A CONCERN-ORIENTED APPROACH TO SOFTWARE ARCHITECTURE

THÈSE N^{O} 2796 (2003)

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Institut des systèmes informatiques et multimédias

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Mohamed Mancona KANDÉ

Diplom-Ingenieur, Technische Universität Berlin, Allemagne
de nationalité suisse et originaire de Bâle (BS)

acceptée sur proposition du jury:

Prof. A. Strohmeier, directeur de thèse
Prof. S. Ducasse, rapporteur
Prof. R. France, rapporteur
Prof. R. Keller, rapporteur
Prof. A. Wegmann, rapporteur

Lausanne, EPFL
2003

Abstract

A major cause of many complications in the field of software architectures is the lack of appropriate abstractions for separating, combining and encapsulating concerns of various kinds in architectural descriptions.

Architectures of most complex software-intensive systems involve concerns that inherently crosscut the natural boundaries of the elements composing the architecture descriptions. Crosscutting concerns intersect the common modularity of systems as prescribed by their architecture descriptions, by traversing both the components and connectors, i.e., the relationships among the components. Crosscutting concerns are critical aspects of many architectural problems. However, architectural descriptions written in special-purpose languages (ADLs) like Wright, Darwin, Rapide and Acme should support descriptions of multiple structures, which include diagrams, models and views, that intentionally address different kinds of concerns. ADLs should show how various concerns affect each other in architectural designs; they should also allow one to identify, analyze and elaborate architectural concerns that cut across several software components, such as transactions, security, load balancing, synchronization, reuse, customization, scalability, etc.; they should, but they do not.

This dissertation presents a new approach to software architecture that is suitable for supporting concern-oriented development and documentation of architectures for software-intensive systems. This approach allows for creating and documenting a multidimensional software structure that is referred to as *concern-oriented software architecture*; it provides new mechanisms for encapsulating individual concerns into independent architectural constructs. The ultimate goal of this new approach is to provide support for achieving *design by concerns* all through the development and description of software architectures. Moving towards this goal, we present a particular concern-oriented architectural framework called Perspectival Concern-Spaces (PCS). The *PCS Framework* offers a flexible and extensible means a) for supporting advanced separation of concerns in architectural design, and in the construction and evolution of software-intensive systems; and b) for filling the gap between architectural descriptions and modern software development artifacts.

To show the feasibility of the proposed approach, we provide new modeling techniques that are used to describe and apply an aspect-oriented architectural pattern, called the On-demand Remodularization pattern. We give several examples of how the PCS Framework can be used to integrate concern-oriented architectural documentations with mainstream software development artifacts.

Kurzfassung

Wenn softwareintensive Systeme komplizierter werden, gewinnt die Softwarearchitektur an Bedeutung als vitales Element für den Bau solcher Systeme. Eines der Ziele im Gebiet der Softwarearchitektur ist die Verbesserung der Art, wie komplexe Softwareinfrastrukturen so organisiert werden können, dass die Kosten der Softwareproduktion sinken. Ein anderes der vielfältigen Ziele ist die Förderung von Software-Bauelementen, die wiederverwendbar, stabil und entwicklungsfähig sind.
Eine der Hauptursachen der vielen Schwierigkeiten bei der Verfolgung dieser Ziele ist das Fehlen von geeigneten Abstraktionen, um verschiedenartige Anliegen (concerns) in Architekturbeschreibungen zu trennen und zu kombinieren. Querlaufende Anliegen sind in manchen Architekturproblemen von kritischer Bedeutung. Allerdings sollten Architekturen, die in Spezialsprachen (ADLs) wie z.B. Wright, Darwin, Rapide und Acme geschrieben sind, die Darstellung von multiplen Strukturen mit Diagrammen, Modellen und Ansichten (views) unterstützen, die beabsichtigen, verschiedenartige Anliegen zu erfassen. ADL's sollten zeigen, wie in Architekturbeschreibungen unterschiedliche Anliegen sich gegenseitig beeinflussen; sie sollten auch ermöglichen, Architekturanliegen zu identifizieren, zu analysieren und Architekturanliegen auszuarbeiten, die quer durch mehrere Softwarekomponenten gehen, wie z.B. Sicherheit, load balancing, Synchronisierung, Wiederverwendung, Anpassung an Kundenbedürfnisse, Skalierbarkeit etc; sie sollten das tun, aber sie tun es nicht.

Diese Dissertation zeigt eine Anliegen-orientierte Vorgehensweise in der Softwarearchitektur, die geeignet ist sowohl die Entwicklung wie auch die Beschreibung von Architekturen für softwareintensive Systeme zu unterstützen. Diese Vorgehensweise, Anliegen-orientierte Softwarearchitektur (Concern-oriented Software Architecture) genannt, ermöglicht es, multidimensionale Softwarestrukturen zu schaffen und beschreiben. Sie bietet neue Mechanismen, um spezifische Anliegen in eigenständige Architekturkonstrukte einzubauen. Das oberste Ziel dieser Vorgehensweise ist, Unterstützung für das anliegen-orientierte Design (design by concerns) während der ganzen Entwicklung und Beschreibung von Softwarearchitekturen zu gewährleisten.

Ich lege ein spezielles anliegen-orientiertes Framework vor, das Perspectival Concern-Spaces (PCS) genannt wird. Das PCS offeriert eine flexible und erweiterungsfähige Möglichkeit, um a) die höhere Trennung von Anliegen im Achitekturdesign sowie den Bau und die Entwicklung von softwareintensiven Systemen zu unterstützen; und b) die Lücke zwischen Architekturbeschreibungen und modernen Softwareentwicklungs-Artefakten zu schliessen. Das PCS-Framework ermöglicht es, Softwarearchitektur als eine multidimensionale Struktur zu behandeln, die einen “Vorrat an Konzepten” anbietet, aus denen ein oder mehrere Softwaresysteme gebaut werden können. Eine solche multidimensionale Softwarestruktur wird Architektur-Anliegenbereich, (architecture concern-space) genannt. Dank

ihrer Fähigkeit, die Schaffung von Architektur für multiple Softwaresysteme zu unterstützen, ist ein Architektur-Anliegenbereich (architecture concern-space) für Softwarearchitekten das, was ein Anwendungsframework (application framework) für die Entwickler von Softwareanwendungen ist.

Um die Realisierbarkeit des vorgeschlagenen Vorgehens zu zeigen, gebe ich Beispiele, die zeigen, wie das PCS-Framework zur Integration von Architekturbeschreibungen in mainstream Softwareentwicklungsartefakte benutzt werden kann und wie die Unterscheidung (Trennung) verschiedener Anliegen im Design und bei der Konstruktion und Entwicklung von softwareintensiven Systemen gefördert werden kann.

To Bettina, Fatou and Aicha

Acknowledgements

Different people have contributed in different ways to help me complete this work over the years. It is now time to express my gratitude to all of them, including those persons whom I might forget to mention. First of all, I would like to thank Alfred Strohmeier for accepting to be my thesis supervisor and allowing me to perform my work at the Software Engineering Lab. He made it possible for me to visit various international conferences, and the Advanced Enterprise Middleware Group at Thomas J. Watson Research Center, IBM Research, where I had the opportunity to work with top-quality researchers. He often asked me many challenging questions, which I believe to be very interesting for exploring the new area of concern-oriented software architecture. Alfred, thank you for all the support you gave me.

Secondly, I would like to thank all the jury members: Stefan Ducasse, Robert France, Rudolf Keller, and Alain Wegmann for having taken the time and effort to review my thesis and to serve on my examination board. Their comments have improved the quality of the final manuscript.

Third, I am deeply indebted to Stefan Tai. His careful comments on my work were enormously constructive and have improved the quality of this thesis. I would also like to thank Isabelle Rouvellou, Stanley Sutton and Peri Tarr for encouraging me and supporting my work on multidimensional separation of concerns (MDSOC) in software architecture. Isabelle, thank you very much for giving me the opportunity to have so many heated, but very nice discussions on MDSOC and UML with your team at Watson, especially with Stefan and Stan.

I am truly grateful to Rich Hilliard. Rich always found time to review my work and a part of this thesis. His thorough comments on the IEEE standard, IEEE-Std-1471, were particularly helpful and have improved the quality of the thesis.

I would like to thank my “roomie” and “predecessor-in-line of PhDs” Shane Sendall for making my time at the lab enjoyable and nice. I thank Shane for discussions on various topics of my work and new ideas. I would like to thank him also for his review and useful comments on this thesis.

Special thank to Valentin Crettaz for his implementation work on numerous tools we have developed together to validate this research. Valentin started convincing me to promote my research through the realization of tool support when he was an undergraduate student. I first was the assistant responsible for supervising Valentin’s diploma work, then his colleague and now we are associates and research partners.

Also, I would like to thank the many students that I had the pleasure to work closely with: Marcos Perez Jurado, Mounir Jarrai, Jean-Philippe Pellet and Grégoire Krähenbühl.

And I would like to thank the other members of the Software Engineering Laboratory (LGL), Anne Crettaz Schlageter, Benjamin Barras, Raul Silaghi, David Hürzeler, Stanislav

Chachkov, Adel Besrour, Xavier Caron, Sandro Costa, Rodrigo Garcia-Garcia, Slavisa Markovic, Jarle Hulaas, and Thomas Baar for a pleasant atmosphere at the lab. I would also like to thank past members who made the effort to welcome me and who assisted me in early times: Didier Buchs, Jörg Kienzle, Thomas Wolf, Enzo Grigio, Nicolas Guelfi, Mathieu Buffo, Cécile Peraire, and Giovanna Di Marzo. Also, I would like to thank the EPFL computer science third year students of the past four years. They taught me many things about the requirements for making good development tools.

Fourth, I would like to thank my parents-in-law Mary and Fritz Stähelin for supporting us, me and my little family, through all this time. Special thanks to Mary for proofreading the English of this thesis and giving me many constructive comments.

I also would like to thank my parents Manian and Sékou and my brothers and sisters. They have always been a stable support to me through my life.

Finally, I would like to thank my wife Bettina and our daughters Fatou and Aicha. Bettina has been a wonderful physical and mental support. Her support throughout both the good and the difficult situations was very significant to me. Bettina, you have always been there for me. I did it! Thanks, Bettina. Now I look forward to being with you hand-in-hand and with the kids.

Table of Contents

Abstract i
Kurzfassung iii
Acknowledgements vii
Table of Contents ix
List of Figures xv

1 Introduction 1
1.1 My Thesis 1
1.1.1 Setting the Scene 1
1.2 Critical Evaluation of Architectural Trends and Practices 2
1.2.1 ADL-based Software Architectures 3
1.2.1.1 Review 3
1.2.1.2 Evaluation 4
1.2.2 View-Oriented Software Architectures 4
1.2.2.1 Review 4
1.2.3 Pattern-Oriented Software Architectures 6
1.2.3.1 Review 6
1.3 The Problem 8
1.4 Main Contributions 9
1.5 Structure of the Dissertation 10

Part I: Motivation & Background

2 Motivating Case Study 15
2.1 Informal Software Architecture Documentation 15
2.1.1 Software Development Problem — The Video Surveillance Service 15
2.1.2 Documenting the Problem Space 16
2.1.3 Documenting Requirements on the Solution Space 16
2.2 Formal Software Architecture Documentation 20
2.2.1 Documenting Architectural Structure in Wright 21
2.2.2 Documenting Architectural Behavior in Wright 22
2.2.3 Remarks 24
2.3 Pattern-Oriented Software Architecture Documentation 24
2.3.1 Documenting Requirements on the Pattern-Oriented Architecture 24
2.3.2 An Application of Architectural and Design Patterns 26
2.3.2.1 Rationale for Applying the Observer Design Pattern 26

2.3.2.2 Documenting An Observer Pattern Occurrence 26
2.3.2.3 Rationale for Applying the Component Configurator Pattern 28
2.3.2.4 Instantiating the Component Configurator Pattern 29
2.3.2.5 Rationale for Applying the Pipe-and-Filter Pattern 31
2.3.2.6 Instantiating the Pipe-and-Filter Architectural Pattern 31
2.3.3 Documenting an Architecture for a Family of Software Systems .. 32
2.4 Concluding Remarks 34

3 Related Work 37
3.1 IEEE Recommended Practice for Architectural Description 37
3.1.1 IEEE-Std-1471 Conceptual Framework 37
3.1.2 General Conformance Requirements 39
3.1.3 IEEE-Std-1471 Lacks Realizations 39
3.2 The Unified Modeling Language 40
3.2.1 UML Is Not Concern-Oriented 41
3.3 Multi-Dimensional Separation of Concerns 41
3.3.1 Conceptual Framework of MDSOC 41
3.3.2 Concern-Space Modeling Schema 43
3.3.3 General Requirements for Achieving MDSOC 44
3.3.4 Existing Realizations of MDSOC 45
3.3.5 Units Are Inside Software, But Not the Concerns 45
3.4 Aspect-Oriented Software Development 46
3.4.1 Aspect-Oriented Concepts 46
3.4.2 Issues in Aspect-Oriented Modeling 47
3.5 Final Remarks 48

Part II: Building & Describing Concern-Oriented Software Architectures

4 Concerns and Software Architecture 53
4.1 Concerns 53
4.1.1 Our Definition of a Concern 54
4.1.2 Discussion 55
4.2 Concern Categories 55
4.2.1 Architectural Concerns 56
4.2.2 Non-Architectural Concerns 56
4.2.3 Reifiable concerns 56
4.2.4 Non-reifiable concerns 56
4.3 Relationship between Concerns 57
4.4 Examples of Concerns 57
4.4.1 Concerns in the Requirements on ADLs 57

4.4.2 Concerns in the Video Surveillance Problem ... 58
4.5 Towards Concern-Oriented Software Architectures ... 60
4.5.1 Objectives of the COSA Approach ... 62
4.5.2 Characteristics and Requirements of the COSA Approach ... 62
4.6 Final Remarks ... 63

5 The PCS Framework ... 65
5.1 Introduction ... 65
5.1.1 Goals, Principles and Key Concepts ... 65
5.1.2 Fulfilling the Requirements of COSA ... 67
5.2 Realizing the IEEE-Std-1471 Conceptual Framework ... 67
5.2.1 Viewpoint Schema ... 69
5.3 Reifying Concerns ... 70
5.4 Realizing an Architecture Concern-Space for UML ... 71
5.4.1 UML Lacks Adequate Support for Modeling Roles ... 71
5.4.2 The X-Syndrome ... 72
5.4.3 UML Space — Overcoming the X-Syndrome ... 74
5.4.3.1 How to Create a UML Space ... 75
5.5 Achieving Architectural Design by Concerns ... 76
5.5.1 Mechanisms for Separating Architectural Concerns ... 76
5.5.2 Linguistic Support for Expressing Architectural Concerns ... 78
5.5.2.1 Tyranny of Dominant Decompositions in UML ... 78
5.5.2.2 Scattering of Concerns ... 79
5.5.2.3 Tangling of Concerns ... 80
5.5.2.4 Crosscutting Concerns ... 80
5.6 Integrating Architecting and Software Development ... 80
5.7 Using UML ... 81
5.8 Final Remarks ... 81

6 Aspect-Oriented Construction PCS ... 83
6.1 Viewpoint Name ... 83
6.2 Sources ... 83
6.3 Concerns ... 83
6.4 Stakeholders ... 84
6.5 Rationale ... 84
6.6 Architectural Problems ... 84
6.6.1 Incentive Aspects ... 84
6.6.2 Towards Perspectival Associations ... 85
6.6.3 Decisional Aspects ... 86
6.6.4 Resultant Aspects ... 86

6.6.5 UML Extensions for Aspect-Oriented Modeling 87
6.7 Resultant View 89
6.7.1 Identifying Causes and Effects for the EventConnector 90
6.7.2 Designing the EventConnector 90

7 On-Demand Remodularization PCS 95
7.1 Viewpoint Name 95
7.2 Sources 95
7.3 Concerns 95
7.4 Stakeholders 96
7.5 Rationale 96
7.6 Architectural Problems 96
7.6.1 Incentive Aspects 96
7.6.2 Decisional Aspects 96
7.6.3 Resultant Aspects 97
7.7 Relationships Among Concerns 97
7.8 Resulting View 97
7.9 The Java Drag & Drop Architecture Case Study 98
7.9.1 Drag Support Initialization Concern 99
7.9.2 DropTarget Installation Concern 103
7.9.3 Drop Event Interception Concern 104
7.10 On-Demand Remodularization Pattern 108
7.10.1 Motivation 108
7.10.2 Structure of the Pattern 109
7.10.3 Achieving Design by Concerns with the ODR Pattern 110
7.11 Final Remarks 112

8 The Service-Oriented PCS 113
8.1 Introduction 113
8.2 Main Model of the Service-Oriented View 114
8.3 Static Structure Model 115
8.3.1 Component Structure Specification 115
8.3.2 Connector Structure Specification 116
8.4 Behavioral Specification Model 118
8.4.1 Component Behavior Specification 118
8.4.2 Connector Behavior Specification 118
8.5 Perspectival Elements 119
8.5.1 Structural Specification of Perspectival Elements 119
8.5.2 Behavioral Specification of Perspectival Elements 120
8.6 Configuration Model 121

8.6.1 Context of the Configuration ... 121
8.6.2 Configuration Manual ... 122
8.6.3 Architecture Background ... 125
8.6.3.1 Design Rationale ... 125
8.6.3.2 Analysis Results ... 125
8.6.3.3 Assumptions ... 126
8.6.4 Other Information ... 126
8.6.5 Related View Packets ... 126
8.7 Final Remarks ... 126

Part III: Other Related Work and Conclusions

9 Integrating the Structural PCS with SADL ... 131
9.1 Introduction ... 131
9.2 Structural Viewpoint ... 131
9.2.1 Structural View of the Compiler System ... 132
9.2.1.1 Static Model ... 133
9.2.1.2 Configuration Model ... 134
9.2.2 Overview of SADL ... 135
9.2.3 Mapping ConcernBASE to SADL ... 138
9.2.3.1 Mapping Data Types ... 138
9.2.3.2 Mapping Architectural Components ... 139
9.2.3.3 Mapping Component Interfaces ... 140
9.2.3.4 Mapping Connections ... 142
9.2.3.5 Putting It All Together ... 143
9.3 Tool Support ... 144
9.4 Final Remarks ... 145

10 Conclusions ... 147
10.1 Summary ... 147
10.2 Contributions ... 147
10.3 Advantages and Limitations ... 148

Part IV: Annexes

A Bibliography ... 153
Curriculum Vitae ... 163

List of Figures

Chapter 1: Introduction

Chapter 2: Motivating Case Study

Figure 2.1: An Informal Documentation of the Problem Space 16
Figure 2.2: Architecture Development Activity Diagram 17
Figure 2.3: An Illustration of Requirements on the Solution Space 18
Figure 2.4: A Wright Description of the Video Surveillance Architecture 21
Figure 2.5: Informal Req. Diagram for the Video Surveillance Architecture ... 25
Figure 2.6: A Higher-Level Description of the Observer Pattern Instance 27
Figure 2.7: An Observer Pattern Instance with Focus on Connection Points.... 27
Figure 2.8: An Observer Pattern Instance with Focus on the Interconnection .. 28
Figure 2.9: Structural Instance of the Component Configurator Pattern 30
Figure 2.10: Behavioral Instance of the Configurator Design Pattern 31
Figure 2.11: A High-Level Structure of a Pipe-and-Filter Instance 32
Figure 2.12: A Structural Instance of the Component Configurator Pattern 33
Figure 2.13: Behavioral Instance of the Component Configurator Pattern 33

Chapter 3: Related Work

Figure 3.1: Conceptual Framework of IEEE-Std-1471 (from [IEEE00]) 38
Figure 3.2: An Overview of the Conceptual Framework for MDSOC 42
Figure 3.3: The Cosmos Perspective of the Software Concern-Space 44

Chapter 4: Concerns and Software Architecture

Figure 4.1: Concerns in the video surveillance service problem in Version 1 .. 59
Figure 4.2: Concerns in the video surveillance service problem in Version 2 .. 60
Figure 4.3: Conceptual Framework of IEEE-Std-1471 (modified) 61

Chapter 5: The PCS Framework

Figure 5.1: A Perspectival Concern-Space in Overview 66
Figure 5.2: Overview of the IEEE-1471 Realization for the PCS Framework.. 68
Figure 5.3: A Viewpoint Schema in Overview..... 69
Figure 5.4: UML Metamodel for Collaboration 72
Figure 5.5: The Description of the UML Metaclass Classifier 75
Figure 5.6: Unidimensional Separation of Concerns Across Different Levels . 79
Figure 5.6: Structural Crosscutting..... 80

Chapter 6: Aspect-Oriented Construction PCS

Figure 6.1: Structural Illustration of Key Elements within a UML Collaboration 85
Figure 6.2: Behavioral Illustration of Key Elements of a UML Collaboration. 86
Figure 6.3: High-Level Package View of the UML Space for AOM..... 87

Figure 6.4: The UML Space for AOM — A Low-Level View of AOM Core . 88
Figure 6.5: A Perspectival Association between the Component Roles..... 90
Figure 6.6: The Cause-Effect Principle Applied on the EventConnector..... 90
Figure 6.7: Static Structure of the EventConnector 91
Figure 6.8: Body of the Perspectival Behavior Associated with newState..... 91
Figure 6.9: Body of the Perspectival Behavior Associated with newConsumer .. 92
Figure 6.10: The Complete Design of the EventConnector..... 92
Figure 6.11: Implementation View of the EventConnector Aspect..... 93

Chapter 7: On-Demand Remodularization PCS

Figure 7.1: Concern-Oriented Model as a Means for Concern Representation. 97
Figure 7.2: Concern-Oriented Model as a Means for Concern Reification..... 98
Figure 7.3: Concern-Oriented Model as a Means for Representing Model Slices 98
Figure 7.4: Concern-Oriented Model as a Means for Representing a Mechanism 98
Figure 7.5: Behavioral Model Slice for the Drag Support Initialization Concern 101
Figure 7.6: Structural Model Slice for the Drag Support Initialization Concern..... 102
Figure 7.7: Behavioral Model Slice for the DropTarget Installation Concern 103
Figure 7.8: Structural Model Slice for the DropTarget Installation Concern 104
Figure 7.9: Behavioral Model Slice for the Drop Event Interception Concern..... 105
Figure 7.10: Structural Model Slice for the Drop Event Interception Concern 106
Figure 7.11: Interaction Model for using the Java Drag & Drop API..... 107
Figure 7.12: Static Structure Model for using the Java Drag & Drop API..... 108
Figure 7.13: Structure of the On-Demand Remodularization Pattern 109
Figure 7.14: Instantiation of ODR for the Traffic Light Control System..... 110
Figure 7.15: ODR Applied on the Java Drag'n'Drop Architecture..... 111

Chapter 8: The Service-Oriented PCS

Figure 8.1: Mapping Between a Viewtype and the PCS Framework..... 114
Figure 8.2: Main Model of the Service-Oriented Architectural View..... 114
Figure 8.3: Component Structure Specification 115
Figure 8.4: Component Types' Interface Elements..... 116
Figure 8.5: The VSServiceConnector Structure Specification..... 116
Figure 8.6: Interface Elements of VSServiceConnector..... 117
Figure 8.7: Protocol State Machine for StreamEndPointSignaling..... 118
Figure 8.8: Protocol Sequence Diagram for StreamEndPointSignaling 119
Figure 8.9: Perspectival Associations and their Properties..... 120
Figure 8.10: The System Context Diagram 121
Figure 8.11: Base Configuration Diagram..... 123
Figure 8.12: A Perspectival Configuration Diagram 124

Chapter 9:

Figure 9.1: Compiler Architecture: taken from [MR97] 132
Figure 9.2: Structural View of the Compiler System 132
Figure 9.3: Static Structure Model of the LexicalAnalyzer Components Type 134
Figure 9.4: Configuration Model of the Compiler System 134
Figure 9.5: Extract of the Level-3 Compiler SADL Specification 137
Figure 9.6: Compiler_Types.sadl 138
Figure 9.7: Translating Architectural Components 139
Figure 9.8: Translating Stream Interface Type 140
Figure 9.9: Translating Behavioral Aspects 141
Figure 9.10: Translating Data Connections 143
Figure 9.11: Putting Everything Together 144
Figure 9.12: A Screenshot of the ConcernBASE Modeler 145

Chapter 1:
Introduction

There is no point in using exact methods where there is no clarity in the concepts and issues to which they are to be applied.

John von Neumann

1.1 My Thesis

A concern-oriented approach to software architecture is feasible and suitable for developing and describing architectures of software-intensive systems, for improving separation of concerns in the design, construction and evolution of such systems, and for integrating their architectural descriptions with modern software development artifacts.

1.1.1 Setting the Scene

Major goals of software architecture [PW92][SG96] are to improve the way of organizing complex software infrastructures, while reducing costs of software production; and to facilitate software construction that fosters reuse, maintainability and evolution. Other important goals are to provide the ability to make descriptions of software architectures that serve as vehicles for communication among the stakeholders (i.e., participants) of software development projects, for manifesting the earliest design decisions in those projects, and for representing reusable and transferable abstractions of the systems under construction [BCK98].

To achieve these goals, research in the software architecture community has shown that software-intensive systems—i.e., systems whose design, construction, deployment and evolution are essentially influenced by software [IEEE00]—should not be developed from scratch. Instead, they should be constructed, based on well-defined architectural descriptions that allow software engineers to understand how to combine various components; such components are possibly developed at different times, using different technologies. However, putting the result of this research into practice has several implications, especially when supporting component-based software engineering [Szy98].

One key implication is the following: when building a complex software system by composition, the components you select will be required to fulfill some “roles” anticipated in the software architecture of the system under construction. A particular component may fulfill one or more roles in order to achieve some stakeholders’ concerns. In reality, the same concerns can pertain to one or more aspects of the system under construction. Some aspects of the system can be relevant to a specific stakeholder who considers the composition problem from a particular perspective; but also, different stakeholders may have differ-

ent concerns which pertain simultaneously to various perspectives. On the other hand, a given role defined by the architecture can be fulfilled by many components of the system at the same time or at different times; an individual concern can affect many roles, but also a role may have an effect on the realization of many concerns. Thus, it becomes rapidly difficult to understand and reason about the software system's composition if we cannot represent and manipulate, adequately, the roles involved in the software architecture.

At this stage, a fundamental issue to be addressed is how to relate the stakeholders' *concerns*, belonging to the composition problem, to the *roles* in the architecture (i.e., abstract and reusable solution to the problem). Providing the capability to properly address this issue will have a great benefit for both software architecture and component-based software engineering. Lacking such a capability will result in a number of complications that hinder accomplishing the goals of software architecture and make component-based software engineering difficult to achieve. For example, developers of individual software components will have to provide mechanisms for implementing not only the concerns that crosscut many components playing different roles, but also they will have to offer techniques for supporting individual roles that affect the realization of multiple concerns. Both cases are very tricky and require appropriate means for separating concerns, at different levels.

1.2 Critical Evaluation of Architectural Trends and Practices

To begin with, we have to admit that there is no agreement on what is called software architecture, and that there is no single, accepted framework for codifying architectural concepts. This lack of agreement does not facilitate the emergence of common practices in software architecture and their controlled evolution [IEEE00]. However, the diversity in the realm of software architecture stems from the variety of issues that reflect the concerns of the authors.

When producing architecture descriptions in software projects, most practitioners (in the recent area of software architecture) prefer to draw diagrams consisting of boxes and lines with vaguely defined meanings, if they are defined at all. The diagrammatic notations used in this practice are typically informal and ad hoc, i.e., based on the whiteboard approach. However, the software architecture community has become aware of the problems caused by the use of informal notations for architecture descriptions, and there is an agreement on the need for proper notations and modeling techniques for software architectures [PW92][SG96]. Modern notations are either formal, i.e., based on formal and special-purpose modeling languages, called Architecture Description Languages (ADLs) [GMW97][MT00][Cle96], or they are positioned somehow between informal and formal notations, i.e., based on semi-formal modeling languages. Semi-formal modeling languages present an alternative to using informal and formal modeling techniques and notations, but

semi-formal languages also complement both. Examples of languages supporting semi-formal techniques and notations include OMG's standard, the Unified Modeling Language (UML) [Omg01]. Semi-formal and formal languages are both being used in different ways, following different trends.

The following subsections summarize some of the most important trends and practices in software architecture: ADLs (architecture description languages), multiple views and pattern-oriented software architecture.

1.2.1 ADL-based Software Architectures

1.2.1.1 Review

To encourage calling off the popular practice based on the whiteboard approach, numerous ADLs have been specifically designed to support representing, analyzing and reasoning about key characteristics of software systems. This goal of ADLs is mainly achieved by providing linguistic constructs, such as components, connectors, properties, styles and systems, which aim at supporting the encapsulation of certain kinds of architectural concerns. A typical ADL allows one to encapsulate architectural concerns into linguistic constructs or units model elements: computations and data store (into components), interactions (into connectors), constraints (into properties), reusability of collections of design elements and the conditions for reusing such collections (into styles) and configurations (into systems).

The ADL trend, essentially driven by academics, is motivated by the creation and improvement of ADL-based approaches to software architecture, including associated tools; to this practice belongs the promotion of ADLs as suitable languages for solving architectural problems. ADLs have the advantage of being mathematically founded, facilitating analysis of architectural models. However, due to their formal nature, architectures described in existing ADLs can be hard to understand and to use as a vehicle for communication among the stakeholders of a project. For instance, developers in need of ADL-based software architectures will have to learn the corresponding mathematical models.

Furthermore, the lack of integration of ADL-based descriptions with other artifacts commonly used by software developers leads to considerable obstacles in the promotion of software architecture as a new field. This is perhaps a reason why ADLs are not so widely used in industry. Having become aware of these obstacles recently, ADL specialists and others have started a development effort which addresses mapping strategies allowing for integration between specific ADLs and well-accepted description languages, for instance, UML [RMR+98][GK00][CKS+01], and XML [Sch01]. This is a valuable approach to achieve integration of ADLs with modern software technologies; however, it still remains insufficient, as ADLs address only a limited number of kinds of architectural concerns. For example, ADLs generally lack support for representing crosscutting concerns; as a result, ADL-based architecture descriptions are difficult to maintain and adapt once they are developed.

1.2.1.2 Evaluation

Major problems encountered with ADLs are related to their lack of flexibility. Architectural descriptions made in special-purpose languages (ADLs) like Wright [All97], Darwin [MDE+95], Rapide [LAK+95] and Acme [GMW97] should support descriptions of multiple structures, including diagrams, models and views, that intentionally address different kinds of concerns. ADLs should show how various concerns affect each other in architectural designs; they should also allow you to identify, analyze and elaborate architectural concerns that cut across several software components at the same time, such as transactions, security, load balancing, synchronization, reuse, customization, scalability, etc.; they should, but they do not.

ADLs do, however, allow architects to achieve good support for non-crosscutting concerns. They provide significant means to represent, analyze and manipulate non-crosscutting or "naturally localizable" kinds of concerns through encapsulation in particular linguistic constructs found in most existing ADLs. But, they failed to allow architects to achieve support for "non-localizable" concerns; though, a good ADL should also allow architects to provide descriptions that address compositions of both crosscutting and non-crosscutting concerns.

1.2.2 View-Oriented Software Architectures

1.2.2.1 Review

View-oriented software architecture practice, especially driven by practitioners in the software industry, is motivated by the idea that an architecture involves multiple structures which, considered from different perspectives, present different aspects of the system. An important characteristic of this trend is its ability to represent an architecture as a collection of diagrammatic descriptions that can be depicted by different views and at different levels of abstraction; each view can address its own set of concerns and allow for further elaboration of its elements in various ways. This provides the capability to create multiple structures simultaneously, while allowing mechanisms for repeatedly refining each structure; it also includes the facility to integrate architectural descriptions with modern software development artifacts.

View-oriented approaches allow you to express different aspects of an architecture in different views and to compose these aspects to build the system structure as a whole. Unlike ADL-based approaches, view-oriented approaches are not just limited to a few kinds of architectural concerns; instead, they allow one to address all significant kinds of concerns involved in a software architecture. View-oriented approaches are usually based on semi-formal notations.

Examples of approaches following the view-oriented trend include the Krutchen 4+1 Views [Kru95] and the Siemens Four Views [HNS99]. Further examples include applica-

tions of the ISO Reference Model for Open Distributed Processing (RM-ODP) [ISO95] with UML in areas such as telecommunications [KTM97][KMP+98].

While multiple views allow one to focus on different structures of a system, individual views do not address stakeholders' concerns explicitly. Therefore, it is difficult to know precisely whether an architecture description resulting from a view-oriented approach has addressed clearly all the concerns of importance to the stakeholders; it is most important to know how views and concerns relate to each other.

1.2.2.2 Evaluation

Essential problems encountered in using multiple views of a software architecture include:

- *Lack of support for associating architectural concerns to individual views*. What is missing in existing view-oriented approaches is a clear understanding of the kinds of architectural concerns that pertain to the structures represented in individual views. Certain kinds of architectural concerns (e.g., functionality and feature) will have an effect on one or many representational units or model elements. Other kinds of architectural concerns (e.g., performance, interaction, configuration and security) will not only affect several linguistic constructs in one partial description, but they will also crosscut many elements in various partial descriptions: diagrams, models and views.
- *Lack of appropriate support for software connectors.* Mechanisms used to encapsulate interaction among components, as described in most architectural views, do not support modularization and reuse of connectors. However, inadequate modularization of interactions increases maintenance costs and complicates reasoning about architectural concerns, such as security policies and synchronization.
- *Lack of consistency check between views*. Current approaches do not have the ability to check consistency among multiple views, since there is no mechanism to support a "conceptual repository" that encompasses all elements of all different views, models and diagrams, and allows checking for consistency among the elements.
- *Lack of separation between architectural viewpoints and architectural views.* The view-oriented software architecture approaches mentioned above support each only a few, limited number of viewpoints and views; they do not clearly define the distinction between architectural viewpoints and views. The ANSI IEEE Recommended Practice for Architectural Description (known as ANSI/IEEE-1471-2000 standard [IEEE00]), in contrast, allows for distinguishing architectural viewpoints from architectural views, and for creating and using an arbitrary number of views. Unfortunately, implementations of the ANSI/IEEE-1471-2000 standard are still missing. (I will return to this point in section 1.4.)

1.2.3 Pattern-Oriented Software Architectures

1.2.3.1 Review

A well-known approach in the style- or pattern-oriented architectural practice is *pattern-oriented software architecture* [BMR+96][SSR+00]. The principal idea of this approach is based on the (re-)use of patterns for developing software architectures. A pattern for software architecture addresses a specific recurring design problem, which occurs in particular design contexts and offers a well-understood general solution schema. The solution schema describes the constituent components, their responsibilities and relationships, as well as the collaboration among those components.

An important characteristic of a pattern-oriented approach to software architecture is its ability to cross several levels of abstraction. Such approaches do not focus on design-level patterns only; instead, they cover consistently system-level or architectural patterns, including design patterns [GHV+95] and low-level idioms. Another characteristic of this approach is the selection of appropriate architectural patterns. Selecting an architectural pattern requires early decision-making that is significant to the design, construction and evolution of the system at hand; the selected architectural pattern will affect not only the fundamental organization of the system, but also the architecture of its subsystems and their interconnections.

The pattern-oriented software architecture approach supports a new mechanism for separation of concerns, called pattern systems. Pattern systems are containers of heterogeneous patterns that allow software architects to separate collections of patterns according to various criteria. When following this approach, design patterns are required to structure individual subsystems and relationships among them, but they have no effect on the organization of the system itself.

Another critical element in an architecture description is a style [SG96]. According to Dwayne E. Perry and Alexander L. Wolf [PW92], “an architectural style defines a family of software systems in terms of their structural organization. An architectural style expresses components and the relationships between them, with the constraints of their application, and the associated composition and design rules for their construction”. Architectural patterns are very similar to architectural styles, but they have some differences.

However, according to [BMR+96], every architectural style can be described as an architectural pattern; but to achieve this, it is important to observe the following distinctions:

- *There are no dependencies among architectural styles, but patterns do relate to each other.* As mentioned above, when applying the pattern-oriented software architecture approach, various design patterns can be used to structure the individual subsystems and the relationships among them; these can, in turn, make use of idioms in their

implementations, etc. The dependency relationships between patterns (from architectural patterns via design patterns to idioms) can be documented in pattern systems.

- *Architectural styles are less problem-oriented than architectural patterns*. An architectural style is a high-level design mechanism that is intended to prescribe a software system's organization independently of the context of a particular design problem. In contrast, a pattern addresses a specific recurring design problem which occurs within a particular design context and offers a well-understood generic solution schema.
- *Architectural styles only describe the global organizational structures of a system*. However, they do not care about the detailed design structures for individual components and interconnections of the system. On the other hand, architectural patterns involve not only the overall system organization, they also affect the design of diverse subsystems and their interconnections.
- *Architectural styles include reference models, but architectural patterns do not*. Reference models are system organizations that impose rules for constructing applications in some particular domains. Examples include: the reference model for open distributed systems, RM-ODP [ISO95], that prescribes the rules for building open distributed applications; the ISO Open Systems Interconnection reference model which recommends the 7-layer model for communication networks [Tan92]; and the reference model for compiler construction [PW92]. In effect, the level of abstraction for using architectural patterns is not the same as for reference models; but one or more architectural patterns can be applied when using a reference model.

1.2.3.2 Evaluation

Major difficulties with the pattern-oriented approach are:

- *Lack of support for identifying forces in the solution scheme*. Architectural patterns do not have means for identifying and reasoning about the relations between individual forces of recurring problems and the associated generic solution schemes. Without this support, instantiation and reuse of patterns remain difficult; this is particularly the case when the concerns related to individual forces crosscut various components, interconnections, and associated design patterns or idioms.
- *Lack of support for crosscutting roles*. Patterns describe their reusable elements in terms of roles rather than concrete components. However, most pattern descriptions do not provide mechanisms for encapsulating interactions and properly addressing dynamically changing roles. Roles that change dynamically are frequently found in complex interactions; they often cut across many components at the same time.

1.3 The Problem

A major cause of many complications in software architecture is the lack of abstraction for separating and combining concerns of various kinds in architectural descriptions. A given concern might cut across many elements of an architecture description. Crosscutting concerns are critical aspects of the software composition problem.

However, despite the large amount of ongoing good research in software architecture, many of the difficulties in the creation of software compositions are related to deficiencies in addressing crosscutting concerns. The representation of crosscutting concerns is very limited in current practices; for example, interactions are difficult to modularize and reason about. Existing software architecture approaches either require that interactions be scattered across the participant components, or they concentrate on just a few kinds of interaction concerns, such as the protocol of communication among components. This makes it easier, perhaps, to design special-purpose description languages to support architectural representations. However, it also complicates considerably the creation of software connectors [SG96] that can combine more than one kind of interaction concern. Thus, it becomes difficult to promote connectors as first-class citizens in software architecture and to describe them as a specification of interaction. Unfortunately, the software architecture community still continues to neglect the importance of crosscutting concerns, although existing approaches have failed to address them appropriately.

Without new techniques to explicitly address crosscutting concerns at system-level, software architects are condemned to produce descriptions in which crosscutting and non-crosscutting concerns will be entangled. When fulfilling unanticipated technological constraints during implementations, developers will have to customize and elaborate descriptions that are already intertwined. As a result, developers will likely produce software that cannot be tested for conformance to the architecture. Lacking conformance between architecture and its realization will continue to limit considerably the benefits of software architecture.

We believe that focusing on only non-crosscutting concerns impedes attaining major goals of software architecture: it inhibits communication among the stakeholders of a software development project, restrains the manifestation of the earliest design decisions in the project, and limits the ability to reuse and transfer the abstraction of a software-intensive system. New approaches are required to improve the current situation in order to achieve the goals of software architecture.

1.4 Main Contributions

The main contribution of this thesis to solving the above problem is a new approach for developing and documenting software architectures, called the *concern-oriented software architecture* approach—or the COSA Approach.

The concern-oriented software architecture approach provides new mechanisms for encapsulating individual concerns into independent architectural constructs. It introduces concern-oriented modeling techniques to encourage software architects to address their systems' requirements in a "concern-oriented way". The COSA Approach allows one to achieve:

- *Integration of architecting activities with common software development*. This capability enables the creation of software architectures in non-isolated ways—that is, COSA supports architecture development at multiple stages within the global context of the software life cycle, not only at one particular stage or during a specific phase of software development.
- *Reification of stakeholders' concerns into software*. This ability allows one to address individual concerns at different levels of abstraction simultaneously, while distinguishing each concern from the software development artifacts reifying it.
- *Architectural design by concerns*. Practicing design by concerns allows one to clearly separate the software architecture of a system from its description, and to ensure that both the software architecture and its description address properly the stakeholders' concerns and nothing more.
- *Realization of an architecture concern-space*. Realizing an architecture concern-space provides support for developing software architectures, while applying mechanisms of multidimensional separation of concerns (MDSOC).
- *Realization of the IEEE-Std-1471 Conceptual Framework*. This allows one to produce architecture descriptions that conform to the IEEE and ANSI standard for architectural descriptions, ANSI/IEEE-Std-1471-2000.
- *Effective use of UML*. The COSA approach is a new way of using UML that allows one to integrate the resulting concern-oriented architecture descriptions with other architectural descriptions written in standard modeling languages, such as UML; this includes, for example, UML support for key ADL constructs: components, connectors, styles, systems and properties.

Another key contribution of this thesis is a particular implementation of the COSA approach, called the PCS Framework. The PCS Framework provides a UML-based linguistic toolkit, called *UML Space*, and combines the realizations of IEEE-Std-1471 and MDSOC. Essentially, the PCS Framework achieves the requirements for fulfilling the COSA approach in the following steps:

- The PCS Framework realizes the IEEE-Std-1471 Conceptual Framework through its concept of a *viewpoint schema*.
- The reification of concerns is achieved by means of the *projection* mechanism.
- The realization of an architecture concern-space is achieved through the notion of *UML Space*—the PCS Framework uses standard UML to create a UML Space which architects need to use *to develop and apply various viewpoint languages* at will.
- The PCS Framework achieves "architectural design by concerns" through *concern-oriented modeling* using one or more viewpoint languages.
- The integration of the building of software architectures with common software development is achieved through a *combination of MDSOC with IEEE-Std-1471 and UML*.

This thesis also provides three different Perspectival Concern-Spaces (PCS's) which together build the current PCS Framework: an *Aspect-Oriented Construction PCS*, an *On-Demand Remodularization PCS*, and a *Service-Oriented PCS*. The On-Demand Remodularization PCS includes a new concern-oriented pattern, which we refer to as On-Demand Remodularization pattern. Furthermore, the dissertation provides different examples, each showing how to use an individual PCS.

1.5 Structure of the Dissertation

Part I- Motivation & Background

Chapter 2: Motivating Case Study

This chapter presents an introductory case study that shows the motivation for going a step further. The case study shows how to apply current software architecture practices, based on the example of a video surveillance service. The example includes an application of three different approaches—an informal, a formal and a semi-formal approach. The chapter also illustrates the limitations of these approaches and argues for a concern-oriented approach to software architecture development and documentation.

Chapter 3: Related Work

This chapter introduces the foundations of the concern-oriented approach to software architecture. It presents two conceptual frameworks: an architecture description framework which is part of the ANSI IEEE standard for architecture documentation, called IEEE-Std-1471; and an aspect-oriented software engineering framework, known as concern spaces, which is part of Multidimensional Separation of Concerns (MDSOC). The Unified Modeling Language (UML) is introduced as a linguistic framework that can be used to establish a bridge between both conceptual frameworks. The chapter concludes with the needs for implementing these frameworks using UML.

Part II- Building & Describing Concern-Oriented Software Architectures

Chapter 4: Concerns and Software Architecture

This chapter introduces a new perception of concerns in relationship to software architecture. The first part of the chapter considers and discusses several definitions of the notion of concern and presents a definition used by this dissertation. It introduces concern categories and gives several examples of concerns from both a given requirements definition and a software development problem. The second part of the chapter introduces the concern-oriented approach to software architecture as a general methodology that can be realized in different ways to achieve architectural design by concerns.

Chapter 5: The PCS Framework

This chapter introduces the PCS Framework, a particular methodology implementing the concern-oriented approach to software architecture. It presents a fulfillment of the general requirements on concern-oriented approaches to software architecture, which uses UML to combine the realizations of the conceptual frameworks of MDSOC and IEEE-Std-1471. The PCS Framework introduces new mechanisms, such as projections and UML Space, to support integrating software development into the building of software architectures.

Chapter 6: Aspect-Oriented Construction PCS

This chapter presents the concern-oriented approach to software architecture from the perspective of aspect-oriented software development, using multi-dimensional separation of concerns (MDSOC). It describes a perspectival concern-space, called Aspect-Oriented Construction PCS. This specific PCS demonstrates how MDSOC helps deal with software complexity by supporting the composition of independent components software along different interaction concerns. The chapter introduces a UML Space for Aspect-Oriented Modeling.

Chapter 7: On-Demand Remodularization PCS

This chapter presents an approach to implementing MDSOC for UML with a specific focus on on-demand remodularization. It introduces techniques for achieving architectural design by concerns with concern-oriented modeling. Moreover, the chapter describes a concern-oriented pattern, called On-Demand Remodularization pattern.

Chapter 8: The Service-Oriented PCS

This chapter presents an integration of the Service-Oriented PCS with a well-known Viewpoint-Oriented approach to documenting software architectures. This chapter uses the Video Surveillance System example introduced in chapter 2 to show the applicability of the PCS Framework on other architectural description approaches.

Part III - Other Related Work and Conclusions

Chapter 9: Integrating the Structural PCS with SADL

This chapter presents the case of a compiler architecture to validate the integration of ConcernBASE, an early structural PCS, with SADL, which is a software architecture description language based on architectural refinement.

Chapter 10: Conclusions

This chapter provides concluding remarks. It also presents some limitations of concern-oriented approaches to software architecture and the PCS Framework, and it indicates some directions in which research in the new area of concern-oriented software architecture could be pursued.

Part IV - Annexes

Part I

Motivation & Background

Chapter 2:
Motivating Case Study

> In describing the world, formalization and formal reasoning can show only the presence of errors, not their absence.
>
> *Michael Jackson*

This chapter presents an introductory case study that shows the motivation for going a step further. The case study shows how to apply current software architecture practices, based on the example of a video surveillance service. The example includes an application of three different approaches—an informal, a formal and a semi-formal approach. The chapter also illustrates the limitations of these approaches and argues for a concern-oriented approach to software architecture development and documentation.

2.1 Informal Software Architecture Documentation

As discussed in the previous chapter, there are many different approaches that might be applied to document a system's software architecture. But which of these approaches is most appropriate for solving a given problem? If there is one at all, why is it the most appropriate? Frankly, there is no simple answer to such questions; in reality, each particular approach has its advantages and limitations. This section helps get an idea of what an informal architecture documentation approach can be good for and for what it is not appropriate. It shows the example of an informal problem description and presents cases of documenting the problem space and documenting the requirements on the solution space for that problem. The following section presents the software development problem that we consider throughout the motivating case study.

2.1.1 Software Development Problem — The Video Surveillance Service

When the number of crimes increases in society and security becomes a concern, it is often necessary to make use of new technologies to control the situation. A video surveillance service can be useful in such a situation. For this purpose, a collection of geographically distributed video cameras is to be controlled and monitored by security agents from a central video surveillance station. Each video camera captures images and produces a video stream that is transmitted to the central surveillance station. In case of an emergency, the security agents alert the police; for analysis purposes, security agents can command the surveillance system to store the sequence of images related to the urgent situation in a database

of emergencies. Moreover, the police can ask for the video stream produced from a particular location and in a specific time period.

2.1.2 Documenting the Problem Space

Creating a software architecture for the video surveillance service is a central part of the development task, but the importance of the architecture depends on how much it facilitates the job of stakeholders. For example, developers will much appreciate it if they can find swiftly all the information they need to know about an architecture in order to develop software that conforms to that architecture. A good way to achieve this objective consists in providing clear and systematic documentation of the software architecture.

However, before you can document a software architecture, it must first be developed. The remainder of this section is about documenting the problem space. Creating a documentation of a problem space allows for better understanding of the software development problem at hand. The elements of the problem space should provide an idea of what is present in the problem description and must be realized in the solution space.

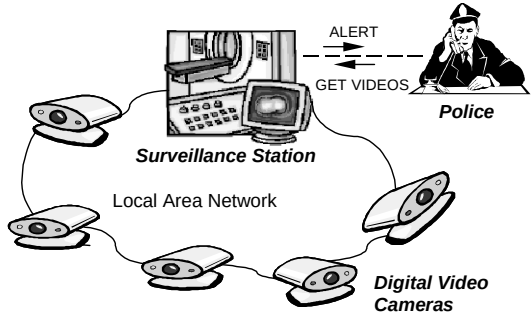


Figure 2.1: An Informal Documentation of the Problem Space

Figure 2.1 shows an informal illustration of a problem space that helps understand and explore the video surveillance service problem; it describes the problem space as a network of elements. An element in a problem space is either a collaborating part (i.e., surveillance station, video cameras or police station) or an interconnection among those parts. Each element of the problem space must be relevant to understand and design the solution.

2.1.3 Documenting Requirements on the Solution Space

Documenting requirements on the solution space is about determining a set of constraints that must be fulfilled by an architectural solution before that solution is actually developed. This allows us to examine how to put into software architecture practice a notion similar to

the popular rule of extreme programming [Beck99], known as “code the unit test first”, and which can be re-formulated as: “*document the requirements on the solution space first*”.

What Architecture Development Process is Needed?

To motivate the use of the above rule, let us have a look at an architecture development process depicted in figure 2.2. This figure shows an approach that helps ensure that the video surveillance service will fulfill the stakeholders' needs, as it allows you to evaluate the software architecture against the expectations of the stakeholders.

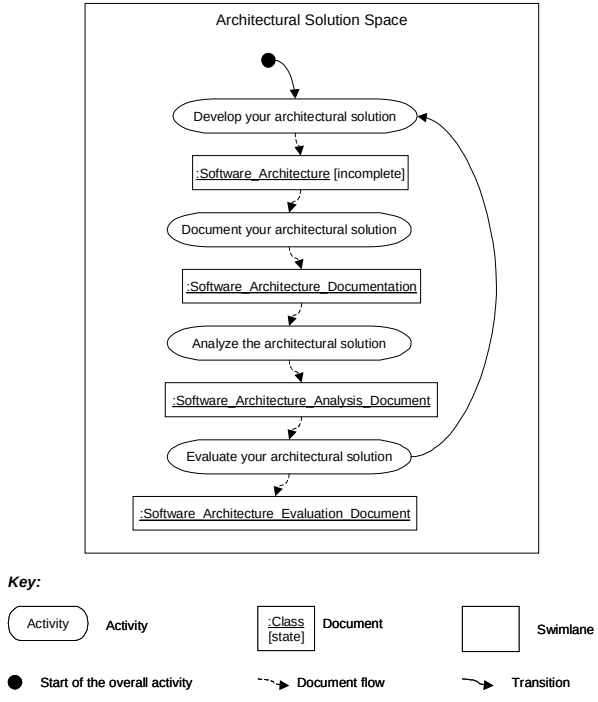


Figure 2.2: Architecture Development Activity Diagram

Figure 2.2 depicts a UML activity diagram describing the development process followed by many architects when building software architectures. Each step in this process is represented by an activity that takes place at one particular stage of the software architecture development process. As shown in the figure, the process itself is documented as part of the architectural solution space (which is represented by the swimlane). The architecture development activity diagram should be interpreted as follows: 1) develop an architectural solu-

tion for the video surveillance problem to be part of the software architecture; 2) document your architectural solution to produce a handbook that is useful for different stakeholders to perform their tasks, respectively, within the project; 3) use this handbook as an input for analyses and reasoning about the essential properties of the video surveillance service; and 4) evaluate your architectural solution against the needs of the stakeholders and against the realizations of the elements introduced in the problem space.

In fact, how practical is such an architecting process? First of all, creating a good software architecture (that is useful for developers and maintainable over time) requires a lot of effort and practical experience. Therefore, waiting to evaluate the software architecture at the end of an architecting process is risky. For instance, following the transition back—from the evaluation step to develop another solution or improve an existing one—can be very expensive. As time and budget are both major concerns that are so relevant to software architects, they can be seen as important decision factors which may lead to the success or failure of a software project.

Which Constructs are Needed in the Architectural Solution?

The following presents an example of informal documentation of requirements on the solution space. Figure 2.3 focuses on fundamental constructs required to explicitly document the video surveillance service architecture.

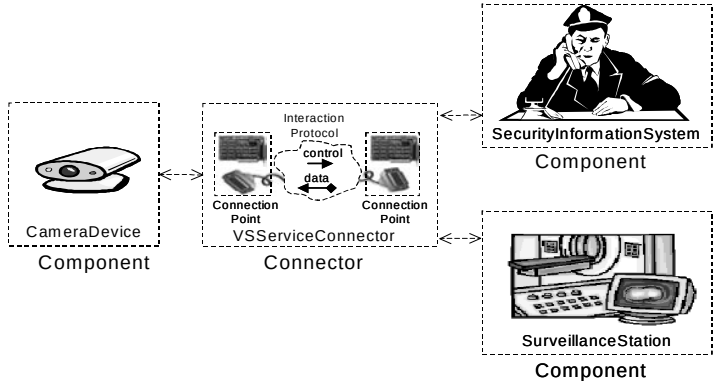


Figure 2.3: An Illustration of Requirements on the Solution Space

Figure 2.3 depicts a set of architectural elements: they are, CameraDevice, SurveillanceStation, SecurityInformationSystem and VSServiceConnector. The video cameras interact with a control station over a particular communication platform. The first three architectural elements embody each an architectural abstraction reified as a *Component*, that is, a kind of represen-

tational construct. The last architectural element represents another kind of representational construct, called a *Connector*.

Both kinds of architectural constructs are equally required when documenting software architectures: a Component represents a location for computation and data store within a system, while a Connector represents a location for interactions among the components. The dashed boxes are used as graphical symbols for both kinds of constructs and for the elements they are composed of. Each dashed box visually encapsulates the realization of a specific element of the problem space, i.e., the picture contained in the box.

A major requirement of the architectural solution consists of providing specifications for the different types of representational constructs. For example, to fulfill the requirements as illustrated in figure 2.3, the architectural solution for the video surveillance service could specify the component type CameraDevice as an abstract representation of individual, geographically distributed video cameras. The SurveillanceStation could be an abstraction for that part of the system that remotely controls the cameras and continuously receives the video streams from the CameraDevice. Similarly, the SecurityInformationSystem could abstract the information system of the police that interacts with the SurveillanceStation. The VSServiceConnector could specify an abstraction for the communication platform; it could have connection points and a protocol of interaction between these connection points, as shown by the nested elements.

Each *connection point* could represent an interface element for the connector that a component will require to interact with other components. But in order to be able to participate in a communication mediated by a connector, a component must implement a mechanism to connect to the connector interface element; the component interface element must match the interface element of the connector. The implementation of such a mechanism might be achieved in software or in hardware; however, the architecture abstracts above any specific implementation details. For instance, in figure 2.3, the two circuit boards, which implement the interface of the connector to the camera device and the video control station, are each encapsulated in a separate connection point.

The *protocol of interaction* specifies the way to perform the communication between the connection points. For example, an architectural solution for the Video Surveillance Service should specify what element in the architecture realizes the notion of the duct carrying the control/data and protocol information exchanged between the components (i.e., the hardware part of the protocol of interaction, shown as cable in figure 2.3). In addition, the dashed cloud shows what is required to realize the software part of the interaction protocol. The bi-directional dashed arrows binding the components to the connector illustrate the connections. *Connections* are required to associate (connect) the elements of a component interface with the connection points of the connector in order to build concrete systems from a Video Surveillance architecture.

2.1.4 Remarks

The description given above is only a small sample of a type of informal documentation. However, an essential goal of the previous two sections is to motivate the need of filling the gap between the specification of software development problems [Jack01] and the description of software architectures [MT00].

A major difficulty with the presented approach for documenting the requirements on the solution space is the lack of means for automating the software architecture evaluation process. Finding an appropriate architecture description language that can be integrated with different software architecture evaluation methods and tools is another problem [CKK02]. A key question is: how to make the idea "*document the requirements on the solution space first*" become a reality in the field of software architecture?

At this stage, formal methods and techniques cannot help much since it is very difficult, or even impossible, to formalize descriptions of early requirements on an architectural solution before developing the actual solution itself.

The informal documentation produced above lacks preciseness. It highlights the need for more precise formalisms that can serve as a basis for communication among stakeholders and for performing analyses on the architectural decisions or properties when documenting software architectures. For example, when considering figure 2.3, it is very difficult to understand how the components interact, or in which sequence they communicate; it is difficult to know what the interaction points for individual components are, or how to specify the connection points precisely.

2.2 Formal Software Architecture Documentation

This section introduces an architecture description language, called Wright [ABV92], which exemplifies a formal software architecture documentation.

Wright is a typical architecture description language that allows one to focus explicitly on the formal specification of a software architecture. Architectures described in Wright are centered around the following ADL constructs: components, connectors, properties, systems and styles. Wright formally represents the notions of component as computation, connector as pattern of interaction, property as constraints, system as configuration, and style as a collection of reusable architectural elements. Thus, describing architectures in Wright allows you to distinguish from one another different kinds of architectural concerns, such as computation/data store, interaction, constraint, configuration, and reuse. Moreover, Wright permits separating the *structure* of a system from its *behavior*. For example, figure 2.4 shows a Wright specification of a software architecture that provides a formal solution to the video surveillance problem; this solution addresses separately the structural aspects and behavioral aspects of the system.

Describing software architectures with Wright requires one to make use of extensions to the CSP notation [MPW92], in order to distinguish events that are initiated (i.e., sent) by a specific type of component from those that are observed (i.e., received). In Wright, a component type describes a localized, independent computation. A Wright connector type specifies interactions among a collection of components. The aim of an explicit use of connectors in Wright specifications is to enhance reuse of recurring patterns of interaction not only in one specific situation, but in many different contexts of communication: connectors allow for decoupling of components.

2.2.1 Documenting Architectural Structure in Wright

The Wright specification of the video surveillance service has two essential parts, called *Style* and *Configuration*. Figure 2.4 shows both parts. Style represents the declaration of a set of properties that need to be fulfilled by any concrete video surveillance service that implements VSServiceArchitecture. Configuration embodies the declaration of one particular instantiation of the VSServiceArchitecture style; this is named VideoSurveillanceSystem.

```
Style VSServiceArchitecture

  Component SurveillanceStation
    Port ssp = videostreamrequest → start → ssp □ §
    Computation = internalCompute → ssp.videostreamrequest → ssp.start → Computation □ §

  Component CameraDevice
    Port cdp = videostreamrequest → start → cdp □ §
    Computation = videostreamrequest → p.internalCompute → cdp.start → Computation □ §

  Connector VSServiceConnector
    Role sink = videostreamrequest → start → sink □ §
    Role source = videostreamrequest → start → source □ §
    Glue = source.videostreamrequest → sink.videostreamrequest → Glue
           □ sink.start → source.start → Glue
           □ §

  Constraints
    ∃ ss : Components, ∀ cd : Components
        ● Type(cd) = CameraDevice ∧ Type(ss) = SurveillanceStation ⇒ connected(cd,ss)

EndStyle

Configuration VideoSurveillanceSystem
  Style VSServiceArchitecture
  Instances station : SurveillanceStation;
            digitalCamera : CameraDevice;
            vssConnector : VSServiceConnector
  Attachments station.ssp as vssConnector.sink;
            digitalCamera.cdp as vssConnector.source
EndConfiguration
```

Figure 2.4: A Wright Description of the Video Surveillance Architecture

The structure of the VSServiceArchitecture style is described by three kinds of architectural elements: Component, Connector and Constraints.

Two component types are described in figure 2.4 which are named SurveillanceStation and CameraDevice. To keep the Wright specification of the architecture simple, we deliberately left out the component type SecurityInformationSystem and its interaction with the VSServiceConnector.

Component (i.e., the component type description) is structured in two parts, the interface and the computation. An interface consists of numerous Ports. Each *Port* represents a point of collaboration at which a component may participate in some interaction. *Computation* specifies what the component does and what it requires to work (i.e., what the component expects from the environment which it interacts with).

The *Connector* consists of two parts: a set of roles and the glue. *Roles* indicate what is expected of a component that will participate in an interaction mediated by the connector. The Glue describes how the participants collaborate to create an interaction. For example, the VSServiceConnector provides two roles (source and sink).

A style description in Wright supports the declaration of properties that must be obeyed by any particular configuration. In the example of the VSServiceArchitecture style, the predicate specified in the Constraints clause indicates that there must exist a component instance of type SurveillanceStation, that is named ss. This component instance must be connected to all digital cameras (i.e., every instance CameraDevice) contained in the system.

The component and connector instances of a Wright description must be composed within a configuration in order to provide a complete description of a particular system; a configuration must conform to a given architectural style. For example, the configuration VideoSurveillanceSystem is described as a collection of instances of two component types: SurveillanceStation and CameraDevice (called station and digitalCamera); ports of component of those types (i.e., ssp and cdp) are attached to two roles (sink and source) of the connector instance vssConnector. The declarations in the *Attachments* clause describes how to assemble different architectural elements to build a particular system; having this as a separate clause allows one to focus on the topology of a configuration for that system. According to figure 2.4, attaching "station.ssp **as** vssConnector.sink" specifies that the station component will fulfill the sink role in the interaction mediated by the vssConnector; the station component will play this role while engaged in communication through the port called ssp. To be exact, all of the information that station outputs to port ssp will be delivered to any component that plays the sink role of the vssConnector. "digitalCamera.cdp **as** vssConnector.source" should be interpreted in a similar way.

2.2.2 Documenting Architectural Behavior in Wright

The Wright specification for the behavior of the Video Surveillance architecture provides an architecture documentation describing: 1) a set of significant events that can be processed

by the components of the system at hand, and 2) the sequences in which these events occur. The notation for behavioral description indicates the direction of the interaction; this allows one to explicitly distinguish events that are initiated by a sending component (overlined events) from those that are observed by receiving components (not overlined).

To describe the behavior of component and connector types, Wright allows us to specify a process for each of the following elements: port, role, computation and glue.

In figure 2.4, the computational behavior description for the component type SurveillanceStation shows that an instance of SurveillanceStation first performs some internal computation (while engaged in internalCompute); then it repeatedly initiates an $\overline{\text{ssp.videostreamrequest}}$ event on the port called ssp and waits to observe an ssp.start event on the same port, "or" it terminates successfully; the successful termination is indicated by the § symbol. "Or" means an internal choice; it indicates that the SurveillanceStation will decide by itself (without any external influence) whether it should make another request or terminate. An internal choice in the behavior is represented by the symbol $\overline{\text{|}}$. The computational behavior of the CameraDevice component type is similarly defined. But, in contrast to the SurveillanceStation component type, the specification of the computational behavior of CameraDevice makes use of an external choice. This is indicated by the symbol □, which means that the computation process is expected to reply to each request, and is not allowed to terminate in advance.

Furthermore, using Wright, you can assign the port of a component to a particular CSP process. This capability allows one to specifically define a local interaction protocol for a particular port. In figure 2.4, the local interaction protocol for the ssp port covers the same behavioral pattern as defined in the computation process of SurveillanceStation, except the internal part (specified by the internalCompute event).

A separate CSP process is required to specify each of the connector roles (source and sink) played by the participant components. In the example, the source and sink roles are deliberately kept simple to allow you to easily comprehend the behavior of the roles, and to explain the matching required between ports and roles when describing Configuration.

In the example, the description of Configuration shows how to attach the port ssp of the station component instance to the connector role, VSServiceConnector.sink, and how to attach the cdp port of the digitalCamera component instance to the VSServiceConnector.source role; these attachments are significant parts of the construction of the particular system, called VideoSurveillanceSystem.

Finally, the Glue process specifies the protocol of interaction among the roles of a connector type. The behavior of the Glue process shown in figure 2.4 can be explained as follows: the SurveillanceStation initiates a $\overline{\text{videostreamrequest}}$ event (via the ssp port) to request video streams from a CameraDevice; the request is mediated by the connector as $\overline{\text{sink.videostreamrequest}}$; it is observed by a CameraDevice as source.videostreamrequest event; the response of that CameraDevice is mediated back as $\overline{\text{source.start}}$ and received as sink.start by the SurveillanceStation component.

2.2.3 Remarks

Wright provides good support for formal expression of the architectural abstractions introduced in figure 2.3. However, the Wright mechanism for separating “the structure” of a system from its behavior suffers from the lack of support for visually modeling multiple kinds of system-level structures; architecture involves more than just one structure.

Moreover, for a connector to mediate interaction among various components, two conditions must be fulfilled:

1. The ports of the components must match the connector roles
2. Both the ports and roles must adhere to the interaction protocol that is specified by the Glue.

Fulfilling these conditions allows one to treat software connectors as first-class citizens in architecture descriptions, and thus to support analysis and reasoning about interaction among components. But requiring protocol adherence in which each port has to be *perfectly matched* to (at least) a role leads to *inflexible* software architectures, because perfect matching is hard to achieve when evolution is a relevant architectural concern. Typically, when requirements change, the architecture must evolve: some ports will have to match many connector roles and vice-versa; thus, maintaining perfect matching becomes problematic. To reduce maintenance effort, software architects might anticipate as many changes as possible in their designs; however, this is not feasible because it complicates the architecture and the development of software that conforms to that architecture. One consequence could be that the project might not finish on time.

2.3 Pattern-Oriented Software Architecture Documentation

This section presents an example of pattern-oriented documentation for the video surveillance service architecture. It starts with documentation of requirements on the pattern-oriented software architecture; then it demonstrates how to apply simultaneously two additional design patterns—the Observer pattern ([GHV+95]) and the Component Configurator ([SSR+00]) pattern. Finally, the section introduces an application of the Pipe-and-Filter architectural pattern ([BMR+96][SG96]) on the entire video surveillance service, and discusses problems related to combining different instances of multiple patterns in a software architecture.

2.3.1 Documenting Requirements on the Pattern-Oriented Architecture

Figure 2.5 presents an informal requirements diagram consisting of a set of constraints to be filled by architectural constructs required: 1) for early evaluation of an architectural solution applied to the problem at hand and 2) for documenting the architectural solution so that it allows one to understand the composition of design and architectural patterns. The

requirements diagram also shows a separation of concerns that is reified in the distinction between “base” and “perspectival” constructs.

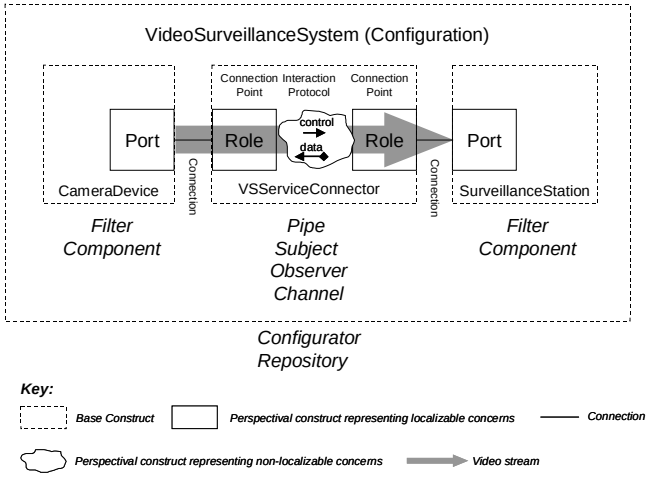


Figure 2.5: Informal Req. Diagram for the Video Surveillance Architecture

Base constructs represent the basic architectural elements that should be globally visible to all stakeholders and from all perspectives. Base constructs should be described in any structural documentation of a software architecture; they are “objective” in nature. The base constructs shown in figure 2.5 include the two component types, the connector type and the system; these are depicted by the dashed boxes.

A *perspectival construct* represents a facet of a base construct—or an aspect of a collection of base constructs. Perspectival constructs have a “subjective” nature; they do not need to be globally visible; instead, they should be relevant to some particular kinds of concerns that are significant from certain perspectives, but irrelevant from others. Indeed, the most significant constructs for understanding, documenting and reusing the video surveillance service architecture are not the base constructs (i.e., the components and connectors) themselves. Instead, the architectural elements of focus are the perspectival constructs that encapsulate the externally visible properties of base constructs. Examples of perspectival constructs include: the roles played by the components and connectors (shown as Ports and Roles), the protocol of interaction between the Roles, and the connections between Ports and Roles.

Clearly, the notion of role as used in the requirements diagram is general and it includes the interface elements of the component types and the connector type. The names shown below the dashed boxes in the requirement diagram designate the roles defined by

the selected patterns. Each *pattern role* should be assigned/realized by one or more perspectival constructs in the architectural solution. For example, the pattern roles named *Filter* and *Component* should be realized by the ports of the CameraDevice and SurveillanceStation component types. The pattern roles called *Pipe*, *Subject*, *Observer*, and *Channel* should be realized by the connector roles and the interaction protocol of the VSServiceConnector. The configuration roles named *Configurator* and *Repository* should be fulfilled by the VideoSurveillanceSystem.

The lines between the component ports and connectors indicate the requirement for a mechanism to connect the ports of components to the roles of connectors. Finally, the transparent arrow (from the CameraDevice to the SurveillanceStation) exemplifies the need for a means to document the direction of the video stream flowing through the VSServiceConnector.

2.3.2 An Application of Architectural and Design Patterns

This section documents a software architecture for the video surveillance service that applies three different patterns—the Observer design pattern, the Component Configurator design pattern and the Pipe-and-Filter architectural pattern.

2.3.2.1 Rationale for Applying the Observer Design Pattern

The rationale for this design decision is to enable a one-to-many interconnection between the connection points so that when one connection point changes, all the related connection points are notified and updated automatically.

2.3.2.2 Documenting An Observer Pattern Occurrence

The following shows an example that documents an Observer pattern occurrence used to model the VSServiceConnector as a “pattern” of interaction utilizing the Observer protocol.

A pattern occurrence (or instance) binds Classifier Roles to Classifiers. As depicted in figure 2.6, a pattern occurrence can be modeled by a named UML Collaboration which is described by a set of roles: Classifier Roles and Association Roles. A *Classifier Role* represents a placeholder for a specific Classifier within a concrete system. Subject and Observer are two examples of Classifier Roles. *Classifiers* involved in the pattern occurrence are called participants. For example, the Classifiers Publisher and Subscriber are two participants of the Observer pattern occurrence shown in figure 2.6. A Classifier that fulfills a specific role (called Classifier Role) is said to be bound to that role. A *binding* is shown as a dashed line drawn from the pattern to the *Classifier*s.

The bindings between the Classifier Roles and the Classifiers should be interpreted as described below.

- *Publisher is bound to Subject*: the Publisher Classifier must satisfy the Subject Classifier Role (i.e., the Publisher must realize the corresponding responsibilities of Observer pattern role, as defined in [GHV+95]);

- *Subscriber is bound to Observer*: the Subscriber Classifier must fulfill the Observer Classifier Role;

Using Classifiers, we have more flexibility in the design: for example, the design of the VSServiceConnector leaves open whether Publisher and Subscriber will be modeled as classes, subsystems or interfaces. The specifications of the Classifier Roles are taken from the pattern description (in the catalogue).

Figure 2.6: A Higher-Level Description of the Observer Pattern Instance

So far, the structural description of the Observer pattern application shown at this stage does not address the connection points. The next design step concentrates on the connection points.

Figure 2.7 presents a refinement of the above instance of the Observer design pattern. It shows the specifications of both Classifier Roles (Subject and Observer) and describes how they are associated with one another. The relationship between the Classifier Roles is represented by an unnamed *Association Role*. This Association Role can be dynamically bound to associations among Classifiers.

Figure 2.7: An Observer Pattern Instance with Focus on Connection Points

publisher and subscriber each represent an endpoint of the unnamed Association Role, which is referred to as the *Association-end Role*; these Association-end Roles specify the connections between the Classifier Roles Subject and Observer, and the unnamed Association Role as shown in figure 2.7. The multiplicities attached to the Association-end Roles (0..1 and *) indicate that at most one “publishing” connection point can be interconnected with many “subscription” connection points.

The features of the Classifier Roles are defined in the two compartments below the name compartment. Having a definition of features in the Classifier Role means that all Classifiers acting as a Subject will have to provide an attribute to keep the subject state; Any Classifier that acts as a Subject must also provide the capability to publish notifications, to read and write the subject state attribute. On the other hand, all the potential Observers must be able to receive the notifications and to perform updates.

Figure 2.7 shows a facet of the instantiation of the Observer pattern and is useful documentation if your goal is to understand and represent the connection points. But it is less

useful when your goal is to describe and reason about the protocol of interaction between the connection points.

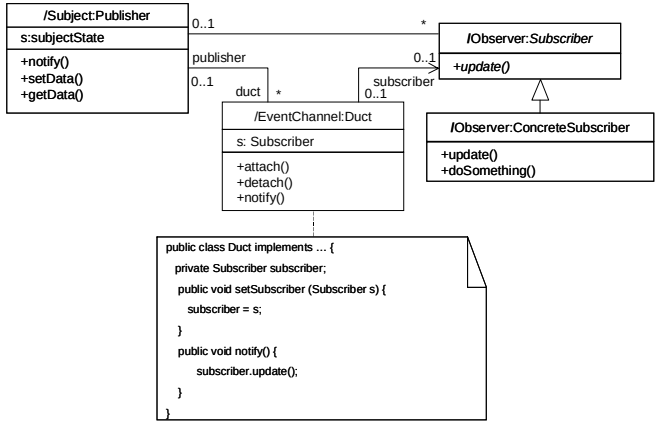


Figure 2.8: An Observer Pattern Instance with Focus on the Interconnection

Figure 2.8 provides more details on realizing the different roles involved in the connector design. These include the unnamed Association Role specifying the interconnection between the connection points and the Classifier Roles that specify the connection points themselves. The figure shows the static structure of the Observer pattern occurrence that consists of the Classifier Roles Subject, Observer, and EventChannel, and the Classifiers Publisher, Duct, Subscriber and ConcreteSubscriber. The Subscriber connection point is modeled as an abstract Classifier to allow developers of the connector to define various ConcreteSubscribers that can be needed for different purposes.

The interconnection is explicitly modeled by the Duct Classifier to allow for explicit specification of the event propagation behavior. There must be only one instance of the Classifier Duct that can act as an event channel interconnecting two connection points. This instance is visible by the publisher only; it must be bound to the Association-end Role called duct. The note attached to the Duct shows an example of Java code describing how the notification events are propagated along the interconnection. This piece of Java code implements the Duct Classifier as a class.

2.3.2.3 Rationale for Applying the Component Configurator Pattern

We will now use the Component Configurator Design Pattern to address the configurability dimension of the software architecture. A major motivation for instantiating the Component

Configurator [SSR+00] in this example is to help understand and document two architectural concerns:

1. How to configure the components of the video surveillance service into various processes without having to shut down and re-start the running application processes.
2. How to link and unlink application components dynamically at runtime, without modifying, recompiling or relinking them.

Each of the concerns presents a different facet of the configuration problem addressed by the pattern: one facet of the problem is the configuration of components, and the second one is the configuration of interconnections among components.

2.3.2.4 Instantiating the Component Configurator Pattern

The documentation of the Component Configurator pattern instance can be separated in two parts: *structural documentation* and *behavioral documentation*.

Structural Documentation

The structural documentation of a Component Configurator instance describes a partial solution to our configuration problem; this solution is depicted in figure 2.9 as a UML collaboration diagram. The diagram shows the structural characteristics required to configure individual components of the system[1].

The system is represented by the Classifier, called VideoSurveillanceSystem, which involves two Classifier Roles, Repository and Configurator. The Classifier VideoSurveillanceSystem is said to play both Classifier Roles and each of the roles must be bound to some concrete elements of the system. The Classifier Roles Repository and Configurator are implicitly bound, that is, the internal Classifiers of the system to be bound to the roles are either hidden in this diagram or the architect has not decided yet which of the internal Classifiers will play the given roles.

An essential condition for the VideoSurveillanceSystem to act as repository (i.e., to play the Repository role) is to be able to add different configurable components to its "configuration space"; it should provide support for *initialization*, *termination*, *suspension*, and *resumption* of the components; VideoSurveillanceSystem should also allow one to get information about each of the components present in the repository. The part of the system that plays the Repository role must be able to store all the configurable component types SurveillanceStation and CameraDevice. Both component types and their abstract super type Configurable (written in italic) are modeled as Classifiers.

1. The term "system", as used here, refers to a software application rather than a system in the sense of ADLs (as used in section 2.2).

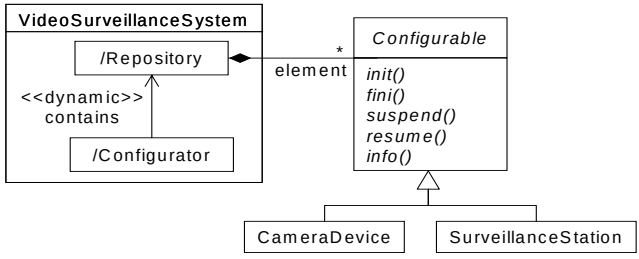


Figure 2.9: Structural Instance of the Component Configurator Pattern

The Classifier Role Configurator is shown as a placeholder for a specific software infrastructure that should allow you to load existing components from the repository, to configure those components individually, and to compose them for the purpose of generating new systems.

The Association Role between the Classifier Roles, Repository and Configurator, denotes a dynamic relationship, which indicates that every configurator must have its own configuration space (or repository) that holds the configurable elements.

Behavioral Documentation

The way the system interacts with its components to achieve the configuration is shown as UML sequence diagram in figure 2.10.

The interaction among the VideoSurveillanceSystem and its components is achieved in three phases:

- *Phase 1: Component initialization*. This initializes each individual component that can be selected from the Repository and added to the Configurator.
- *Phase 2: Component processing*. Once added to the Repository (temporary configuration space), the services provided by one component can be used by another component.
- *Phase 3: Component termination*. This involves the selection of components and their removal from the Repository.

The diagram shown in this figure addresses another facet of the configuration problem which focuses on the dynamic structure of the pattern occurrence. It documents the messages (e.g., init(), insert(), etc.) that are exchanged among the instances of the participant Classifiers.

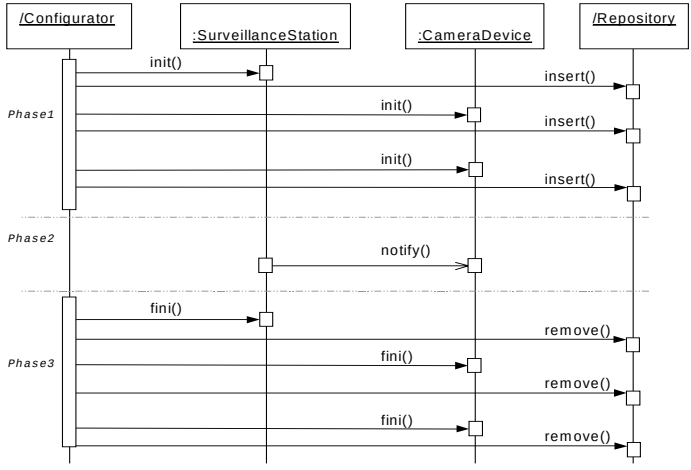


Figure 2.10: Behavioral Instance of the Configurator Design Pattern

The sequence of interactions required to configure the components of the VideoSurveillanceSystem is indicated by the timelines (the vertical lines). The vertical rectangles placed on the timelines show the activation boxes that indicate when instances are involved in some computation. The underline indicates that the model elements are instance-level elements. Instances are represented by the boxes containing some text in the following format: "Classifier name/Classifier Role:Classifier". This text format expresses an explicit bindings between Classifiers and Classifier Roles.

2.3.2.5 Rationale for Applying the Pipe-and-Filter Pattern

According to [BMR+96], the *Pipe-and-Filter* architectural pattern provides an organization for a family of systems that process stream of data. Such systems are essentially composed of two kinds of elements: *pipes* and *filters*. Applying this architectural pattern to the the video surveillance system allows one to encapsulate each processing step into a component filter and to convey video streams through the pipes between neighboring filters.

2.3.2.6 Instantiating the Pipe-and-Filter Architectural Pattern

The Pipe-and-Filter pattern instantiation, shown in figure 2.11, provides three elements: two component types, the CameraDevice and the SurveillanceStation, and the connector type VSServiceConnector. Each component type has a port that represents a filter. From the connector

point of view, each processing element is encapsulated in a Filter role (i.e., component port). Thus, video streams can be passed through the pipe positioned between the filters.

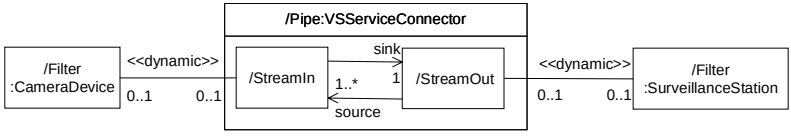


Figure 2.11: A High-Level Structure of a Pipe-and-Filter Instance

The connector type VSServiceConnector itself provides two kinds of roles—static roles and dynamic roles. Static roles (e.g., StreamIn and StreamOut) were previously referred to as connector roles; they are part of the structure of the connector type. The dynamic roles (e.g., source and sink) are rather behavioral; they are transient representations of the participants in an interaction.

The Classifier CameraDevice is bound to the Filter Classifier Role; it must act as a stream producer port that is connected (i.e., attached) to the connector role StreamIn. Similarly, the filter on the SurveillanceStation side must behave as a stream consumer port that is connected to the connector role StreamOut. The connections between the component ports and the connector roles are modeled as dynamic associations.

2.3.3 Documenting an Architecture for a Family of Software Systems

As mentioned earlier, the description shown in figure 2.9 is practical for describing an individual software application, but it is not viable for an architecture that can serve as a means for communication among stakeholders in the construction of a family of applications. To describe a family of applications, Configurable should not be modeled as a Classifier; instead, it should be modeled as a Classifier Role—i.e., a placeholder for the configurable elements, such as SurveillanceStation and CameraDevice.

This solution provides more flexibility: it allows one to decide at will how and when to bind the Configurable Classifier Role to the different Classifiers (i.e., the component types) of the applications. The relationship stereotyped with <<bind>> represents the bindings between Configurable, SurveillanceStation and CameraDevice. A documentation of this solution is depicted in figure 2.12.

In contrast to the previous design, this solution provides an architectural model that applies to a family of systems; it can be realized in different ways (by using different bindings) to create different software applications.

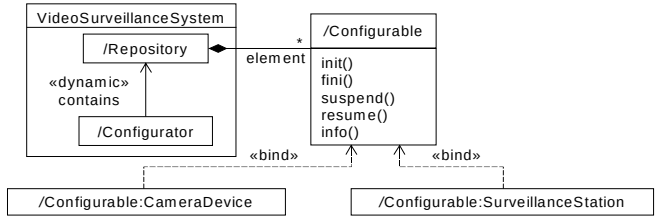


Figure 2.12: A Structural Instance of the Component Configurator Pattern

The sequence diagram shown in figure 2.10 can be easily updated to fit the new architectural model documenting the Component Configurator pattern instance. To achieve this, we just need to add the Configurable Classifier Role in the interaction diagram and remove the Classifiers, as shown in figure 2.13.

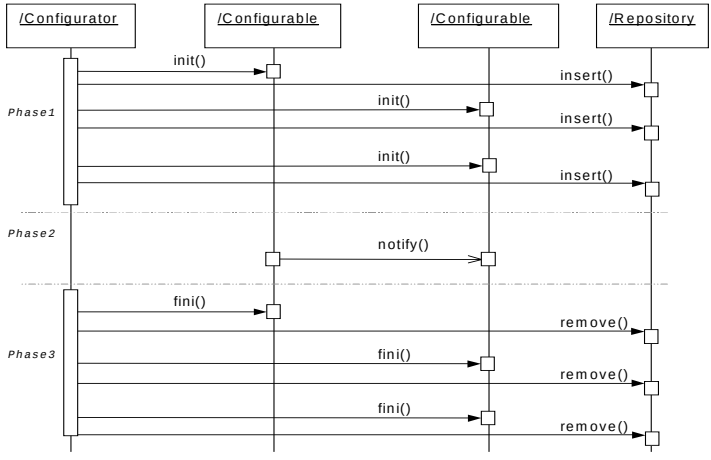


Figure 2.13: Behavioral Instance of the Component Configurator Pattern

In figure 2.13, the messages are exchanged among the objects of classes playing the Classifier Roles: that is, Classifier Roles become more important in the architecture documentation than Classifiers. As a consequence, the participant components themselves (represented by the Classifiers) are not first-class citizens of the architectural description.

Instantiating a pattern at the design or architectural level of abstraction is a very human-intensive process; therefore, providing good instantiation of a pattern depends strongly on the experience of the developer or architect. This makes it difficult to know when and at which level of abstraction a pattern should be used.

2.4 Concluding Remarks

The motivating example presented in this chapter shows how informal notations, architecture description languages, and design and architectural patterns can be used to document different characteristics of an architecture. The example highlights some complementarity between the existing trends in software architecture; the view-oriented software architecture trend has not yet been presented.

The problem presented in this chapter has many different aspects that a software architect needs to separate from one another. Some of the aspects have architectural significance and others do not. Throughout the case study, we have referred to the architecturally significant aspects of the video surveillance problem as *architectural concerns*, and we observed that those architectural concerns are relevant to both the problem space and the solution space.

The informal approach taken in section 2.1 provides the software architect with an idea of what is required to express the structural characteristics of an architecture before starting to build that architecture. The approach shows that informal descriptions can be helpful to establish a bridge between a given problem and an architectural solution to that problem. It demonstrates how informal descriptions help get an idea about various kinds of abstractions that are required to formally describe an architecture.

As mentioned in section 2.2.3, following the ADL-based trend to address other kinds of concerns is rather problematic. An ADL such as Wright hinders the identification, understanding and separation of multiple kinds of concerns [KCS+02]. For example, problems related to the use of Wright are its:

- inability to localize information about interaction concerns other than protocols of interaction
- lack of mechanisms for describing crosscutting aspects of both components and connectors (e.g., synchronization and security)
- lack of support for integrating architecture descriptions with common software development artifacts.

Applying patterns in architectural descriptions can hardly be achieved with formal notations. Pattern applications have been documented by using combinations of both informal and semi-formal notations. While using formal notations, section 2.3.1 motivated the distinction between base constructs and perspectival constructs.

A major requirement on perspectival constructs is to provide the capability to support explicit representations of roles and the binding of such roles to concrete components and connectors in different architecture descriptions. Specifically, this capability involves support for:

- representing and instantiating pattern roles appropriately, and
- complementing an ADL-based solution

A lesson we have learned from the example presented in section 2.3.3 is that a design pattern can be intentionally applied to structure individual elements of a software architecture. If the application of the pattern has no crosscutting effects, then the resulting design can be referred to as “non-architectural”. Otherwise it is said to be architectural, that is:

- *the design pattern instance involves a crosscutting structure which might be relevant to the software architect*. For example, the Observer protocol behavior has a crosscutting effect on the participant components, because it governs the rules for mediating the interaction among the components.
- *the design pattern instance addresses kinds of concerns that are known to be architectural in nature*. For example, the application of the Component Configurator design pattern addresses two facets of the configuration problem, though configuration is an architectural concern.

The approaches presented in this chapter complement each other; each approach is best suited to understand and describe only some particular characteristics of software architectures, not all. Each approach presented has focused on the description of the software architecture of the video surveillance service: none of the approaches supported the distinction between the software architecture and its description; and none of them has allowed one to explicitly address, simultaneously, individual concerns, at different levels of abstraction.

While the solutions presented throughout this chapter have implicitly addressed various concerns, it is not possible to localize any of these concerns in the architecture. Consequently, we cannot find out whether the concerns of importance to the problem at hand have been addressed in the software architecture. For instance, all we know from the presented descriptions is that the software architecture addresses a number of generic architectural concerns, such as configuration, interaction, reuse, etc. What about the concerns that are specific to the given problem? We believe that in order to build software architectures that address a problem at hand, we need a *concern-oriented approach to software architecture* that supports the *design by concerns* paradigm.

Chapter 3:
Related Work

This chapter introduces the foundations of the concern-oriented approach to software architecture. It presents two conceptual frameworks: an architecture description framework which is part of the ANSI IEEE standard for architecture documentation, called IEEE-Std-1471-2000; and an aspect-oriented software engineering framework, known as concern spaces, which is part of Multidimensional Separation of Concerns (MDSOC). The Unified Modeling Language (UML) is introduced as a linguistic framework that can be used to establish a bridge between both conceptual frameworks. The chapter concludes with the needs for implementing these frameworks using UML.

3.1 IEEE Recommended Practice for Architectural Description

The IEEE recommended practice for architectural description for software-intensive systems was first developed as an IEEE standard known as IEEE-Std-1471-2000 [IEEE00]. It has also been called ANSI/IEEE-Std-1471-2000 since its adaptation by ANSI (American National Standards Institute). We will refer to this standard as IEEE-Std-1471 in the remainder of this dissertation.

The purpose of the IEEE-Std-1471 is to facilitate the expression and communication of architectures. It puts emphasis on the creation, analysis, and sustainment of architectures of software-intensive systems, and the recording of such architectures in terms of architectural descriptions. Essentially, the recommendations of IEEE-Std-1471 focus on two proposals: a conceptual framework for architectural description, and a pronouncement of what is required to evaluate the conformance of an architecture description to the standard IEEE-Std-1471.

3.1.1 IEEE-Std-1471 Conceptual Framework

The *conceptual framework* of the IEEE-Std-1471 establishes several terms and concepts that are relevant to the content and use of architectural descriptions; it also includes the relationships among the concepts. Especially, the conceptual framework defines the term architecture as “the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution”. In addition, it refers to an architecture description as “a collection of products to document an architecture”. By these definitions, IEEE-Std-1471 makes clear the *distinction between an architecture description and an architecture* of a software-intensive system.

According to IEEE-Std-1471, a *viewpoint* is a specification of the conventions for constructing and using a view, while a *view* is a representation of a whole system from the perspective of a related set of concerns. *Concerns* are those interests which pertain to the system development, its operation, or any other aspects that are critical or otherwise important to one or more stakeholders. Concerns can be abstract or logical concepts from a certain domain, but they may also include system considerations such as performance, reliability, security, distribution, and evolvability.

In this framework, the terms architectural viewpoints and views are first-class citizens. Figure 3.1 illustrates in a UML class diagram a summary of the conceptual framework. In this diagram, each class stands for an element of the conceptual framework. Every association has two roles that are optionally named. The name of a role played by a class in an association is shown on the opposite side of the association. The associations in the diagram are unnamed. Each role name indicates the direction in which an association should be read. Each role has a multiplicity that is optionally shown. Multiplicities are one-to-one if not otherwise written.

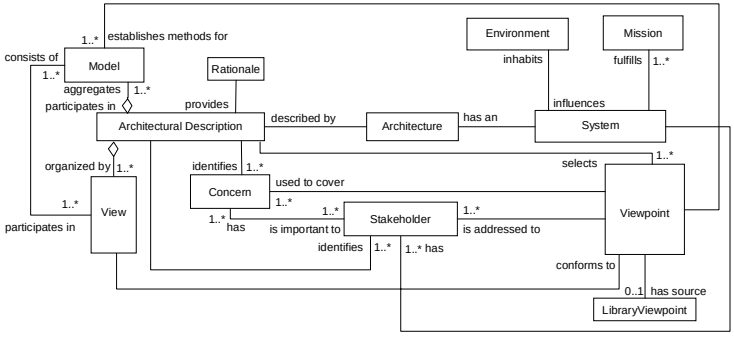


Figure 3.1: Conceptual Framework of IEEE-Std-1471 (from [IEEE00])

Figure 3.1 should be interpreted as follows: Every System has an Architecture, an Environment, and one more Missions. Each Architecture is described by an Architectural Description. An Architectural Description is organized into one or more constituents called architectural Views and one or more Models; it allows one to select one or more Viewpoints and to identify many Concerns and many Stakeholders.

A Viewpoint may have at most one source, called Library Viewpoint; it addresses a set of Stakeholders and covers many Concerns. A Viewpoint establishes the rules by which Models composing a View are created, depicted and analyzed. The View must conform to the Viewpoint. A Concern can be relevant to many Stakeholders and a Stakeholder may have many Concerns.

3.1.2 General Conformance Requirements

An architectural description conforms to IEEE-Std-1471 if it comprises the following elements:

Overview of architectural documentation. This includes the identification information, summary, context, glossary, change of history, and references.

Identification of stakeholders and concerns. The standard gives a minimum list of stakeholders and concerns that must be identified by any conforming architectural description.

Architectural viewpoints. Selected viewpoints must be identified. Each identified viewpoint must be specified by:

- the viewpoint name
- the stakeholders to be addressed by the viewpoint
- the concerns to be addressed by the viewpoint
- the language, modeling techniques, or analytical methods to be used
- a rationale for the selection of each viewpoint, and
- other additional information, including consistency and completeness checks, evaluation techniques, heuristics, patterns, or any useful guideline can be incorporated.

Architectural views. Each view must correspond to the specification of exactly one viewpoint. The architectural models included in that view must fulfill the specification of the corresponding viewpoint.

A record of every inconsistency among all views. This includes an analysis of inconsistencies among the architectural views all together.

Architectural rationale. This includes the rationale for the architectural concepts selected, if possible together with evidence of the consideration of alternatives and the rationale for the choices made.

3.1.3 IEEE-Std-1471 Lacks Realizations

Despite the importance of the IEEE-Std-1471, the software industry still lacks automated and systematic architectural approaches that enable effective production of architecture descriptions conforming to the conceptual framework of IEEE-Std-1471.

Furthermore, according to the conceptual framework, an architecture represents a set of abstractions that is manifested by an architecture description, which is itself a set of concrete software artifacts. A viewpoint (i.e., abstraction) is similarly related to a view (i.e., concrete artifact). Remarkably, an architecture description is organized into multiple views, but an architecture is not related to multiple viewpoints, although the standard says that each architecture should be built or “viewed” from various viewpoints.

Without a relationship between architecture and viewpoints, we cannot relate a given architecture to various sets of stakeholders' concerns. Lacking such a relationship complicates any realization of the IEEE-Std-1471 that enables producing concern-oriented architecture descriptions. Thus, software architects cannot develop and describe architectures of software-intensive systems in a concern-oriented way.

3.2 The Unified Modeling Language

The Unified Modeling Language (UML) is the result of an effort that was instantiated by three prominent (object-oriented) methodologists, Booch, Jacobson, and Rumbaugh [BRJ98][RJB98]. UML has been submitted to and approved by the Object Management Group (OMG) as a standard [Omg01]. The standardization of UML not only provides a better possibility for tool compatibility, but it also joins research on improving expressiveness and preciseness of a single language. UML is process independent and therefore it does not prescribe how its notations should be used.

Using UML notations provides several advantages, including:

- UML offers a common language, uniting different development methods in terms of notation and vocabulary, and allowing tool interoperability between different vendors.
- UML provides a rich set of notations that can be used to describe various aspects of a software-intensive system. It offers eight diagram types: activity diagrams, implementation diagrams (component and deployment diagrams), interaction diagrams (sequence and collaboration diagrams), statechart diagrams, class diagrams, and use case diagrams.
- UML provides built-in extension mechanisms (stereotypes, constraints and tagged values) that can be selectively applied to model elements. These extension mechanisms assist methodologists in defining new model elements that are not found in standard UML and which are required for some particular purpose. For example, we proposed in previous work [KS00a] a UML-based approach to software architecture description using the IEEE-Std-1471, which focused on incorporating key abstractions, found in nearly all-existing ADLs, into UML.
- UML provides a textual constraint language known as OCL, the Object Constraint Language [WK98]. OCL is a formal language based on set theory and first-order predicate logic that is used for describing expressions. OCL can be used in different ways to increase preciseness of UML models beyond the limitations of the graphical diagrams. Typically, it can be used to define a set of constraints, to define the properties of a stereotype, or to express invariants of systems, and the pre- and postconditions for operations.
- UML provides a metamodel that is itself defined in UML and which describes, together with different sets of OCL constraints, the semantics of the language itself.

3.2.1 UML Is Not Concern-Oriented

Because of its origin in object-oriented methods, UML has a bias towards software decomposition along multiple kinds of concerns (see next section). UML was not intended for modeling software architectures; however, various constructs have been incorporated that are convenient for describing architectures.

Architecture descriptions resulting from existing UML-based approaches are not concern-oriented: although these approaches enable implicitly the "reification of stakeholders' concerns into software", software architects cannot identify the model elements into which individual concerns have been reified; architects using UML are not aware of the process of reifying the stakeholders' concerns (outside the software) into the architectural models they build (inside the software). Consequently, using current UML-based approaches makes validation of software architectures against the stakeholders' concerns very difficult.

3.3 Multi-Dimensional Separation of Concerns

A major goal of *separation of concerns* [Par72] is to lessen the amount of complexity software developers must contend with, while reducing significantly the impact of change. Multidimensional separation of concerns (MDSOC) provides a conceptual framework for advanced separation of concerns. It was first introduced by Tarr et al. [TOW+99] and calls attention to a set of mechanisms for composition and decomposition throughout the software life cycle, including architecture.

The MDSOC introduces a number of concepts and issues that are useful for understanding the organization of concerns in software and for supporting advanced separation of concerns appropriately across different software engineering approaches; however, the concepts and issues defined in MDSOC are independent of any specific approach. MDSOC describes a set of goals and requirements that must be fulfilled by any specific approach realizing it.

3.3.1 Conceptual Framework of MDSOC

The notion of *concern space* forms the foundation of the conceptual framework. It provides a locus for expressing the concern structure for multiple systems. A concern space constitutes a kind of "multi-dimensional repository" within which the body of software—including all artifacts belonging to the software development effort and the product of the software development process itself—can be explored, selected, analyzed, combined or otherwise manipulated.

Figure 3.2 illustrates an overview of the conceptual framework for MDSOC that is shown as a software concern space. The concern space consists of a set of artifacts written in different languages. As example, the figure shows two models m_i and m_j that consist each

of two diagrams Di1 and Di2, and Dj1 and Dj2, respectively. While built separately, both models are used in the same documents Doc1 and Doc2 (indicated by the lines between models and documents). Other software artifacts related to the models, including the source code implementing each model, can be stored in different databases shown as DB1 and DB2. The models, their diagrams and the program code realizing each model are made up of various units that work together to achieve the goals of the software system under development. Units of the same or different kinds can be combined in various ways to address concerns of importance to one or more developers. Concerns that are significant to developers can be of different kinds, and developers need to distinguish such kinds of concerns from one another to facilitate the development tasks.

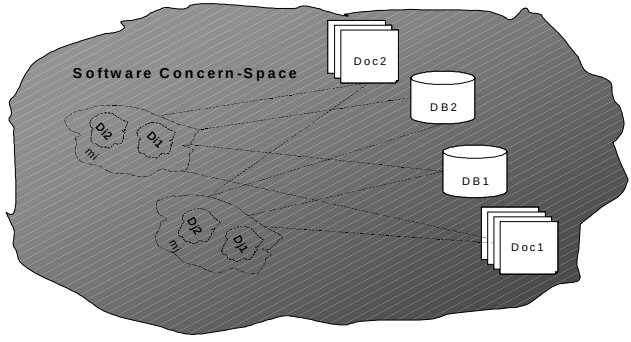


Figure 3.2: An Overview of the Conceptual Framework for MDSOC

The key concepts and issues that make up the conceptual framework for MDSOC are defined as follows:

Software artifact. Software is made of artifacts that consist of descriptions in appropriate artifact languages. Artifacts represent a major part of the body of software; they comprise all kinds of documents of interest in software development.

Artifact language. An artifact language is a formalism for describing software. It includes programming languages, specification languages and standard modeling languages, such as Java, AspectJ, UML, OCL, etc.

Unit in software. A unit is an identifiable portion of software within an artifact. Criteria for determining the nature of a unit are dictated by the artifact language and the level of granularity at which the unit can be used.

Kinds of units. Units may be of different kinds. Some units, called compound units, can be obtained through the composition of other units. The composite units that cannot be decom-

posed are referred to as primitive units. Some units can be defined within the context of other units; they are said to be contextual units.

Area of interest in a body of software. Different stakeholders may have different areas of interest in a given concern space. Each area of interest can be seen as a specific concern that pertains to the needs of a particular stakeholder.

Mapping between concerns and units . Stakeholders interested in different areas of interest need to focus on different artifacts to address different concerns, including configuration, use cases, security, features, etc. Each individual concern has a set of units that pertain to it. A concern is defined as a predicate of units. The units in a given set are “affected” by the concern corresponding to that set.

Dimensions of concern. A dimension of concern is a way of/or an approach to decomposing software according to one particular kind of concerns. Each dimension of concern corresponds to one kind of concern. For example, UML allows one to decompose software along classes, use cases, diagrams, components, or subsystems.

Separation of concerns along multiple dimensions. Separation of concerns along multiple, arbitrary dimensions allows one to keep the units pertaining to different concerns separate from one another. A collection of units that pertains to the same concern is called a module. The module is said to encapsulate that concern.

On-demand remodularization. Developers may recognize the need for a new decomposition that was not foreseen when they started a software project. On-demand remodularization allows one to achieve such decomposition without invasive change.

3.3.2 Concern-Space Modeling Schema

A ***co***ncern-***s***pace ***mo***deling ***s***chema, known as *Cosmos* [SR02], has been proposed by Sutton and Rouvellou to define a concern space as a structured representation of concerns and their relationships. A Cosmos schema complements MDSOC; however, in contrast, it focuses explicitly on matters of interest pertaining to a body of software rather than the software itself. The authors of Cosmos have argued for the need of explicit concern-space modeling that goes far beyond the production of software artifacts, giving emphasis to the intent of artifacts.

According to [SR02], a general-purpose concern-space modeling schema should provide support for:

- representing arbitrary kinds of concerns
- representing composite concerns
- representing arbitrary relationships among concerns
- associating concerns with arbitrary software units, work products, or system elements

- modeling of concerns independently of any specific methodology, programming language, development formalism or stage of the software lifecycle

Figure 3.3 shows the Cosmos perspective of a software concern-space. It complements figure 3.2 through an explicit representation of a set of related concerns, shown as c0, c1, …c14. These concerns are referred to as *logical concerns*: they represent the concepts in which stakeholders are interested with respect to a system or artifact. On the other hand, *physical concerns* represent the system elements or software artifacts which stakeholders need to work with; this category of concerns also includes artifacts or system elements to which logical concerns can be applied.

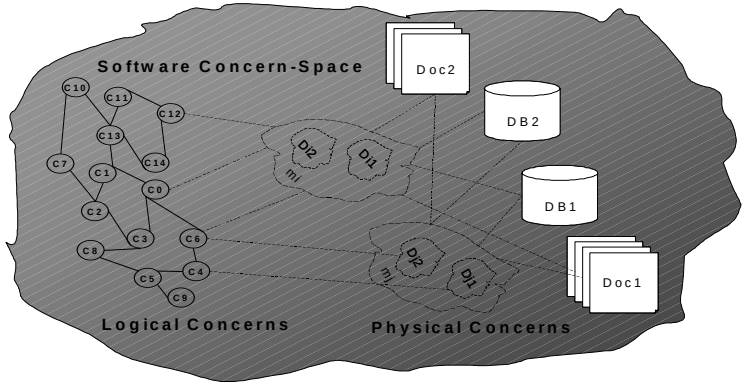


Figure 3.3: The Cosmos Perspective of the Software Concern-Space

3.3.3 General Requirements for Achieving MDSOC

For now, we focus on the general requirements for realizing MDSOC in any software engineering approach. According to [TO00], in order to achieve MDSOC, developers must:

- be able to identify multiple concerns and dimensions (i.e., kinds of concerns), *simultaneously.* "Tyrant" (predominant) dimensions must not be allowed to disqualify decomposition of software along other kinds of concerns;
- be able to identify further concerns and dimensions of importance, *incrementally,* at any time, all through the software life cycle;
- not be forced to take into account concerns that do not pertain to their needs;
- be able to represent and manage *overlapping* and *interacting* concerns, and to identify the points of interaction and maintain proper relationships across the interacting concerns;
- be able to integrate separate concerns and to raise new ones;

- be able to accomplish *on-demand remodularization*: that is, to impose new decompositions on existing software, without invasive change, explicit refactoring, re-engineering, etc.;
- be able to choose, at any time, the best modularization for the development task at hand without, perturbing the existing ones.

3.3.4 Existing Realizations of MDSOC

So far, the most popular realization of MDSOC has been developed by its authors themselves. Their realization is known as an aspect-oriented software development approach called *hyperspaces* [TO00]; the tool supporting the hyperspaces for Java is called *HyperJ* [Hyp03].

The hyperspaces approach takes the premise that each concern of importance has a collection of software units pertaining to it. This is an important characteristic of a concern space as it allows one to distinguish between “reifiable” and “non-reifiable”[1] concerns. Non-reifiable concerns are those for which the pertaining set of units is empty. Currently, the hyperspaces approach only supports multidimensional separation of concerns for Java. Hyperspaces support for UML is still lacking.

3.3.5 Units Are Inside Software, But Not the Concerns

As an advanced principle and technique of software engineering, MDSOC provides superior mechanisms for separation of concerns, but what it really achieves is separation of multiple units along multiple dimensions, at multiple stages of the software lifecycle. MDSOC deals with units explicitly, while remaining rather silent on concerns.

Units are identifiable pieces of software that can be composed using an artifact language. A particular composition of units can be relevant to a particular concern, in which case the composition of units is said to provide the reification of the given concern into software. However, it is critical to understand that concerns are not inside of software; instead, they are part of the stakeholder’ world, which is external to software. MDSOC focuses on new mechanisms for separating concerns, but indeed the notion of concern itself is implicitly defined—as a general predicate over a set of units—within the conceptual framework of MDSOC. Without mechanisms supporting explicit identification of concerns independently of the units that encapsulate them, software architects will not be able exploit the power of MDSOC. Lacking clear separation between units and concerns, MDSOC becomes inefficient in developing and describing concern-oriented software architectures.

1. Note that the terms “reifiable” and “non-reifiable” concerns are new notions introduced in this dissertation; they are not part of MDSOC, Hyperspaces or HyperJ.

3.4 Aspect-Oriented Software Development

Aspect-oriented software development (AOSD) [AOSD] aims at identifying, encapsulating, and explicitly representing characteristics of software that are very difficult, or even impossible, to capture with object-oriented software development approaches, including approaches based on UML. Such characteristics typically involve concerns cutting across the boundaries of several model elements and modules, for example, security, logging, tracing, etc.

An essential awareness in aspect-oriented software development is that the boundaries of stakeholders' *concerns* hardly ever correspond to the boundaries of the modules reflecting those concerns in software systems. Based on this awareness, the AOSD community has recognized that the ability to understand, to model and to reason about concerns, cutting across both structure and behavior of components, is critical to the design of any complex software system.

A major contribution of AOSD to software engineering is multidimensional separation of concerns. A different contribution of AOSD is a recent programming paradigm, called *aspect-oriented programming* (AOP) [KHH+01], which provides mechanisms for implementing crosscutting concerns as separate modules.

Another (emerging) contribution of AOSD to the model-driven software engineering community is *aspect-oriented modeling.* Aspect-oriented modeling (AOM) is an up-and-coming area of aspect-oriented software development that aims at providing techniques, principles, and mechanisms for identifying, analyzing, managing, and representing crosscutting concerns in software design and architecture [AOM03]. A major objective of AOM consists of filling the gap between aspect-oriented requirements engineering (AORE) [RMA03] and aspect-oriented programming (AOP).

So far, diverse approaches using UML built-in extension mechanisms have been proposed to support the description of aspects at design level [CW00][KKS02][SHU02]. However, these approaches suffer from both the limitations of the built-in UML extension mechanisms [GH02], and the inflexible hierarchical structure of the UML metamodel (more details on this can be found in section 3.2). Without appropriate support for modeling concerns in UML, we will not be able to understand, model and reason about crosscutting concerns at any other level of abstraction that is higher than the program code. Moreover, lacking UML support for concern modeling, model-driven software developers will not be able to benefit from the advantages of the aspect-oriented technology.

3.4.1 Aspect-Oriented Concepts

In a similar way as object-orientation allows us to encapsulate “commonly localizable” concerns (data store and computation) into objects, aspect-orientation provides a mechanism, called *aspect*, for modularizing crosscutting concerns in software.

Major aspect-oriented concepts include a join point model, support for aspectual behavior, and support for structural amendment—or introductions:

- A *join point model*: This model allows one to potentially select any execution point in a program, thus providing a flexible approach to defining the dynamic structure of crosscutting concerns of importance;
- *Support for aspectual behavior*: This mechanism provides a means for adding aspectual (crosscutting) behavior at various execution points captured by the related join point model;
- *Support for structural amendment*: This support provides a means for modifying the structure (hierarchy, feature declarations, etc.) of existing software artifacts.

While the focus of this thesis is rather on aspect-oriented modeling, our work is based on lessons learned from aspect-oriented programming, including HyperJ and AspectJ. We consider the AspectJ programming language for two reasons: 1) it is the most popular programming language that supports aspect-oriented concepts explicitly and we believe that raising the concepts of AOP to the level of aspect-oriented modeling is beneficial; 2) supporting multidimensional separation of concerns at the modeling level provides flexible mechanisms for composing and decomposing elements of UML models that can be refined to support both AspectJ and HyperJ programs.

AspectJ realizes explicitly the aspect-oriented concepts mentioned above by providing different language constructs, such as, *pointcut*, *advice*, and *introduction*. All these constructs can be declared within another construct that is called *aspect*, and materializes the conceptual aspect mechanism for encapsulating crosscutting behavior.

3.4.2 Issues in Aspect-Oriented Modeling

Although the number of researchers and practitioners working on various issues of aspect-oriented modeling [AOM03] is increasing, many problems related to the design-level description still remain to be solved.

An essential issue in aspect-oriented modeling consists of separately expressing *base* elements and *aspectual* elements. We consider base elements as the fundamental elements of standard UML that are typically defined in the foundation package of the metamodel, such as ModelElement, Classifier, Namespace, etc. Aspectual elements are rather treated in UML as second class citizens, which are usually named, but not specified explicitly. This work uses the term perspectival element rather than aspectual element. Examples include roles (e.g., Association Roles, Classifier Roles, Association End Roles) in addition to other kinds of features that can crosscut the boundaries of a group of classes and objects of different types.

UML was not originally designed with modeling support for crosscutting concerns in mind, but research has demonstrated that the built-in extension mechanisms of UML can be used to address issues of aspect-oriented modeling [AEB03][CW00][SHU02]. However,

experience has shown that simply using built-in UML extension mechanisms for attaching a set of constraints, tagged values, or a stereotype to a model element does not make it support modeling of some concerns of importance to developers [AKH02][BGJ99][GH02]. In addition, extending UML to support aspectual elements other than the predefined ones—i.e., those that are already part of the standard UML—is problematic: built-in extension mechanisms should be carefully declared, and used only if the concerns they encapsulate cannot be adequately expressed using any other elements of UML [GH02].

3.5 Final Remarks

This chapter introduced two conceptual frameworks, IEEE-Std-1471 and concern spaces, and the Unified Modeling Languages to serve as a basis for the concern-oriented approach to software architecture.

IEEE-Std-1471 provides a mechanism for separating the set of architecturally significant concerns involved in a software system along multiple viewpoints; however, it does not support separation of concerns along other dimensions (than just viewpoints). Moreover, it does not specify how an architect should identify and categorize the concerns that pertain to individual viewpoints. To serve as a recommended practice of general interest, IEEE-Std-1471 remains intentionally silent on a number of issues, including the following:

- the way an architecture relates to a viewpoint (note that an architecture description consists of a composition of many architectural views)
- the way of representing concerns in architectural views
- how to deal with concerns that crosscut viewpoints
- how to verify that a view conforms to its viewpoint

Clearly, the notion of concern is central to achieving these goals. However, for the sake of generality, MDSOC does not provide a standard definition for the concept of concern. This is beneficial as it allows one to consider MDSOC as a domain-independent conceptual framework that can be implemented in different ways and for various purposes.

Existing realizations of MDSOC focus on aspect-oriented programming; they do not have explicit support for modeling concern-oriented software architectures. Without support for concern-oriented software architecture modeling, it becomes very difficult to apply aspect-oriented software development in the context of large-scale software systems—developers need software architecture documentation to understand, elaborate, implement and reason about the key properties of any large-scale system. Moreover, lacking support for concern-oriented software architecture modeling, tool vendors will not be able to implement MDSOC mechanisms in their products and users will be unable to fully apply the AOSD technology when building large-scale software systems.

Research on modeling crosscutting concerns is ongoing; however, there is no consensus on how to express an aspect using existing modeling languages, including UML. While

some researchers argue that UML is well suited for modeling aspects, many others believe that UML lacks appropriate mechanisms for addressing crosscutting concerns [AOM03]. We believe that much research is still required to find out what is the most appropriate way to model crosscutting concerns in UML. Aspect-oriented modeling as addressed in this thesis is an integral part of the concern-oriented approach to software architecture.

Realizing the conceptual frameworks, IEEE-Std-1471 and concern spaces, using UML allows one to benefit from the advantages of both architecture description and multi-dimensional separation of concerns when developing complex software. Throughout this dissertation, we illustrate how UML can be used for implementing IEEE-Std-1471 and multidimensional separation of concerns. To do so, we provide a new mechanism called UML Space, which realizes the notion of concern space, while allowing one to flexibly and efficiently support advanced separation of concerns in UML.

Part II

Building & Describing Concern-Oriented Software Architectures

Chapter 4:
Concerns and Software Architecture

This chapter introduces a new perception of concerns in relationship to software architecture. The first part of the chapter considers and discusses several definitions of the notion of concern and presents a definition used by this dissertation. It introduces concern categories and gives several examples of concerns from both a given requirements definition and a software development problem. The second part of the chapter introduces the concern-oriented approach to software architecture as a general methodology that can be realized in different ways to achieve architectural design by concerns.

4.1 Concerns

Every software engineer is familiar with some mechanisms for separating concerns in analyzing existing software systems or architecting, designing or programming new systems. However, few software engineers are *fully* aware of the nature of the concerns they deal with in their everyday jobs. Most of them concentrate on concrete artifacts which they can directly manipulate in software. Existing approaches to software architecture typically do not consider concerns as abstract "things" existing outside the computer, but rather as concepts which software engineers need to reify into elements of software systems. To explain what we mean by that, let us first take a look at different definitions given to the notion of concern; we proceed from a general concern definition to that introduced purposely by this work.

1. A concern is a *"... matter that engages a person's attention, interest or care"* [Web97]. This is a dictionary definition (from Random House Webster's Dictionary) which helps get some idea of what a concern is; but it is far too general for software development needs.
2. A concern is an *"... area of interest in a body of software (e.g., artifacts, aspects, etc.)"*[TO00]. This definition clearly focuses on software artifacts representing conceptual "things" in software rather than the conceptual "things" themselves.
3. A concern is *"... any matter of interest in a software system"*[SR02]. This definition is specific to the needs of software development; however, it concentrates on what exists in the computer (i.e., both software and hardware). Also, the definition does not consider a concern as a "matter of interest" that is outside of the software system.
4. Concerns are *"... those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more*

stakeholders. Concerns include system considerations such as performance, reliability, security, distribution, and evolvability" [IEEE00]. This definition is close to our needs, since it focuses on stakeholders' needs relative to the development and operation of the system. However, it lacks any mention of the problem to be addressed.

5. A concern is *"... an aspect of a problem demanding the developer's attention"* [Jack01]. This is the closest definition to what we are looking for; we adopt it as a basis for this work and enhance it to cover the needs not only of developers, but also of other software project participants.

4.1.1 Our Definition of a Concern

When we use the notion of concern in this document, we do not mean any concrete artifact that software engineers can directly manipulate, but the following:

A concern is an aspect of a problem that is critical or otherwise important to one or more stakeholders.

Though an "aspect of a problem" can be seen as a sub-problem (or a problem) itself, we prefer to use the term "aspect" to give emphasis to the projection of a problem rather than a partition of it. Moreover, our definition offers two essential components that need to be kept in mind when talking about concerns: aspect of a problem (problem) and interest to stakeholders (goal). Both components define what we mean by concern. By putting these two components together, it should be possible to formulate any concern in terms of a question that may be answered by an architectural design solution (or by a low-level design solution or an implementation) [Hil01].

To make this clear, let us take a look at what is not a concern. Consider the following:

1. A subsystem X in a given software system:

- Does the subsystem X represent *an aspect of a problem*? *No*, as the subsystem X is part of the software solution, it cannot be a problem.
- Is the subsystem X *of interest to some stakeholders* of the system? *Yes*, because any part of a system must be of interest to at least one stakeholder.
- As a result, the *subsystem X is not a concern*. This also confirms what we have mentioned before that: an artifact is not a concern.

2. Refactor the design of the software application Y, using the strategy design pattern.

- Does refactoring represent *an aspect of a problem*? *Yes*.
- Is it *of interest to some stakeholders*? *Yes*.
- *Refactoring is a concern*. However, the software application Y and the strategy design pattern are both artifacts, not concerns.

4.1.2 Discussion

Concerns are often confused with requirements [TD97], though they are different. Usually requirements reflect some aspects of a software development problem, but not all. Some aspects of the problem existed earlier or will exist later than requirements and others may occur throughout the software life cycle. For example, when developing object-oriented software, problems may arise at different stages: analysis, design, implementation, etc. The solution of an analysis problem can trigger a design problem, which in turn can cause an implementation problem. Aspects of such problems, particularly those of design and implementation problems, are certainly not reflected in the requirements, though they are concerns themselves.

Concerns have also been considered as non-functional or crosscutting considerations [RMA03]. Compared to our definition, this view of concerns is somewhat restrictive for the following reasons:

- Non-functional considerations are common parts of requirements, and as discussed above, not all concerns are requirements.
- Limiting concerns to non-functional considerations prevents functional considerations from being concerns themselves; however, in reality, functionality is the very kind of concern that software engineers understand the most.

With our concern definition, one cannot achieve much separation of concerns using existing formalisms, such as UML, because we regard a concern as an abstract concept which developers cannot directly deal with.

For example, UML does not provide support for adequately modeling software development problems, nor does it allow one to focus explicitly on various aspects of development problems. Researchers have argued that concern modeling should be achieved at a different level than what UML can currently offer [SR02][ST02]. However, it is a goal of this thesis to establish a bridge between UML and concern-oriented architecture documentation.

4.2 Concern Categories

A software project typically involves many concerns of different kinds, which for various reasons can be categorized in different ways. There is no consensus on how to categorize concerns in software development. This thesis distinguishes between four categories of concerns: architectural, non-architectural, reifiable and non-reifiable concerns.

Our decision for choosing these four categories was based on the fact that they are general enough to cover more detailed classifications of concerns. However, it is not the goal of this thesis to provide such a detailed classification of concerns. A good example of concern classification is given by Cosmos [SR02].

4.2.1 Architectural Concerns

An architectural concern is an aspect of a problem of interest to stakeholders that globally affects a software system. Architectural concerns usually include quality attributes and most kinds of crosscutting concerns.

The notion of crosscutting, as used in this context, is relative to the system's decomposition approach and to the level of abstraction considered. For instance, what is crosscutting in a procedural system's decomposition is not necessarily crosscutting in a modular decomposition (where a module is similar to the Modula-2 language concept), and vice versa. What you call crosscutting in a process-oriented decomposition may not be crosscutting in an object-oriented decomposition, and vice versa. Or what is crosscutting in a component-based decomposition may be well localized in an aspect-oriented decomposition. Further examples of architectural concerns include the forces of an architectural pattern that apply to the pattern as whole, while cutting across the boundaries of various pattern roles. Moreover, what is crosscutting for a group of objects and components might be local to a subsystem; but also, concerns that crosscut a group of subsystems might be local to a system. Examples of architectural concerns include security, performance, adaptation, etc.

4.2.2 Non-Architectural Concerns

A non-architectural concern is an aspect of a problem that can be reified into a localizable part of a software system. Non-architectural concerns include many kinds of non-crosscutting concerns that can be projected onto a single part of a system. Typical examples of non-architectural concerns include: computation, data store, responsibilities, etc.

4.2.3 Reifiable concerns

Tangible concerns are those concerns that can be reified into software through a non-empty set of model elements. These include architectural and non-architectural concerns that have explicit representations in software. For example, a computation concern can be addressed by means of its reification into operations (described in interfaces), methods (in classes) or functions (in modules); a persistence concern can be addressed by being reified into a crosscutting module.

4.2.4 Non-reifiable concerns

Non-reifiable concerns are those concerns that cannot be reified into software (at least not in the software under construction). The set of model elements that pertains to such concerns is empty. Examples include many aspects of software development, such as usability, time-to-market, costs reduction, etc.

4.3 Relationship between Concerns

Due to the multitude of the kinds of concerns encountered in software development, we do not have a predefined set of relationships among concerns. Relationships among non-reifiable concerns are non-reifiable themselves, whereas relationships among reifiable concerns are reifiable. We consider that because each reifiable concern can be reified into a collection of model elements, the relationships among such concerns can be reified into model elements as well.

4.4 Examples of Concerns

The following sections present two examples of concerns, from a given requirements definition for an architecture description language and from the video surveillance software development problem.

4.4.1 Concerns in the Requirements on ADLs

This section shows an example of concerns obtained from the requirements on architecture description languages. In [SG96], Shaw and Garlan have identified general concerns in an architecture description problem (they call them characteristics of an ideal ADL). These concerns are: composition, abstraction, reusability, configuration, heterogeneity, and analysis.

We consider the general architecture description problem as a set of different subproblems, which each in turn can have various aspects, as shown below:

Composition problem. *"It should be possible to describe a system as a composition of independent components and connections"* [SG96].
Aspects of the composition problem can be shown as follows:

- The ability to decompose complex systems hierarchically into smaller, convenient components and conversely, while allowing one to compose a system from its constituent elements;
- The ability to modularize each individual element;
- The ability to separate externally visible properties of individual elements from implementation details.

Abstraction problem. *"It should be possible to describe the components and their interactions within software architecture in a way that clearly and explicitly prescribes their abstract roles in a system"* [SG96].
Key aspects of the abstraction problem can be considered as:

- The ability to support abstract roles as first-class abstractions to represent multiple kinds of components, interactions among those components, and architectural patterns.
- The ability to characterize roles in terms of obligations, permissions and prohibitions.

Reusability problem. *"It should be possible to reuse components, connectors and architectural patterns in different architectural descriptions, even if they were developed outside the context of the architectural system"* [SG96].
Aspects of the reusability problem can be defined by the following concerns:

- The ability to reuse the abstract roles of individual components and connectors, not the components and connectors themselves;
- The capability to reuse collections of those roles of components and connectors that together characterize the architectural patterns that designers can reuse.

Configuration problem. *"Architectural descriptions should localize the description of system structure, independently of the elements being structured. They should also support dynamic configuration"* [SG96].
Aspects of the configuration problem can be defined by the following concerns:

- The ability to bind individual roles to individual components and connectors;
- The ability to bind a collection of roles to a collection of components and connectors.

Heterogeneity problem. *"It should be possible to combine multiple, heterogeneous architectural descriptions"* [SG96].
Aspects of the heterogeneity problem can be defined by the following concerns:

- The capability to combine various kinds of components and connectors into different architectural patterns within a particular system;
- The capability to integrate various kinds of components into a system;
- The ability to manage co-existence of multiple kinds of architectural concerns

Analyzability problem. *"It should be possible to perform rich and varied analyses of architectural descriptions"* [SG96].
Aspects of the architectural analysis problem can be defined by the following concern:

- The ability to support reasoning about architectural descriptions.

4.4.2 Concerns in the Video Surveillance Problem

This section shows another example of concerns that are obtained from the description of the video surveillance problem in section 2.1.1 on page 15.

To facilitate understanding of the concerns, let us recall the video surveillance service problem:

When the number of crimes increases in society and security becomes a concern, it is often necessary to make use of new technologies to control the situation. A video surveillance ser-

vice can be useful in such a situation. For this purpose, a collection of geographically distributed video cameras is to be controlled and monitored by security agents from a central video surveillance station. Each video camera captures images and produces a video stream that is transmitted to the central surveillance station. In case of an emergency, the security agents alert the police; for analysis purposes, security agents can command the surveillance system to store the sequence of images related to the urgent situation in a database of emergencies. Moreover, the police can ask for the video stream produced from a particular location and in a specific time period.

Figure 4.1 shows an example of key aspects of the video surveillance service problem. To identify the important aspects of the given problem description, we performed an experiment with different software engineers. The outcome of this experiment is a set of concerns listed in the following figure.

```
Number of crimes
Security
Geographic distribution
Centralized monitoring
Image capture
(Video) Streaming
Secure transmission
Police call
Image sequence storage
Specific image sequence retrieval
```

Figure 4.1: Concerns in the video surveillance service problem in Version 1

An alternative to the concern list shown above is presented in figure 4.2.

```
Reduce crime (main goal)
    Close watch of locations by security agents
      Centralization of security agents
      Distribution of video cameras
      Capturing images and video streams and passing them to the centralized
       station
      The surveillance (is achieved by)
        Observation of the video stream
        Reporting to police any problem
        The police acting upon the report
        TThe police reviewing the corresponding video sequence
```

Figure 4.2: Concerns in the video surveillance service problem in Version 2

4.5 Towards Concern-Oriented Software Architectures

The *Concern-oriented approach to software architecture* (COSA approach) aims at providing a solid foundation for concern-oriented software architectures. The COSA approach introduces a new methodology for building architectures, which is driven by a set of concerns of interest to stakeholders. A key idea in this methodology consists of relating individual *concerns* to the *model elements representing them* in software, just as, in a similar way, an *architecture* is related to *its description.*

Concerns are aspects of a problem that are critical or otherwise important to one or more stakeholders; the model elements are artifacts or concrete work products that capture the concerns in the body of software. Concerns are abstract things outside the computer, whereas model elements representing them are rather concrete work products that commonly exist inside the computer. Moreover, according to IEEE-Std-1471, an architecture is “the fundamental organization of a system *embodied* in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution”. An architecture description is “a collection of products to document an architecture”.

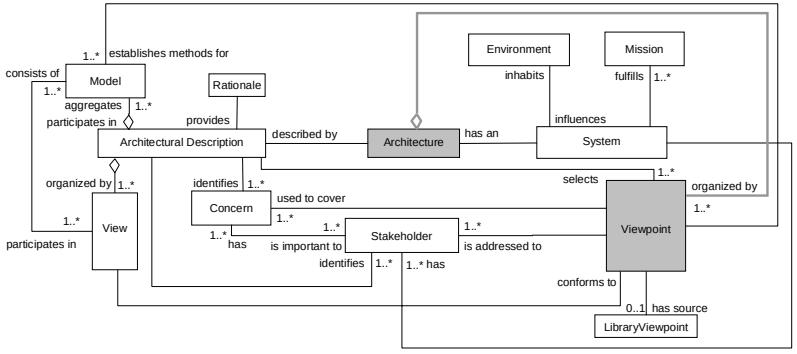


Figure 4.3: Conceptual Framework of IEEE-Std-1471 (modified)

By establishing an analogy between the mutual relationships of an architecture and its description on one hand, and of a concern and its representation on the other, we can make the following observation: every organization of a system embodied in components and their relationships to each other reflects a certain combination of different concerns that are relevant to some stakeholders. The concerns can be seen as projected on the elements of the system (i.e., its components and their relationships) in order to facilitate a description of the system structure and reasoning about its essential properties. Different perspectives of the system's organization might reflect different or overlapping combinations of concerns. Each combination of concerns results in a particular set of concerns that characterizes a specific viewpoint. A viewpoint is the specification of a view, that is, a view expresses an architecture of the system from the perspective of that viewpoint (i.e., a set of related concerns).

Figure 4.3 shows a modified version of the IEEE-Std-1471 conceptual framework. As we see in this conceptual framework, a view is a component of an architecture description. Such a relationship between a viewpoint and an architecture is lacking in the original IEEE-Std-1471 conceptual framework. However, based on the analogy between architecture vs. architecture description and concern vs. model elements, we believe that an architecture is a composition of viewpoints, in the same way as an architecture description is a composition of views. We illustrate this by adding a new aggregation relationship between Architecture and Viewpoint in figure 4.3. This addition implies another, similar relationship between Architecture and Concern. However, we have decided not to add such a second relationship for two reasons: 1) because its effect can be achieved indirectly via Viewpoint; and 2) because viewpoints allow one to separate the concerns involved in the architecture along multiple perspectives. Given that different concerns are relevant from different perspectives, this decision provides more flexibility to architects in organizing the stakeholders' concerns in a "software space" realizing the conceptual framework.

We refer to the resulting *composition of concerns* as *concern-oriented software architecture*. A concern-oriented software architecture is an architecture whose development is essentially driven by a set of concerns of interest to stakeholders. On the other hand, it is important to notice that the extension made to the conceptual framework is an enhancement; it in no way contradicts the standard.

4.5.1 Objectives of the COSA Approach

A major objective of the concern-oriented approach to software architecture is to provide a methodology for overcoming the limitations of existing architectural approaches as identified in section 1.2 on page 2 and section 1.3 on page 8. This methodology should be suitable: for developing and describing architectures of software-intensive systems; for improving separation of concerns in the design, construction and evolution of such systems; and for integrating architectural descriptions with modern software development artifacts. The ultimate goal of this new methodology is to provide support for achieving *design by concerns* all through the development and description of software architectures.

4.5.2 Characteristics and Requirements of the COSA Approach

To build a software architecture in a concern-oriented way, it is important to understand that a *software architecture is a multidimensional concern structure* that provides a conceptual but reusable solution to a software decomposition problem.

Naturally, concerns exist outside of the computer. Therefore, the multidimensional concern structure characterizing the architecture of a software system cannot subsist inside the software system itself. However, such a multidimensional concern structure can be described from multiple perspectives. Each perspectival description represents a particular view of the concern organization.

Different architects can produce different collections of views depending on the needs of stakeholders. Each collection of views can be implemented in different ways to build a different system. Thus, a system consists of a set of (physical and concrete) artifacts that *implement* an architecture description. An architecture description consists of a collection of work products, which together *reify* the (abstract and logical) concerns in the multidimensional structure.

Perceiving software architecture in a concern-oriented way, we believe that the *architecture is outside the system*, but *its implementation exists inside the system*. This perception has two advantages: 1) it provides a clear distinction between an architecture and an architecture description; and 2) it facilitates the evaluation of a software system against its architecture.

Distinguishing between architecture and architecture description, and facilitating software architecture evaluation are key characteristics of any concern-oriented software architecture. To help address these characteristics, we have identified a set of minimal

requirements that need to be fulfilled by every concern-oriented software architecture approach. A concern-oriented approach to software architecture must provide support for:

1. *Realizing the IEEE-Std-1471 Conceptual Framework*. It should be possible to produce architecture descriptions that conform to the IEEE and ANSI standard for architectural descriptions, ANSI/IEEE-Std-1471-2000.
2. *Reifying concerns*. It should be possible to address individual concerns at different levels of abstraction simultaneously, while distinguishing each concern from the software development artifacts reifying it.
3. *Realizing an architecture concern-space*. It should be possible to achieve multidimensional separation of concerns in software architecture through the realization of an architecture concern-space. This requires a mechanism for reifying a software architecture into an architecture description that can be implemented by different systems built from the same architecture concern-space.
4. *Achieving architectural design by concerns*. It should be possible to clearly separate the software architecture of a system from its description, while achieving architectural design by concerns from multiple perspectives, not just a single viewpoint.
5. *Integrating architecting with common software development*. The creation of software architectures should not be addressed in isolation—i.e., it is not sufficient to have "one specific" level or phase for software architecture development—instead, architecting can and should be performed at multiple stages within the global context of the software life cycle.
6. *Using UML*. It should be possible to integrate the resulting concern-oriented architecture descriptions with other architectural descriptions written in standard modeling languages, such as UML; this includes, for example, UML support for key ADL constructs: components, connectors, styles, systems and properties.

4.6 Final Remarks

It is essential to comprehend that concerns are aspects of the problems we, the humans, consider when we start building software. These problems are outside of the computer (both hardware and software) [Jack01]. Therefore, the computer cannot distinguish one concern from another, and it cannot identify the relationships between different concerns. It is the job of software engineers to identify the concerns and their relationships, to reify them into both model elements and code, and to manipulate concerns via their representations.

The relationship between concerns and model elements is similar to the relationship between an architecture and an architecture description.

We believe that what makes a concern is: 1) its significance to stakeholders and 2) the level of abstraction at which it can be reified. For example, a given problem may have many

aspects, but only the most significant ones are considered to be concerns of interest. Furthermore, the concerns in the video surveillance service problem exist at a completely different level of abstraction than those of the architecture description problem.

The concerns shown in figure 4.1 and figure 4.2 build together what we refer to as problem space. As discussed in chapter 2, such a problem space is required for building software architectures effectively. The thesis argues that building software architectures should be driven by the concerns of interest to the stakeholders, not by a specific artifact language or modeling technique. This is the very objective of the concern-oriented software architecture approach introduced as a general methodology for achieving architectural design by concerns. An example of realization for the COSA approach is the subject of the next chapter.

Chapter 5:
The PCS Framework

This chapter introduces the PCS Framework, a particular methodology implementing the concern-oriented approach to software architecture. The PCS Framework uses UML to combine the realizations of the conceptual frameworks of MDSOC and IEEE-Std-1471, and it presents a fulfillment of the general requirements on concern-oriented approaches to software architecture. The framework also introduces new mechanisms, such as projections and UML Space, to support integrating software development into the building of software architectures.

5.1 Introduction

In simple terms, a *Perspectival Concern-Space* (PCS) represents a technique of depicting concerns of multiple kinds (or dimensions) in an architectural view consisting of one or more models and diagrams. A perspective is a "way of looking" at a multidimensional space of software concerns from one specific viewpoint. Similar to model orientation [Ross78], every perspective has an orientation. The *orientation of a perspective* is determined by a set of related concerns, and by a purpose, a context and a viewpoint.

More specifically, a PCS represents the perspective of a particular viewpoint together with a mechanism needed for reifying a set of related concerns (relevant to the given viewpoint) into the body of software. The *PCS Framework* is one implementation of the concern-oriented approach to software architecture that provides a means for composing and decomposing different PCSs.

5.1.1 Goals, Principles and Key Concepts

A major goal of the PCS Framework is to provide mechanisms for building, describing and implementing concern-oriented software architectures in a flexible and incremental way. It allows one to identify, separate, modularize and integrate various software artifacts that pertain to different kinds of concerns.

An important principle for the PCS Framework is the *recursive separation of concerns* along multiple dimensions—that is, the capability to separate concerns along multiple dimensions called viewpoints, so that the set of related concerns viewed from the perspective of a viewpoint can be separated recursively into further (sub)viewpoints. This is the result of our interpretation of the IEEE-Std-1471. With this interpretation, we consider viewpoints to be an architectural mechanism for separating stakeholders' concerns into dif-

ferent sets of related concerns; each view expresses only those aspects of the system that can be "seen" from the perspective of a given viewpoint: when new problems arise, aspects thereof may be grouped to define a new perspective of the system at hand.

Figure 5.1 gives a general idea of the key concepts used in the PCS Framework and summarizes the combination of the realizations of the conceptual frameworks for MDSOC and IEEE-Std-1471, and UML.

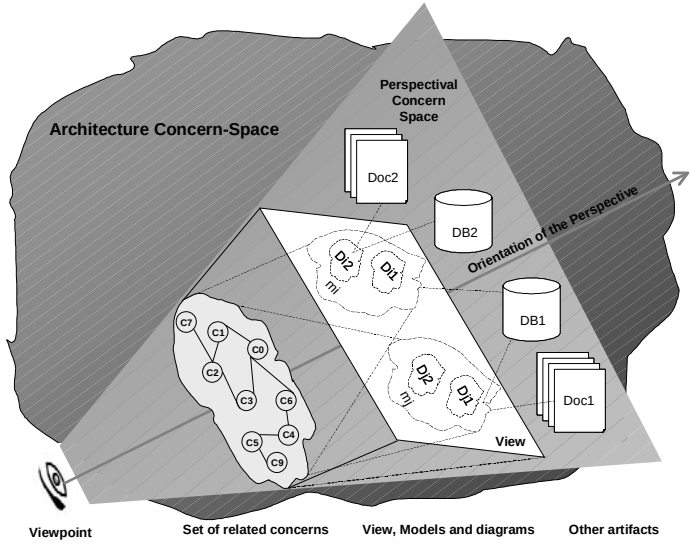


Figure 5.1: A Perspectival Concern-Space in Overview

Figure 5.1 illustrates the notion of perspectival concern-space as a projection of a concern-space that involves a set of related concerns, their reifications into models, and the realizations of these models. Essentially, the PCS Framework represents a composition of multiple Perspectival Concern-Spaces that work together to support integration of software development with building software architectures.

Moreover, the PCS Framework provides a means to overcome many of the problems identified in section 1.2 and section 1.3.

Finally, the realization of the IEEE-Std-1471 Conceptual Framework for the PCS Framework is shown as an *Architecture Concern-Space*. Further details on the concern space are given in section 5.4.

5.1.2 Fulfilling the Requirements of COSA

Essentially, the PCS Framework fulfills the general requirements on concern-oriented approaches to software architecture by providing a UML-based linguistic toolkit, called *UML Space*, which combines the realizations of the conceptual frameworks of IEEE-Std-1471 and MDSOC. The PCS Framework achieves the requirements for fulfilling the COSA approach in the following steps:

1. The PCS Framework realizes the IEEE-Std-1471 Conceptual Framework through its concept of *viewpoint schema*.
2. The reification of concerns is achieved by means of the *projection* mechanism.
3. The realization of an architecture concern-space is achieved through the notion of *UML Space*—the PCS Framework uses standard UML to create a UML Space which architects need to use *to develop and apply various viewpoint languages* at will.
4. The PCS Framework achieves "architectural design by concerns" through *concern-oriented modeling* using one or more viewpoint languages.
5. The integration of the building of software architectures with common software development is achieved through a *combination of MDSOC with IEEE-Std-1471 and UML*.

The following sections describe in more detail how the PCS Framework fulfills of the requirements of COSA.

5.2 Realizing the IEEE-Std-1471 Conceptual Framework

To realize the IEEE-Std-1471 Conceptual Framework, we premise that a software architecture is multidimensional in nature. That is, when constructing complex software, an architect should represent the system at hand from multiple perspectives in order to be able to understand, communicate and reason about its important properties. Each representation of the system is considered as a different view of the system's software architecture, and each view consists of one or more architectural models. Each architectural model reflects some aspects of the system relevant to the view at hand, while hiding other aspects that pertain to

other views. An overview of our realization of the IEEE-Std-1471 Conceptual Framework for the PCS Framework is illustrated in figure 5.2.

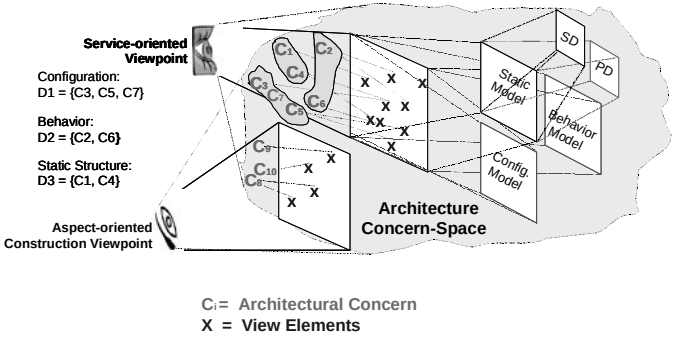


Figure 5.2: Overview of the IEEE-1471 Realization for the PCS Framework

Figure 5.2 illustrates some key concepts of IEEE-Std-1471, including architectural viewpoints, views, models and concerns, and shows how these concepts relate to each other. The relationships among the concepts conform to the description given in section 3.1.1, together with the extensions discussed in section 4.5 (see also figure 4.3).

Figure 5.2 shows two examples of viewpoints, a Service-Oriented Viewpoint and an Aspect-Oriented Construction Viewpoint (see "Service-Oriented PCS" on page 109 and "Aspect-Oriented Construction PCS" on page 83). For example, stakeholders who are interested in the system from the perspective of the service-oriented viewpoint will need to focus on different aspects of the software development problem at hand than those interested in the system from the perspective of another viewpoint. Considering the service-oriented viewpoint, each aspect of the problem is documented as a separate concern, called C_1, C_2, C_3, C_4, C_5, C_6 or C_7.

Concerns can be grouped into separate dimensions to define new viewpoints, recursively. For instance, C_3, C_5 and C_7 are shown as three different Configuration concerns that are grouped into one kind of concern, called the dimension D1. Similarly, the Behavior concerns (C_2 and C_6), and the Static Structure concerns (C_1 and C_4) are separated into a behavioral dimension (D2) and into a structural dimension (D3), respectively.

The Service-Oriented View of the software architecture of a system (unnamed in the figure) represents a partial architecture description of the system. Such a partial description can be further transformed and represented in different ways, for different reasons. As an example, in order to focus on the architectural dimensions introduced above, we decided to transform (or refine) the Service-Oriented View into three different architectural models shown as Static Model, Behavior Model and Configuration Model. Each of these architectural models per-

tains to the concerns in a corresponding dimension. Such models need to express only the concerns at hand and nothing more. We refer to them as *concern-oriented models*.

As a result, architectural models can be transformed into other models (or diagrams) in a "concern-oriented way". We call this process *concern-oriented model transformation*. For example, the Static Model is transformed into further models shown in two diagrams, the service description diagram (SD) and perspectival association diagram (PD). Each descriptive *unit* in a view (i.e., a view element depicted as "x") consists of a collection of one or more model elements.

5.2.1 Viewpoint Schema

A summarized description of the template for a viewpoint schema is shown in figure 5.3.

Viewpoint name		An expressive identifier that reflects the architectural perspective documented in an associated architectural view
Sources		Sources for documents providing additional information related to the viewpoint, including overview information, summary, context, references, history of change, and glossary.
Concerns		A list of critical aspects of the architectural problem(s) which: 1) need to be addressed by the present viewpoint, and 2) affect the context of the system, its development, operation, maintenance or its stakeholders
Stakeholders		An explicit record of who is interested in or needs what concern
Rationale		A description that explains the *motivation* for making decisions, the *decisions* themselves, the *impact* of the decisions, and the degree of *satisfaction* achieved by a solution.
Architectural Problem(s)	**Incentive aspects**	A set of concerns that *motivate* the need for making new decisions
	Decisional aspects	A set of (helpers) concerns allowing one to specify what *decisions* to take
	Resultant aspects	A set of concerns characterizing the degree of *satisfaction* with the decisions taken, their *impacts* on the architecture. Options: *Noteworthy* – all observable/measurable results that characterize the degree of satisfaction with the decisions, and the impacts of the decisions *Unsatisfactory* - dissatisfied with or indifferent to the decisions
Relationships among Concerns		Describes how different concerns present in the viewpoint relate to each other
Resultant View		A partial description of the software architecture of the system from the perspective reflected in the viewpoint

Figure 5.3: A Viewpoint Schema in Overview

To allow architects to produce architecture descriptions that conform to the IEEE-Std-1471, the PCS Framework has introduced the notion of *viewpoint schema*, which:

- defines a unique name for the viewpoint at hand;
- identifies a set of stakeholders along with a set of various kinds of concerns that pertain to those stakeholders;
- provides an approach to facilitate the definition of a viewpoint language;

- identifies the associated architectural view that represents the stakeholders' concerns in one or more architectural models;
- provides the sources for key information used in or related to the viewpoint definition.

The PCS Framework provides a template for defining viewpoint schemas. In addition, it allows one to state the rationale for a viewpoint and to provide some relationships between the different kinds of concerns to be addressed, using a viewpoint schema.

5.3 Reifying Concerns

Besides the key elements of the IEEE-Std-1471 Conceptual Framework, figure 5.2 illustrates the notion of reification of concerns. The lines between the concerns ($\{C_i\}_{i=1..10}$), and the view elements (the x's) illustrate the reification of the stakeholders' concerns into viewpoint language elements. The mechanism supported by the PCS Framework to address concern reification is the *projection*.

A concern that is relevant to a viewpoint is reified into an architecture description by means of its projection onto one or more view elements. The PCS Framework supports two kinds of projections: projections between descriptive units and projections between concerns and descriptive units.

1. *Projections between descriptive units* are concern-oriented model transformations that transform a set of descriptive units into another. Examples of such concern-oriented model transformations are given in the previous section, showing transformations of an architectural view into concern-oriented models, and of a model into diagrams.
2. *Projections between concerns and descriptive units* are different as they address the reification of a given concern (or set of concerns) into a set of descriptive units. For example, when designing a system using UML, you might need to reify a given responsibility (concern) into one or more methods (descriptive units) encapsulated in one or more classes (composite descriptive units).

Essentially, a *projection* is an architectural abstraction that defines the relationship between a viewpoint and a view—or between a view and a set of models. It consists of a set of rules that specifies how to reify (one or more) concerns into (zero or more) descriptive units, which are typically model elements. Projections must take into account that some concerns might not have adequate representations in the description language at hand.

A descriptive unit can be simple or composite. A simple descriptive unit can be, for example, any basic UML element, such as a link, attribute, parameter, etc. Examples of composite descriptive units include classes, subsystems, packages, and any type of UML diagrams. Basically, a projection can be any set of rules that specifies how to decompose, organize, and structure software according to a specific dimension. Projections can be used, simultaneously, at different levels of abstraction. Different projections along different

dimensions result in different models, but also different projections along the same dimension may result in the same or different models. All sets of rules defining projections are defined and maintained as parts of the concern space.

Supporting the projection in the PCS Framework makes it possible to reify, and concurrently separate concerns into different artifacts, at various levels of abstraction, while distinguishing each concern from the software artifacts reifying it.

5.4 Realizing an Architecture Concern-Space for UML

The requirement of concern-oriented models to express the concerns at hand, and nothing else, increases the need to have flexible viewpoint languages. In practice, UML seems to be a good candidate for documenting software architectures from multiple perspectives [HNS99][CBB+02]. On the contrary, using "members" of the UML family of languages (i.e., languages for expressing the different diagram types) as viewpoint languages can be very problematic. For example, the notion of software connector as found in ADLs does not exist in standard UML [KS00a], and introducing it into UML is complicated, as 1) UML lacks adequate support for modeling roles as first-class citizens and expressing roles is critical to connector modeling, and 2) extending UML suffers from the *X-Syndrome* (section 5.4.2).

5.4.1 UML Lacks Adequate Support for Modeling Roles

The UML notation used to model roles is part of the UML language family called Collaboration. The metamodel describing the meaning of this language is shown in figure 5.4.

This metamodel is written itself in UML, using the notation for class diagrams. Each class in this diagram is a meta-class that represents a concept defined in the Collaboration language. An association between two meta-classes is a meta-association. The meta-classes shown in gray are linguistic concepts borrowed from other members of the UML language family. For instance, Action is borrowed from the BehavioralElements::Actions language and all the other concepts in gray are from the Core language.

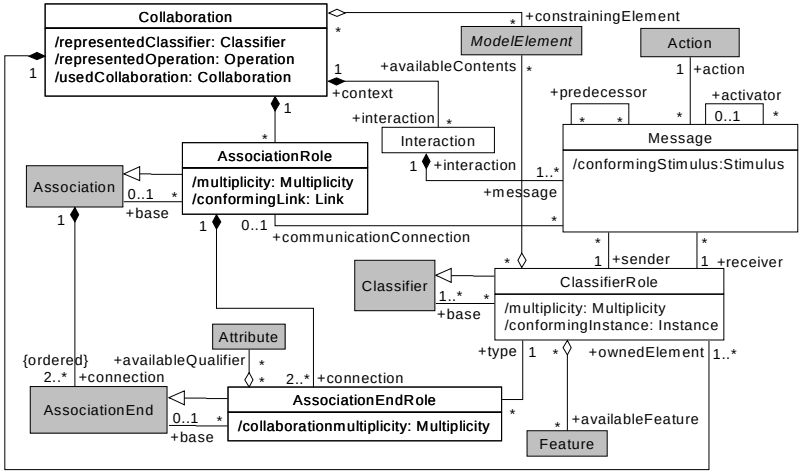


Figure 5.4: UML Metamodel for Collaboration

The Collaboration language, as it stands, is very useful for modeling interactions among objects and components, as long as there is no need 1) to have a specific locus of definition for the interaction (i.e., the interaction is scattered among the participants), and 2) to separate collaboration roles from the entities playing such roles. Otherwise, using the UML Collaboration is not appropriate.

As shown in the metamodel, roles (i.e., AssociationRole, AssociationEndRole and ClassifierRole) inherit from their base meta-classes (i.e., Association, AssociationEnd and Classifier), respectively: that is, roles cannot be separated from their base entities. Thus, every model element that is not a Classifier, an Association or an AssociationEnd cannot play a role in UML. This makes it impossible to use UML for composing interactions, associations, collaborations, features, etc.

However, it should be possible to define a collaboration role by what base elements fulfilling that role *must*, *must not* or *may* accomplish (i.e., obligations, prohibitions and permissions).

5.4.2 The X-Syndrome

As mentioned earlier, UML provides built-in extension mechanisms which assist methodologists in defining new model elements that are not found in the standard specification of the language and which are required for some particular purpose. However, using built-in extension mechanisms of UML can be problematic to such a point that it may turn the attention of methodologists and software architects from their goals; thus, methodologists and

architects are often forced to deal with linguistic issues that have little or nothing to do with their concerns: if we do not provide a solution to this problem, model-driven software engineering cannot be successful and using UML (both as it is and in its upcoming version UML 2.0 [Omg03]) will rather lead to the distraction and frustration of modelers.

Through research and practical work we have done over the last five years, we observed a syndrome characterizing the distraction and frustration of both architects and methodologists using UML. Our experience was gained from work in two different areas: 1) in using, teaching and enhancing UML-based methodologies, and 2) in developing new UML extensions and profiles for structural architecture descriptions [KMP+98][KCS+02] as well as for aspect-oriented modeling [KKS02].

We have identified a pattern of activity in modeling that tends to occur repeatedly whenever modelers try to extend UML. This pattern of activity is not a useful software pattern; it is an anti-pattern that we describe to help architects and methodologists avoid frustration and distraction when using standard UML to create architectural models or models resulting from a software development process. We call this anti-pattern the *X-Syndrome* or “Extension Hell Syndrome” of UML.

The X-Syndrome anti-pattern represents a group of symptoms that together characterize a large number of problems related to using current UML's extensibility mechanisms.

The X-Syndrome can be best observed whenever you try to model a new kind of concern in UML that is not already supported. A typical process for extending UML looks like the following:

1. You find a model element that is most similar to the kind of construct you would like to support in UML
2. You extend this model element (e.g., by defining a new stereotype)
3. You neutralize or “deactivate” all features you do not need (e.g., by defining OCL[1] constraints—using OCL you can only restrict; however, does this cover all your neutralizations?)
4. You add all the additional features you wish to have (e.g., by defining new tagged values)
5. Eventually, you get a new model element that resembles what you are looking for, *but* you might not succeed in getting exactly what you want!

Summing up, UML currently forces modelers to inherit from elements that contain features they don’t need.

We believe that the X-syndrome is a direct consequence of the rigid hierarchy of the UML metamodel. In UML, primitive model elements and groups thereof suffer from the predominance of Classifiers and predefined diagram types: elements of the UML metamodel can be composed and decomposed along a few dominant dimensions only; compositions and decompositions along other dimensions are not supported.

1.The Object Constraint Language, described in the UML specification [Omg01].

5.4.3 UML Space — Overcoming the X-Syndrome

In addition to the problem described above, it is also difficult to use and understand the standard UML specification [Omg01]. Even experienced users of UML cannot easily find out the exact meaning of certain model elements from the standard specification, because the information pertaining to a specific element is scattered throughout the whole document (which is about 700 pages). Typically, information on a meta-class is found in different diagrams, associations, and parent meta-classes, etc.

Figure 5.5 shows an example of the kind of information an experienced user of UML would need to decide whether a specific model element is appropriate for his/her modeling needs or not.

This example describes the Classifier meta-class, including its attributes, parents, scope of definition, children, meta-diagrams by which it used, as well as the roles it plays.

```
Classifier
     has attribute
          - association
          - classifierInState
          - classifierRole
          - collaboration
          - feature
          - instance
          - objectFlowState
          - powertypeRange
          - specifiedEnd
          - typedFeature
          - typedParameter
     is child of
          - GeneralizableElement
          - Namespace
     is owned by
          - Core Package
     is parent of
          - Actor
          - Artifact
          - Class
          - ClassifierInState
          - ClassifierRole
          - Component
          - DataType
          - Interface
          - Node
          - Signal
          - Subsystem
          - UseCase
     is used by
```

```
    - Activity Graphs Package
    - Collaborations Package - Interactions
    - Collaborations Package - Roles
    - Common Behavior Package - Instances
    - Common Behavior Package - Signals
    - Core Package - Backbone
    - Core Package - Classifiers
    - Core Package - Relationships
    - Model Management Package
    - Use Cases Package

acts as (roles)
    - base
    - classifier
    - owner
    - participant
    - powertype
    - representedClassifier
    - specification
    - type
```

Figure 5.5: The Description of the UML Metaclass Classifier

Normally, this information about the Classifier meta-class is scattered across many pages of the standard specification document, and there is no way to gather it quickly and efficiently. Techniques and tools are needed to facilitate the exploration and navigation within the space of UML concepts.

UML Space is the PCS Framework mechanism for realizing an architecture concern space. In general terms, a concern space represents a conceptual repository that contains all relevant information related to a set of different viewpoints. A concern space takes a set of viewpoint schemata as an input and refines the information contained in each individual viewpoint schema to help define its viewpoint language.

Using an architecture concern space, the PCS Framework allows one to organize, recursively and incrementally along different dimensions, the set of all concerns from the perspectives of multiple viewpoints; it also allows one to specify the relationships between the dimensions and maintain changes in the concern structure. Therefore, we consider a concern space as a tool for reifying a software architecture into a “multi-dimensional model of systems”—that is, the set of all systems that can be built from the same software architecture.

5.4.3.1 How to Create a UML Space

Creating a UML Space allows one to overcome many problems related to development of extensions to UML [BGJ99][GH02], including the X-Syndrome. UML Spaces provide an

effective and flexible approach to supporting domain-specific modeling in UML, and developing different viewpoint languages and profiles of UML. We found useful to go along the following steps when developing a UML Space:

1. *Create a conceptual repository* to which you can add the UML metamodel
2. *Provide support for exploring and navigating* across the UML Space
3. *Fragment the UML metamodel* to get independent units of manipulation; this allows you to achieve multidimensional separation of concerns, by acquiring the necessary flexibility to combine units of the metamodel along multiple dimensions of concern
4. *Compose different units* contained in the UML Space to build the viewpoint language which you wish to have
5. *Defragment the resulting metamodel* (i.e., recompose the units) to build the model elements or meta-classes of your new language
6. *Inherit from the smallest element of UML* to make your viewpoint language become a variant or profile of UML
7. *Store the new metamodel* as an extension to UML or a new UML profile

To validate this approach, we have developed a UML Space for Aspect-Oriented Modeling that is used to model component interactions. See “Aspect-Oriented Construction PCS” on page 83. Furthermore, to provide support for exploring and navigating across the UML Space, we have developed a Topic Map for UML 1.5 that will be available very soon to the UML community at the Internet address given in [US03]. As an example, the Classifier description shown in figure 5.3 was generated from this Topic Map for UML.

Because of their extensive use of units in the UML metamodel, we believe that UML Spaces provide a good linguistic toolkit to integrate concern-oriented architecture descriptions with standard UML models, including UML extensions to support ADL constructs.

5.5 Achieving Architectural Design by Concerns

The suitability of any formalism to support concern-oriented architecture description depends on its capabilities to facilitate separation, representation, and reasoning about multiple kinds of concerns involved in software development.

To address the achievement of architectural design by concerns, this section concentrates on two points: mechanisms for separating architectural concerns and linguistic support for expressing those concerns.

5.5.1 Mechanisms for Separating Architectural Concerns

Up to now, we have introduced viewpoints as a mechanism for separating architectural concerns. We have also demonstrated the use of viewpoints in a recursive and incremental way. This approach provides a great flexibility to produce architectural models from a given

viewpoint and to transform these models into further models, incrementally and in a concern-oriented way. However, it does not address adequately the representation of those concerns that crosscut multiple viewpoints, causing consistency problems among views (recall that in IEEE-Std-1471, each view conforms only to one viewpoint).

To solve this problem, we need more flexible mechanisms for separating concerns in software architecture. This leads us to examine the role of multidimensional separation of concerns in software architecture.

An important goal of multidimensional separation of concerns is to allow support for "ilities" throughout the software lifecycle. Thus, it represents a compelling means for achieving architectural qualities (as introduced by [BCK98]). Examples of "ilities" supported by MDSOC, and which are referred to as architectural qualities, are the following:

Understandability. Lack of understanding what concerns are relevant to a software architecture is the cause of the failure in most software projects. Good understanding of architectural design involves comprehension of various concerns of importance to an architect. Using MDSOC, software architects can focus on one concern at a time and document its representation in an architecture description. This facilitates achievement of architectural design by concerns.

Reusability. Achieving architectural design by concerns allows one to produce concern-oriented models of the software architecture and to reuse them at will, when constructing large-scale systems.

Traceability. Traceability between concerns and the elements (or units) in the body of software that represent those concerns [CHO+99]; using MDSOC, developers can find, at any time, all units that are relevant for a given concern.

Modifiability. Modifiability is essentially a function of the locality of any change [BCK98]. It defines the circumstances under which an architect has to limit the impact of change. MDSOC helps provide means for restricting the impact of change in software.

Evolvability. Limitation of change allows architects to address new concerns in their software architectures or change the design of existing concerns by modifying the corresponding concern-oriented models.

Integrability. Integrability is an important driving force in any concern-oriented software architecture, because it stands for the ability to make separately developed concern-oriented models work together. MDSOC refers to this as a mechanism for composing different modules.

While these advantages are critical to building software architectures, realizations of separation of concerns found in software architecture practice are very limited. There is a complete lack of mechanisms for advanced separation of concerns in the software architec-

ture community. Software architects should learn from work done in the aspect-oriented software development community.

The path we follow in this dissertation is aligned with an important observation by Clements and Northrop: "*aspect-oriented programming is an architectural approach because it provides a means for separating concerns that would otherwise affect a multitude of components that were constructed to separate a different, orthogonal set of concerns* ([CN02], pp.68)."

Furthermore, the implementation of MDSOC for software architecture, as proposed by the PCS Framework, goes beyond aspect-oriented programming. We adopt aspect-oriented modeling (AOM) and extend it to cover the notion of reification of concerns into software. Thus, our work does not just address AOM to complement object-oriented modeling techniques with aspects. Instead, the PCS Framework focuses on the idea of design by concerns which leads to an extension of AOM to cover concern-oriented modeling.

5.5.2 Linguistic Support for Expressing Architectural Concerns

As mentioned in section 5.4, UML is quite inflexible. Achieving multidimensional separation of concerns for UML requires one to break the tyranny of dominant decompositions. This can be achieved by model fragmentation/de-fragmentation to create more flexible design languages (See "UML Space — Overcoming the X-Syndrome" on page 74.) What we need now is to focus on design by concerns.

5.5.2.1 Tyranny of Dominant Decompositions in UML

Current UML allows one to separate certain kinds of concerns along individual dimensions to achieve *unidimensional separation of concerns*. For example, using UML you can discriminate different responsibility concerns and reify them into different Classifiers (e.g., Classes, Interfaces and Use Cases).

Using Classifiers, a modeler is able to separate the specification of a given responsibility concern from its realization, by means of reification of the given concern into model elements—an interface and a class realizing that interface. Such model elements can be described at two different levels of abstraction, specification and realization (covering both design and implementation). If we consider the specification, design and implementation as three different levels of abstraction, then we can say that the given responsibility concern pertains to all three levels, but what does this mean? Does this responsibility concern crosscut the levels?

Indeed, responsibility is a dimension of concern; it should not be confused with the individual concerns that belong to it. Examples of specific responsibility concerns include: CUSTOMER MANAGEMENT, BOOK MANAGEMENT, and TRADING. These concerns can be reified into descriptive units.

Figure 5.6 shows an example of how the TRADING concern can be reified into the public operations, buy() and sell(). The reification of the CUSTOMER/PRODUCT MANAGEMENT concern is more complicated because the concern itself can be seen as a problem with two aspects: ASSORTMENT and PERSISTENCY. These aspects represent each different concern that can be addressed separately.

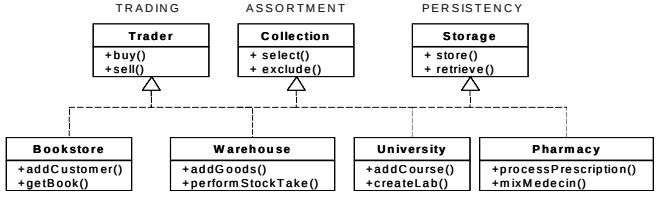


Figure 5.6: Unidimensional Separation of Concerns Across Different Levels

As exemplified in figure 5.6, the ASSORTMENT concern is reified into two operations: select(), exclude(); the PERSISTENCY concern is also reified into store() and retrieve().

To realize design by concern, the descriptive units reifying each of the concerns (TRADING, ASSORTMENT and PERSISTENCY) must be encapsulated separately. This is achieved by organizing the units into three different interfaces called Trader, Collection and Storage, respectively. This is concern-oriented modeling in the context of unidimensional separation of concerns.

While concern-oriented modeling can be achieved in this context, using standard UML, it does not work in many other situations where we have to deal with multiple dimensions. A well-known example of a development method based on unidimensional separation of concerns is the Responsibility-Driven Design method [WWW90].

One problem with concern-oriented modeling for unidimensional separation of concerns is scattering and tangling.

5.5.2.2 Scattering of Concerns

Consider the example of figure 5.6. Through the realization relationships between the interfaces (Trader, Collection and Storage) and the design classes (Bookstore, Warehouse, University, and Pharmacy), the realizations of the responsibilities separated previously are scattered across all classes. Consequently, if for some reason, an argument within a signature of an operation declared in one of the interfaces changes, the change may become invasive, leading to maintenance problems.

5.5.2.3 Tangling of Concerns

In the example of figure 5.6, scattered responsibility concerns overlap and become entangled with each other and with the features of the design classes.

5.5.2.4 Crosscutting Concerns

Figure 5.7 shows a different view of the scattering and tangling discussed above. It motivates the need for concern-oriented modeling, while demonstrating the inability of UML to support advanced separation of concerns. This expresses the crosscutting effect that results from the unidimensional separation of concerns achieved through the use of responsibility-driven design.

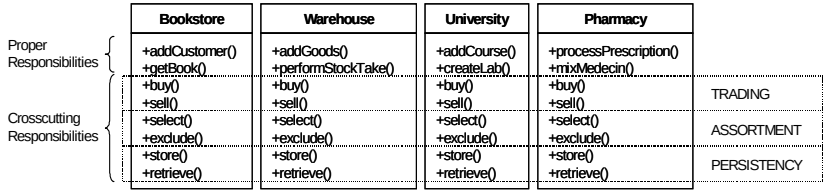


Figure 5.7: Structural Crosscutting

To realize multidimensional separation of concerns in UML, we need to make UML support the notion of perspectival elements. We refer to such *perspectival model elements* as model slices that can be created by using a UML Space. *Model slices* can be used for representing the reifications of individual (crosscutting and non-crosscutting) concerns in concern-oriented models. Many concrete examples of model slices are given in the next chapters.

5.6 Integrating Architecting and Software Development

The PCS Framework addresses integration between software architecture building and software development through the realization of a general-purpose software engineering approach (MDSOC), a standard recommended practice for architecture description (IEEE-Std-1471) and a standard modeling language (UML). By combining these elements to support separation of concerns at multiple levels, simultaneously, the PCS Framework shows that the creation of software architectures should not be addressed in isolation. Software architectures are created, described and elaborated from different perspectives at the same time. The combination of MDSOC, UML and IEEE-Std-1471 has been extensively discussed throughout this chapter with several examples. More practical examples are shown in the validating chapters.

The description of any particular PCS is based on the use of a viewpoint schema as shown in figure 5.3 (see for example, “Aspect-Oriented Construction PCS”).

5.7 Using UML

As you might have observed, we have been using UML notions throughout this chapter. In particular, supporting the UML Space within the PCS Framework allows one to effectively use UML to quickly create viewpoint languages and use them to provide concern-oriented descriptions of software architectures.

5.8 Final Remarks

This chapter introduced the PCS Framework. It provided an approach to implement two conceptual frameworks, the IEEE-Std-1471, and multi-dimensional separation of concerns (MDSOC). The PCS Framework provides means for integrating these two frameworks by using the Unified Modeling Language (UML). Furthermore, it introduces the notion of concern-oriented modeling to support the paradigm of architectural design by concerns. The concern space realization proposed by the PCS Framework is called UML Space.

Chapter 6:
Aspect-Oriented Construction PCS

This chapter presents the concern-oriented approach to software architecture from the perspective of aspect-oriented software development, using multi-dimensional separation of concerns (MDSOC). It describes a perspectival concern-space, called Aspect-Oriented Construction PCS. This specific PCS demonstrates how MDSOC helps deal with software complexity by supporting the composition of independent software components along different interaction concerns. The chapter introduces a UML Space for Aspect-Oriented Modeling.

6.1 Viewpoint Name

Aspect-Oriented Construction Viewpoint

6.2 Sources

Overview Information. The Aspect-Oriented Construction viewpoint provides the ability to compose software components that have been separately implemented using, possibly, different technologies.

Summary. This viewpoint presents an aspect-oriented technique to create software connectors based on modeling different kinds of interaction.

Context. This viewpoint is part of the Aspect-Oriented Construction PCS.

References. [GMW97] [MMP00]

History of change. None.

Glossary. None.

6.3 Concerns

The major concerns for the Aspect-Oriented Construction PCS are the following: Interaction, modifiability, reusability, understandability and decoupling.

6.4 Stakeholders

The stakeholders include: architects, maintainers, designers, developers and programmers.

6.5 Rationale

1. *Developing software systems by composing existing independent components*. Architects have to reuse existing components to build new systems.
2. *Enterprises need to adopt component-based software development to build new components faster.* The architecture must allow one to:
   - plug-in new components into existing environments
   - respect time-to-market constraints
3. *Managers said: We need to adopt the aspect-oriented software development along with software architecture.*

6.6 Architectural Problems

6.6.1 Incentive Aspects

To explain our approach to achieving multidimensional separation of concerns in UML, consider the notion of collaboration as defined in standard UML: *"a Collaboration contains a set of ClassifierRoles and AssociationRoles, which represent the Classifiers and Associations that take part in the realization of the associated Classifier or Operation. The Collaboration may also contain a set of Interactions that are used for describing the behavior performed by Instances conforming to the participating ClassifierRoles."*[Omg01]

The structure resulting from this definition can be illustrated as shown in figure 6.1. The figure contains two ClassifierRoles, each of which is connected to an AssociationRole through an AssociationEndRole. However, the UML metamodel is based on object-oriented modeling techniques. In object-orientation, things that happen within objects can be described and reasoned about. But what occurs between objects is not well understood. For instance, UML does not allow associations between classes to exist independently of the classes between which they establish a relationship. This enslavement of Associations in UML is reflected on AssociationRole because of the inheritance relationship between both (page 2-113 in [Omg01]). Furthermore, while AssociationEndRole and ClassifierRole represent two metaclasses, the relationship between them is treated as a second-class citizen. Therefore, the *attachment* between these roles cannot be expressed.

We believe that if we understand how to attach ClassifierRoles to AssociationEndRoles—which are part of an AssociationRole—we can express ClassifierRoles and AssociationRoles inde-

pendently from one another, and treat both as first-class citizens in Collaborations. Lacking support for expressing ClassifierRoles and AssociationRoles independently, we will not be able to understand the behavior that cuts across the boundaries of Classifiers and Associations playing these roles respectively.

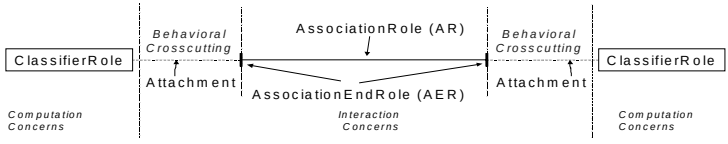


Figure 6.1: Structural Illustration of Key Elements within a UML Collaboration

Supporting explicit separation between ClassifierRoles and AssociationRoles would allow us to reify computation and data store concerns within ClassifierRoles, and reify interaction concerns by means of AssociationRoles. Unfortunately, AssociationRole is based on (i.e., inherits from) Association, and Association inherits from GeneralizableElement and Relationship, which in turn extend ModelElement. This leads to the X-Syndrome, as it forces an AssociationRole to take over all the features from all the super meta-classes including those that are not desired. To avoid this, another perspective on AssociationRoles is required.

6.6.2 Towards Perspectival Associations

We believe that taking another perspective on an AssociationRole will allow one to understand what happens between instances of different Classifiers, and thus to express various aspects of an interaction among them. Figure 6.2 addresses the structural crosscutting shown in figure 6.1 and figure 5.7.

Figure 6.2 describes a collaboration between two instance-level ClassifierRoles, Client and Storage. In particular, it focuses on what happens at the ends of and within the links that connect the ClassifierRoles. For example, the point at which a Client invokes the retrieve() method on an instance of a Classifier (bs, wh, and ph) playing the role Storage (1) joins the point at which the retrieve() method starts executing within one of these instances (2). The call on the retrieve() method made at (1) can be executed at several points (2). As a consequence, the concept of *perspectival associations* allows us to understand and express how individual calls can crosscut the boundaries of different links as well as instances of Classifiers attached to those links.

The point at which to get the value of a variable (i.e., an attribute) (3) joins the point at which that variable is de-referenced (4). Also, the point at which to set a value to a variable (5) joins the point at which the value is assigned to that variable (6).

Similarly, the point at which software causes, raises or throws an exception (7) joins the point at which that exception is handled (8). The points through which the control flow passes illustrate our join point model which is based on the *cause-effect* principle. For

example, a call (a cause) triggers an execution (an effect). In this model, we distinguish between implicit and explicit join points. An explicit join point is a point in the software at which a new branch of the control flow starts. An implicit join point is the point at which the newly created branch of the control flow ends.

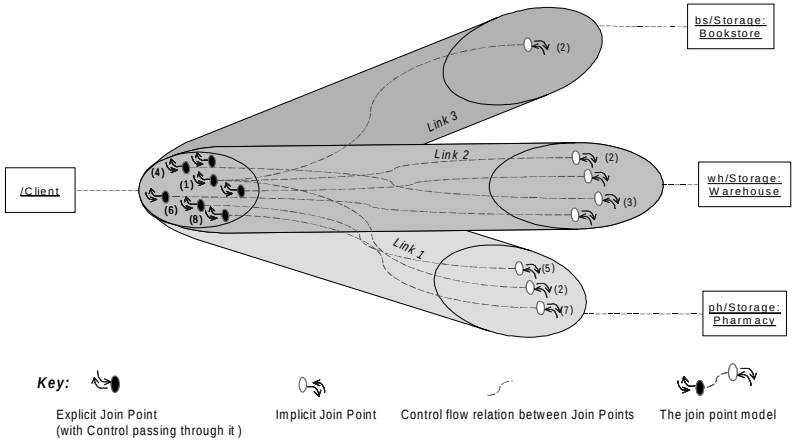


Figure 6.2: Behavioral Illustration of Key Elements of a UML Collaboration

6.6.3 Decisional Aspects

The decisional aspects for the Aspect-Oriented Construction PCS are:

- The need for a mechanism to support concern-oriented software composition
- The need for supporting aspect-oriented modeling
- The need for a mechanism to support reifying interaction concerns

6.6.4 Resultant Aspects

The resultant aspects for the Aspect-Oriented Construction PCS are:

- The ability to reify interaction concerns into software connectors is satisfactory
- The ability to compose interaction concerns without using the aspect mechanism is unsatisfactory
- The ability to support aspect-oriented modeling is satisfactory
- The ability to support model fragmentation and de-fragmentation is noteworthy. Tool support is very much required

6.6.5 UML Extensions for Aspect-Oriented Modeling

The UML extensions for aspect-oriented modeling presented in this section are based on the use of MDSOC in building the UML Space. As mentioned in section 5.4.3, we copy the UML metamodel into the UML Space and fragment it into primitive units (attributes, elements, features, etc), which we compose to build new model elements for aspect-oriented modeling.

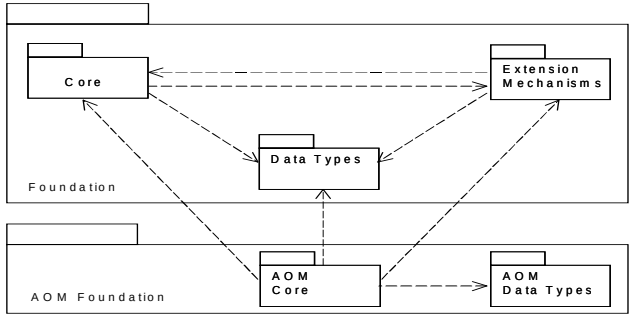


Figure 6.3: High-Level Package View of the UML Space for AOM

The new model elements are defined within the package called AOM Foundation (figure 6.3). This package is decomposed into two subpackages called AOM Core and AOM Data Types.

The figure further illustrates the relationships between the AOM Foundation and the UML Foundation packages. It also shows the internal structure of each of those packages. All relationships between the packages shown in this diagram are UML dependency relationships.

The AOM Core package specifies the basic AOM constructs necessary to model aspect-oriented software. The AOM Data Types package defines basic data types.

We will first give a broad overview of the extension packages, followed by a detailed description of the abstract syntax of the new model elements.

6.6.5.1 AOM Core: The Aspect-Oriented Construction Viewpoint Language

This package is a fundamental subpackage that composes the AOM Foundation package. It defines the basic metamodel constructs needed for the development of aspect-oriented models, and can be extended as the AOM community gains more experience.

The following section presents very briefly the abstract syntax. The well-formedness rules and detailed semantics of the AOM Core package have been left out for space reasons.

6.6.5.2 Abstract Syntax

The abstract syntax for the AOM Core package is expressed in graphical notation, and is decomposed into three different diagrams. For readability reasons, the model elements are described in alphabetical order just after the diagram that contains them. Figure 6.4 shows the model elements that form the AOM Core metamodel. This metamodel includes two sub-diagrams that are represented each in a separate slice (dynamic and static crosscutting slices)

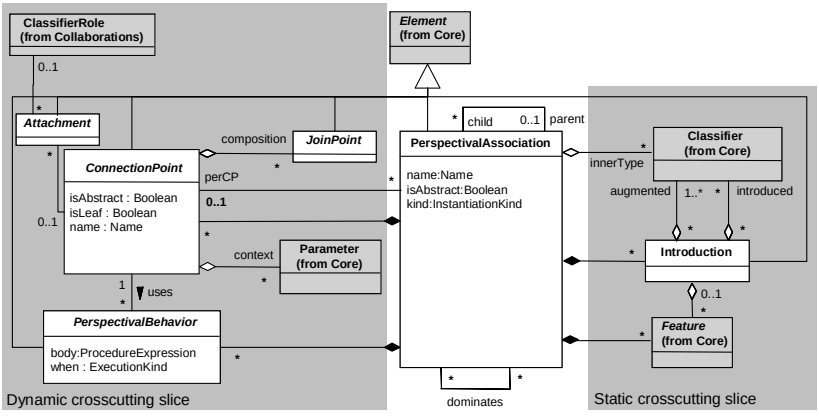


Figure 6.4: The UML Space for AOM — A Low-Level View of AOM Core

PerspectivalAssociation. A perspectival association is a mechanism for representing interaction aspects among a group of collaborating parts. It is similar to an AssociationRole viewed from a specific perspective that focuses on identifying and reifying crosscutting concerns into a separate module. To avoid the X-Syndrome, PerspectivalAssociation extends the UML meta-class Element, which is the smallest element within UML. Moreover, it consists of a collection of ConnectionPoints, PerspectivalBehavior, and Introductions. It may also declare features and inner types. A PerspectivalAssociation can be related with another PerspectivalAssociation in precedence or inheritance relationships.

ConnectionPoint. A ConnectionPoint is an element of a PerspectivalAssociation that provides a mechanism for composing JoinPoints to which PerspectivalBehavior can be added. A ConnectionPoint exposes its context in terms of Parameters that can be used in the body of the associated PerspectivalBehavior.

Attachment. An Attachment is a mechanism for applying the specification of a ConnectionPoint to the context of an associated ClassifierRole. This context is obtained from the relationship

between the ClassifierRole and its base Classifiers. We use this because UML does not treat roles as explicit types.

PerspectivalBehavior. PerspectivalBehavior defines crosscutting behavior that can be executed at the associated ConnectionPoints. The additional behavior can be executed at different points in time (before, after, or around) relative to the moment at which the associated ConnectionPoint is reached.

Introduction. An Introduction provides support for structural amendments. For example, it allows one to declare supplementary behavioral and structural features, and to modify an existing hierarchy.

6.7 Resultant View

The resultant view is a partial description of a software architecture. This is best explained by means of an example: the *Traffic Light Control System*.

The following describes a running example of an application of the UML Space for AOM. It presents an aspect-oriented solution to the traffic light problem:

This problem consists of developing a software system that should be built by composing different types of components developed at different times by different people: four traffic lights, and a timer. The requirements of the system are quite simple:

- The timer component is responsible for triggering an event at regular time intervals
- A traffic light component should always switch on the same light as its opposite peer
- A traffic light component should never show the same light as its direct neighbors
- A traffic light component should not maintain any knowledge of the state of its peers

An important dimension of the design addressed in the solution to this problem is the interaction among individual components. The solution does not describe the components themselves; it rather focuses on the roles played by those components within the context of the interaction. Figure 6.5 shows a perspectival association between components' roles that is based on the architectural configuration model shown in figure 6.7. It describes an aspect-oriented model for an event connector. This event connector consists of two ClassifierRoles, called Producer and Consumer, that are attached via two ConnectionPoints (shown as black circles), named as newState and newConsumer. This event connector provides a flexible way of making components communicate without any coupling between them. This is achieved by modeling the EventConnector as a PerspectivalAssociation (shown as a dashed ellipse) that modularizes the interaction concerns among the participating components. These interaction concerns are mainly related to the adaptation of components to be combined and configured to build a new application. They include the establishment of interconnections among components, the maintenance of those interconnections, and the mediation of the events exchanged between the participating components

Figure 6.5: A Perspectival Association between the Component Roles

The connector model shown in figure 6.5 allows us to overcome the limitations stated in section 5.5.2 for two reasons. First, it allows one to modularize multiple interaction concerns; and second, the use of roles instead of components allows us to overcome the limitations stated in section 5.5.2.

6.7.1 Identifying Causes and Effects for the EventConnector

The EventConnector mediates events among the interacting components based on the cause-effect principle introduced earlier.

Cause	*Effect*
When the producer triggers a state change ...	...inform consumers to change their state.
When a consumer is created...	...keep a reference on it.

Figure 6.6: The Cause-Effect Principle Applied on the EventConnector

Figure 6.6 shows a table that illustrates the cause-effect principle applied on the example of the traffic light system. The content of the table can be interpreted as follows: When a component playing the Producer role triggers an event (at predefined time intervals), each of the components playing the Consumer role is instructed to change its internal state. In order to deliver events to components playing the Consumer role, references to those components must be obtained at their creation time.

6.7.2 Designing the EventConnector

An aspect-oriented way of accomplishing the causes and effects identified in section 6.7.1 consists of the following two steps: 1) using ConnectionPoints to capture the causes; and 2) realizing the effects within the body of the PerspectivalBehaviors associated to the ConnectionPoints.

Figure 6.7 describes the main structure of the `EventConnector`. It provides an elaborated view of the model (depicted in figure 6.5) that gives details on both the Connection-

Points (previously shown as black circles), and the PerspectivalAssociation (previously shown as a dashed ellipse)

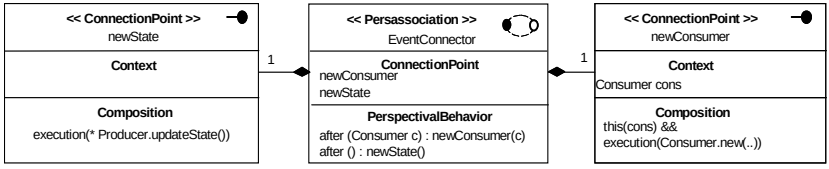


Figure 6.7: Static Structure of the EventConnector

6.7.2.1 Capturing the Causes

Capturing the causes drives the design of the ConnectionPoints. The EventConnector has two ConnectionPoints that are: newState and newConsumer. The structure of a ConnectionPoint is shown with the <<ConnectionPoint>> stereotype and has two compartments: one for exposing the context, and another one containing the JoinPoint composition. We use the AspectJ notation as language for expressing the JoinPoint composition.

A major reason for designing the newState ConnectionPoint is to capture the exact moment at which the state changes within the Producer (i.e., when the updateState() method is invoked). Similarly, the newConsumer ConnectionPoint detects the creation of components playing the Consumer role, and it exposes a reference to that new component within its context.

6.7.2.2 Realizing the Effects

The realization of the effects drives the design of the PerspectivalBehavior. This is mainly achieved by determining: 1) when the body of this PerspectivalBehavior should be executed; and 2) which ConnectionPoint is associated with this PerspectivalBehavior. Right now, we are using the AspectJ notation for *advice* as language for expressing the signature of a PerspectivalBehavior.

Figure 6.8 shows two pieces of Java code implementing the bodies of the PerspectivalBehaviors associated with the newState and the newConsumer ConnectionPoints.

Note that the body of the perspectival behavior is not shown on the design illustrated in figure 6.7, as it is an implementation artifact.

```
java.util.Iterator it = consumers.iterator();
while (it.hasNext()) {
   ((Consumer) it.next()).changeState();
}
```

Figure 6.8: Body of the Perspectival Behavior Associated with newState

Figure 6.8 shows the piece of Java code describing the notification of Consumers on state changes within the body of the perspectival behavior associated with newState.

```
if (cons != null) {
    consumers.add(cons);
}
```

Figure 6.9: Body of the Perspectival Behavior Associated with newConsumer

Similarly, figure 6.9 shows the Java code within the body of the perspectival behavior associated with newConsumer. It describes how to store the reference to the newly created Consumers into an internal list container.

6.7.2.1 Refining the Design of the Connector Model

Figure 6.10 presents a detailed design of the connector model given in figure 6.5. This figure shows explicitly the attachments between the ConnectionPoints and ClassifierRoles. The attachment relates the specification of a given ConnectionPoint with a component via the role played by that component. For example, the attachment to the left uses the JoinPoint composition of the newState ConnectionPoint in the contextual relationship between components and their Consumer roles.

Additional information in figure 6.10 is the declaration of the list container needed by the EventConnector to store the Consumers.

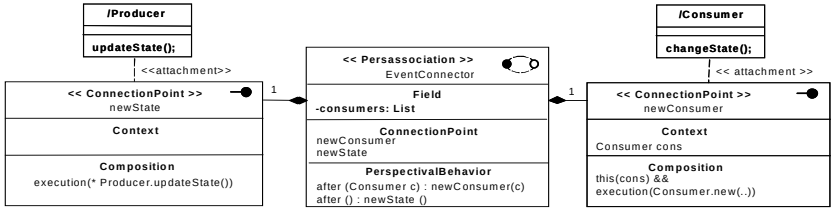


Figure 6.10: The Complete Design of the EventConnector

6.7.3 Mapping the Aspect-Oriented Model to an AspectJ Program

This section describes how to map our connector model onto an aspect (as defined in AspectJ) and introduces a running example that evaluates our contribution. An important issue in the mapping of the connector model onto an AspectJ program is to describe how to express the Producer and Consumer roles in the program code. One way to address this issue consists of allowing arbitrary components to play these two roles. This is currently done by hand.

To map the connector design to an AspectJ program, we proceed as follows: each ConnectionPoint is mapped onto a pointcut with the same name as shown in the code of figure 6.10. Each PerspectivalBehavior is realized by a separate advice whose bodies con-

tain the code given in figure 6.8. Finally, the PerspectivalAssociation itself is mapped onto an aspect named `EventConnector` that contains the pointcuts and advice declared above, as well as a private list member.

```
aspect EventConnector {
   private java.util.List consumers = new java.util.LinkedList();

   pointcut newConsumer (Consumer cons) :
      this(cons) &&
      execution(Consumer.new(..));

   pointcut newState () :
      execution(* * Producer.updateState());

   after (Consumer cons) : newConsumer(cons) {
      if (cons != null) {
         consumers.add(cons);
      }
   }

   after () : newState() {
      java.util.Iterator it = consumers.iterator();
      while (it.hasNext()) {
         ((Consumer) it.next()).changeState();
      }
   }
}
```

Figure 6.11: Implementation View of the EventConnector Aspect

6.8 Final Remarks

This chapter presented a case study on software composition by interaction concerns. It introduced an approach to building complex software systems by using aspect-oriented connectors. It provided a UML Space for aspect-oriented modeling which allows us to model interaction aspects for adapting independent components to a new environment. The approach proposed in this chapter allows us to express different aspects of software interactions into aspect-oriented models. However, one of the limitation of this approach is related to the composition of interaction aspects themselves. Further research is required to come out with a more appropriate language for modeling PerspectivalBehavior.

Another limitation is related to the assignment of roles to components. This is currently done manually; the provision of mechanisms for enabling dynamic assignment of roles to component requires further research..

Chapter 7:
On-Demand Remodularization PCS

This chapter presents an approach to implementing MDSOC for UML using the projection mechanism for reifying concerns into software in order to achieve remodularization of software on-demand. It introduces techniques for achieving architectural design by concerns, using concern-oriented modeling. Moreover, the chapter describes a concern-oriented pattern, called On-Demand Remodularization pattern.

7.1 Viewpoint Name

On-Demand Remodularization Viewpoint

7.2 Sources

Overview Information. The On-Demand Remodularization (ODR) viewpoint provides the ability to remodularize a software-intensive system according to new architectural concerns, non-invasively, and without eliminating encapsulations based on prior decompositions.

Summary. The ODR viewpoint presents a software architecture recovery approach that aims at reconstructing a view on the Java Drag'n'Drop architecture *as-built*. This viewpoint is part of the On-Demand Remodularization PCS. Using this viewpoint allows us to focus on two aspects of software architecture recovery:

- evaluating the usability of the Java Drag'n'Drop architecture
- remodularizing various architectural concerns without invasive change

Context. This viewpoint is part of the On-Demand Remodularization PCS.

References. [TOW+99] [TO00]

History of change. None.

Glossary. None.

7.3 Concerns

The major concerns for the On-Demand Remodularization PCS are the following: Integrability, modifiability, reusability, usability, understandability and decoupling.

7.4 Stakeholders

The stakeholders include: architects, managers, maintainers, designers, application developers and programmers.

7.5 Rationale

1. *Analysis of the system reveals the need for evolution in the software architecture*. Architects have recognized maintenance problems
2. *New organization of the system without invasive change*. The architecture must allow one to:
    - Add new concerns without worrying about existing modularization
    - Preserve existing relationships among concerns
3. *Managers said: It is time for major challenges in the enterprise*

7.6 Architectural Problems

This section describes different aspects of an architectural modeling problem. These aspects are described in the following sub-sections.

7.6.1 Incentive Aspects

The incentive aspects for the On-Demand Remodularization PCS are:

- The need for representing concerns before they can be populated with units
- The need for representing concerns together with the units that pertain to them
- The need for expressing interactions among units pertaining to a particular concern as a separate model slice
- The need for expressing the static structure of a group of units pertaining to a particular concern as a separate model slice
- The need for expressing an aspect mechanism as a concern-oriented model

7.6.2 Decisional Aspects

The decisional aspects for the On-Demand Remodularization PCS are:

- The need for a mechanism to support model fragmentation and de-fragmentation
- The need for a mechanism to support concern reification
- The need for expressing relationships among concerns

7.6.3 Resultant Aspects

The resultant aspects for the On-Demand Remodularization PCS are:

- The ability to reify concerns into units is satisfactory
- The ability to compose concerns without using the aspect mechanism is unsatisfactory
- The ability to support on-demand remodularization of existing software systems is satisfactory
- The ability to support model fragmentation and de-fragmentation is noteworthy. Tool support is very much required

7.7 Relationships Among Concerns

None.

7.8 Resulting View

This On-Demand Remodularization view is a collection of concern-oriented models that provide a partial description of the Java Drag'n'Drop architecture. These concern-oriented models can be used to express concerns at different levels:

Concern at the viewpoint level. This shows a concern representation that contains only the name of the concern. Reifications are not shown at this level, but additional documents might be attached to give more description of the concern. Figure 7.1 shows an example of such a concern representation.

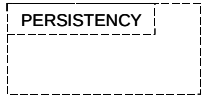


Figure 7.1: Concern-Oriented Model as a Means for Concern Representation

Concern at the view level. This shows only reifications of concerns into units. There are three options to show concerns at this level. These are shown in what follows.

Figure 7.2 shows the variant that provides a solution to the limitations of UML as described in figure 5.7. The names of the concerns (e.g., TRADING) are shown together with the units that reify them.

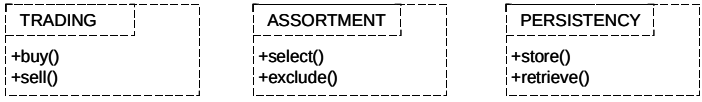


Figure 7.2: Concern-Oriented Model as a Means for Concern Reification

Figure 7.3 shows model slices that together build a concern-oriented model. Several examples of slices are given below. Model slices are UML-based hyperslices [TO00]. Similarly to hyperslices, model slices must be declaratively complete: that is, they must declare all the units they use. This allows one to reuse model slices in different design contexts.

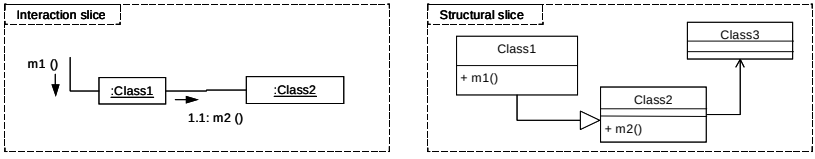


Figure 7.3: Concern-Oriented Model as a Means for Representing Model Slices

Finally, Figure 7.4 shows a concern-oriented model as an aspect-oriented mechanism. It builds on the notation introduced in the previous chapter.

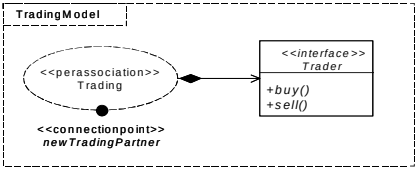


Figure 7.4: Concern-Oriented Model as a Means for Representing a Mechanism

The language constructs introduced above are the main elements used in the On-Demand Remodularization PCS. The On-Demand Remodularization view can be best explained by means of a concrete example. The next example presents the case of the Java Drag'n'Drop architecture.

7.9 The Java Drag & Drop Architecture Case Study

This section presents an application of the On-Demand Remodularization PCS on the Java Drag & Drop Architecture.

According to some computer users, Drag’n’Drop is one of the most user-friendly inventions in software. Every time it is used, the Drag’n’Drop operation happens between

two software components that are referred to as source and target components. The source component is the component on which a drag operation is initiated, while the target component is the one on which a drop operation is performed. The goal of a Drag'n'Drop operation is to transfer an object from one point (the source component) to another (the target component). This goal is usually achieved by using a pointing device, such as a mouse, for selecting the element to be transferred, and dragging it towards the target component where it should be dropped to perform the operation.

Because of its importance to users of computers, most programming languages and systems provide support (i.e., application programming interfaces or APIs) for facilitating the realization of the Drag'n'Drop operation in software applications. For example, using the Java language, all Swing components have the ability to act as source or target components—that is, they may provide an element to be dragged, or receive a dropped element.

According to the Java Drag'n'Drop architecture [LEW+02], Drag'n'Drop operations are carried out by means of collaborations among helper components, called TransferHandlers. The source and target components must be associated to a TransferHandler (via the setTransferHandler() method call) in order to be able to transfer and/or receive transferable elements.

In contrast to users of applications supporting Drag'n'Drop operations, designing a Drag'n'Drop architecture is less user-friendly. For example, the internal machinery of the Java Drag'n'Drop architecture is quite complicated: it spans over several software layers (from the native operating system layer up to the application programmer's layer). It is not the intention of this thesis to provide an exhaustive insight into the original design of the Java Drag'n'Drop architecture. We prefer to take a maintainer's perspective of the architecture as-built.

This On-Demand Remodularization view is a collection of concern-oriented models that provide a partial description of the Java Drag'n'Drop architecture.

7.9.1 Drag Support Initialization Concern

This section presents two model slices, called *structural model slice* and *behavioral model slice*, which describe together the Drag Support Initialization concern.

Figure 7.5 shows a behavioral model slice that describes the Interaction pertaining to the Drag Support Initialization concern. It shows the interaction that takes place on the side of the source component when a drag operation is initiated by invoking exportAsDrag() on the TransferHandler of the source component. First, the TransferHandler creates a DragHandler that will manage the drag operation on behalf of the source component (1).

The TransferHandler then creates a component that is able to recognize dragging gestures, and tracks the state of those gestures on the side of the source component (2). The gesture recognizer builds an entity responsible for the initiation of the Drag'n'Drop operation, called DragSource (2.1). Upon successful creation of the gesture recognizer, the TransferHandler triggers a gesture operation on the recognizer (3).

This causes the recognizer to fire a DragGestureEvent that is dispatched to the DragHandler listening to such events (3.1, 3.2). The DragHandler retrieves the transferable element from the TransferHandler (4), and passes it over to the DragGestureEvent (5), which in turn propagates the start of the dragging operation to the DragSource component (5.1).

The DragSource then creates a native component for keeping contextual information during the drag'n'drop operation (6), associates it with a Java-level context component (7), and starts the drag operation on the native layer (8). The contextual component is responsible for notifying its associated DragHandler when drag events occur within the system, and for providing the transferable element when the drag operation is about to terminate. When the pointing device is dragging a transferable element over Swing components, the native layer is notified (9), and the notification percolates up to the DragHandler responsible for handling that event (9.1, 9.1.1).

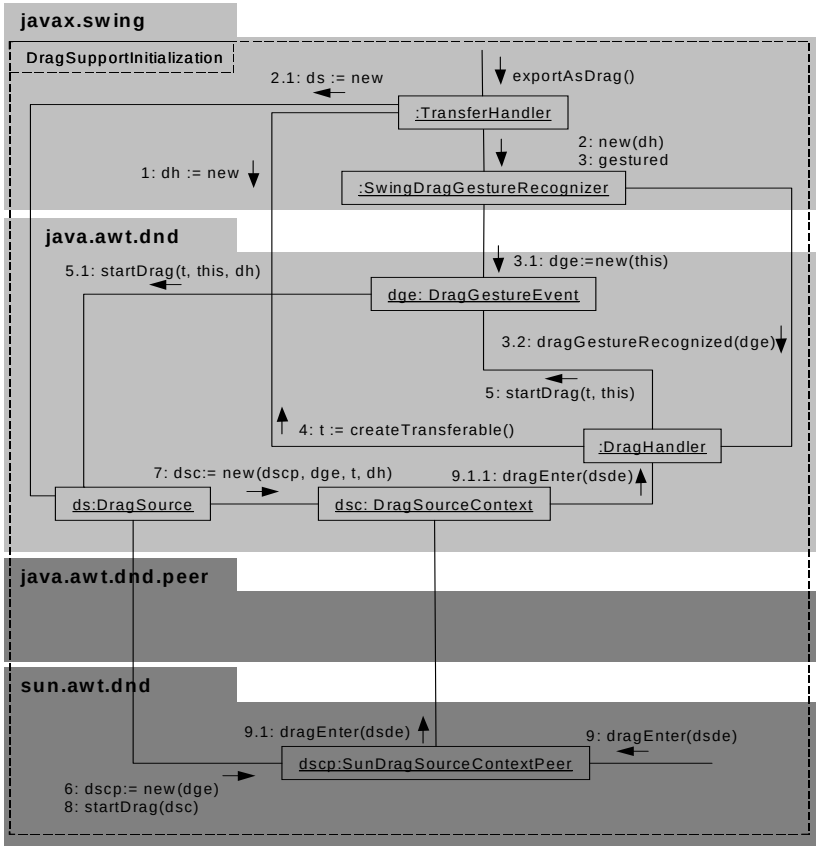


Figure 7.5: Behavioral Model Slice for the Drag Support Initialization Concern

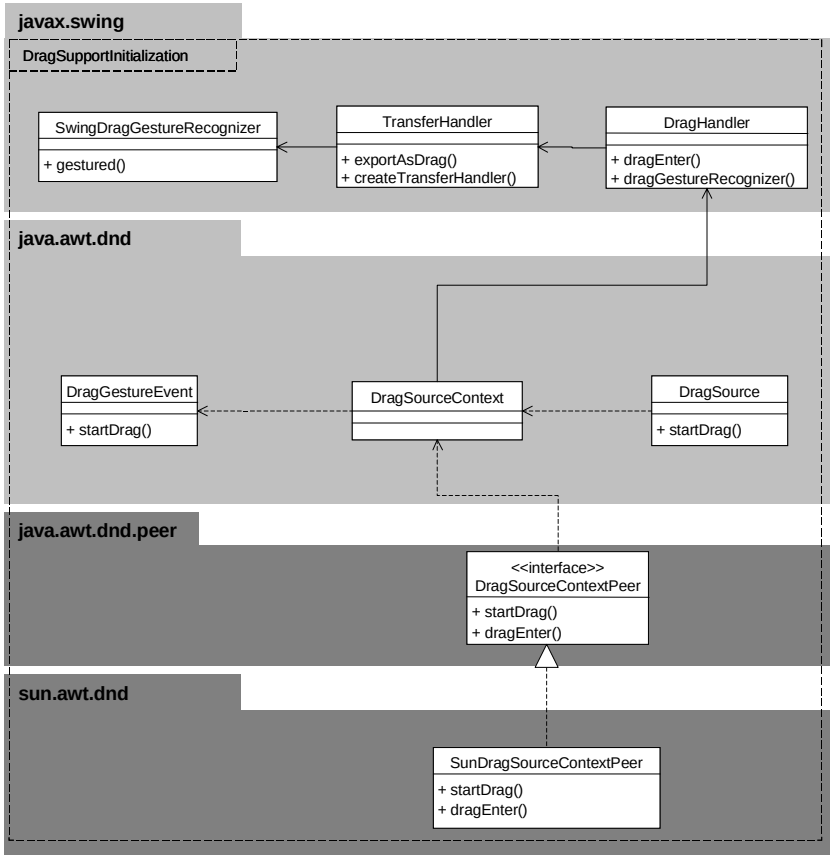


Figure 7.6: Structural Model Slice for the Drag Support Initialization Concern

The context of the interaction shown in the behavioral model slice for the Drag Support Initialization Concern is described as a separate structural model slice for the same concern. This is a concern-oriented context diagram shown in figure 7.6 which contains all the information about the reference structure pertaining to the Drag Support Initialization Concern. This concern-oriented diagram is different from the standard UML class diagram designed for a particular concern in that it is declaratively complete and contains no hidden features that belong to other concerns.

7.9.2 DropTarget Installation Concern

This section presents the behavioral model slice and the structural model slice for the DropTarget Installation Concern.

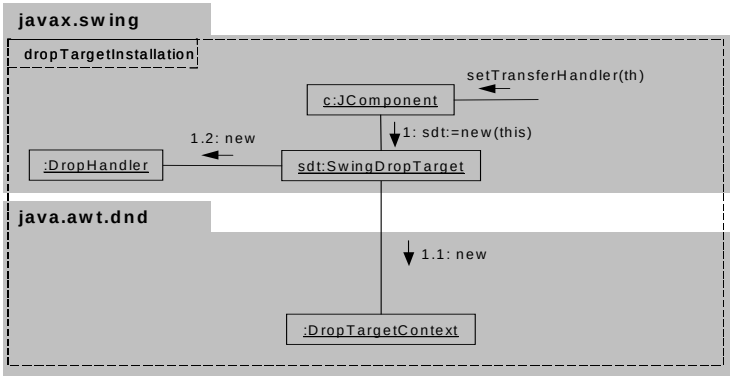


Figure 7.7: Behavioral Model Slice for the DropTarget Installation Concern

Figure 7.7 illustrates the few steps that are necessary to set up the Drag'n'Drop on any Swing components.

The only thing that is required from the developer is to install a TransferHandler on the target component via the method call setTransferHandler(). When this method is invoked, the component will create a DropTarget for listening to drop operations on the associated component. The DropTarget internally builds a DropTargetContext for holding contextual information during a Drag'n'Drop operation (1.1), and a DropHandler to which it delegates the handling of all events received during a Drag'n'Drop operation (1.2). At this point, the target component is able to handle and manage a Drag'n'Drop operation.

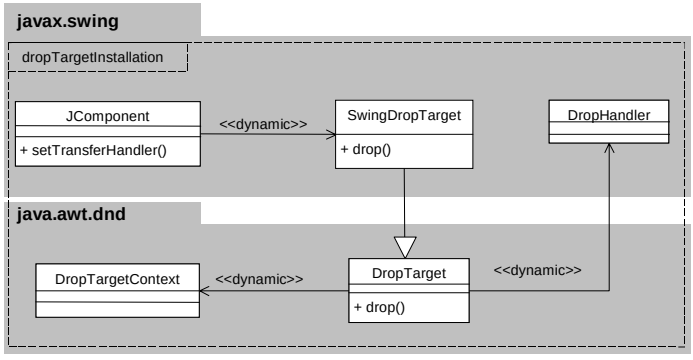


Figure 7.8: Structural Model Slice for the DropTarget Installation Concern

Figure 7.8 illustrates the context of the interaction among the units pertaining to the DropTarget Installation Concern. This figure describes in a concern-oriented model the structure of a class model that has only those units that pertain to the DropTarget Installation concern.

7.9.3 Drop Event Interception Concern

Another interesting part of a Drag'n'Drop operation is depicted in figure 7.9. The figure illustrates what happens when a transferable element is dropped on a receiving component during a Drag'n'Drop operation. First, a DropTargetDropEvent is created to hold the state of the drop operation until the drop completes (10). Once the DropTargetContext is retrieved from the drop event, we can obtain the DropTarget from that context (11) and initiate the drop operation on it (12). The handling of the drop operation is delegated to the DropHandler component (13). The latter retrieves the DropTargetContext (13.1), obtains the target component from that context (13.2), asks the target component for its TransferHandler (13.3), and acquires the element to be transferred from the DropTargetDropEvent (13.4, 13.4.1, 13.4.2). The transferred object is then imported into the TransferHandler of the receiving component, thereby terminating the whole Drag'n'Drop operation.

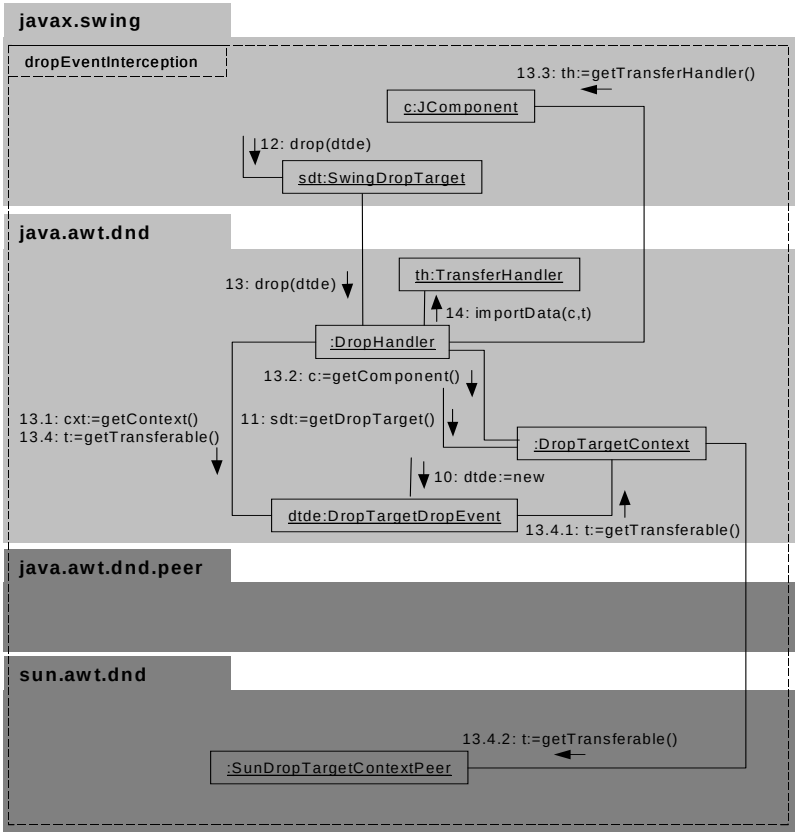


Figure 7.9: Behavioral Model Slice for the Drop Event Interception Concern

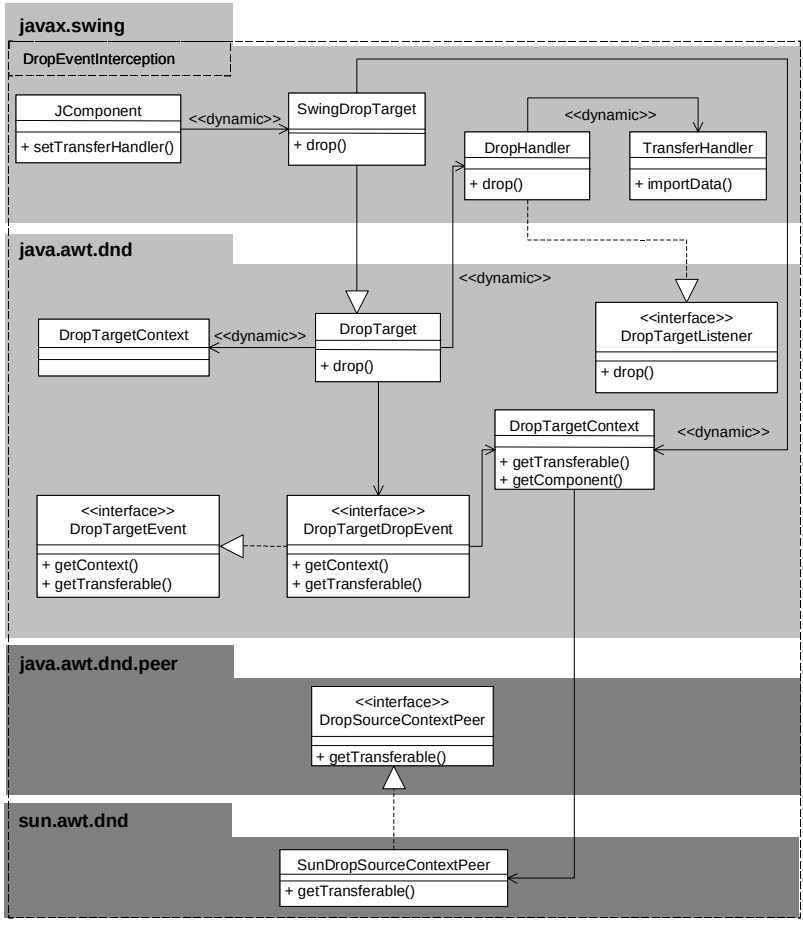


Figure 7.10: Structural Model Slice for the Drop Event Interception Concern

As for the previous concerns, figure 7.10 illustrates the context of the interaction among the units that are relevant to the Drop Event Interception Concern.

When designing the above concerns, we deliberately decided not to show all the packages of the original Java Drag'n'Drop architecture (depicted in gray, for example the java.awt.dnd package). Instead, we tried to make clear where the individual concerns are located within the different architectural models. Another reason for this decision was to

give information about the layered architectural style adopted by the Java Drag'n'Drop architecture team. Clearly, one can observe how the usability concerns lead to the production of more and more layers. Following this reasoning, we can state to some extent usability has a *negative impact* on complexity and even performance. These are relationships among concerns which we could not address earlier (for instance, not when we were filling the viewpoint schema).

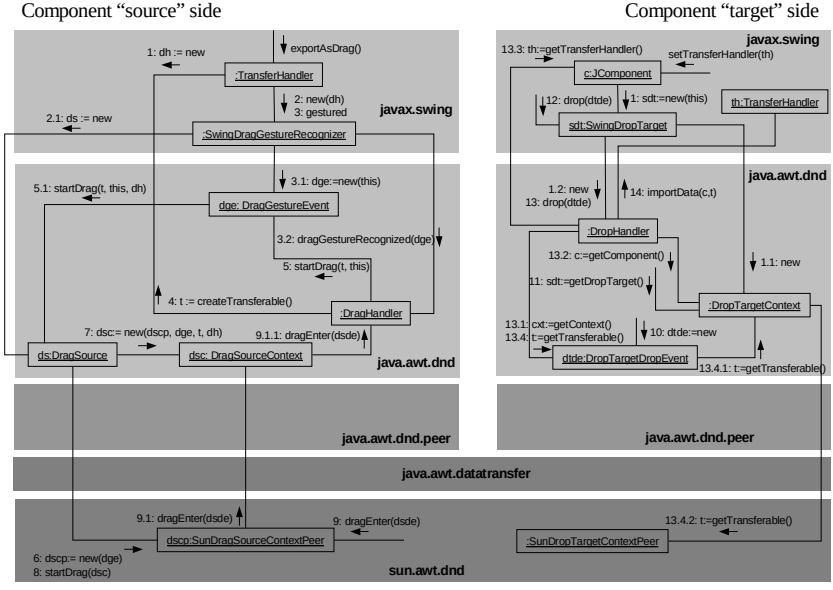


Figure 7.11: Interaction Model for using the Java Drag & Drop API

Figure 7.11 and figure 7.12 illustrate a more complete architecture recovery view of the Java Drag'n'Drop architecture. These two figures can be seen as the superposition of all the other architectural models shown previously.

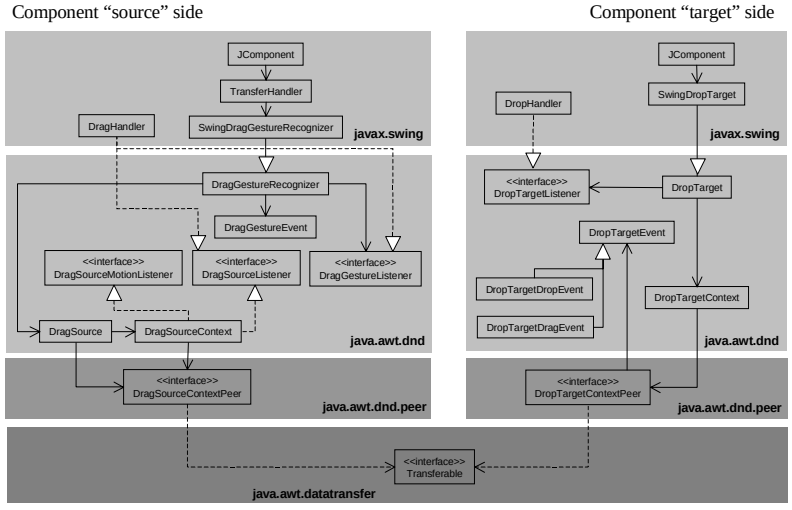


Figure 7.12: Static Structure Model for using the Java Drag & Drop API

7.10 On-Demand Remodularization Pattern

This section introduces a concern-oriented pattern which we refer to as the *On-Demand Remodularization Pattern*, ODR Pattern in short. This pattern is the mechanism we propose for supporting On-Demand Remodularization (as stated in section 7.6.3).

Like all patterns, the ODR pattern provides a solution for recurring design problems; however it is the one with a specific focus on remodularizing existing software without invasive change; and as a concern-oriented pattern it can be used at both low-level design and architectural level. When to use it as design pattern or architectural pattern depends on the kind of concern you need to remodularize.

7.10.1 Motivation

To motivate the use of this pattern, we consider one of the major problems that could not be addressed by the “Aspect-Oriented Construction PCS” on page 83. In the following, we present the solution provided by the ODR pattern to that problem.

7.10.2 Structure of the Pattern

The ODR pattern consists of three *model slices* that help remodularize an existing system into a more coherent, maintainable one:

- a managing model slice
- an enabling model slice
- a binding model slice

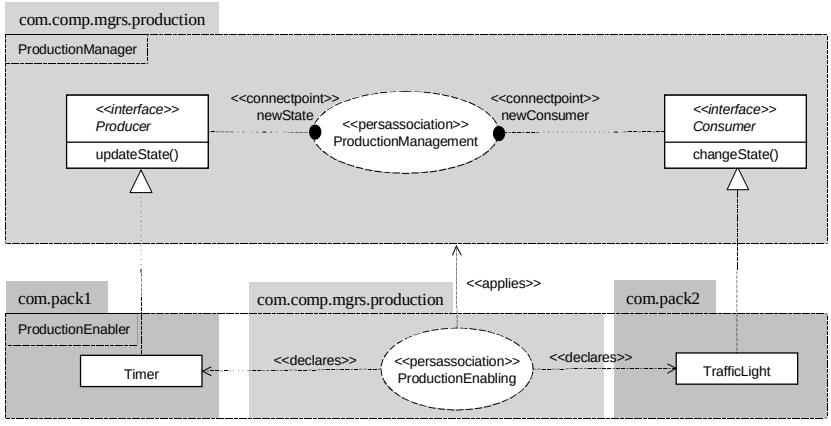


Figure 7.13: Structure of the On-Demand Remodularization Pattern

The structure of the *managing model slice* is normally described as a composition (as shown in figure 7.13) of a perspectival association and one or more interface type declarations. These types represent roles that are fundamental for instantiating the pattern. Furthermore, they may declare behavior (e.g., methods) that must be provided by components to which the roles they represent will be bound. The managing model slice may also contain any other utility types, such as classes, that would help the perspectival association perform its job. In the managing model slice, the perspectival association is responsible for defining the interaction protocol among the roles defined previously as interface types.

The *enabling model slice* is mainly in charge of attaching the pattern instance to a part of a system at hand. We say that this model slice enables the remodularization of a system by binding existing components to the roles defined in the managing model slice. This binding is achieved by introducing the interface type that defines a role to be played by a component into the inheritance tree of that component, and giving the component appropriate behavior for playing that role. In Java terms, we make the component implement the interface type. In short, the enabling model slice is the glue that allows one to apply the ODR pattern to an existing system.

The *binding model slice* is responsible for customizing and fine-tuning the behavior of components whose parent has already been adapted in the enabling model slice. This model slice can be made optional when the behavior of the parent component is perfectly suitable for its children, and does not necessitate any amendments.

The structure that results from the instantiation of the ODR pattern on the Traffic Light Control System is shown in figure 7.14. The diagram shown in this figure is different from the one depicted in figure 6.5 in that the binding between the roles and the components are explicitly shown (e.g., the Timer plays the Producer role).

This example demonstrates the integration between the Aspect-Oriented Construction PCS and the On-Demand Remodularization PCS.

7.10.3 Achieving Design by Concerns with the ODR Pattern

Figure 7.14: Instantiation of ODR for the Traffic Light Control System

In section 7.9, we have applied the On-Demand Remodularization PCS on the Java Drag'n'Drop example to support concern-oriented modeling by providing different slices (structural and behavioral) for the same concern and combining those slices to build larger models. However, concern-oriented modeling as addressed in this example does not support modeling of dynamically changing roles (e.g., when a component changes its role from source to target and vice versa, dynamically). In the same way, the solution provided above in section 7.9 is not desirable when roles are entangled with one another. Dynamically changing roles and tangling of roles are major problems when architecting software systems, especially for remodularizing existing systems.

The ODR pattern plays a key role in capturing dynamically changing roles and entangled roles. An application of the ODR pattern on the Java Drag'n'Drop architecture is illustrated in figure 7.15, and described in what follows.

In the Drag'n'Drop example, the three model slices are named, respectively, DnDManager, DnDEnabler, and DnDBinder. The DnDManager model slice contains a perspectival association called DnDManagement, and an interface type called DnDParticipant. DnDParticipant defines the behavior any component willing to participate in a Drag'n'Drop operation has to provide. A brief overview of each method declared in the DnDParticipant interface is given below:

- initDnD(): sets up a participating component for a future Drag'n'Drop operation
- getTransferable(): returns an object to be transferred when a Drag'n'Drop operation has been initiated on a participating component
- setTransferable(Object): provides the object that has been dropped on (transferred to) a participating component

- acceptTransferable(Flavor): returns a boolean indicating whether the participating component accepts the current object (denoted by its flavor or type) being transferred
- getDropLocation(): returns the exact point on the component where the transferred object has been dropped.
- setDropLocation(): sets the location where the transferred object has been dropped.

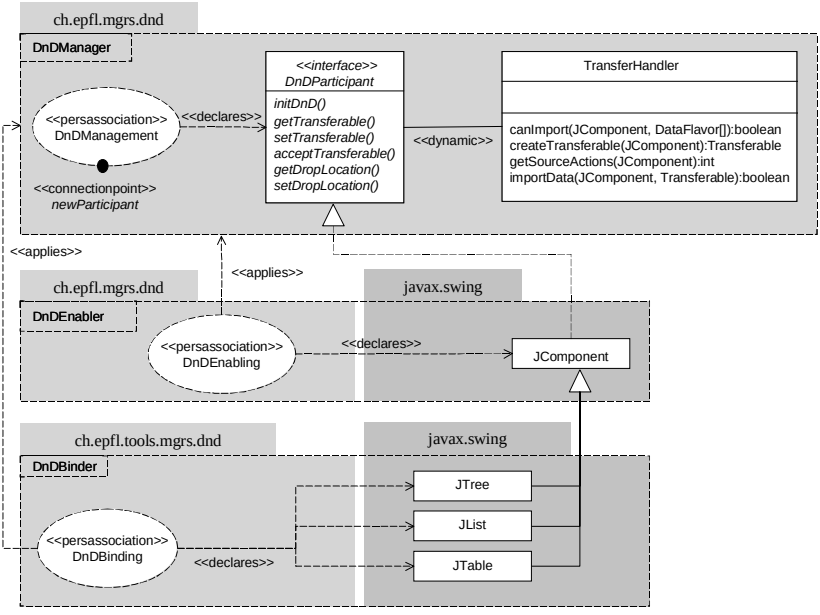


Figure 7.15: ODR Applied on the Java Drag'n'Drop Architecture

In addition, DnDManagement defines a connection point that will detect when a new DnDParticipant is created within the system. When such a creation is underway, the perspectival association invokes initDnD() upon the participant in order to set it up for upcoming Drag'n'Drop operations.

The DnDEnabler model slice contains the DnDEnabling perspectival association whose primary goal is to allow any JComponent (of javax.swing) to participate in a Drag'n'Drop operation by giving to it the capability of playing the DnDParticipant role (defined in the DnDManager model slice). Furthermore, DnDEnabling injects into JComponent behavior that shall be common to any JComponent (i.e., getDropLocation(), and setDropLocation()). All other methods need specific implementation that cannot be factored out into this model slice.

The task of the DnDBinder model slice is precisely to bridge that gap, and to introduce into subclasses of JComponent specific behavior for each of the remaining methods (i.e., initDnD(), getTransferable(), setTransferable(), and acceptTransferable()). The DnDBinding perspectival association achieves this by inserting the implementation of an appropriate behavior, specified by the interface type DnDParticipant, into each of the participant components, such as JTree, JTable or JList. This is necessary because these components may realize the same role in different ways. For instance, a JList does not handle the drop of an object the same way as a JTable or a JTree.

In summary, the DnDManager model slice defines the interaction protocol between components engaged in a Drag'n'Drop operation (playing the DnDParticipant role). The DnDEnabler model slice applies the pattern instance to a system at hand by making components play the DnDParticipant role defined in the DnDManager model slice. Finally, the DnDBinder model slice customizes the Drag'n'Drop behavior to each participating component. It is worth noting that this Drag'n'Drop ODR instance encloses all Drag'n'Drop concerns; there are no other places in the system where Drag'n'Drop concerns are being handled.

7.11 Final Remarks

This chapter presented the On-Demand Remodularization PCS by defining a viewpoint language and using that viewpoint language to remodularize concerns in the Java Drag'n'Drop architecture.

The notations used in this chapter provide an example based on the use of the UML Space as proposed by the PCS Framework. Beyond the notations, the On-Demand Remodularization PCS contributed a concern-oriented pattern, called ODR pattern. Clearly, this pattern can be applied at both levels: architectural level and design level. However, when applied at the programming level, using for example AspectJ, it is important to notice that programmers must provide additional elements that must inherit from the legacy part to be remodularized. This is due to the current limitations of the aspect-oriented programming language AspectJ, as it does not allow one to make introductions into legacy components.

Chapter 8:
The Service-Oriented PCS

This chapter presents an integration of the Service-Oriented PCS with a well-known Viewpoint-Oriented approach to documenting software architectures. This chapter uses the Video Surveillance System example introduced in chapter 2 to show the applicability of the PCS Framework on other architectural description approaches.

8.1 Introduction

This chapter presents an example of how the PCS Framework supports architectural documentation based on other well-accepted practices. To validate the PCS Framework with well-known software architecture documentation approaches other than the IEEE-Std-1471, we have decided to apply a viewpoint-oriented approach as described in [CBB+02]. The example we use is based on the motivating case study introduced in chapter 2.

Using PCS, we have developed a *service-oriented view* of the video surveillance service development problem. The approach taken to document the software architecture of the video surveillance service is based on the notation of the UML profile for structural descriptions [KS00a].

The service-oriented view is documented, using a *view documentation template*. The view documentation template consists of the following parts:

- The primary presentation
- The element catalog
- A context diagram
- A variability guide
- An architecture background (with rationale, results of analysis, and assumptions made)
- The related view packets
- Other information

Figure 8.1 summarizes our mapping strategy relating the view documentation template to the PCS Framework. Details of the mapping strategy are demonstrated in applying the Service-oriented PCS on the example of video surveillance service.

Viewtype Documentation Template	Service-Oriented PCS
Primary presentation	Main architectural model
Element catalog	Static structure model Behavioral specification model *Perspectival elements*
Context diagram	Configuration model • Context of the configuration
Variability guide	Configuration model • Configuration manual o Base configuration o *Perspectival configuration*
Architecture background (rationale, results of analysis, and assumptions made)	Textual descriptions
Related view packets	Models, diagrams or textual descriptions
Other information	Textual descriptions

Figure 8.1: Mapping Between a Viewtype and the PCS Framework

8.2 Main Model of the Service-Oriented View

Figure 8.2 depicts the main model of the video surveillance service architecture. This diagram documents the video surveillance service from a bird's-eye-view. The main model involves three essential elements: the service component types CameraDevice and SurveillanceStation, and the connector type VSServiceConnector. The relationships among these elements are shown as attachments between component types and the connector type: to be exact, relationships represent the attachments between realizations of the component types and connector roles.

In this figure and in the remainder of the chapter, a key for icons or diagram elements is placed below each diagram that explains the meaning of symbols used.

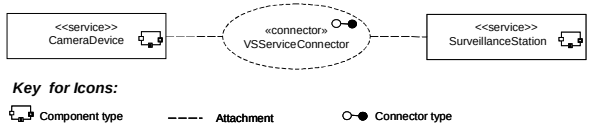


Figure 8.2: Main Model of the Service-Oriented Architectural View

8.3 Static Structure Model

In this section, the *service-oriented model elements* are described in more details; it provides a documentation of the elements and their properties, relations and their properties, element interfaces and element behavior.

8.3.1 Component Structure Specification

The documentation of the architectural component types and their properties is depicted in figure 8.3; this figure describes the service component types CameraDevice and SurveillanceStation.

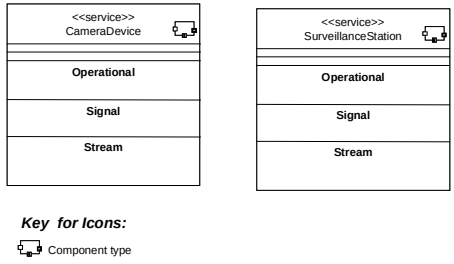


Figure 8.3: Component Structure Specification

The service component types documented in figure 8.3 do not show properties. However, some properties of component types can be shown as attributes (e.g., the format of video streams produced) in the first unnamed compartment. Other properties can be shown as operations that describe, for example, certain life-cycle management responsibilities of the component type; such responsibilities are usually shown in the second unnamed compartments. In addition, the interaction-related properties can be described in the last three compartments with the names: Operational, Signal and Stream.

Component Interface Elements

The following sub-section documents the interface elements of the component types present in the architecture.

Figure 8.4 presents three standard UML interfaces, CamConfiguror, CamController and Alarm. Each of these UML interfaces must be either provided or required by realizations of the SurveillanceStation and CameraDevice component types.

The interface CamConfiguror consists of a set of operations that allow one to configure various video cameras. CamController specifies the operations allowing one to control realiza-

tions of camera devices. The Alarm interface specifies the mechanisms required for alerting and informing the police in an emergency situation.

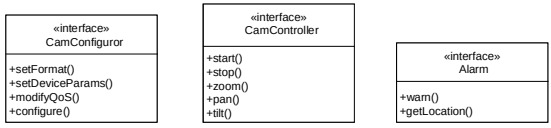


Figure 8.4: Component Types' Interface Elements

8.3.2 Connector Structure Specification

The connector type specifies the connector instances that mediate interaction among communicating components according to different interaction protocols. As an example, the structural specification of the VSServiceConnector is shown in figure 8.5. According to this figure, a significant property of the VSServiceConnector is that each interaction protocol must be applied to interconnect at least two or more Connectionpoints of the following types: CamConfiguration, StreamEndPointSignaling, CamControl, VideoStream and StreamConnectionMgmt. Each Connectionpoint type is presented as a small circle on the boundary of the connector type. A more elaborated description of these five Connectionpoint types is given in figure 8.6.

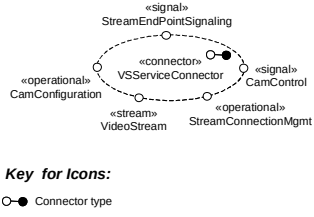


Figure 8.5: The VSServiceConnector Structure Specification

Each interaction mediated by an instance of the connector type VSServiceConnector must enter that connector through a specific connection point, which conforms to one of the Connectionpoint types listed above.

Connector Interface Elements

Figure 8.6 presents the static structure of the connector type VSServiceConnector, together with its interface elements, i.e., its Connectionpoint types. In this figure, VSServiceConnector

is documented as a composition of various Connectionpoint types: CamConfiguration, StreamEndPointSignaling, CamControl, VideoStream and StreamConnectionMgmt.

The keyword "Connector::" preceding the name of a Connectionpoint type indicates the scope of that Connectionpoint type. This allows one to document the locus of definition of individual Connectionpoint types.

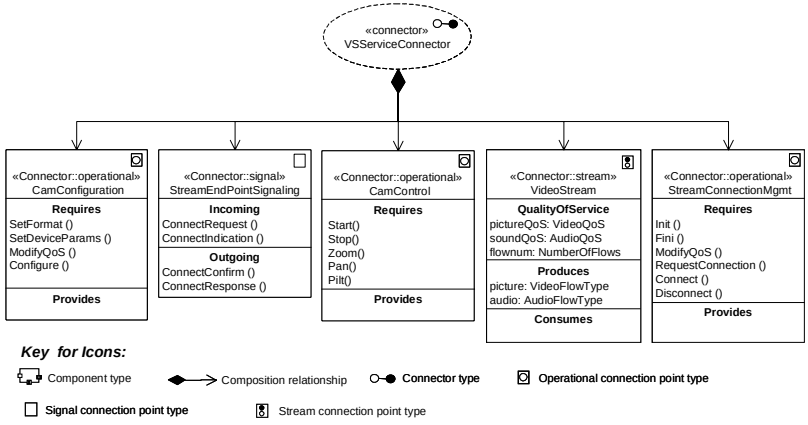


Figure 8.6: Interface Elements of VSServiceConnector

CamConfiguration defines all configuration related interactions that can take place between the video surveillance station and compatible camera devices. This Connectionpoint type documents the types of possible configuration calls that are exchanged between the interacting parties.

StreamConnectionMgmt specifies the Connectionpoint for mediating interactions for initiating and finishing the negotiation procedure in point to point multimedia connections. It describes how to control and coordinate the connection activities that are particular to stream connections between multimedia devices. Moreover, StreamConnectionMgmt determines the interactions related to the control of individual flow endpoints composing a stream endpoint. It also describes how to control and manage flow connections between multimedia devices.

VideoStream describes a Connectionpoint type that defines a set of multimedia data flows, where each flow represents a continuous sequence of objects in a specific direction. This Connectionpoint type defines the continuous media transfer between components and describes the quality of service constraints that are related to it.

The Connectionpoint type StreamEndPointSignaling is needed to mediate a set of signals for the establishment and release of stream connections (in a nonsophisticated wide area or local network).

Finally, the CamControl Connectionpoint type represents an extension of the CamController interface (shown in figure 8.4) and specifies the operations allowing one to control realizations of camera devices. In contrast to the CamController interface, the CamControl Connectionpoint type provides the ability to specify calls on operations that are provided (i.e., calls defined within the provides compartment) and required (i.e., calls defined within the requires compartment) by a participant component.

8.4 Behavioral Specification Model

In this section, the *behavioral specification model* for the Video Surveillance System is presented.

8.4.1 Component Behavior Specification

No component behavior was defined for this case study.

8.4.2 Connector Behavior Specification

The behavior of an architectural element can be described by using different types of UML diagrams, including activity, collaboration, sequence and statecharts diagrams.

Figure 8.7 presents an example of protocol state machine, shown in a statechart diagram, that describes the interaction protocol between two conjugated connection points; both connection points are specified by the Connectionpoint type StreamEndPointSignaling, as shown in figure 8.6.

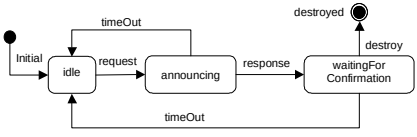


Figure 8.7: Protocol State Machine for StreamEndPointSignaling

The protocol state machine of StreamEndPointSignaling describes the allowable sequences of signal events that are related to the establishment and release of stream connections between two communicating components.

The initiating component must be in the state idle to send an instance of the signal type ConnectRequest (e.g., the signal ConnectRequest), which results in the request signal. When the ConnectRequest signal arrives at the connection point on the site of the receiver, an indication signal (entry event of announcing state, not shown) occurs.

The receiver component gets a ConnectIndication signal announcing that a component wants to connect to it. Then the receiver component may send the ConnectResponse signal to tell whether it wants to accept or reject the pending connection request. During this period, the initiating component will be in the state waitingForConfirmation. If the connection request is accepted, the response signal occurs at the connection point on the site of the initiating component. This results in the arrival of the ConnectConfirm signal at the interface of the component issuing the connection request. In both the announcing and waitingForConfirmation states, it is possible that the process restarts when the timeOut signal occurs. In the waitingForConfirmation state, the connection is deleted when the destroy signal occurs.

Figure 8.8 provides another view on the interaction protocol shown in figure 8.7. In addition, it focuses on the messages exchanged between the connection points.

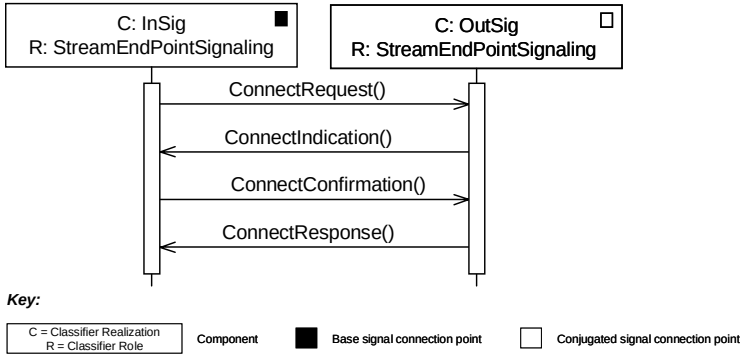


Figure 8.8: Protocol Sequence Diagram for StreamEndPointSignaling

8.5 Perspectival Elements

In this section, the perspectival elements for the Video Surveillance System are presented.

8.5.1 Structural Specification of Perspectival Elements

Figure 8.9 shows a static structure diagram that focuses on the attachments between the CameraDevice and SurveillanceStation, as well as the properties of these attachments.

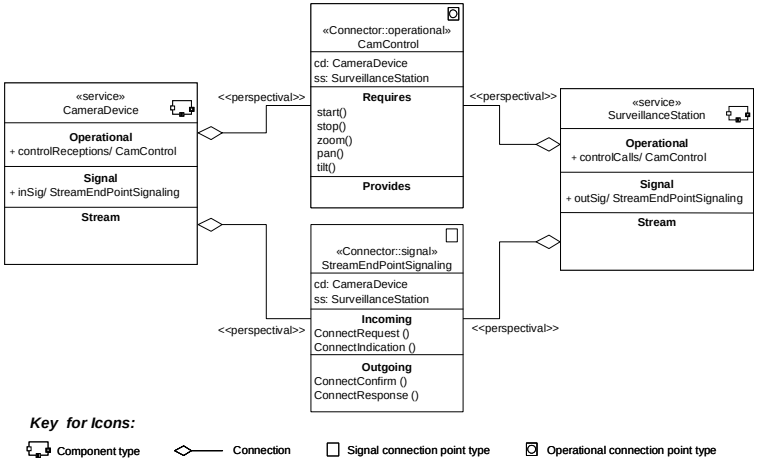


Figure 8.9: Perspectival Associations and their Properties

Each of the component types is documented as a set of interaction roles shown in the Operational and Signal compartments. The interaction roles (named controlReceptions, inSig, controlCalls and outSig) do not belong to the design-time structures of the CameraDevice and SurveillanceStation, respectively; instead, they are assigned to the component types when attachments are introduced. Similarly, the *perspectival* attributes ss (of type SurveillanceStation) and cd (of type CameraDevice) are "woven" (i.e., added) into the Connectionpoint types, CamControl and StreamEndPointSignaling.

Figure 8.9 shows four attachments; each attachment is represented by a perspectival association, i.e., the connection, that is navigable from the Connectionpoint types to the component types and vice versa. By adding attachments between the Connectionpoint types and the component types, you connect the component types with the interaction roles and you weave perspectival elements dynamically into both types. In this case, the description of the weaving rules must be part of the documentation of the perspectival association, i.e., the connection.

8.5.2 Behavioral Specification of Perspectival Elements

None.

8.6 Configuration Model

In this section, the *configuration model* for Video Surveillance System is presented.

8.6.1 Context of the Configuration

Figure 8.10 depicts the context diagram for the video surveillance service architecture. This diagram draws up the boundaries of the developers' tasks; it is useful for describing the VideoSurvellanceSystem for stakeholders who are interested in the externals of the system rather than the internals. Thus, the context diagram allows you to answer questions like: What belongs to the system of focus? What does not belong to it? How does the system communicate with its environment?

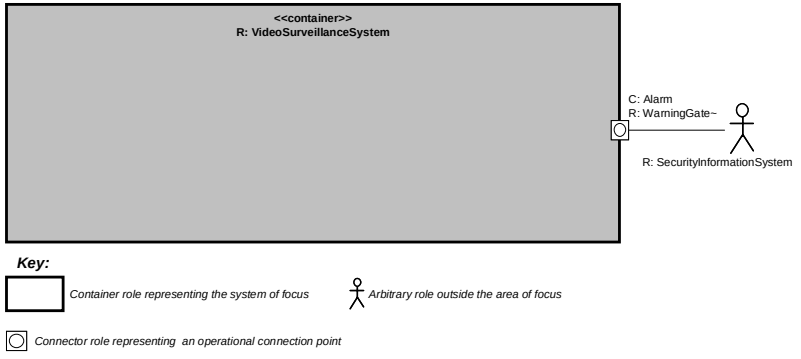


Figure 8.10: The System Context Diagram

In the context diagram, VideoSurveillanceSystem is our *system of focus*; it is modeled as a Classifier role that represents a container for the configuration which encloses a collection of realizations of the component types and the connector type shown in section 8.3 as elements of the architecture. The system of focus is colored in gray to indicate that details on the internal structure of the configuration are not relevant for the purpose of this diagram; these are documented in the next section.

SecurityInformationSystem is another Classifier role that describes a placeholder for the information system of the police that will receive and handle all alerts sent by the video surveillance system. Although SecurityInformationSystem is an important part of the video surveillance service architecture, it is not considered as the system of focus in this context. The SecurityInformationSystem is said to be a part of the environment, it would become the system of focus in another context.

VideoSurveillanceSystem communicates with its environment through the Connectionpoint alarm, that is specified by the Connectionpoint type WarningGate (the specification of this type has been left out). alarm, on one hand, represents the point of contact through which the VSServiceConnector transfers alerts to the police via the SecurityInformationSystem; on the other hand, it represents the point through which the system can receive queries from the police, concerning the video sequences it requires.

Ideally, once the interaction between the VideoSurveillanceSystem and the SecurityInformationSystem has been documented, architects and developers can concentrate on the system at hand to perform further development tasks. For example, a developer team could be in charge of providing an internal design and implementation of the VideoSurveillanceSystem; another team could be responsible for testing it, while a third team could be responsible for developing the SecurityInformationSystem.

8.6.2 Configuration Manual

This section examines variation points in the component-and-connector view of the video surveillance service architecture. When working out the variation points, we found it useful to try to achieve variability around the attachments between the roles and the realizations of the types specified in the architecture.

The following two *variants* of configuration show examples of systems that fulfill the Classifier role representing the system of focus (i.e., VideoSurveillanceSystem).

Base Configuration

This variant of configuration is shown in figure 8.11. It identifies and characterizes a variation point within the VideoSurveillanceSystem which is surrounded by the following three Connection points: outStream, controlReceptions and inSig.

Figure 8.11 shows a configuration of the system that consists of two instances of the component types, SurveillanceStation and CameraDevice. Each instance fills a placeholder representing an expected realization for a particular component type. The component type instances (realizations) shown in the figure are unnamed. The placeholders are shown by the Classifier roles, named DigitalCamera and Station.

The realization of each component type is attached not only to its Classifier role (DigitalCamera or Station), but also to three different Connectionpoints of types: CamControl, VideoStream and StreamEndPointSignaling. For example, the realization CamDev of the component type CameraDevice fulfills the Classifier role DigitalCamera. This is expressed as CamDev/DigitalCamera:DigitalCamera, by following the notation "*realization name/Classifier role name: Classifier name*". (Note that this notation is not part of standard UML, it is an extension specific to the structural profile for software architecture descriptions). Moreover, the realization CamDev is attached to the Connectionpoints outStream, controlReceptions and inSig. Morover,

every Connectionpoint is linked to its *conjugate* via a duct. We took the concept of "conjugated" elements from B. Selic et al. in [SR98] and [SGW94].

A conjugate Connectionpoint is an end point of an interconnection that is compatible with the other end point (i.e., both have the Connectionpoint type), but one end point is the inverse of the other.

The types of the interconnections involved in the static system configuration depend on the types of the Connectionpoint (or the form of communications) supported by the VSServiceConnector. The Connection Points depicted in the figure determine the interconnections involved in the VSServiceConnector.

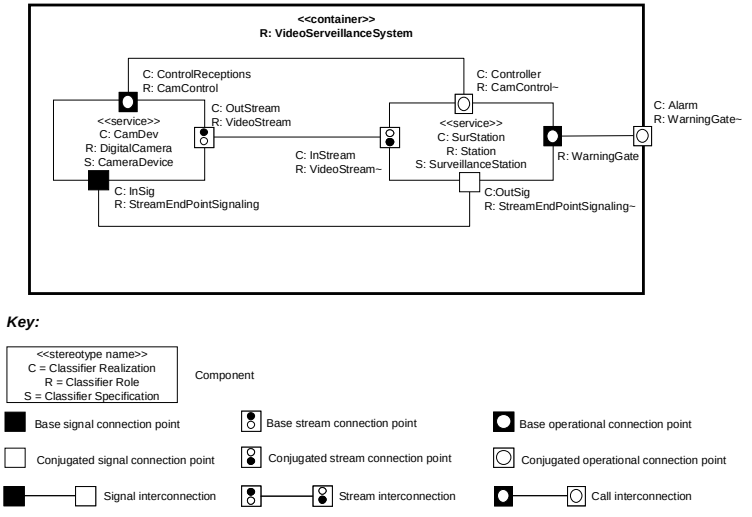


Figure 8.11: Base Configuration Diagram

There are two issues to be addressed by the diagram shown in figure 8.11:

First, what are the *elements affected* by this variant? Well, to remain simple, the configuration of the system leaves this open. The elements affected are not documented. No dynamism is observable. The configuration is rather static and less variable. However, this option can be seen as defining an implicit variation point, because it allows you to replace a simple realization of a component type by another, possibly a more complex one. For example, when considering the placeholder for realizations of the Component type CameraDevice as a variation point, various systems (or applications) can be built, based on the same architecture.

Second, what is the *binding time* of this variant? Clearly, binding in this case can be achieved at low-level design time, implementation time or at initial load time. The notion of

binding, as defined by the authors of the viewtype model ([CBB+02], p. 209), refers to “the decisions that will be made by a member of a development team prior to system deployment”.

Perspectival Configuration

This documents the configuration of the VideoSurveillanceSystem from a fault tolerance perspective. The diagram shown in figure 8.12 describes a perspectival configuration that is structured in two parts.

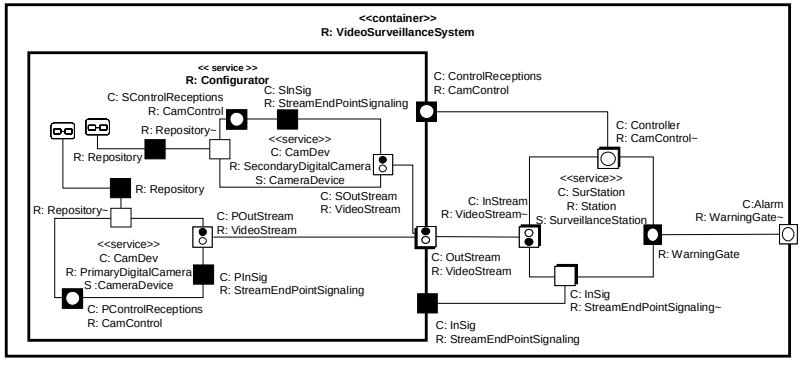


Figure 8.12: A Perspectival Configuration Diagram

One part of the configuration (on the right hand side) remains the same as in variant 1. It includes the component type’s placeholder, named station, as well as all its attached Connectionpoints. The other part of the configuration shows some variability.

In this variant, we identify and characterize the variation point explicitly, by applying the *Component Configurator* design pattern on the “portion of the design space” surrounding the three Connection points mentioned in variant 1 (OutStream, ControlReceptions and InSig). The Classifier role Configurator contains two different realizations of the Component type CameraDevice, playing the roles PrimaryDigitalCamera and SecondaryDigitalCamera, respectively.

The Classifier role Configurator essentially describes a placeholder for a Configurator component type that is responsible for realizing the following configuration services:

1. Configure the components into various processes without having to shut down and restart the running application processes.
2. Link and unlink the application components dynamically at runtime, without modifying, recompiling or relinking them.

Some questions that come to mind are:

- How can we combine the configuration model shown in section 8.6 with the Component Configurator pattern introduced in chapter 2?
- What are the elements affected by the options?
  The variation point is explicit; it is represented by the unnamed classifier role Configurator. The elements affected by this variation point are shown in figure 8.11. Dynamism is observable within the variation point. The introduction of the Configurator in the design makes the configuration more complex, but it also becomes more dynamic and more variable.
  The elements affected by the variation point are explicitly shown within. This makes the configuration simple, less variable and static.
- What is the binding time of an option?
  Binding cannot be completed at low-level design or implementation time; it must be achieved at runtime. Examples of runtime bindings include the interconnections between connection points: OutStream is bound to PoutStream, OutStream is bound to SOutStream, etc.

8.6.3 Architecture Background

Roles are the placeholders for the realizations of component types and Connectionpoint types in a configuration. Each role represents a variation point within a particular configuration of the software architecture; a specific role can be attached to different realizations of different types to build different systems: what makes up the system depends on how you attach a *realization* of a specific *type* to a specific *role*. The use of roles to deal with variability allows one to flexibly design and document different configurations from the same architecture.

8.6.3.1 Design Rationale

- Interfaces shown in Figure 8.4 are potential candidates for defining ports of the components at configuration time.
- In figure 8.10, the architecture was left unbound to the design decision on the exact description of the duct between the Connectionpoint, Alarm, and the unnamed Classifier role, SecurityInformationSystem. The decision is left to the good judgment of lower-level designers and implementers, as the sis (SecurityInformationSystem) port is missing.

8.6.3.2 Analysis Results

- Configuration is a system-level concern that can be designed along an architectural dimension, called *configurability*. However, as shown in figure 8.11 and figure 8.12,

useful analyses of the configurability dimension depend on several other architectural dimensions, including *simplicity, complexity, variability* and *dynamism*.
- While characterizing the variation points, section 8.6.2 evaluates four architectural dimensions against each other: *simplicity* versus *complexity*, and *variability* versus *dynamism*.

8.6.3.3 Assumptions

At realization time, the interfaces CamConfiguror, CamController, shown in figure 8.4, will be provided by the CameraDevice component type and used by the SurveillanceStation component type. The interface Alarm, in contrast, will be provided by the SurveillanceStation and used by the system itself. Each of these interfaces can be seen as a specification of a static port with the same name that provides a particular view of the service component type.

8.6.4 Other Information

None.

8.6.5 Related View Packets

None.

8.7 Final Remarks

The approach presented in this chapter is similar to the IEEE-Std-1471; however, it does not address the concerns as first-class elements that determine the way large-scale software systems should be designed. In addition, though it provides good support for architecture description languages, it does not support expressing crosscutting concerns in software architecture. Moreover, it lacks mechanisms for building systems from an architecture concern-space.

A lesson learned from this case study is that a software architecture is a design solution, but not every design solution is a software architecture. Developing a software architecture requires decision making and trade-offs that affect and guide the tasks of developers at different stages in the software life cycle. The decisions made and the trade-offs have a positive impact on some aspects of the software development problem at hand, but they negatively affect other aspects. Consequently, the question of making the right decision during an architecting process depends primarily on the concerns—or aspects of the problem—of importance.

Here again, this demonstrates the need to complement ADL-based approaches with concern-oriented modeling techniques. Relying solely on the use of ADLs to solve architec-

tural problems makes one think of a situation in which someone tries to develop a solution for a given problem without knowing exactly what the different aspects are of the problem to be solved, which aspects of the problem are of importance, and how the different aspects affect each other.

Of course, lacking knowledge about key aspects of a problem, one can provide perhaps solutions that work somehow for the given problem, but such solutions are certain not to be the ones a customer is willing to pay for; they may be applicable to some other problems, but perhaps not to the one for which we are being asked to find a solution.

Part III

Other Related Work and Conclusions

Chapter 9:
Integrating the Structural PCS with SADL

This chapter presents the case of a compiler architecture to validate the integration of ConcernBASE, an early structural PCS, with SADL, which is a software architecture description language based on architectural refinement.

9.1 Introduction

This chapter introduces the ConcernBASE approach to software architecture description. The development of this approach was influenced by our experience that most ADLs provide support for describing structural aspects of software architecture. Thus, ConcernBASE defines first a structural viewpoint that supports the key concepts of ADLs. To validate this viewpoint, we discuss how to map a ConcernBASE structural description of software architecture, written in UML, onto an architectural description developed in a particular ADL, called SADL (Structural Architecture Description Language)[MR97]. The mapping to SADL was motivated on one hand by its explicit focus on structural aspects of system descriptions, and on the other hand by the verification capabilities of SADL tools for ConcernBASE. The mapping has been validated in ConcernBASE Modeler, a UML-based tool prototype that supports the ConcernBASE approach and its integration with SADL tools [CKS+01].

9.2 Structural Viewpoint

This section presents an example that illustrates the benefits of the ConcernBASE approach by applying its techniques to a well known compiler example. Figure 9.1 depicts an informal representation of a Level-3 Compiler architecture taken from [MR97][MR97], which uses the reference model for compiler construction.

Despite the box-and-arrow architecture representation, figure 9.1 shows that the compiler has a batch-sequential architectural style. The Main component coordinates the correct execution sequence of the components composing the compiler system. First, it transfers the control to the LexicalAnalyzer, then to the Parser, then to the AnalyzerOptimizer, and finally, to the CodeGenerator. The rounded-edge components, SymbolTable and Tree, are shared-memory components. The former holds binding information and makes it available to the LexicalAnalyzer and AnalyzerOptimizer. The latter keeps abstract syntax trees and is accessed by the Parser, AnalyzerOptimizer and CodeGenerator. Note that some components have read

and write access, while others are only granted read or write access. The Parser component is directly receiving tokens from the LexicalAnalyzer via the unidirectional pipe relating them and not through shared-memory components.

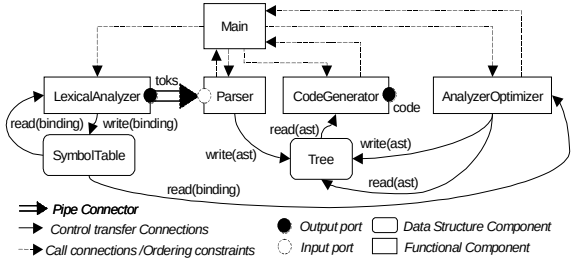


Figure 9.1: Compiler Architecture: taken from [MR97]

9.2.1 Structural View of the Compiler System

Figure 9.2 depicts the set of significant architectural elements that define the structural view of the compiler system. It contains six components: LexicalAnalyzer, Parser, AnalyzerOptimizer, CodeGenerator, SymbolTable and Tree, which are all connected via a complex connector, named CompilerConnector. As shown below, the connector plays a central role in this example. It mediates different kinds of communications between the components of the system and encapsulates all the communication paths. The CompilerConnector also coordinates the interactions among participant components. Therefore, it may enforce a particular communication protocol among the components.

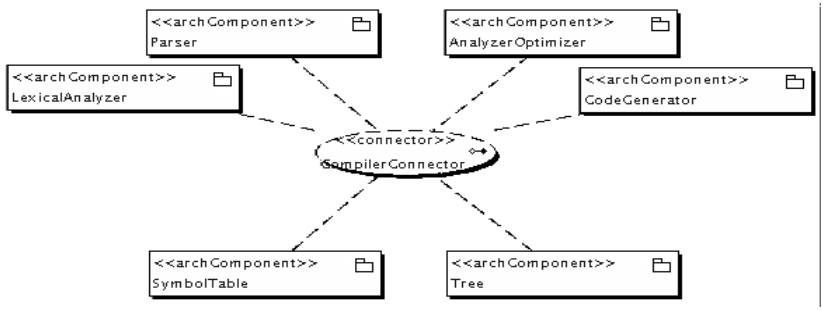


Figure 9.2: Structural View of the Compiler System

9.2.1.1 Static Model

The static structure of the LexicalAnalyzer component is shown in figure 9.3. Its component interface is composed of five interface elements, where each element defines a logical interaction point between the component and its environment. The ExecutionControl interface element provides the operation start with the meaning that another component can activate the LexicalAnalyzer, i.e. can start it by implementing this interface. The MemoryAccessControl interface element requires two operations: read and write. This means that the LexicalAnalyzer requires these operations to be provided by another component. The ControlFlowSignaling interface element declares incoming and outgoing signals necessary to control the execution of the LexicalAnalyzer, while the MemoryFlowSignaling interface element enumerates signals needed for communication with the shared-memory components.

Lastly, the Dataflow interface element defines two streams produced by the LexicalAnalyzer, namely a stream of tokens and a stream of bindings, as well as two consumed streams conveying characters and bindings. It is important to remark that bindings can be produced and consumed by the component. For example, as shown in figure 9.3, the LexicalAnalyzer component reads and writes bindings, i.e. produces and consumes them. All the interface elements shown in figure 9.3 are involved in a composition relationship with the LexicalAnalyzer component that realizes them. The interface elements are externally visible parts of the component.

The use of communication-specific interface elements clearly exhibits separation of concerns when defining specialized interaction points (referred to as static ports in the configuration model), since each interface element type is responsible for a particular communication type.

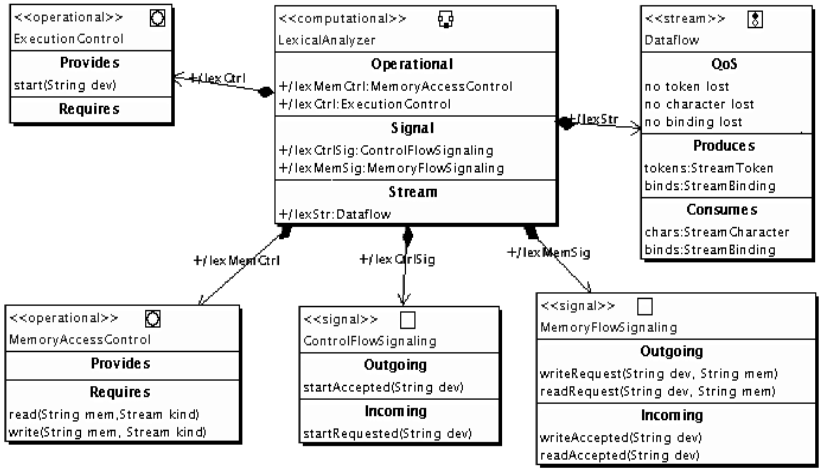


Figure 9.3: Static Structure Model of the LexicalAnalyzer Components Type

9.2.1.2 Configuration Model

The configuration model of the compiler system is illustrated in figure 9.4, which consists of an instance of the LexicalAnalyzer component type, an instance of the Parser component type and three simple connector instances that interconnect the ports and mediate the communication between the components. Instantiating a component type means to instantiate all its interface elements that are required in the configuration model. In figure 9.4, one connector instance is depicted to interconnect the conjugated <<operational>> static ports, named ExecutionControl and ExecutionControl~ together; a second one is shown to link the <<stream>> Dataflow and Dataflow~ ports; and finally, a third connector is used to interconnect the <<signal>> ControlFlowSignaling and ControlFlowSignaling~ ports.

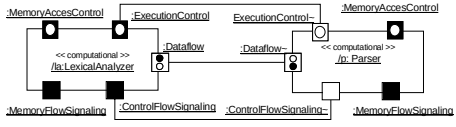


Figure 9.4: Configuration Model of the Compiler System

9.2.2 Overview of SADL

This section gives a brief introduction to the concepts of the Structural Architecture Description Language (SADL). SADL is a particular ADL that focuses on understanding, specifying and refining the representation of structural concerns in complex software systems. SADL is different from other ADLs, such as Wright [ABV92], as it supports structural decomposition at multiple levels. This is called refinement of high-level system structures in the SADL terminology. However, SADL is only capable of providing support for structural decomposition along a limited number of dimensions (e.g., components, connectors, configurations). The SADL support for behavioral modeling is very restricted.

Figure 9.5 shows a portion of the architecture description of the compiler_L1 example in SADL. The topmost section of an SADL architectural description declares the imported and exported architectural elements. This is achieved by using the keyword IMPORTING, indicating where the definitions can be found. In our example, IMPORTING Function FROM Functional_Style tells us that the Function construct is imported from an SADL style named Functional_Style.

The next section, called ARCHITECTURE, encloses further lower-level SADL sections. We can see that an architecture section is referenced by the identifier compiler_L1. The architecture description given after the ARCHITECTURE keyword includes data exchanged with its environment using input and output ports. The compiler_L1 has an input port, named chars_iport, and an output port, called code_oport. chars_iport receives a sequence of characters (Finite_Stream(character)), and code_oport sends code data.

An SADL architecture description contains three different sections dealing with various aspects of its software architecture, namely COMPONENTS, CONNECTORS and CONFIGURATION. The first and the second sections contain the declaration of the components and connectors, respectively, whereas the third section describes constraints on the configuration of the architectural elements defined in the first and second sections.

The COMPONENTS section contains mainly elements like ARCHITECTURE, Function, Variable and Operation. In SADL, all of these elements are considered as being components. The ARCHITECTURE section allows us to define sub-architectures that can be contained in a higher-level architecture. For instance, in figure 9.5, lexicalAnalyzerModule is a sub-architecture contained in the compiler_L1 architecture. Note that through this feature, SADL provides support for modularization.

Functionality of architectures can be expressed through the definition of Function components. As an architecture element, a Function component may have input and output ports through which data can be received or sent. In figure 9.5, the sub-architecture lexicalAnalyzerModule contains a function called lexicalAnalyzer representing the main functionality of the sub-architecture.

In SADL, Operation and Function components have similar meanings. The difference between them lies in the fact that the input ports of an Operation are seen as the parameters

and the output port as the return value of the operation. However, the number of output ports of an Operation component is limited to one.

Variable components are used to hold different types of data and make them available to other components in the sense of shared-memory, which is local to a sub-architecture. One Variable component is able to keep only a single type of data, which means that we need different Variable components for different types of data. For instance, the lexicalAnalyzer-Module contains three different Variable components (character-, token- and bindingVariable), the only three that are used by the sub-architecture.

The CONNECTORS section contains the definitions of different kinds of connectors, e.g., Pipe. Connectors enable communication among components. A Pipe connector carries data from an output port of one component to an input port of another. The transmitted data must be of the same type supported by the related output and input ports. An Enabling_Signal connector mediates signal communication that is likely to occur between two components.

```
IMPORTING Function FROM Functional_Style
...
compiler_L1 : ARCHITECTURE [ chars_iport : Finite_Stream(Character) -> code_oport : Finite_Stream(code)]
BEGIN
  COMPONENTS
    lexicalAnalyzerModule : ARCHITECTURE
       [chars_iport : Finite_Stream(Token), bind_iport: Finite_Stream(Binding) ->
        bind_oport: Finite_Stream(Binding), token_oport : Finite_Stream(Token)]
    BEGIN
       COMPONENTS
          lexicalAnalyzer : Function
                [chars_iport : Finite_Stream(Token), bind_iport: Finite_Stream(Binding) ->
                 bind_oport: Finite_Stream(Binding), token_oport : Finite_Stream(Token)]
          characterVariable : Variable(Character)
          tokenVariable : Variable(Token)
          bindingVariable : Variable(Binding)
       CONNECTORS
          ...
       CONFIGURATION
          token_read  : CONSTRAINT = Reads(lexicalAnalyzer, tokenVariable)
          token_write : CONSTRAINT = Writes(lexicalAnalyzer, tokenVariable)
          ...
    END
    ...
  CONNECTORS
    tokenPipe : Pipe[Finite_Stream(Token)]
    ...
  CONFIGURATION
    tokenFlow : CONNECTION = Connects(tokenPipe,lexicalAnalyzerModule!token_oport,parserModule!token_iport)
    ...
END
```

Figure 9.5: Extract of the Level-3 Compiler SADL Specification

The CONFIGURATION section defines the configuration constraints on the previously described components and connectors. These constraints may state, for instance, which Function or Operation component has read/write access to a Variable component, which component sends a signal, which component receives it, the direction of the data flow between two components, and from which component an Operation is called. We use two different types of statements, namely CONNECTION and CONSTRAINT (examples of constraints are shown at the bottom of figure 9.8 and figure 9.9). The former defines data flow connections and the latter specifies all other kinds of constraints.

9.2.3 Mapping ConcernBASE to SADL

This section presents our approach for translating a ConcernBASE architectural description written in UML into a textual form written in SADL.

The mapping consists of 5 steps. The first step identifies all data types utilized in the ConcernBASE architectural description and maps them to SADL. The second step requires that all the architectural components be found and mapped to SADL. The third step requires that all the interface elements of each architectural component be found and mapped to SADL. The fourth step identifies data flow connections and maps them to SADL. And finally, the fifth step puts the pieces together.

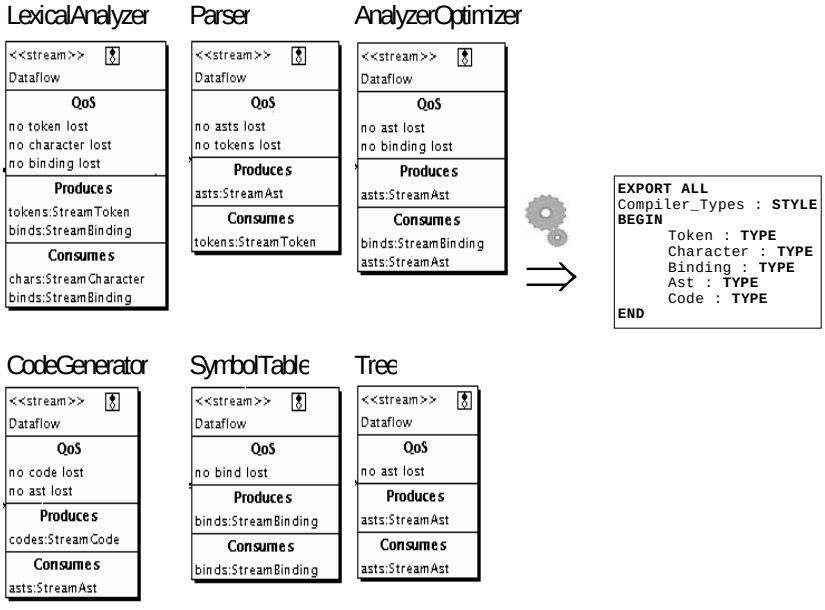


Figure 9.6: Compiler_Types.sadl

9.2.3.1 Mapping Data Types

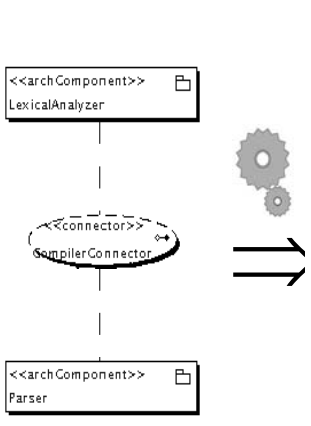
To perform this task, we use an SADL feature that allows SADL styles to be defined anytime [MR97]. Figure 9.6 shows an SADL style which defines the data types used in the level-3 compiler (see section 9.2.2).

Basically, we define a new style that consists of all data types contained in the current architectural description. To do this, we have to look at every stream interface in the static model of all the components and connectors. Then we build up the data types list by gathering every data type supported by the different streams. Then we simply define a new style having the name of the current architecture appended with the suffix Types in a file having the name of the style with the extension ".sadl".

9.2.3.2 Mapping Architectural Components

Before mapping ConcernBASE components to SADL, we look at the structural view and identify all the architectural components that are contained in the system.

We translate every architectural component (subsystem) as an SADL sub-architecture with the suffix Module and declare it in the COMPONENTS section of the main architecture. We then declare a Function component with the same name as the component and the same input and output ports. The Function represents the main functionality of the sub-architecture and will be referred to as the sub-architecture's main component. However, this mapping strategy does not exclude that other UML artifacts (for instance, high-level connectors) can be modeled as components. Such artifacts will be discovered during the next steps. Figure 9.7 shows how the structural view is translated into SADL.




```
...
COMPONENTS
    lexicalAnalyzerModule : ARCHITECTURE
        [ ... -> ... ]
    BEGIN
        COMPONENTS
            lexicalAnalyzer : Function
                [ ... -> ... ]
    END

    parserModule : ARCHITECTURE
        [ ... -> ... ]
    BEGIN
        COMPONENTS
            parser : Function
                [ ... -> ... ]
    END
CONNECTORS
...
```

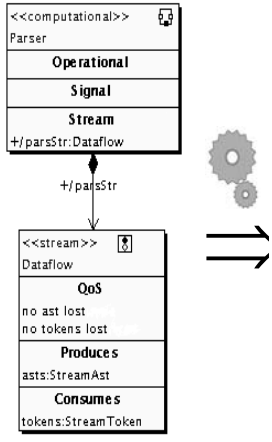
Figure 9.7: Translating Architectural Components

9.2.3.3 Mapping Component Interfaces

To translate the component interface, we have to look at its static model. The component interface is composed of three different interface element types, each of which supports a different communication pattern.

Stream Interface Type

Clearly, the <<stream>> interface element type is the easiest type to map, since it is equivalent to an SADL port. A stream interface element may produce and consume different kinds of streams, e.g., video and audio streams. Each stream declared in the Produces and Consumes compartments is translated into an output and an input port of the component, respectively. Figure 9.8 illustrates this idea.




```
...
COMPONENTS

      parserModule : ARCHITECTURE
          [tokens_iport : Finite_Stream(Token)->
           asts_oport : Finite_Stream(Ast)]
      BEGIN
          COMPONENTS
             parser : Function
               [tokens_iport : Finite_Stream(Token)->
                asts_oport : Finite_Stream(Ast)]

             tokenVariable : Variable(Token)
             astVariable : Variable(Ast)
          CONFIGURATION
             ...Reads(parser,tokenVariable)
             ...Writes(parser,tokenVariable)
             ...Reads(parser,astVariable)
             ...Writes(parser,astVariable)
      END
CONNECTORS
...
```

Figure 9.8: Translating Stream Interface Type

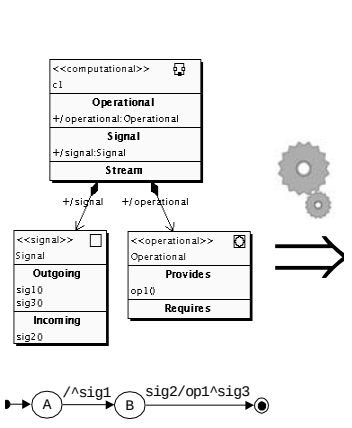
Also, we declare a Variable component in the COMPONENTS section of the sub-architecture for every different type of stream. A Variable component simply holds the data and acts as a shared-memory component within the sub-architecture. Moreover, it should only be accessed by internal components of the sub-architecture that owns it, using Reads/Writes predicates. These are configuration constraints that need to be specified in the sub-architecture itself. The reason for doing so is to differentiate between functional and data-holding concerns of components. Thus, all data consumed by a component is stored within an inter-

nal Variable component (that belongs to the sub-architecture) that deals with the corresponding data type.

Operational and Signal Interface Types

SADL lacks precise formalism for the definition of operational connectors, i.e. connectors that mediate operation calls between two components. However, the SADL style, Procedural_Style, contains the definition of the Called_From predicate taking the invoked Operation and the calling COMPONENT as parameters. For instance, Called_From(B!start,A) means that the component A calls the operation start implemented by component B. Note that start is declared as an Operation in the COMPONENTS section of the sub-architecture B.

The Outgoing compartments of the <<signal>> interfaces of a component allow us to identify the set of signals defined by that component. We therefore declare the signals in the CONNECTORS section of the sub-architecture representing the architectural component. To retain their behavior, we have to translate the ordering constraints on the signals. To do this we analyze the behavioral model, which provides all the information we need to get the correct sequencing of signals.




```
...
COMPONENTS
        c1Module : ARCHITECTURE
              [ ... -> ... ]
        BEGIN
             COMPONENTS
                c1 : Function
                   [ ... -> ... ]
                op1 : Operation
                   [ ... -> ... ]
             CONNECTORS
                sig1, sig3 : Enabling_Signal
        END
CONNECTORS
CONFIGURATION
        ...Sender(c1Module!sig1,c1Module)
        ...Receiver(c1Module!sig1,c2Module)
        ...Sender(c2Module!sig2,c2Module)
        ...Receiver(c2Module!sig2,c1Module)
        ...Called_From(c1Module!op1,c1Module)
        ...Sender(c1Module!sig3,c1Module)
...
```

Figure 9.9: Translating Behavioral Aspects

Figure 9.9 shows the translation of the behavior of a component into SADL with respect to the mediation of signal and operational communication. The static model is helpful for identifying operations and signals, while the behavioral model helps discover the temporal ordering of signals and operation calls.

Furthermore, C1 sends the signal sig1 and enters state B. The component C2 (not shown in the figure) receives sig1 and immediately sends sig2, which is in turn received by C1. Upon reception of sig2, C1 calls the operation op1 and sends the signal sig3. The ordering is translated by means of SADL predicates (Sender, Receiver, Called_From) indicating the kind of relationships existing between the predicate's arguments. For instance, Sender(c1Module!sig1, c1Module) means that c1Module is the sender of the signal sig1. Outgoing signals are declared within the sub-architecture. The constraints that specify the correct sequencing of the signals are declared in the CONFIGURATION section of the main architecture.

Translating the behavior of connectors is another very important thing that has to be taken into account in order to retain the semantics of the source model. ConcernBASE and SADL differ on the fact that connectors may have behavior, too. We cannot specify the behavior of a connector in SADL. In section 9.2.3.2, we mentioned that we may have to create an additional SADL component to represent a ConcernBASE connector with behavior. For instance, in the level-3 compiler, the CompilerConnector is responsible for controlling the execution flow of the components that are part of the compiler system. In SADL, we would model this feature as a component that would transfer the control to each component in a sequential manner (see the main component in figure 9.1). This simply means that we create an SADL sub-architecture for each simple ConcernBASE connector that has behavior. To achieve this, we have to find all state machines of a connector that do not transfer signals and operation calls further. Such an SADL component, standing for a ConcernBASE connector, has no precise functionality and therefore does not own any internal component (Functions, Operation or Variable component). This new component is only responsible for transferring the control to other components, much like a main procedure calling other sub-procedures to delegate different sequential sub-tasks.

9.2.3.4 Mapping Connections

In the SADL formalism, a connection represents a data link between two components. It is further specified as being a CONNECTION constraint relating an output port of a component with an input port of another component via a data connector (e.g., a Pipe).

The identification of SADL ports has been shown under the heading Stream Interface Type on page 140. Now let us have a look at the interconnection between the ports that support data exchange among the components. This is described in the configuration model shown in figure 9.10. This figure illustrates the instantiation of a simple stream connector type between two components c1 and c2. The component c1 produces a finite stream of characters that are consumed by c2. The connector between the static ports (with the <<stream>> stereotype) is the carrier of the character stream. Both the connector and the connection are respectively declared in the CONNECTOR and the CONFIGURATION sections of the main architecture.

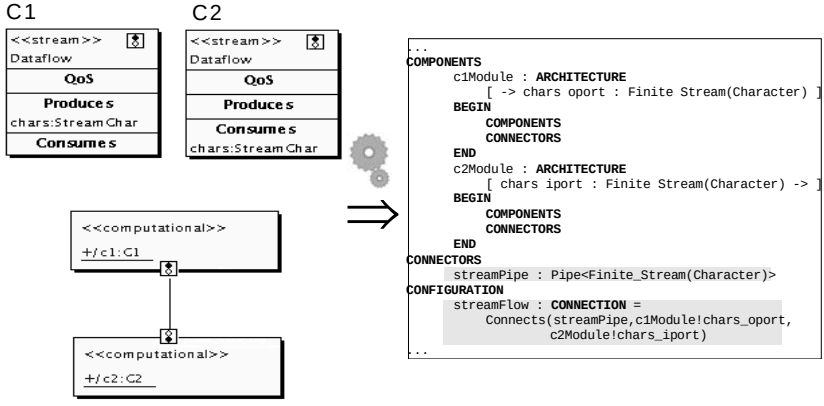


Figure 9.10: Translating Data Connections

9.2.3.5 Putting It All Together

The last task consists of composing all the partial descriptions that result from the previous steps, and adding IMPORTING and EXPORTING statements. An excerpt of the resulting description is shown in figure 9.11.

Conceptually, this figure corresponds to the illustration of the structural view as shown in figure 9.2. It represents a significant result that validates the ConcernBASE profile for structural descriptions, by allowing us to define architectural models in the ConcernBASE profile of UML and to translate them into SADL for analysis. Unfortunately, the translation from SADL descriptions back into ConcernBASE models is not supported yet.

```
IMPORTING Character,Binding,Ast,Token,Code FROM Compiler_Types
IMPORTING Function FROM Functional_Style
IMPORTING Operation,Called_From FROM Procedural_Style
IMPORTING Sender,Receiver,Before,Enabling_Signal FROM Control_Transfer_Style
IMPORTING Pipe,Finite_Stream FROM Process_Pipeline_Style
IMPORTING Variable,Reads,Writes FROM Shared_Memory_Style
compilerL3 : ARCHITECTURE [ ... -> ... ]
BEGIN
COMPONENTS
      lexicalAnalyzerModule : ARCHITECTURE   [ ... -> ... ]
      BEGIN
          COMPONENTS
              lexicalAnalyzer : Function [ ... -> ... ]
              start : Operation [ ... -> ... ]
              tokenVariable : Variable(Token)
              ...
      END
CONNECTORS
...
```

Figure 9.11: Putting Everything Together

9.3 Tool Support

The ConcernBASE Modeler is an integrated tool for developing architectural descriptions using the ConcernBASE approach. The tool allows one to translate UML architectural models into SADL descriptions, providing at the same time a new and elegant way to supply verification support for UML models using the existing SADL tools. Tool pro-activeness supports the developers in their modeling tasks because it actively handles the consistency between different overlapping models. For instance, when the user wants to instantiate a component type in the configuration model, the tool proposes a list of components that have already been defined in the structural view. When the user is modeling the behavior of architectural elements by means of state machines, the trigger and call event lists are populated with signals and operations that already exist, i.e. that have been defined in the corresponding interface elements. These features reduce erroneous editions and maintain consistency between different aspects of the same model. Figure 9.12 illustrates a usage view of the ConcernBASE Modeler tool.

The software is single-project based, which means that it only allows one architecture to be modeled at a given time. One project may contain several model files depicting the architecture. The structural view is shown as a high-level model that can be refined by defining more detailed models; each architectural element declared in the structural view has its own static model and behavioral model; the configuration structure is also defined as a separate model. All models are stored on disk using the standard XMI file format.

The graphical user interface is simple, ergonomic and intuitive. It has a menu bar that provides different options, a tool bar containing frequently-used functions, a left pane dis-

playing a structured view of the architecture, a right pane allowing one to graphically modify architectural diagrams, and a message pane keeping the user informed of what is going on within the system. The interface is completely event-driven and all resources, i.e. labels, texts, messages, images, etc., are internationalized; this means that the aspect of the interface can be changed and localized without having to rebuild the system. A complete built-in help system offers information on the system itself, its functionalities, and its application domain (ConcernBASE and SADL).

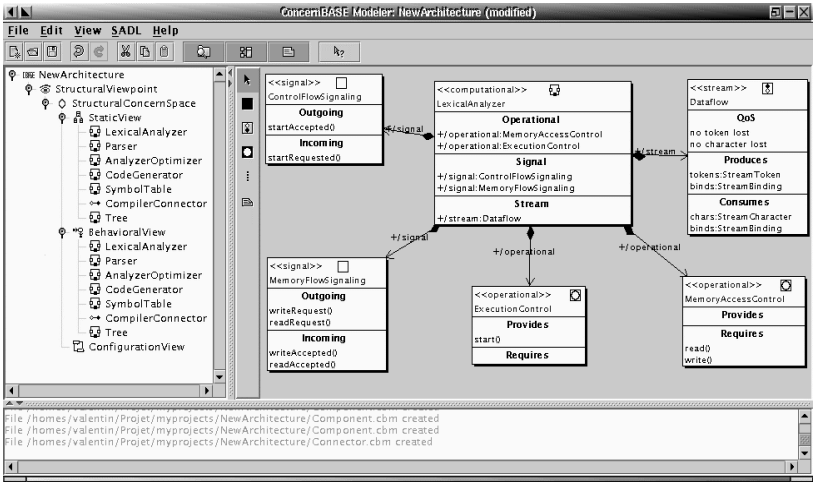


Figure 9.12: A Screenshot of the ConcernBASE Modeler

9.4 Final Remarks

In this chapter, we have proposed a particular way of establishing a bridge between an ADL, UML and the IEEE-Std-1471.

ADLs provide expressive notations that many architects would like to integrate with UML. A contribution of ConcernBASE was to facilitate such an integration by implementing two conceptual frameworks: the IEEE-Std-1471 and MDSOC.

The chapter described a method for translating ConcernBASE models using UML into SADL specifications. This translation enabled us to make use of SADL verification tools and integrate them with the ConcernBASE Modeler tool. Although the example of integration shown was based on SADL, the ConcernBASE approach is general and allows

one to define various viewpoints and viewpoint languages to represent different aspects of software architecture.

We distinguish consistency between the views and models (both described by model elements) from consistency between viewpoints (characterized by a set of concerns). Therefore, support for consistency between viewpoints could not be achieved. Also, it would be nice to integrate other ADLs and to have a backward mapping from SADL to UML, so that we do not have to learn SADL when using the ConcernBASE Modeler tool.

Finally, the ConcernBASE approach and the tool supporting it are both undergoing refinement and improvement, but they are already being applied in projects. Although the tool is not yet complete, one can already develop models, translate them to SADL, edit and syntax-check the resulting SADL descriptions and save the models to disk.

Chapter 10:
Conclusions

This chapter provides concluding remarks. It also presents some limitations of concern-oriented approaches to software architecture and the PCS Framework, and it indicates some directions in which research in the new area of concern-oriented software architecture could be pursued.

10.1 Summary

This thesis studied the feasibility and suitability of a concern-oriented approach for developing and describing architectures of software-intensive systems. It proposed improvements for separation of concerns and in the design, construction and evolution of such systems, and for integrating their architectural descriptions with modern software development artifacts.

10.2 Contributions

The thesis has argued for the necessity to comprehend that concerns are aspects of the problems we, the humans, consider when we start building software. These problems are outside of the computer (both hardware and software) [Jack01]. Therefore, the computer cannot distinguish one concern from another, and it cannot identify the relationships between different concerns. It is the job of software engineers to identify the concerns and their relationships, to reify them into both model elements and code, and to manipulate concerns via their representations.

The relationship between concerns and model elements is shown to be similar to the relationship between an architecture and an architecture description.

The dissertation presented two elements that makes up a concern: 1) its significance to stakeholders—that is, there must be a goal for a stakeholder—and 2) a characteristic that makes it appear as an aspect of a problem—it must characterize a problem to be solved. Putting these two elements together, it should be possible to formulate any concern in terms of a question that may be answered by an architectural design solution (or by a low-level design solution or an implementation).

Moreover, this thesis argues that building software architectures should be driven by the concerns of interest to the stakeholders, not by a specific artifact language or modeling tech-

nique. This is the very objective of the concern-oriented software architecture approach introduced as a general methodology for achieving architectural design by concerns.

This work introduced the PCS Framework as an approach to implement two conceptual frameworks, the IEEE-Std-1471, and MDSOC. The PCS Framework provides means for integrating these two frameworks by using the Unified Modeling Language (UML). It introduced the notion of building model slices, a new concern-oriented modeling technique, which aims at supporting the paradigm of architectural design by concerns. The thesis also proposes the concept of UML Space as a new mechanism for achieving multidimensional separation of concerns in UML, and for realizing the notion concern space as part of the PCS Framework to develop concern-oriented architectures.

The thesis also presented numerous case studies. It describes an On-Demand Remodularization PCS by defining a viewpoint language and using that viewpoint language to remodularize concerns in the Java Drag'n'Drop architecture. The On-Demand Remodularization PCS contributed the ODR pattern, which can be applied at both levels: architectural and low-level design.

Moreover, this work also demonstrated how to achieve software composition by focusing only on interactions—i.e., not on components that interact; thus, it introduced a new approach for rapidly building software systems by means of aspect-oriented connectors.

An example has been provided which shows how to create a UML Space for aspect-oriented modeling. This allows us to model interaction aspects for adapting independent components to a new environment. The connector modeling approach proposed in this thesis supports the expression of different aspects of software interactions in aspect-oriented models.

10.3 Advantages and Limitations

One of the limitations of this concern-oriented approach to software architecture is related to the composition of interaction aspects themselves. Further research is required to figure out more appropriate languages for supporting reification of concerns into model elements. Another limitation is related to the assignment of roles to components when applying the ODR pattern to an existing software system. The ability to assign roles to components is currently not supported. Though this assignment is similar to connecting aspects with classes, reasoning on composition of roles is still unexplored. This is also similar to the composition of aspects and requires further research.

The basic idea of this thesis is that software architectures need to be developed in a similar way as the software that implements an architecture. However, there are many directions in which research in this new area of concern-oriented software architecture could be pursued, including tools for UML Spaces, and new techniques for identifying and reifying

concerns into software at all levels of abstraction. Another research direction is concern-oriented software development, which includes concern-oriented patterns, concern-oriented web services, concern-oriented analysis, concern-oriented design, concern-oriented programming, concern-oriented testing, etc.

Part IV

Annexes

Annex A: Bibliography

[ABV92] M. Aksit, L. Bergmans, and S. Vural. *An object-oriented language-database integration model: The composition-filters approach*. In Proc. ECOOP, pp. 372-395, 1992. LNCS 615S.

[AEB03] O. Aldawud, T. Elrad and A. Bader. *A UML Profile for Aspect-Oriented Software Development*. Workshop on Aspect-Oriented Modeling with UML, AOSD'2003, Boston, USA. (http://lgl-www.epfl.ch/workshops/aosd2003/)..

[All97] R. Allen. *A Formal Approach to Software Architecture*. Ph.D. Thesis, Carnegie Mellon University, School of Computer Science, available as TR# CMU-CS-97-144, May (1997).

[AKH02] C. Atkinson, T. Kühne, and B. Henderson-Sellers. *Stereotypical Encounters of the Third Kind*. In J.-M. Jézéquel, H. Hussmann, S. Cook (Eds.): UML2002 (Dresden, Germany, October 2002), LNCS 2460, pp. 100-114.

[AOM03] AOM Website. http://lglwww.epfl.ch/workshops/aosd2003/links.html

[AOSD] AOSD Website: http://aosd.net

[BCK98] L. Bass, P. Clements, R. Kazman: *Software Architecture in Practice*. Addison-Wesley (1998).

[Beck99] K. Beck: *Extreme Programming Explained: Embrace Change*. Addison-Wesley (1999).

[BGJ99] S. Berner, M. Glinz, and S. Joos. *A Classification of Stereotypes for Object-Oriented Modeling Languages*. In R. France, B. Rumpe (Eds.): UML1999 (Fort Collins, Colorado, USA, October 1999), LNCS 1723, pp. 249-263.

[BRJ98] G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language User Manual*. Addison-Wesley, 1998.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley and Sons Ltd (1996).

[CHO+99] S. Clarke, W. Harrison, H. Ossher, P. Tarr. *Subject-Oriented Design - Towards Improved Alignment of Requirements, Design and Code*. in Proc. of OOPSLA '99 (Denver, CO, Nov. 1999), SIGPLAN Notices 34(10), 325-339.

[CW00] S. Clarke and R. J. Walker. *Composition Patterns: An Approach to Designing Reusable Aspects*. Proceedings of the International Conference on Software Engineering - ICSE'2001 (May 2001).

[CBB+02] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord and J. Stafford: *Documenting Software Architectures — Views and Beyond*. SEI Series in Software Engineering. Addison-Wesley (2002).

[CN02] P. Clements, L. Northrop. *Software Product Lines —Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley (2002).

[Cle96] P. Clements. *A Survey of Architecture Description Languages.* 8th International Workshop on Software Specification and Design, Germany, March, 1996.

[CKK02] P. Clements, R. Kazman, M. Klein: *Evaluating Software Architectures*.Addison-Wesley (2002).

[Con03] ConcernBASE: http://lglwww.epfl.ch/research/concernbase/index.html

[CKS+01] V. Crettaz, M. M. Kandé, S. Sendall and A. Strohmeier. *Integrating the ConcernBASE Approach with SADL*. UML 2001 - The Unified Modeling Language: Modeling Languages, Concepts and Tools, Fourth International Conference, Toronto, Canada, October 1-5, Martin Gogolla (Ed.), LNCS (Lecture Notes in Computer Science), no. 2185, Springer Verlag, 2001, pp. 166-181.

[Dij74] E. W. Dijkstra. *EWD447: On the role of scientific thought* (1974). Reproduced in "*Selected Writings on Computing: A Personal Perspective*", Springer Verlag 1982. SBN 0-387-90652-5.

[DR99] E. Di Nitto and D. Rosenblum. *Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures*. Proceedings of the International Conference on Software Engineering - ICSE'99 (May 1999).

[EAK+01] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. *Discussing Aspects of AOP.* Communications of the ACM 44(10), pp. 33–38, October 2001.

[Eng97] R. Englander. *Developing Java Beans*. O'Reilly (1997).

[FPLP99] P. Fradet, D. Le Metayer, and M. Perin. *Consistency Checking for Multiple View Software Architectures.* O. Nierstrasz, M. Lemoine (Eds.): ESEC/FSE '99, LNCS 1687, pp. 410-428, (1999). Springer-Verlag.

[GHV+95] E. Gamma, R. Helm, J. Vlissides, R. Johnson: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley (1995).

[Gar00] D. Garlan. *Software Architecture: A Roadmap*. In The Future of Software Engineering; Anthony Finkelstein (Ed). 22nd International Conference on Software Engineering, ICSE (2000).

[GK00] D. Garlan and A. Kompanek. *Reconciling the Needs of Architectural Description with Object-Modeling Notations*. In UML 2000 - The Unified Modeling Language: Advancing the Standard, Third International Conference, S. Kent and A. Evans (Ed.), York, UK, October 2-6, LNCS (Lecture Notes in Computer Science), no. 1939, Springer Verlag, 2000, pp. 498-512.

[GMW97] D. Garlan, R. T. Monroe and D. Wile. *ACME: An Architecture Description Interchange Language*. Proceedings of CASCON '97 (1997), pp. 169-183.

[GRF02] G. Georg, I. Ray and R. France. *Using Aspects to Design a Secure System*. In proceedings of ICECCS 2002, (2002).

[GH02] M. Gogolla and B. Henderson-Sellers. *Analysis of UML Stereotypes within the UML Metamodel*. In J.-M. Jézéquel, H. Hussmann, S. Cook (Eds.): UML2002 (Dresden, Germany, October 2002), LNCS 2460, pp. 84-99.

[GL03] J. D. Gradecki, N. Lesiecki. *Mastering AspectJ - Aspect-Oriented Programming in Java*. John Wiley Publishing, 2003.

[IEEE00] The Institute of Electrical and Electronics Engineers (IEEE) Standards Board. *Recommended Practice for Architectural Description of Software-Intensive Systems* (ANSI/IEEE-Std-1471). September 2000.

[ISO95] ISO/IEC 10746-1/2/3. *Reference Model for Open Distributed Processing - Part 1: Overview/Part2: Foundations/Part3: Archictecture*. ISO/IEC (1995).

[HO93] W. Harrison, H. Ossher. *Subject-Oriented Programming - A Critique of Pure Objects*. In Proc. of OOPSLA'93 ('Washington DC, Oct. 1993), SIGPLAN Notices 28(10), 411428

[HM00] S. Hermann and M. Mezini. *PIROL: A Case Study for Multidimensional Separation of Concerns in Software Engineering Environments*. In OOPSLA 2000 Proc. , (2002) pp.188-207.

[Hil01] R. Hilliard: *Viewpoint modeling*. ICSE Workshop on Describing Software Architecture with UML (2001).

[HJP+02] W.-M. Ho, J.-M. Jézéquel, F. Pennaneac'h, N. Plouzeau. *A toolkit for weaving aspect oriented UML designs*. In Proceedings of the 1st International Conference on Aspect-Oriented Software Development - AOSD'2002 (April 2002).

[HNS99] C. Hoffmeister, R. Nord, D. Soni: *Applied Software Architecture*. Addison-Wesley (1999).

[Hyp03] HyperJ web site. http://www.alphaworks.ibm.com/tech/hyperj

[Jack01] M. Jackson. Problem Fames - Analyzing and structuring software development problems. Addison-Wesley (2001).

[KCS+02] M. M. Kandé, V. Crettaz, A. Strohmeier, S. Sendall. *Bridging the gap between IEEE 1471, an architecture description language, and UML*; in Journal on Software and Systems Modeling, Springer-Verlag, ISSN: 1619-1366 (printed version), Volume 1 Issue 2 (2002); pp 113-129.

[KKS02] M. M. Kandé, J. Kienzle and A. Strohmeier. *From AOP to UML: Towards an Aspect-Oriented Architectural Modeling Approach*. Technical Report, EPFL, http://ic2.epfl.ch/publications/documents/IC_TECH_REPORT_200258.pdf

[KMP+98] M. M. Kandé, S. Mazaher, O. Prnjat, L. Sacks, and M. Wittig. *Applying UML to Design an Inter-Domain Service Management Application*. In proceedings of UML'98

[KS00a] M. M. Kande and A. Strohmeier. *Towards an UML Profile for Software Architecture Descriptions*. UML'2000 - The Unified

Modeling Language: Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, S. Kent, A. Evans, B. Selic (Ed.), LNCS (Lecture Notes in Computer Science), no. 1939, Springer Verlag, 2000, pp. 513-527.

[KS00b] M. M. Kande and A. Strohmeier. *On The Role of Multi-Dimensional Separation of Concerns in Software Architecture*. Position paper for the OOPSLA'2000 Workshop on Advanced Separation of Concerns. (Online at http://lglwww.epfl.ch/~kande/Publications/role-of-mdsoc-in-swa.pdf)

[KTM97] M. M. Kandé, S. Tai, M. Wittig. *On the Use of UML for ODP-Viewpoint Modeling*, OOPSLA'97 Workshop on Object-Oriented Technology for Service, System and Network Management, Atlanta, Georgia, U.S.A. (1997).

[KK03] M. Katara, S. Katz. *Architectural views of aspects*. Proc. of the 2nd international conference on Aspect-oriented software development - AOSD'2003 (March 03).

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, K. Kersten, J. Palm and W. G. Griswold. *An Overview of AspectJ*. in Proc. of ECOOP'01 (Budapest, Hungary, June 2001), LNCS 2072, 327-252.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-Oriented Programming*. In M. Aksit and S. Matsuoka (Eds.), 11th European Conference on Object–Oriented Programming (ECOOP '97), Jyváskylá, Finland, LNCS (Lecture Notes in Computer Science) no 1241, Springer Verlag, 1997, pp. 220 – 242.

[Kru95] P. B. Kruchten. *The 4+1 view model of architecture*. IEEE Software, 12(6):42-50, (1995).

[LEW+02] M. Loy, R. Eckstein, D. Wood, J. Elliott, B. Cole. *Java Swing*, 2nd Edition. O'Reilly Publishing (2002).

[LAK+95] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Vera, D. Bryan and W. Mann. *Specification and Analysis of System Architecture Using Rapide*. In IEEE Transactions on Software Engineering, 21(4):336-355, (April 1995).

[MDE+95] J. Magee, N. Dulay, S. Eisenbach and J. Kramer. *Specifying Distributed Software Architectures*. Proc. 5th European Software Engineering Conf. (ESEC 95).

[MT00] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, Vol. 26, No.1, January 2000, pp. 70-93.

[MMP00] N. Medvidovic, N. R. Mehta and S. Phadke. Towards a Taxonomy of Software Connectors. Proceedings of the International Conference on Software Engineering - ICSE'00 (May 2000), pp. 178-187.

[MPW92] R. Milner, J. Parrow, and D.Walker. *A calculus of mobile processes*. Journal of Information and Computation, 100:1-77, 1992.

[MR97] M. Moriconi and R. Riemenschneider. Introduction to SADL 1.0. SRI Computer Science Laboratory, *Technical Report SRI-CSL-97-01*, March 1997.

[Omg01] OMG Unified Modeling Language Revision Task Force. *OMG Unified Modeling Language Specification*. Version 1.4 draft (February 2001). http://www.omg.org.

[Omg03] http://www.omg.org/

[OKL02] J. Ovlinger, K. Lieberherr and D. Lorenz. Aspects and Modules Combined. Technical Report, NU-CCS-02-03, Northeastern University, March 2002.

[Par72] D. Parnas. *One the criteria to be used in decomposing systems into modules*. In Communications of the ACM, volume 15, pages 1053-1058, 1972.

[PW92] D. E. Perry and A. L. Wolf. *Foundations for the Study of Software Architecture.* ACM SIGSOFT Software Engineering Notes, 17(4), (1992).

[RMA03] A. Rashid, A. Moreira and J. Araújo. *Modularisation and composition of aspectual requirements*. Proc. of the 2nd international conference on Aspect-oriented software development - AOSD' 2003 (March 03).

[RMR+98] J. E. Robbins, N. Medvidovic, D. F. Redmiles and D. S. Rosenblum: *Integrating Architecture Description Languages with a Standard Design Method.* In Proceedings of the 20th International Conference on Software Engineering (ICSE'98), pp. 209-218, Kyoto, Japan, April 19-25 (1998).

[Ross78] D. T. Ross. *Structured Analysis (SA): a language for communicating ideas*. IEEE Transactions on Software Engineering, SE-3(1), January 1977. Also appears in Programming methodology : a collection of articles by members of IFIP WG2.3 edited by David Gries. New York : Springer-Verlag (1978).

[RJB98] J. Rumbaugh, I. Jacobson, G. Booch: *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[Sch02] S. R. Schach. *Object-Oriented and Classical Software Engineering*. Fith Edition. Mc Gray Hill (2002)

[Sch01] B. Schmerl. *xAcme: CMU Acme Extensions to xArch*. URL: http://www-2.cs.cmu.edu/~acme/pub/xAcme/guide.pdf.

[SSR+00] D. C. Schmidt, M. Stal, H. Rohnert and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*.Wiley and Sons, 2000.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hill (1996)

[SGW94] B. Selic, G. Gullekson and P. T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, (1994)

[SR98] B. Selic, J. Rumbaugh. *Using UML for Modeling Complex Real-Time Systems*. ObjecTime, (1998).

[SHU02] D. Stein, S. Hanenberg, and R. Unland. *A UML-based Aspect-Oriented Design Notation For AspectJ*. In Proceedings of the 1st International Conference on Aspect-Oriented Software Development - AOSD'2002 (April 2002).

[SR02] S. M. Sutton Jr. and I. Rouvellou. *Modeling of Software Concerns in Cosmos*. In Proceedings of the 1st International Conference on Aspect-Oriented Software Development - AOSD'2002 (April 2002).

[ST02] S. M. Sutton, Jr. and P. Tarr. *Aspect-Oriented Design Needs Concern Modeling*. Position paper in the Aspect-Oriented Design workshop in conjunction with AOSD 2002, Enschede, The Netherlands, April 2002.

[Szy98] C. Szyperski. *Component Software - Beyond Object-Orented Programming*. ACM Press, New York, Adisson-Wesley, 1998.

[Tai98] S. Tai. *Constructing Distributed Component Architectures in Continuous Software Engineering*. Wissenschaft und Technik Verlag, Berlin (1999)

[Tan92] A. S. Tannenbaum. Computer netzworks, 4/e. Prentice Hall, (2002).

[TO00] P. Tarr and H. Ossher. *Multi-Dimensional Separation of Concerns and The Hyperspace Approach.* In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer (January 2000)

[TOW+99] P.Tarr, H. Ossher, W. Harrison, and S. Sutton Jr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. Proceedings of the International Conference on Software Engineering - ICSE'99 (May 1999), pp. 107-119.

[TD97] Thayer, R,. Dorfman, M. (Eds.), *Software Requirements Engineering* (2nd Edition). IEEE Computer Society Press, 1997.

[US03] UML Space Website: http://www.umlspace.org

[WK98] J. Warmer, A. Kleppe. The Object Constraint Language: Precise Modeling With UML. Addison-Wesley (1998).

[Web97] *Random House Webster's College Dictionary*, 2nd Edition. Random House (1997).

[Wei00] O. Weigert (moderator). *Panel: Modeling of Architectures with UML*. In UML 2000 - The Unified Modeling Language: Advancing the Standard, Third International Conference, S. Kent and A. Evans (Ed.), York, UK, October 2-6, LNCS (Lecture Notes in Computer Science), no. 1939, Springer Verlag, 2000, pp. 513-527.

[WWW90] R. Wirfs-Brock, B. Wilkerson, L. Wiener: *Designing Object-Oriented Software*, Prentice Hall (1990).

Curriculum Vitae

Mohamed Mancona Kandé
Dipl. Ing. Technische Informatik (M.Sc. in Computer Engineering)

Education

1988	Baccalauréat “Sciences Mathématiques”.
1988 - 1990	Studies in Mathematics, Université de Conakry, Guinea.
1992 - 1998	Studies in Computer Engineering, Technische Universität Berlin, Germany.

Research and Development

1996 - 1998	Fraunhofer Institute for Open Communication Systems (FOKUS), Berlin, Germany.
Summer 2001	Visit to IBM Research, Thomas J. Watson Research Center, Advanced Enterprise Middleware Group, New York, USA
1998 - 2003	Research Assistant at the Software Engineering Lab, School of Computer and Communication Sciences, Swiss Federal Institute of Technology in Lausanne (EPFL).
1998 - 2003	Ph.D. thesis in the field of Software Architecture.
Jan. 2003 —	Project Manager for "Aspect-Oriented Software Design" (this project is supported by Soft[net], the Swiss National Innovation Program for Software Technology)
May 2003 —	Senior Researcher at the Department of Management of Technology and Entrepreneurship, EPFL

Teaching

1998 - 2003	Assistant responsible for the design phase of the Software Engineering Course given by the Software Engineering Lab., EPFL.

Supervision

2000 - 2003	Supervision of numerous semester and diploma projects in software engineering.