNames</i>	The list of the names of the parameters at the join point.

The method execution join point occurs when the body of a method is called. It is not at the call site, but in the method body. For example, in the following code the method execution join point shadow for *Hello.SayHello()* is from line 2–4. The signature object returned by *Eos.Runtime.IJoinPoint.Signature* property for this join point also implements the *Eos.Runtime.IJoinPoint.Signature.IMethodSignature* interface shown in Table A.6.

```

1 public class Hello{
2     public static void SayHello(){
3         System.Console.WriteLine(`Hello World`);
4     }
5     public static void Main(string[] args){
6         SayHello();
7     }
8 }
```

Table A.7: The `Eos.Runtime.Signature.IFieldSignature` Interface

Property	Description
FieldType	The <code>System.Type</code> representation of the type of the field.

### A.1.3 Field & Property Access Join Points

The field access join point occurs when a field is referenced or set. For example in the following code, on line 5 the field `HelloString` is referenced, so it is a field get join point shadow.

```

1 public class Hello{
2     static string HelloString = ``Hello World;
3     public static void Main(string[] args){
4         System.Console.WriteLine(Hello.HelloString);
5     }
6 }
```

In the following code, on line 4 the field `HelloString` is set, so it a field set join point shadow. There is also a field get join point shadow on line 5.

```

1 public class Hello{
2     static string HelloString;
3     public static void Main(string[] args){
4         Hello.HelloString = ``Hello World;
5         System.Console.WriteLine(HelloString);
6     }
7 }
```

The property access join points are similar to field access join points. A property get join point occurs when the get accessor for a property executes. The value returned from this join point is of the same type as the property. The property set join point occurs when the set accessor for a property executes. The value returned from this join point is considered to be void. The variable bound to *value* is considered as the argument available at the join point and it can be accessed using the property *Args* of the interface `Eos.Runtime.IJoinPoint`. The signature object returned

by *Eos.Runtime.IJoinPoint.Signature* property for the field set and get join points and property set and get join points also implement the *Eos.Runtime.IJoinPoint.Signature.IFieldSignature* interface shown in Table A.7.

#### A.1.4 Object & Static Initialization Join Points

Table A.8: The *Eos.Runtime.Signature.IConstructorSignature* Interface

Property	Description
ParameterTypes	The list of the types of the parameters at the join point.
ParameterNames	The list of the names of the parameters at the join point.

The object initialization join point occurs when an instance constructor actually executes after the chained base (*:base(..)*) and this constructors (*:this(..)*). The object being constructed is the currently executing object, and so may be accessed with the property *This* of the interface *Eos.Runtime.IJoinPoint*. No value is returned from a instance constructor execution join point, so its return type is considered to be void. The property *ReturnValue* thus returns *null* for this join point.

Static initialization join point occurs when the type constructor for a class executes. No value is returned from a type constructor execution join point, so its return type is considered to be void. The property *ReturnValue* thus returns *null* for this join point as well. The signature object returned by *Eos.Runtime.IJoinPoint.Signature* property for the object initialization and static initialization join points also implement the *Eos.Runtime.IJoinPoint.Signature.IConstructorSignature* interface shown in Table A.8.

## A.2 Pointcuts

A pointcut expression in Eos can be a named pointcut expression or either of the pointcut designators. The pointcut expressions can be composed using *and(&&)*, *or(||)* and *not(!)* operators. The rest of this section describes each of the pointcut designators briefly.

- *Execution*: The syntax of the execution pointcut descriptor is *execution(method-pattern)*. It selects method execution join points where the return type, fully qualified name and the arguments of the method matches with the method pattern. For example, the pointcut expression *execution(public void SomePackage.any(..))* selects the execution of all public methods in the package *SomePackage* that return void.
- *Field Get/Set*: The syntax of the field get pointcut descriptor is *fget(field-pattern)* and the syntax of the field set pointcut descriptor is *fset(field-pattern)*. The field get pointcut descriptor selects the field reference join points where the field type and the fully qualified name of the referenced field matched the field pattern. The field set pointcut descriptor selects the field assignment join points where the field type and the fully qualified name of the field being assigned matched the field pattern.
- *Handler*: The syntax of the handler pointcut descriptor is *handler(type pattern)*. It selects exception handler join points where the type of the exception being caught matches the type pattern. For example, the pointcut expression *handler(System.ArgumentException)* selects all exception handler join points (i.e. catch clauses) where the exception *System.ArgumentException* is being caught.
- *Initialization*: The syntax of the initialization pointcut descriptor is *initialization(constructor pattern)*. It selects constructor execution join points, where the signature of the join points matches the constructor pattern.
- *Property Get/Set*: The syntax of the property get pointcut descriptor is *pget(field-pattern)* and the syntax of the property set pointcut descriptor is *pset(field-pattern)*. The property get pointcut descriptor selects the property reference join points where the property type and the fully qualified name of the referenced property matched the field pattern. The property set pointcut descriptor selects the property assignment join points where the property type and the fully qualified name of the property being assigned matched the field pattern.



- *Static Initialization*: The syntax of the static initialization pointcut is `staticinitialization(type pattern)`. It selects the type constructor of each type whose signature matches type pattern.
- *Lexical Pointcuts*: There are two lexical pointcut descriptors that select join points based on the type scope and the method scope. The within pointcut descriptor's syntax is `within(type pattern)`. It selects join points that are defined in the type that matches type pattern. For example, the pointcut expression `within(SomePackage.SomeClass)` selects all join points in the class `SomeClass`. The withincode pointcut descriptor's syntax is `withincode(method pattern)`. It selects join points that are defined in all methods whose signature matches the method pattern. For example, the pointcut expression `withincode(SomePackage.SomeClass.SomeMethod)` selects all join points in the method `SomeMethod`.

### A.3 Bindings

The binding construct in Eos is similar to the event-method binding constructs of implicit invocations systems. When a method is bound to a join point, whenever the join point is reached that method is called. The syntax of the binding declaration is shown in the listing below. It has four parts. The first, `opt-static`, specifies whether a binding is static. Non-static bindings result in instance-level advising: selective advising of the join points of individual object instances. Static bindings affects all instances of advised classes. The second part of a binding (`after/before/around`) states when the advising method executes: after, before, or around. The third part, pointcuts, selects the join points at which an advising method executes. The final part specifies the advising method.

Binding-declaration

```
: opt-static after pointcuts : call method-bindings;
| opt-static before pointcuts : call method-bindings;
| opt-static type around pointcuts : call method-bindings;
;
```

Method-bindings

```

    : method-binding , method-binding
    | method-binding
    ;
Method-binding
    : IDENTIFIER(opt-formal-parameter-list)
    ;
Opt-static
    : Empty
    | static
    ;

```

A binding provides a list of methods to execute at a join point, in the order specified. A binding can also pass reflective information about the join point to the methods invoked, by binding method parameters to reflective information using the AspectJ pointcut designators such as `this`, `target`, `args`, etc. As with `around` advice in AspectJ, an Eos-U method bound `around` is allowed to return a value, so `around` bindings must be declared with return types. Methods subject to binding have to follow certain rules. First, a method must be accessible in the class declaring a binding. Second, a method bound before or after a join point can have only `void` as a return type. Third, a method bound around a join point must have a return type that matches the return type at the join point. For example, if method `Foo` is bound around the execution join point `execution(public int *.Bar())`, then it must return `int`.

### A.3.1 Around Bindings

Table A.9: Return Types of Join Points

Join Points	Return Type
Method execution	Return type of the method.
Field Set	<code>void</code>
Field Get	Type of the field
Constructor execution	<code>void</code>

Around bindings can bind methods to execute in place of execution, field get, field set, and instance constructor execution/object initialization join points. An `around` advice may not advice

an exception handler join point. Syntax of an around binding is as follows:

```
return-type around pointcut-expression: Method-Handler-Name(Formal Parameters);
```

It is a compile time error to bind a method returning void to a join point with a non-void return type. It is a run-time error (`System.InvalidCastException`) to return null or objects in a method bound around that may not be casted to the return type of the join point. The return types of some join points are given in Table A.9.

# Appendix B

## Eos Grammar

---

Eos is an extension of the C# language. This appendix presents relevant parts of the C# grammar and the extensions to it by the Eos language. The base grammar is adopted from the C# specification [28]. I use the Extended Backus-Naur Form (EBNF) notation to represent the grammar.

### B.1 Tokens

abstract	as	add	assembly	base	bool
break	byte	case	catch	char	checked
class	const	continue	decimal	default	delegate
do	double	else	enum	error	event
explicit	extern	false	finally	fixed	float
for	foreach	get	goto	if	implicit
in	int	interface	internal	is	lock
long	namespace	new	null	object	operator
out	override	params	private	protected	public
readonly	ref	return	remove	sbyte	sealed
set	short	sizeof	stackalloc	static	string
struct	switch	throw	true	try	typeof
uint	ulong	unchecked	unsafe	ushort	using
virtual	void	volatile	warning	while	

Figure B.1: Keywords in C#

The tokens in Eos are keywords, single character operators and or punctuations and literals. Figure B.1 shows the keywords of the base language (C#). To support aspect-oriented constructs

action	adviceexecution	after	args
around	aroundptr	any	before
call	cflow	cflowbelow	conditional
declare	execution	fget	fset
handler	initialization	instancelevel	introduce
iteration	joinpoint	parents	pget
pointcut	pointcutid	preinitialization	pset
returning	role	static initialization	target
this	throwing	warning	within
withincode			

Figure B.2: Additional Keywords in Eos

---

*Open brace, close brace, open bracket, close bracket, open parens, close parens, dot, comma, colon, semicolon, tilde, plus, minus, band, assign, less then operator, greater then operator, bitwise and, bitwise or, star, percent, division, carret, interr, increment operator, decrement operator, shift left, shift right, less then equal to, greater then equal to, not equal to, and, or, multiply assign, divide assign, modulo assign, add assign, subtract assign, shift left assign, shift right assign, and assign, xor assign, or assign, pointer operator*

---

Figure B.3: Operators and Punctuations

some new keywords were added. Figure B.2 shows these new keywords in Eos. Figure B.3 shows the single character operators and punctuations. Like C# Eos can have *int*, *float*, *double*, *decimal*, *character*, *string* as literals. The next section describes complete production rules of Eos. I have used the Extended Backus-Naur Form (EBNF) notation to describe the production rules. In the rest of the chapter, the notation  $\{ \}$  means 0 or more of the symbol,  $[ ]$  means 0 or 1 of that symbol.

## B.2 Production Rules

Like C# language, the start symbol of the Eos grammar is the *compilation-unit*. A *compilation-unit* defines the overall structure of a source file. A compilation unit consists of zero or more *using-directives* followed by zero or more *global-attributes* followed by zero or more *namespace-member-declarations*.

```

compilation-unit ::=
    {using-directive} {global-attribute} {namespace-member-declaration}

```

```
using-directive ::=  
    using identifier semicolon  
    using identifier assign identifier semicolon
```

A namespace-declaration consists of the keyword `namespace`, followed by a namespace name and body, optionally followed by a semicolon.

```
namespace-declaration ::=  
    namespace qualified-identifier namespace-body [semicolon]
```

```
qualified-identifier ::=  
    identifier { dot identifier }
```

```
namespace-body ::=  
    open-brace {using-directive} {namespace-member-declaration} close-brace
```

A *namespace-member-declaration* is either a *namespace-declaration* or a *type-declaration*.

```
namespace-member-declaration ::=  
    namespace-declaration  
    | type-declaration
```

A compilation unit or a namespace body can contain namespace-member-declarations, and such declarations contribute new members to the underlying declaration space of the containing compilation unit or namespace body.

A type-declaration is a *classpect-declaration*, a *struct-declaration*, an *interface-declaration*, an *enum-declaration*, or a *delegate-declaration*. For the purpose of this appendix, only the *classpect-declaration* is discussed in detail.

```
type-declaration ::=  
    classpect-declaration  
    | struct-declaration  
    | interface-declaration
```

```

| enum-declaration
| delegate-declaration

```

A classpect-declaration is a type-declaration that declares a new classpect.

classpect-declaration:

```

{attribute} {modifier} class identifier [base]
open-brace {classpect-members} close-brace [semicolon]

```

A classpect-declaration consists of an optional set of attributes, followed by an optional set of modifiers, followed by the keyword `class` and an identifier that names the classpect, followed by an optional base specification, followed by a open-brace, followed by the classpect-members, followed by a close-brace, optionally followed by a semicolon.

classpect-members::=

```

binding-declaration
| constant-declaration
| constructor-declaration
| destructor-declaration
| event-declaration
| field-declaration
| indexer-declaration
| inter-type-declaration
| method-declaration
| operator-declaration
| pointcut-declaration
| property-declaration
| type-declaration

```

A classpect member can be either of the binding, constructor, destructor, event, field, indexer, inter-type declaration, method, operator, pointcut, property or another inner type declaration. This production is a modified version of the production for class members in the C# specification [28]. Three new types of members are added, namely bindings, inter-type declarations and pointcuts.

These new members add aspect-oriented capabilities to the classpects. I will discuss the grammar of each of these members in detail in the rest of this section.

```
binding-declaration ::=
    [static] after pointcut-expression colon method-bindings semicolon
  |[static] before pointcut-expression colon method-bindings semicolon
  |[static] type around pointcut-expression colon method-bindings semicolon
```

```
method-bindings ::=
    method-binding {comma method-binding}
```

```
method-binding ::=
    identifier open-paren [formal-parameter-list] close-paren
```

```
pointcut-expression ::=
    pointcut-descriptor
  | bang pointcut-expression
  | pointcut-expression and pointcut-expression
  | pointcut-expression or pointcut-expression
  | open-parens pointcut-expression close-parens
```

```
pointcut-descriptor ::=
    named-pointcut
  | execution open-paren method-pattern close-paren
  | fget open-paren field-pattern close-paren
  | fset open-paren field-pattern close-paren
  | pget open-paren field-pattern close-paren
  | pset open-paren field-pattern close-paren
  | initialization open-paren constructor-pattern close-paren
  | staticinitialization open-paren name-pattern close-paren
  | handler open-paren name-pattern close-paren
  | within open-paren name-pattern close-paren
```



```

    | withincode open-paren method-pattern close-paren
    | this open-paren identifier close-paren
    | target open-paren identifier close-paren
    | return open-paren identifier close-paren
    | aroundptr open-paren identifier close-paren
    | joinpoint open-paren identifier close-paren
    | args open-paren identifier close-paren

named-pointcut ::=
    pointcut identifier open-paren [argument list] close-paren

method-pattern ::=
    [modifiers] type-pattern name-pattern open-paren [parameter-pattern] close-paren

constructor-pattern ::=
    [modifiers] name-pattern open-paren [parameter-pattern] close-paren

field-pattern ::=
    [modifiers] type-pattern name-pattern

```

A binding declaration consists of an optional static keyword (to denote static bindings), followed by either of the keywords `after`, `before`, and `around` (to denote after, before, and around bindings), followed by a pointcut expression, followed by a colon, followed by one or more method bindings, followed by a semicolon. A method binding is an identifier followed by an open parenthesis followed by optional parameter list followed by a close parenthesis. The identifier in the method binding denotes the name of the handler method and the optional parameter list is the list of parameters of the method. The method named in the method binding must be accessible in the lexical scope of the binding.

A pointcut expression in Eos consists of either a single pointcut descriptor (PCD), or a pointcut expression composed with not operator, or two pointcut expressions composed with and operator,

or two pointcut expressions composed with or operator or a pointcut expression parenthesized.

A pointcut descriptor can be a named pointcut or either or the execution, fget, fset, pget, pset, initialization, staticinitialization, handler, within, within, withincode, this, target, return, aroundptr, joinpoint, args, conditional and iteration pointcut descriptor with corresponding patterns.

A method pattern is of the form: optional modifiers, followed by a type-pattern, followed by a name pattern, followed by an open parenthesis, followed by a parameter pattern, followed by a close parenthesis. A constructor pattern is of the form: optional modifiers, followed by a name pattern, followed by an open parenthesis, followed by a parameter pattern, followed by a close parenthesis. A field pattern is of the form: optional modifiers, followed by a type pattern, followed by a name pattern. An unspecified modifier is equivalent to the wildcard, i.e. it matches all modifiers. A '..' in place of optional parameters matches to all parameter sequences.

```
inter-type-declaration ::=
    introduce in type-pattern
    open-brace { classpect-members } close-brace
```

An inter-type declaration is of the form: an introduce keyword, followed by an in keyword, followed by a type pattern followed by an open brace, followed by zero or more classpect members. The type pattern selects the type declarations in which the classpect members are inserted at compile time.

```
pointcut-declaration ::=
    abstract-pointcut-declaration
    named-pointcut-declaration

abstract-pointcut-declaration ::=
    abstract pointcut identifier semicolon

named-pointcut-declaration ::=
    pointcut identifier pointcut-expression semicolon
```

A pointcut declaration is either an abstract pointcut declaration or a named pointcut declaration. An abstract pointcut declaration is of the form: an abstract keyword, followed by a pointcut keyword, followed by a semicolon. The identifier represents the name of the abstract pointcut. An abstract pointcut can only be declared in an abstract classpect or an interface. A named pointcut declaration is of the form: the pointcut keyword, followed by an identifier, followed by a pointcut expression, followed by a semicolon. The identifier here also represents the name of the pointcut (hence named).

# Appendix C

## Eos Implementation

---

This appendix provides an overview of the implementation of the Eos compiler.

### C.1 The Eos Compiler



Figure C.1: Components of Eos

The Eos compiler has seven components as shown in Figure C.1. The AST component handles abstract syntax tree building and provides a representation of the abstract syntax tree nodes in the compiler. The Parser, CodeWeaver, CodeGenerator, and CodeCompiler parse, weave, generate and compile source code. The Utils provides string manipulation, argument construction and similar functions. The ConsoleUI component is the main driver of the compiler and it invokes other components. Total size of the compiler is around 50K line of code and it is organized as shown in Table C.1) among components.

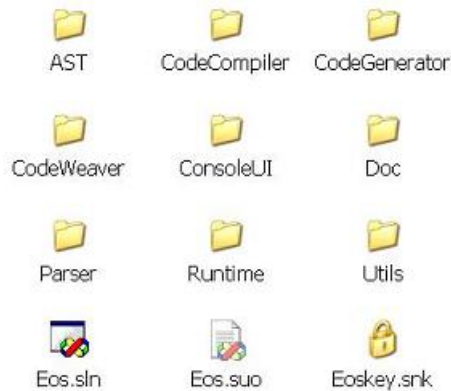


Figure C.2: Directory Structure of Eos

As shown in table C.1, the AST component has 224 classes in separate files. Out of 224 classes, 9 are used internally (Eos.Utils and Eos.AST namespace). The Code Weaver uses approximately 120 classes to access and transform the representations of join points, pointcuts, bindings and classpects and to perform pointcut matching. The Parser and the Code Generator use all types excluding the 9 internal classes to construct and generate abstract syntax tree nodes. The parser, weaver, and generator are highly coupled with the AST component; however, they are kept name-independent of each other.

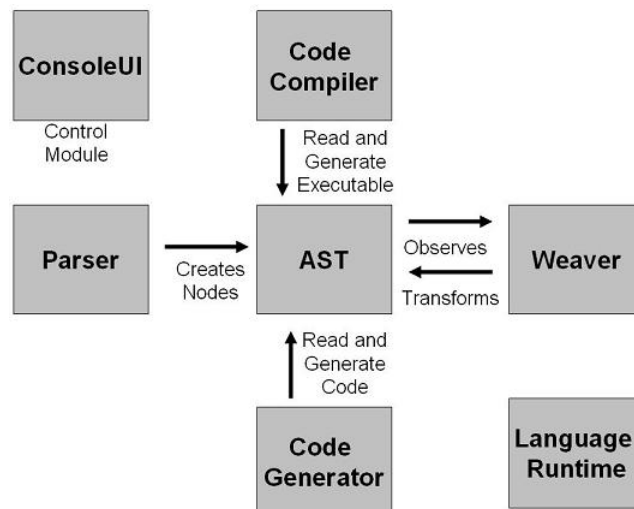


Figure C.3: The Architecture of the Eos System

The architecture of the compiler is shown in Figure C.3. It is essentially blackboard architecture.

Table C.1: Line of Code Distribution for Eos

Component	Namespaces	Number of Files	Line of Code
Eos.AST	Eos.AST.	1	82
	Eos.AST.Base.Attributes.	6	434
	Eos.AST.Base.Expressions.	46	2486
	Eos.AST.Base.Statements.	58	3017
	Eos.AST.Base.TypeMembers.	32	3399
	Eos.AST.Crosscuts.JoinPoints.	18	2551
	Eos.AST.Crosscuts.Matching.	6	218
	Eos.AST.Crosscuts.Patterns.	5	508
	Eos.AST.Crosscuts.Pointcut.	24	1172
	Eos.AST.Crosscuts.TypeMembers.	20	2134
	Eos.AST.Utils.	8	4507
	<b>Total</b>	<b>224</b>	<b>20508</b>
Eos.Utils	Eos.Utils.	17	4293
Eos.Parser	Eos.Parser.	11	16032
	Eos.Parser.Common.	8	413
	Eos.Parser.Crosscuts.	2	136
	Eos.Parser.CSharp.	3	2550
	<b>Total</b>	<b>24</b>	<b>19131</b>
Eos.CodeGenerator	Eos.CodeGenerator.	3	3499
Eos.CodeCompiler	Eos.CodeCompiler.	2	220
Eos.CodeWeaver	Eos.CodeWeaver.	12	1172
Eos.ConsoleUI	Eos.ConsoleUI.	3	275
Eos.Runtime	Eos.Runtime.	10	1494
	Eos.Runtime.Signature.	16	471
	<b>Total</b>	<b>26</b>	<b>1965</b>
	<b>Grand Total</b>	<b>311</b>	<b>51063</b>

The abstract syntax tree (AST) component is the blackboard and other components in the system read and write to it. The Parser creates nodes on the blackboard. The Code Generator and Compiler are readers that read the AST and generate another representation. The Weaver observes changes to the AST.

To decouple observers and announcers, the design of AST uses implicit invocation mechanisms. As shown in Figure C.4, *ASTAnnouncer*, a sub-component of AST declares events for the rest of the world. An event of interest related to AST nodes are announced through *ASTAnnouncer*. An event of a given type can be announced by multiple subjects.

## C.2 Compilation Passes in Eos

To construct the abstract syntax tree Parser takes the C# source code as input, calls the lexer to extract tokens out of it, and on matching a node calls the constructor of that node type in the AST component. The Parser requires only one pass on the complete syntax tree to complete this process.

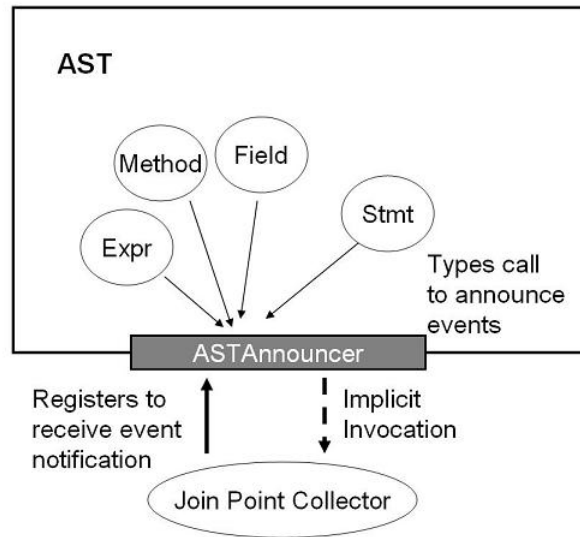


Figure C.4: AST Event Announcer

The Code Weaver performs following ten different transformations or passes on the abstract syntax tree (in the order in which they should be performed). In some cases visiting only up to the type declaration level is necessary. A complete syntax tree visit is called Full Pass and a visit till only the type declaration level is called Short Pass. Note that saving a full pass can significantly reduce compilation time on large systems.

1. Introduction Collection: IntroductionCollector pass collects all the inter-type declarations in the program. The inter-type declarations introduce crosscutting members into other types. As a result definition of other types might change, therefore this pass has to be done in the very beginning. This is a short pass.
2. Declare Collection: DeclareCollector pass collects all the declare statements in the program. The declare statements are of three types: declare parents, declare errors, and declare warnings. The declare parents statement changes the inheritance hierarchy of a type and declare error and warning instruct the compiler to report warnings or errors on encountering specific patterns in the code. These patterns are specified using pointcut expressions. The declare parents statement can also potentially affect the definition of types in the program on which other passes depend. This is a short pass.

3. Introduction Planner: The introduction planner pass processes all the inter-type declaration requests collected in the introduction collection pass. This pass also accesses the declare parents statements collected by the declare collector pass to ensure correct behavior. This is a short pass.
4. Specification Checker: This pass verifies that the resulting syntax tree satisfies the some very basic properties after introduction of inter-type declarations. This is a full pass.
5. Join Point Shadow Collector: This pass collects all join point shadows in the program. A join point shadow is a static location in the program, where a dynamic join point might occur. It is essential to perform this pass after introduction planner pass to collect join points in inter-type declarations as well. This is a full pass. In particular, to collect method call, field set & get, property set & get expression throw join points visiting up to the expression nodes is necessary.
6. Binding Collector: This pass collects all method bindings in the program. It is essential to perform it after introduction collector and planner as an inter-type declaration is allowed to introduce a binding into another type.
7. Declare Processor: This pass processes all declare statements collected previously. This is a long pass.
8. Action Collector: This pass collects all the directives for generalized concern processing. This is a short pass.
9. Crosscutting Infrastructure Builder: This pass prepares the necessary crosscutting infrastructure for weaving. For example, it creates the necessary instantiation and binding mechanisms in classpects. This is a short pass.
10. Join Point Weaving Pass: This pass performs the necessary instrumentation at join points to introduce invocation mechanisms to execute methods that are bound to that join point. This is a long pass.



Some of these transformations are performed in parallel. As a result the abstract syntax tree is visited five times by the weaver out of which only two passes are full passes. In total including the code generation and parsing passes, the current compiler implementation performs four full passes and three short passes in the following order:

1. *Full Pass*: Parsing
2. *Short Pass*: Introduction Collector and Declare Collector
3. *Short Pass*: Introduction Planner
4. *Full Pass*: Specification Checker, Join Point Collector, Binding Collector, Declare Processor, and Action Collector
5. *Short Pass*: Crosscutting Infrastructure Builder
6. *Full Pass*: Join Point Weaving
7. *Full Pass*: Code Generation

Further reduction in compilation time was achieved by applying Eos programming model to Eos implementation.

### C.3 Limitations and Future Extensions

Current implementation of Eos has the following limitations.

- **Robust Test Suite**: Currently testing methodology employed by the Eos project is ad hoc in nature. A more systematic unit-test and regression-test suite is badly needed.
- **Language Manual**: The documentation provided with Eos is limited to release notes and research publications. A detailed language manual is essential to encourage adoption.
- **Source Level Weaving**: Currently Eos compiler operates on source code only. It does not weave binaries. A possible enhancement would be to add MSIL level weaving to Eos. This

enhancement will enable cross-language weaving. By cross-language weaving, I mean writing classpects in one language and allowing them to affect classpects written in other language.

- **Debugging Support:** The output executable in Eos does not emit any debugging information inside it. This makes it difficult to debug Eos programs. Due to the proprietary nature of the debugging standard it is difficult to solve this problem.
- **C# Specification 2.0:** Eos extends C# language as defined in the specification 1.0. The support for new C# language specification 2.0 is needed.

# Appendix D

## Additional Source Listings

---

This appendix provides listings that were too long to be interleaved with the text of the dissertation.

### D.1 Concern Coverage Tool- NUnit Integration Code

#### D.1.1 Introducing GUI Elements into NUnit

```
1 public class GUIMixin{
2     introduce in NUnit.Gui.NUnitForm {
3         public System.Windows.Forms.TabPage coverage;
4         public System.Windows.Forms.RichTextBox coverageTab;
5         public System.Windows.Forms.Button displayResultButton;
6         public void CInit(){
7             this.coverage.SuspendLayout();
8             coverageTab.Enabled = true;
9             DisplayResultButton.Enabled = false;
10            this.resultTabs.Controls.Add(this.coverage);
11            this.groupBox1.Controls.Add(this.displayResultButton);
12        }
13        public void CInitComponents(){
14            /// Construct and initialize the tab page
15            this.coverage = new System.Windows.Forms.TabPage();
```

```
16         this.coverage.Controls.Add(this.coverageTab);
17         this.coverage.Location = new System.Drawing.Point(4, 22);
18         this.coverage.Name = "coverage";
19         this.coverage.Size = new System.Drawing.Size(423, 247);
20         this.coverage.TabIndex = 4;
21         this.coverage.Text = "Code Coverage Analysis Results";
22
23         /// Construct and initialize the tab text box
24         this.coverageTab = new System.Windows.Forms.RichTextBox();
25         this.coverageTab.Dock = System.Windows.Forms.DockStyle.Fill;
26         this.coverageTab.Font = new System.Drawing.Font("Courier New", 9F,
27                                                         System.Drawing.FontStyle.Regular,
28                                                         System.Drawing.GraphicsUnit.Point,
29                                                         ((System.Byte)(0)));
30         this.coverageTab.Location = new System.Drawing.Point(0, 0);
31         this.coverageTab.Name = "coverageTab";
32         this.coverageTab.ReadOnly = true;
33         this.coverageTab.Size = new System.Drawing.Size(423, 200);
34         this.coverageTab.TabIndex = 0;
35         this.coverageTab.Text = "";
36         this.coverageTab.WordWrap = false;
37         //Construct and initialize the coverage button
38         this.displayResultButton = new System.Windows.Forms.Button();
39         this.displayResultButton.Location =
40             new System.Drawing.Point(181, 16);
41         this.displayResultButton.Name = "displayResultButton";
42         this.displayResultButton.Size = new System.Drawing.Size(74, 31);
43         this.displayResultButton.TabIndex = 5;
44         this.displayResultButton.Text = "&Analyze";
45         this.displayResultButton.Click +=
46             new System.EventHandler(this.displayResultButton_Click);
47     }
```

```

48     private void displayResultButton_Click(object sender,
49                                           System.EventArgs e){
50         coverageTab.Clear();
51     }
52 }
53 static after objectinitialization( public NUnit.Gui.NUnitForm(..) && this(form):
54     CallCInit(NUnit.GUI.NUnitForm form);
55 public void CallCInit(NUnit.GUI.NUnitForm form){
56     form.CGUIInit();
57 }
58 static after execution(private NUnit.Gui.NUnitForm.InitializeComponent()) &&
59     this(form): CallCInitComponents(NUnit.GUI.NUnitForm form);
60 public void CallCInitComponents(NUnit.GUI.NUnitForm form){
61     form.CInitComponents();
62 }
63 }

```

### D.1.2 Collecting Coverage Information

```

1  public class IntroduceFilter{
2      public static CoverageInfoCollector coverageCollector;
3      static around execution(public static void
4          NUnit.UiKit.Init(Form, TextWriter, TextWriter) &&
5          args(Form, tW, TextWriter)) && aroundptr(p)
6          && joinpoint(jp):
7          Replace(TextWriter tW, Eos.Runtime.IJoinPoint jp, Eos.Runtime.AroundADP p);
8
9      static void Replace(TextWriter tW,
10         Eos.Runtime.AroundADP p, Eos.Runtime.IJoinPoint jp) {
11         coverageCollector = new CoverageInfoCollector(tW);
12         jp.Args[1] = coverageCollector;
13         p.InnerInvoke(); // Invoke the underlying join point

```

```
14     }
15 }
```

### D.1.3 Displaying Coverage Information

```
1 public class FormCovColMediator{
2     public FormCovColMediator(
3         NUnit.Gui.NUnitForm form,
4         CoverageInfoCollector collector){
5         this.form = form;
6         this.collector = collector;
7         addObject(form);
8     }
9     NUnit.Gui.NUnitForm form;
10    CoverageInfoCollector collector;
11    after execution(private void
12        NUnit.Gui.NUnitForm.displayResultButton_Click($...$)):
13        Display();
14    public void Display(){
15        collector.Display(
16            new TextBoxWriter(form.coverageTab ));
17    }
18 }
```

## Bibliography

---

- [1] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting Object Interactions Using Composition Filters. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791, pages 152–184. Springer-Verlag, 1994.
- [2] Jonathan Aldrich. Open modules: Modular reasoning about advice. In *2005 European Conference on Object-Oriented Programming (ECOOP'05)*, page To Appear, July 2005.
- [3] James H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 187–209, London, UK, 2001. Springer-Verlag.
- [4] AspectJ programming guide.  
<http://www.eclipse.org/aspectj/>.
- [5] Lodewijk Bergmans and Mehmet Akşit. Principles and design rationale of composition filters. In Filman et al. [34], pages 63–95.
- [6] Jonas Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 5–6, New York, NY, USA, 2004. ACM Press.
- [7] Antonio Brogi and Jean-Marie Jacquet. On the expressiveness of coordination via shared dataspace. *Sci. Comput. Program.*, 46(1-2):71–98, 2003.

- [8] Fredrick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison Wesley, Reading, Mass., second edition, 1995.
- [9] Glen Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. muabc: A minimal aspect calculus. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004: Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224, London, August 2004. Springer.
- [10] Avi Bryant and Robert Feldt. AspectR - simple aspect-oriented programming in Ruby, Jan 2002.
- [11] Craig Chambers, Bill Harrison, and John Vlissides. A debate on language and tool support for design patterns. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 277–289, New York, NY, USA, 2000. ACM Press.
- [12] Christina Chavez and Carlos Lucena. A theory of aspects for aspect-oriented software development. In *SBES 2003 – XVII Brazilian Symposium on Software Engineering*, pages 19–32, October 2003.
- [13] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *SPLAT 2003: Software engineering Properties of Languages for Aspect Technologies at AOSD 2003*, March 2003. Available as Computer Science Technical Report TR03-01a from <ftp://ftp.cs.iastate.edu/pub/techreports/TR03-01/TR.pdf>.
- [14] Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. Technical Report 03-13, Iowa State University, Department of Computer Science, October 2003.
- [15] Adrian Colyer and Andrew Clement. Large-scale aosd for middleware. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 56–65, New York, NY, USA, 2004. ACM Press.



- [16] C. Constantinides and T. Skotiniotis. Reasoning about a classification of cross-cutting concerns in object-oriented systems. In Pascal Costanza, Günter Kniesel, Katharina Mehner, Elke Pulvermüller, and Andreas Speck, editors, *Second Workshop on Aspect-Oriented Software Development of the German Information Society*. Institut für Informatik III, Universität Bonn, February 2002. Technical report IAI-TR-2002-1.
- [17] Constantinos Constantinides, Therapon Skotiniotis, and Maximilian Stoerzer. AOP considered harmful. In *1st European Interactive Workshop on Aspects in Software (EIWAS 2004)*, Berlin, Germany, September 2004.
- [18] Constantinos A. Constantinides and Tzilla Elrad. Composing concerns with a framework approach. In Ziad Choukair, editor, *Proc. Int'l Workshop on Distributed Dynamic Multiservice Architectures (ICDCS-2001)*, Vol. 2, pages 133–140, April 2001.
- [19] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Commun. ACM*, 9(9):671–678, 1966.
- [20] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.
- [21] Edsger W. Dijkstra. Notes on Structured Programming. circulated privately, April 1970.
- [22] Edsger W. Dijkstra. On the role of scientific thought. *EWD 477*, August 1974.
- [23] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [24] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about implicit invocation. *SIGSOFT Software Engineering Notes*, 23(6):209–21, November 1998.
- [25] R. Douence and M. Südholt. A model and a tool for event-based aspect-oriented programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.
- [26] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on*

- Aspect-oriented software development*, pages 141–150, New York, NY, USA, 2004. ACM Press.
- [27] ECMA. *Standard-335: Common Language Infrastructure (CLI) Specification*, December 2001.
- [28] ECMA. *Standard-334: C# Language Specification*, 2002.
- [29] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner Jr. Seesoft — a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, pages 957–68, November 1992.
- [30] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.
- [31] Matthias Felleisen. On the expressive power of programming languages. In N. Jones, editor, *Proceedings of the third European symposium on programming on ESOP '90*, volume 432, pages 134–151, New York, NY, USA, 1990. Springer-Verlag New York, Inc.
- [32] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75, 1991.
- [33] Robert E. Filman, Stuart Barrett, Diana D. Lee, and Ted Linden. Inserting ilities by controlling communications. In Filman et al. [34], pages 283–295.
- [34] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.
- [35] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-oriented Software Development*, pages 21–35. Addison-Wesley Professional, 2004.
- [36] Andreas Gal, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. AspectC++: Language proposal and prototype implementation. In Kris De Volder, Maurice Glandrup, Siobhán

- Clarke, and Robert Filman, editors, *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, October 2001.
- [37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [38] Alessandro Garcia, Uir&#225; Kulesza, Cl&#225;udio Sant’Anna, Carlos Lucena, Eduardo Figueiredo, and Arndt von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD ’05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 3–14, New York, NY, USA, 2005. ACM Press.
- [39] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM ’91: Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I*, pages 31–44, London, UK, 1991. Springer-Verlag.
- [40] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993. Large-scale architecture patterns: pipes and filters, layering, black-board systems.
- [41] Adele Goldberg. Introducing the Smalltalk-80 System. *Byte*, 6(8):14–26, August 1981.
- [42] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
- [43] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
- [44] Jennifer Hamilton. Language integration in the common language runtime. *SIGPLAN Not.*, 38(2):19–28, 2003.

- [45] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.
- [46] William Harrison, Harold Ossher, and Peri Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. In Lodewijk Bergmans, Johan Brichau, Peri Tarr, and Erik Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, March 2003.
- [47] Robert Hirschfeld. Aspect-oriented programming with AspectS. In Mehmet Akşit and Mira Mezini, editors, *Net.Object Days 2002*, October 2002.
- [48] Robert Hirschfeld. Aspects - aspect-oriented programming with squeak. In *NODE '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.
- [49] Radha Jagadeesan, Alan Jeffrey, and James Riely. A calculus of untyped aspect-oriented programs. In *Proc. European Conf. Object-Oriented Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 415–427. Springer-Verlag, 2003.
- [50] Shmuel Katz. Diagnosis of harmful aspects using regression verification. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL: Foundations Of Aspect-Oriented Languages*, pages 1–6, March 2004.
- [51] Alan Kay and Adele Goldberg. Personal dynamic media. *IEEE Computer*, 10(3):31–41, March 1977.
- [52] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Budapest, Hungary, June 2001.

- [53] Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering*, pages 481–90, Boston, Massachusetts, 17–23 May 1997. IEEE.
- [54] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.
- [55] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.
- [56] Howard Kim. AspectC#: An aosd implementation for c#. Technical Report TCD-CS-2002-55, Department of Computer Science, Trinity College, Dublin, 2002.
- [57] Stephen Kleene. *Introduction to Metamathematics*. Number 1 in Bibliotheca mathematica. North-Holland, 1952. Revised edition, Wolters-Noordhoff, 1971.
- [58] Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, 1974.
- [59] P. Koopmans. Sina user’s guide and reference manual. Technical report, Dept. of Computer Science, University of Twente, 1995.
- [60] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. *SIGSOFT Softw. Eng. Notes*, 29(6):137–146, 2004.
- [61] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Muller-Pedersen, and Kristen Nygaard. Abstraction mechanisms in the beta programming language. In *POPL ’83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 1983. ACM Press.

- [62] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.
- [63] John Lamping. The role of base in aspect-oriented programming. In Cristina Videira Lopes, Andrew Black, Liz Kendall, and Lodewijk Bergmans, editors, *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)*, June 1999.
- [64] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM*, 44(10):39–41, 2001.
- [65] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-Oriented Programming with Adaptive Methods. Technical Report NU-CCS-2001-05, College of Computer Science, Northeastern University, Boston, MA, February 2001.
- [66] Karl Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.
- [67] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Co., Boston, MA, USA, 1995.
- [68] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, 1974.
- [69] Cristina Videira Lopes and Sushil Krishna Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 15–26. ACM Press, 2005.
- [70] Bruce J. MacLennan. *Principles of programming languages: design, evaluation, and implementation (2nd ed.)*. Holt, Rinehart & Winston, Austin, TX, USA, 1986.
- [71] Joshua Marshall, Doug Orleans, and Karl J. Lieberherr. DJ: Dynamic structure-shy traversal in pure Java. Technical report, Northeastern University, May 1999.

- [72] Hidehiko Masuhara and Gregor Kiczales. Modular crosscutting in aspect-oriented mechanisms. In Luca Cardelli, editor, *ECOOP 2003—Object-Oriented Programming, 17th European Conference*, volume 2743, pages 2–28, Berlin, July 2003.
- [73] Ren Meier, Barbara Hughes, Raymond Cunningham, and Vinny Cahill. Towards real-time middleware for applications of vehicular ad hoc networks. In Lea Kutvonen and Nancy Alonistioti, editors, *Distributed Applications and Interoperable Systems: 5th IFIP WG 6.1 International Conference, DAIS 2005*, pages 1–13. Springer-Verlag GmbH, jun 2005.
- [74] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
- [75] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.
- [76] Peter Norvig. Design patterns in dynamic programming. In *Object World 96*, Boston, MA, May 1996.
- [77] H. Ossher and P. Tarr. Multi-dimensional separation of concerns using hyperspaces. IBM Research Report 21452, IBM, April 1999.
- [78] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–8, December 1972.
- [79] David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–66, March 1985.
- [80] Somkutas Pter. Ect (eos compiler tool): an add-in for visual studio, 2005. URL: <http://ect.jate.hu/>.
- [81] Hridesh Rajan and Kevin Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference*

- held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 297–306, New York, NY, USA, 2003. ACM Press.
- [82] Hridayesh Rajan and Kevin Sullivan. Need for instance level aspect language with rich pointcut language. In Lodewijk Bergmans, Johan Brichau, Peri Tarr, and Erik Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, mar 2003.
- [83] Hridayesh Rajan and Kevin Sullivan. Aspect language features for concern coverage profiling. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 181–191, New York, NY, USA, 2005. ACM Press.
- [84] Hridayesh Rajan and Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 59–68, New York, NY, USA, 2005. ACM Press.
- [85] Steven P. Reiss. Connecting tools using message passing in the field environment. *IEEE Softw.*, 7(4):57–66, 1990.
- [86] Jon G. Riecke. Fully abstract translations between functional languages. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–254, New York, NY, USA, 1991. ACM Press.
- [87] Daniel Sabbah. Aspects: from promise to reality. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 1–2, New York, NY, USA, 2004. ACM Press.
- [88] Kouhei Sakurai, Hidehiko Masuhara, Naoyasu Ubayashi, Saeko Matsuura, and Seiichi Komiya. Association aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 16–25, New York, NY, USA, 2004. ACM Press.
- [89] Olaf Spinczyk, Andreas Gal, and Wolfgang Schroeder-Preikschat. AspectC++: an aspect-oriented extension to the c++ programming language. In *CRPITS '02: Proceedings of the*



- Fortieth International Conference on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.
- [90] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 2000.
- [91] Gregory T. Sullivan. Advanced programming language features for executable design patterns. Lab Memo AIM-2002-005, MIT Artificial Intelligence Laboratory, 2002.
- [92] Kevin Sullivan, Lin Gu, and Yuanfang Cai. Non-modularity in aspect-oriented languages: integration as a crosscutting concern for aspectj. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 19–26, New York, NY, USA, 2002. ACM Press.
- [93] Kevin J. Sullivan, Joanne Bechta Dugan, John Knight, et al. Galileo: An advanced fault tree analysis tool, 1997. URL: <http://www.cs.virginia.edu/~ftree/index.html>.
- [94] Kevin J. Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *The Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, page To Appear, Sept 2005.
- [95] Kevin J. Sullivan, Ira J. Kalet, and David Notkin. Evaluating the mediator method: Prism as a case study. *IEEE Transactions on Software Engineering*, 22(8):563–579, August 1996.
- [96] Kevin J. Sullivan and David Notkin. Reconciling environment integration and component independence. *SIGSOFT Software Engineering Notes*, 15(6):22–33, December 1990.
- [97] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–68, July 1992.
- [98] Clemens Szyperski. Component-oriented programming: A refined variation on object-oriented programming. *The Oberon Tribune*, 1(2), December 1995.

- [99] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
- [100] P. Tarr and H. Ossher. Hyper/J user and installation manual. Technical report, IBM T. J. Watson Research Center, 2000.
- [101] Peri Tarr, Harold L. Ossher, William H. Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.
- [102] Anne S. Troelstra. *Metamathematical investigation of intuitionistic arithmetic and analysis*, volume 344 of *Springer LNM*. Springer-Verlag, Berlin, 1973.
- [103] David B. Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 158–167. ACM Press, 2003.
- [104] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.
- [105] Daniel von Dincklage. Making patterns explicit with metaprogramming. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, pages 287–306, New York, NY, USA, 2003. Springer-Verlag New York, Inc.
- [106] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 127–139, New York, NY, USA, 2003. ACM Press.
- [107] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

- [108] Niklaus Wirth. *Systematic Programming: An Introduction*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1973.
- [109] William Wulf. Trends in the design and implementation of programming languages. *IEEE Computer*, 13(1):14–24, 1980.
- [110] Jia Xu, Hridesh Rajan, and Kevin Sullivan. Aspect reasoning by reduction to implicit invocation. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL: Foundations Of Aspect-Oriented Languages*, pages 31–36, March 2004.
- [111] Jia Xu, Hridesh Rajan, and Kevin J. Sullivan. Understanding aspects via implicit invocation. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, 20-25 September 2004, Linz, Austria, pages 332–335. IEEE Computer Society, 2004.