Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

A META-SEMANTIC LANGUAGE FOR SMART COMPONENT-ADAPTERS

by
Leon K. Jololian

The issues confronting the software development community today are significantly different from the problems it faced only a decade ago. Advances in software development tools and technologies during the last two decades have greatly enhanced the ability to leverage large amounts of software for creating new applications through the reuse of software libraries and application frameworks. The problems facing organizations today are increasingly focused around systems integration and the creation of information flows.

Software modeling based on the assembly of reusable components to support software development has not been successfully implemented on a wide scale. Several models for reusable software components have been suggested which primarily address the wiring-level connectivity problem. While this is considered necessary, it is not sufficient to support an automated process of component assembly. Two critical issues that remain unresolved are: (1) semantic modeling of components, and (2) deployment process that supports automated assembly. The first issue can be addressed through domain-based standardization that would make it possible for independent developers to produce interoperable components based on a common set of vocabulary and understanding of the problem domain. This is important not only for providing a semantic basis for developing components but also for the interoperability between systems. The second issue is important for two reasons: (a) eliminate the need for

developers to be involved in the final assembly of software components, and (b) provide a basis for the development process to be potentially driven by the user. To resolve the above remaining issues (1) and (2) a late binding mechanism between components based on meta-protocols is required. In this dissertation we address the above issues by proposing a generic framework for the development of software components and an interconnection language, COMPILE, for the specification of software systems from components. The computational model of the COMPILE language is based on late and dynamic binding of the components' control, data, and function properties. The use of asynchronous callbacks for method invocation allows control binding among components to be late and dynamic. Data exchanged between components is defined through the use of a meta- language that can describe the semantics of the information but without being bound to any specific programming language type representation. Late binding to functions is accomplished by maintaining domain-based semantics as component meta-information. This information allows clients of components to map generic requested service to specific functions.

A META-SEMANTIC LANGUAGE FOR SMART COMPONENT-ADAPTERS

by
Leon K. Jololian

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer and Information Science

May 2000

APPROVAL PAGE

A META-SEMANTIC LANGUAGE FOR SMART COMPONENT-ADAPTERS

Leon K. Jololian

Dr. Franz J. Kurfess, Dissertation Advisor — Date
Assistant Professor of Computer and Information Science, NJIT

Dr. Murat M. Tanik, Dissertation Advisor — Date
Professor of Electrical and Computer Engineering, UAB at Birmingham, Alabama

Dr. Daochuan Hung, Committee Member — Date
Associate Professor of Computer and Information Science, NJIT

Dr. Joseph Y. Leung, Committee Member — Date
Distinguished Professor of Computer and Information Science, NJIT

Dr. C.V. Ramamoorthy, Committee Member — Date
Professor Emeritus of Electrical and Computer Engineering, U.C. Berkeley

Dr. Donald H. Sebastian, Committee Member — Date
Professor of Industrial and Manufacturing Engineering, NJIT

Dr. Gary L. Thomas, Committee Member — Date
Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Leon K. Jololian

Degree: Doctor of Philosophy

Date: May 2000

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science
  Computer and Information Science Department,
  New Jersey Institute of Technology, Newark, NJ, 2000

- M.S. in Computer Science
  Department of Computer Science,
  Polytechnic University, Brooklyn, New York, 1993

- M.S. in Electrical Engineering
  Department of Electrical Engineering,
  Georgia Institute of Technology, Atlanta, Georgia, 1981

- B.E. in Electrical Engineering
  Department of Electrical Engineering
  Manhattan College, Riverdale, New York, 1980

Major: Computer Science

Publications:

Papers

Caudill, R.J., Jololian, L., and Dogru, A.H., "Design for Environment Tools: Current Status and Beyond," Proceedings of the Third Biennial World Conference on Integrated Design and Process Technology, Berlin, Germany, Vol. 3, July 5-9, 1998, pp. 74-77.

Jololian, L., Kurfess, F., and Healey, M., "A Component-Based Environmental Knowledge Network," To appear in the Proceedings of the Fifth Biennial World

Conference on Integrated Design and Process Technology, North Dallas/Addison, Texas, June 4-8, 2000.

Jololian, L., Tanik, M., and Kurfess, F., M., "Intelligent Integration of Enterprise Software," To appear in the Proceedings of the Fifth Biennial World Conference on Integrated Design and Process Technology, North Dallas/Addison, Texas, June 4-8, 2000.

Jololian, L. and Smith, M.F., "Architectural Framework for Virtual Enterprises," Proceedings of the Third Biennial World Conference on Integrated Design and Process Technology, Berlin, Germany, Vol. 4, July 5-9, 1998, pp. 347-350.

Dogru, A., Tanik, M., Kurfess, M., Healey, M., and Jololian, L., "Green Manufacturing of Ammunition through Knowledge Management With Distributed Access," Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences, Maui, Hawaii, January 5-8, 1999 (CD-ROM), 6 pages.

Rossak W., Kirova V., Jololian L., Lawson H., Zemel T., "A Generic Model for the Use and Specification of Software Architectures," IEEE Software, Vol. 14, No. 4, July/August 1997, IEEE Computer Society Press, Los Alamitos CA, pp.84-92.

Kirova V., Rossak, W., and Jololian, L., "Software Architectures for Mega-System Development – Basic Concepts and Possible Specification", Proc. of the IEEE Third International Conference on Systems Integration, Sao Paolo City, Brazil, August, 1994, pp. 38-45.

Rossak, W., Zemel, T., Kirova, V., Jololian, L., "A Two-Level Process Model for Integrated System Development," Proc. of Symposium and Workshop on Systems Engineering of Computer Based Systems, IEEE Computer Society Press, Los Alamitos, CA, May 1994, pp. 90-96.

Workshop

Electronic Infrastructure for Virtual Organizations, 9th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-97), Electronic Virtual University Workshop, Newport Beach, CA, Nov. 4-7, 1997

Tutorial

Rossak, W., Kirova, V., and Jololian, L., "Systems Integration," invited presentation and seminar for INTAN (The National Academy of Civil Servants) – System Analysts Assoc., Kuala Lumpur, Malaysia, December 1995.

Jololian, L., "From Object-Oriented to Component-Based Development," The Fifth Biennial World Conference on Integrated Design and Process Technology, North Dallas/Addison, Texas, June 4-8, 2000.

To Cecilia, Taleen, and Lory
Whose love I will always cherish

ACKNOWLEDGEMENT

I offer my deepest gratitude to all who have helped me reach my goal. It hasn't been easy and without their help it wouldn't have been possible.

I thank Dr. Tanik for being a constant source of encouragement. He always believed in me and never wavered in his support of my work or me. He has been a mentor and a friend. I also thank Dr. Kurfess for being patient in listening to my ideas and providing me with useful feedback. I am fortunate to have him as my co-advisor.

Special thanks to all the committee members, Dr. Hung, Dr. Leung, Dr. Ramamoorthy, Dr. Sebastian, and Dr. Thomas. Their contribution is greatly appreciated. I thank Dr. Rossak for his support throughout the years and for getting me first interested in the area of software architecture.

I thank Dr. Ng for giving me the opportunity to try things I did not know I was capable of. His trust in my abilities has been inspirational. I also thank Dr. McHugh for his support and understanding. His advice and wisdom will always be remembered.

I thank Dr. Perl and his research group for the strong criticism in an early presentation. It provided an impetus for the improved exposition of my work. I also thank Drs. Scherl, Gerbesiotis, and Oria for their thoughtful comments on my presentation.

I offer my gratitude to Dr. Dogru for his friendship, encouragement, and support. I thank Surasit Nithikasem for the many long discussions we had and for the valuable input and help he has provided me.

I thank Dr. Rana, Dr. Kirova, and Mojgan Mohtashami for their encouragement and friendship. I am fortunate to have Vahid Moghaddasi as a friend. He has always

been generous with his time and in sharing his expert knowledge of systems administration.

I am indebted to the CIS staff, Carole Poth, Barbara Harris, Michelle Craddock, and Rosemarie Giannetta, for their wonderful support. I am also thankful to all the students who have taken my courses; teaching has been a valuable learning tool. I owe them my deepest gratitude.

Last but not least I thank all my family members for their love and understanding: My wife Cecilia whose unselfishness and dedication has allowed me to complete this journey, my parents Juliette and Krikor who have been my first and greatest teachers, my daughters Taleen and Lory for brightening my life, my sisters, brothers, and their spouses Gloria, Haig, Mary, Pauline, Berge, Ohannes, Stanley, and Harry for their unconditional support, my nephews and nieces, Vahe, Armen, Gregoire, George, Arpi, Angelina, and Irene for adding joy to my life.

If there are individuals whom I have not mentioned it is only due to my negligence. I offer them my sincerest apologies.

TABLE OF CONTENTS

Chapter **Page**

1 INTRODUCTION 1

1.1 Component-Based Software Development 2

1.2 The Difficulties in Component-Based Development 4

1.3 Component Adapters 5

1.4 A Component-Interconnection Language 7

1.5 Thesis Organization 8

1.6 Related Issues 9

1.6.1 Semantic Gap in Programming Languages 10

1.6.2 Limitations in Development Methodologies 10

2 RELATED WORK 12

2.1 Reuse Artifacts 12

2.2 Software Reusability 14

2.1.1 Three Levels of Reuse 14

2.1.2 Abstraction and Separation of Concerns 16

2.2 Domain Analysis 17

2.3 Design and Architecture 19

2.3.1 Design Patterns 19

2.3.2 Object-Oriented Frameworks 22

2.3.3 Domain Specific Software Architecture 24

Chapter **Page**

2.3.4 Megaprogramming 27

2.4 Infrastructure 30

2.4.1 Middleware 30

2.4.2 TAFIM 34

3 A COMPONENT-BASED SOFTWARE DEVELOPMENT FRAMEWORK 37

3.1 Programming with Components 37

3.1.1 Component Model 37

3.1.2 Mediating Component Mismatch 39

3.1.3 Simple and Composite Adapters 41

3.1.4 Generating Adapters 41

3.2 Component Binding Time 43

3.2.1 Virtual Functions and Polymorphism 44

3.2.2 Event-Based Programming 44

3.2.3 Late Binding with Components 45

3.3 Software Deployment Environments 45

3.3.1 Automation in Component Deployment 45

3.3.2 Processes to Automate 46

3.4 Factors in Component Interaction 47

3.4.1 Binding Control 47

3.4.2 Binding Data 48

3.4.3 Binding Service 49

Chapter **Page**

3.5 Meta-Semantics Programming 50

3.5.1 Function Meta-Semantics 50

3.5.2 Data Meta-Semantics 51

3.5.3 Control Meta-Semantics 52

3.6 Language Specification 53

3.6.1 Elements of the Language 53

3.6.2 The Semantics of Clusters 54

3.6.3 Adapter Manager 56

3.6.4 Event Management in Adapters 57

3.7 Generating Adapters from Abstract Specifications 59

3.7.1 Standardizing Event Structures 59

3.7.2 Standardization of Components 60

3.7.3 Support for Higher-Level Processes 61

4 A MODEL FOR SOFTWARE COMPONENTS 63

4.1 The Object-Oriented Model 63

4.1.1 A Unifying Model with Objects 64

4.1.2 Reusability of Objects 65

4.1.3 Object Support for Good Design 66

4.1.4 Design Limitations with Objects 67

4.2 The Software Component Paradigm 69

4.2.1 The Role of Components in Design 70

Chapter **Page**

4.2.2 A Model for Components 72

4.2.3 Design Elements of Components 73

4.2 Summary 76

5 CLUSTERS 77

5.1 Component Representation of Channels 77

5.1.1 The Push/Pull Model 77

5.1.2 Channel-Based Communication 78

5.1.3 Extension of the Channel Model 79

5.1.4 A Component Representation of a Channel 82

5.2 Design Environment 85

5.3 Properties of Clusters 88

5.3.1 Centralized vs. Distributed Control 88

5.3.2 Race Conditions 89

5.3.3 Deadlocks 90

5.3.4 Dynamic Configuration Management Control 91

5.4 Design Pattern Components 92

5.4.1 Language Support for Design Artifacts 92

5.4.2 Design Patterns for Components 93

5.4.3 Component Representation for Designs 93

5.4.4 The Structure of Design Pattern Components 94

5.5 Implementation of Selected Design Patterns 96

Chapter **Page**

5.5.1 The Adapter Pattern 96

5.5.2 The Facade Pattern 97

5.5.3 The Decorator Pattern 98

6 COMPILE: A COMPONENT INTERCONNECTION LANGUAGE 100

6.1 Elements of the Language 100

6.1.1 Components 101

6.1.2 Clusters 102

6.1.3 Design Patterns 103

6.2 Properties of the Language 106

6.2.1 Identifiers, Scoping Rules, and Control Structures 106

7 A COMPILER EXAMPLE 108

7.1 The Multiphase Compiler 108

7.1.1 A Multiphase Compiler Model 108

7.2 A Domain Specification for Compilers 109

7.2.1 Functional Model 110

7.2.2 The Data Model 112

7.2.3 The Control Model 116

7.2.4 Combining the Three Models 118

7.3 Specifying the Compiler 118

7.3.1 Specification of a Component 120

7.3.2 Specification of a Cluster 121

Chapter **Page**

7.3.3 Specification of Cluster Hierarchies 123

8 CONCLUDING REMARKS AND FUTURE WORK 125

8.1 Summary 125

8.1.1 Lessons Learned 125

8.1.2 Applying What We Learned 126

8.1.3 Results Achieved 127

8.2 Future Work 130

8.3 Contribution 131

APPENDIX A CONTEXT-FREE GRAMMAR REPRESENTATION 132

APPENDIX B THE COMPILE LANGUAGE 133

APPENDIX C CASE STUDY THE COMPILER EXAMPLE 136

APPENDIX D SPECIFICATION OF THE COMPILER IN COMPILE 139

REFERENCES 142

LIST OF TABLES

Table **Page**

Table 1 Traditional Versus Component-Based Development 3

Table 2 Classifications of Design Patterns (Gamma, 1995) 21

Table 3 Binding time of some object-oriented Features in C++ 43

Table 4 Component Functionality 110

Table 5 Control Table 116

Table 6 The Combined Model 119

LIST OF FIGURES

Figure **Page**

1.1 System Performance 4

2.1 Three Levels of Reuse 15

2.2 Tools in Support of Methodology and Process 17

2.3 Domain Analyses 25

2.4 Generic DoD Technical Reference Model (From TAFIM) 35

3.1 Adapter Between Components 40

3.2 Wrapping a Component 40

3.3 Cost of Developing Adapters 42

3.4 Adapter for unspecified event and handler 51

3.5 Adapters for Event Mismatch 52

3.6 Adapter for Control Mismatch 52

3.7 Run-time Activities 54

3.8 Registering with a Cluster Event 56

4.1 Forward Calling 75

4.2 Callbacks 76

5.1 Event-based Interaction 78

5.2 A Channel 79

5.3 Filtering and QoS on Channels 81

5.4 Interaction Diagram for a Channel 82

5.5 Interaction Diagram for the Observer Pattern 83

5.6 Channel Representation of the Model/View Controller Pattern 84

LIST OF FIGURES
(continued)

5.7 Interaction Diagram of the Model/View/Controller Pattern 85

5.8 Design Patterns Independently Defined 94

5.9 Combining Two Design Patterns 95

5.10 A Composite Component 95

5.11 Nesting of Design Patterns 96

5.12 The Adapter Pattern 97

5.13 The Façade Pattern 98

6.1 Specification of a Component 102

6.2 Specification of a Cluster 103

6.3 Specification of a Design Pattern 104

6.5 Platform Dependencies 107

6.4 Compiling the COMPILE Code 107

7.1 The Multiphase Compiler 109

7.2 Token Encoding 113

7.3 Syntactical Structure Encoding 114

7.4 Intermediate Code Encoding 115

7.5 Target Code Encoding 115

7.6 The Scanner Component Specification 120

7.7 The Parser Component Specification 121

7.8 Design Pattern Specification 122

7.9 A Cluster Specification 123

7.10 A Hierarchical Cluster 124

LIST OF FIGURES
(continued)

8.1 Separation of Control 128

8.2 Separation of Data 128

8.3 Separation of function 129

CHAPTER 1

INTRODUCTION

In order to meet the current and future challenges of software it is no longer possible to develop software by writing code one line at a time. The systematic use of software components is a necessary requirement if we are to go beyond incremental improvements in software development. Components are gradually becoming the units of reuse from which systems with varying requirements are built. However, there are two significant issues that must be resolved before components can become the basis of development. The first issue requires us to decide on a model for defining the functionality of individual components so that the assembly of a system can be based on the functionality of available components. The second issue requires us to address the problem of how to make components interoperable when mismatches between components along their interfaces are encountered. Standardization can play an important role in addressing both issues. However, it is expected that the usual forces that oppose industry-wide standardization in general will also be present here. If past experiences are any indication, it is likely that competitiveness in the software industry will result in more than one component standard, at least for the foreseeable future. Our approach is based on the following assumptions: (1) software development will be based on component assembly, (2) the Internet will increasingly become the infrastructure upon which the components of a system will be deployed and interoperate in a distributed environment, (3) end users will increasingly be driving and have control over the process of software development. In this dissertation we define a model for software components that can

support a development process based on the above assumptions. We provide a mechanism for automating the assembly of components by mediating mismatches between components' interfaces through the generation of "smart" adapters. Finally, we define and implement a high-level language to describe the meta-semantics of a system. Specifications in this language are compiled to produce components that first generate the necessary run-time adapters required by the components to interoperate, and second implement the wiring between components and adapters. Our work differs from others in several major aspects: we have adopted an existing component model, which we have augmented with semantic meta-information. By using a late binding of functionality, data, and control between interacting components we can generate customized component-adapters based on meta-information and the interfaces of components.

1.1 Component-Based Software Development

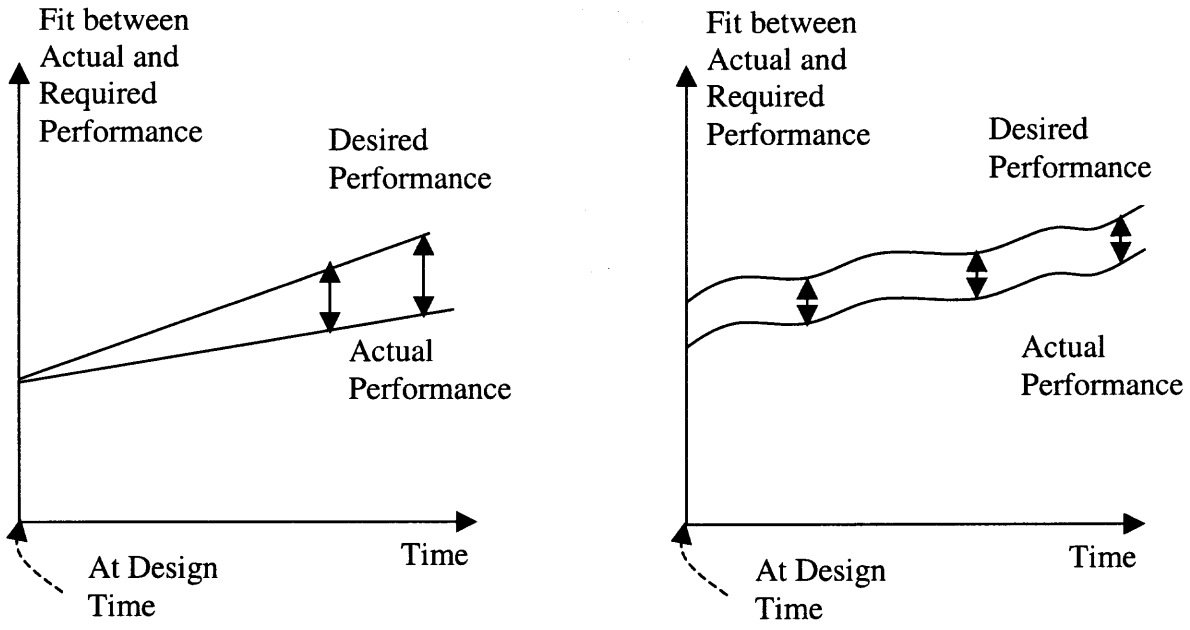
Developing software systems from components requires methodologies that are significantly different from the traditional software development approaches. Whereas the focus of software design has been on deciding which software modules to build to meet the users' requirements for a particular system, we now seek methodologies for developing software components that can later be assembled to satisfy future requirements. The terms "module" and "component" have been used interchangeably in the past. We prefer to use "module" to refer to the traditional units of development and to "components" to refer to prefabricated software building blocks. We first present the merits of each approach and then argue the need to transition to component-based development. Table 1 is a comparison summary of the two approaches based on the criteria of development time, cost, reliability, and performance.

Table 1 Traditional Versus Component-Based Development

	Time	Cost	Reliability	Performance
Traditional Module Development	Depends on module development time	Increasingly expensive	Only partial, through testing	Built to meet exact requirements
Component-Based Development	Components are prefabricated	Economies of scale applies	Increased reliability through reuse	Customization and adaptation

Development time favors a component-based approach since with prefabricated components there is minimal time spent on code development. Similarly cost criteria favor a component-based approach since component development cost is amortized over its reuse across projects (Szyperski, 1998). The reliability of components is established over time through its repeated usage under the assumption that most of the bugs would have been removed. The performance dimension may slightly favor module development in the in the short term of the system usage since its design could meet the exact requirements of the software (Figure 1.1).

Based on this comparison it is becoming increasingly obvious that component-based development is the approach of choice when we consider the rising cost of developing software, the time constraints that software developers must increasingly meet, and the increased concern over code reliability as the complexity of software increases.



(a) Traditional Module Development **(b)** Component-based Development

Figure 1.1 System Performance

1.2 The Difficulties in Component-Based Development

A component is a binary unit of execution that is instantiated at run-time to provide a needed functionality. We use the term "component" interchangeably to refer to the component itself or to its instantiation and rely on the context to distinguish between them. We model a component as a "black-box" that is accessible only through its one or more interfaces. Components interact with each other at run time when one component invokes a function on another component. While this action appears to be simple it requires the developer to address many details. First, the semantics of the components should be complementary, i.e. the service or functionality that one component is requesting must match with the one offered by the other component. Second, the invocation of the function requires knowledge of the function's signature; otherwise, it

will result in type mismatch. Third, cooperating components must assume compatible programming models such as synchronous versus asynchronous or single versus multiple threads of execution. In the absence of component standards it is not likely that the components can be readily assembled into larger systems. Garlan refers to the incompatibility between component interfaces as the "architectural mismatch" problem (Garlan et al., 1995). The solution to this problem must assume that in general components are available only in binary form, which implies that any required changes to either component is not possible. In order to remove this mismatch developers need to find ways to mediate the interaction between components externally. Different approaches based on a narrow problem-domain for the development of systems from components have been implemented with limited success. The Domain-Specific Software Architecture (DSSA) project focused on specific military applications for which standard architectures and component sets can be developed for the purpose of building such systems from reusable parts (Mettala and Graham, 1992; Coglianese and Goodwin, 1992). Object-oriented application frameworks capture the object-oriented model for components and overall structure of a family of applications or product line for instantiating a customized application instance (Schmid, 1997). In chapter 2 we survey approaches to software development that are based on components.

1.3 Component Adapters

The composition of components requires the use of late-binding techniques (Mittermeir and Kofler, 1993). Software engineers have used many ad hoc approaches to deal with architectural mismatch. Shaw proposes nine ways to resolve the mismatch between two components (Shaw, 1995). Assuming we have two components A and B, we generalize

the alternatives given by Shaw into three groups, depending on where the changes are introduced: (1) On the side of A, (2) on the side of B, (3) between A and B. The latter one requires the use of adapters and is the preferable alternative since it does not involve any change to either component. An adapter is a piece of software introduced transparently between two components to mediate their mismatched interfaces. The development of component-adapters can be difficult and time consuming. Writing adapters requires understanding of the interfaces involved and generating the appropriate code that can interact with both components while performing the necessary translation. To make component-based software development a viable approach it should be possible to generate adapters automatically and dynamically. It should be possible for developers to specify only the type of interaction between components and allow for the dynamic creation of adapters at deployment time taking full advantage of domain-based component meta-information. This task requires that components have the property of reification to allow a meta-level programming (Szyperski, 1998). Current software component architectures lack support for such an approach to automate the generation of adapters. In Chapter 3 we present a framework for component-based development that is based on architectural standardization of components and the use of adapters. The framework includes an interconnection language that can be used to specify systems at the architectural level. A compiler for this language translates the specifications into components that are capable of generating adapters at component deployment time to allow interaction between components and mediate mismatches in the interfaces.

1.4 A Component-Interconnection Language

An interconnection language must be able to describe the pattern of interactions between components. In Chapter 6 we introduce a language called COMPILE that can capture the high-level organization of a system in terms of the interactions between components. The expressive nature of the language allows components to be organized into groups called "clusters." There are two important aspects to a cluster: first, it allows the user to create coarse-grained components through the abstraction mechanism; since clusters are implemented as components they can interact uniformly with other components and be included within other clusters. Second, a cluster is implemented as a component that itself coordinates the activities of its constituent components at deployment time. The function of the cluster component is to get a handle on the components through interaction with the run-time environment and to dynamically generate adapters that can mediate component interaction within the cluster.

The computational model of the COMPILE language is based on late and dynamic binding of the components' control, data, and function properties. The use of the push model for method invocation allows control binding among components to be late and dynamic. Data exchanged between components is defined through the use of a meta- language that can describe the semantics of the information but without being bound to any specific programming language type representation. Late binding to functions is accomplished by maintaining domain-based semantics as component meta-information. This information allows clients of components to map generic requested service to specific functions.

1.5 Thesis Organization

In Chapter 2 we provide a survey of related work that provide points of reference to the research conducted in this dissertation. Chapter 3 offers a framework for the development of components. Chapter 4 presents a comprehensive model for software components that is essential for any discussion on component-based software development. The properties and features of software components are discussed to clear any ambiguities about the overloaded meaning of software components. In particular, a reference to the object-oriented paradigm is made to help differentiate between the two concepts of objects and components. Chapter 5 introduces the notion of component clusters as an abstraction mechanism that allows the definition of coarse-grained components. In addition to having an important role in system organization, a cluster defines a pattern of interaction on the set of components it manages. A cluster interacts with its environment to locate the components and generate the adapters they require. Chapter 6 presents COMPILE, an interconnection language that allows developers to specify the desired interconnections between the components of a system. The compilation of the specification produces the design pattern components needed to implement the automated assembly of the system. Using COMPILE developers are able to give a high-level specification of the system without addressing the low-level interconnection issues. Through the use of adapters dynamically generated and bound at run-time, COMPILE provides the needed mediation between components. Chapter 7 presents a case study that illustrates the concepts of clusters and design pattern components. The example used in the case study is to build a compiler from components. We use COMPILE to specify the architecture of the compiler. Finally,

Chapter 8 is a summary of the model and concepts of the research presented in this dissertation. The advantages and alternatives are discussed. Suggestions for future work, which can be used to extend the model is presented.

1.6 Related Issues

Building systems from components is an approach used by all mature engineering disciplines. However, for components to be useful in design and development there has to be a model based on standards that specifies the type of components to produce. By committing components to standards it will be possible to sustain component-markets that can not only speed up the development process but also reduce the cost of systems since components will constitute substantial units of reuse. In software engineering, models for components have been suggested covering the spectrum from functions and procedures to packages and modules. The objective in defining these units has been to find a way to divide the complexity of the systems into more manageable units and the desire to improve productivity through the potential reuse of these units. However, the benefits have been limited to independent compilation with a linking step that allows dynamic binding to the externally accessible entities of the units. However, the level of details required in the specification of software makes it very difficult, if not impossible, to get a substantial level of software reuse without requiring considerable effort in making the interfaces compatible. The software component model needed must support a binary view of components, i.e. no assumption is made about the availability of source code, however, access to components through tools not only will allow exercising the customization options but also the assembly of components through dynamic adapter mediation.

1.6.1 Semantic Gap in Programming Languages

Programming languages have limited support for creating abstractions at the software architectural level. The abstractions typically found in a programming language consist of data types, control structures, functions, and procedures. These abstractions can be used to express the entities of the problem and the operations performed on them. The program becomes an abstract representation of the solution. To the extent that the abstractions within a programming language are closely matched with the concepts of the problem domain, the task of representing the solution can be greatly facilitated. As programming languages evolved, new abstractions were added. Object-orientation introduced a new abstraction that significantly increases the programmer's modeling capability. A significant contribution of objects is in their ability to offer a conceptual representation of many concepts and entities in the problem domain in a way that is meaningful to the programmer as well as the customers and users of the software. The object paradigm groups the state and the behavior of entities into a single structure and allows entities to be organized hierarchically. Recently software components have emerged, offering similar conceptual abstractions as objects and promising a shorter software development cycle. The appeal of software components comes from the tool-supported development environments that allow the components to be assembled into a system. However, existing software development methodologies are not adequate for this new paradigm.

1.6.2 Limitations in Development Methodologies

Developing software from prefabricated components requires an appropriate methodology that can guide developers in building systems. A simple adaptation of an

existing software development method is not adequate and can only produce an incremental improvement. Although the idea of software components as building blocks is not new, the development of systems from existing components defines a set of issues and problems to solve that are significantly different from other known methodologies. A Component-based methodology is likely to have similarities with object-oriented approaches due to the conceptual similarities of objects and components. Where the corresponding methodologies may differ is because objects are primarily specified as a result of system analysis and design while software components are likely to have been built without advance knowledge of the specific application in which it will be used. The greater emphasis on software tools for manipulating components during development distinguishes the component-based methods from all the other development methods.

CHAPTER 2

RELATED WORK

To simplify the development of software requires isolating elements of the software that that can be factored out, generalized, and applied across projects. This is the basic notion of reuse and when applied to software, it can greatly reduce the complexity of the development process. Three major elements present in software are: functionality, data, and communication. In this chapter we survey the ways in which it has been possible to exploit knowledge in these three areas in an effort to simplify the development of software.

2.1 Reuse Artifacts

Domain knowledge is an essential element of software reuse. We define a domain to be any set of entities that share common characteristics. Accordingly, we can identify two types of domains: vertical and horizontal domains. Vertical domains include whole industries such as banking, insurance, and manufacturing, while horizontal domains include specific types of applications such as accounting, inventory, and payroll. Software reuse is the application of knowledge acquired from developing systems towards development and maintenance of new systems (Prieto-Diaz, 1993; Biggerstaff and Perlis, 1989). An important criteria in measuring the state of the art in any domain, be it software or other engineering discipline, is by the level of systematic reuse applied during production processes. Software engineering has been lacking behind most other disciplines when it comes to the definition of, and systematic reuse of artifacts. Expressions such as *software crisis*, *software bottleneck,* and *Very Large Scale Reuse problem* give us a clear indication of the current limited ability in developing large and

complex systems. Software reusability may appear to be a much simpler concept than it really is. Software requires a high level of specificity in its definition. In addition, the fragility of software in the presence of the smallest errors makes the process of development a very complex one. Both of these problems can be addressed by reuse. Reuse in all aspects of software engineering has been studied in the last three decades but success has fallen far short of expectations (Prieto-Diaz, 1993). Some reasons for the failure of reuse is attributed to the lack of planning and support within the organization (Card andComer, 1994; Fafchamps, 1994), economic factors (Joos, 1994), and technical issues (Henninger, 1994; Prieto-Diaz, 1989). Some reuse programs have been reported to achieve anywhere from 30 to 80 percent reuse (Card and Comer, 1994). We believe that reuse when based on a narrow domain of applications supported with industry standards is the most promising approach. This view is supported by many of the results reported in the literature.

In this chapter, we present a survey of software reusability from three points of view: Domain analysis, Design, and infrastructure. These three areas are related to various phases of the software lifecycle addressing system analysis, construction, and runtime environment. Two essential criteria for applying reuse are abstraction (Parnas, Clements and Weiss, 1989) and separation of concerns (Dijkstra, 1968). In software development, the most critical time to apply reuse is during the early phases when the problem and its solution are being formulated. The lack of software artifacts primarily at the design level has caused a software crisis that limits our ability to build large and complex systems.

2.2 Software Reusability

Reuse is a major driving force behind advances in software engineering and has a major impact on our ability to produce large and complex systems. A systematic reuse of software artifacts during the development process allows us to build systems quickly, efficiently, and with high degree of reliability. The three areas that have the most significant impact on software development, domain analysis, design, and infrastructure, are important areas to consider for reusability as shown in Figure 2.1. These areas are related and form a continuum of views on a system differing only by the level of abstraction. It is vitally important for the development of non-trivial systems that define abstractions at each level that can form a basis for reusability. To take full advantage of the reusability artifacts we must build tools around these abstractions that facilitate their use.

2.1.1 Three Levels of Reuse

In building systems, an understanding of the problem domain is of great importance and offers significant opportunity for reuse. A domain analysis can help us identify all the necessary information required for developing a system in a particular domain. A good representation of the domain information can be used in the development of different system or a family of applications in the domain. Since most reuse is based on domain specific information, this area has also an impact on the other two areas.

The design of the system is based on our understanding of the application and its domain. An area of growing importance is how to best capture the collective experience of good designers in developing systems and make that available through reuse artifacts.

In this context, design patterns at all level of design have recently been gaining recognition since they provide valuable information on existing systems and more importantly can be used as a basis for the design of new systems. Design patterns, frameworks, and domain specific software architectures are three of the best examples explored so far in the area of software design reusability.

An infrastructure is the basis on which a design is implemented. It has an essential role in the development of systems since it determines the practical aspects of the implementation and the feasibility of the design. For the last decade, considerable advances have been made in the area of middleware. The complex nature of system requirements and design make it necessary for advanced middleware solutions to be available upon which the system can rely. Two the major areas of advances are in database and distributed communication middleware. Infrastructure is an example of reuse where some of the basic functionality requirements of a system can be defined externally to the application through well-defined service interfaces.

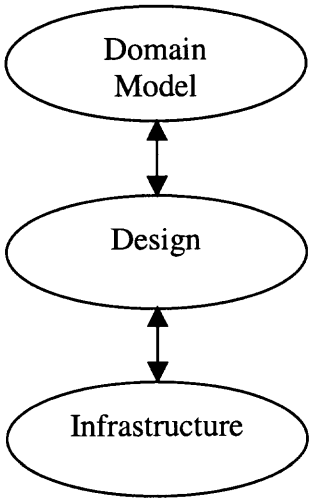


Figure 2.1 Three Levels of Reuse

2.1.2 Abstraction and Separation of Concerns

The process of domain analysis can be summarized in three steps: identification reusable entities, abstract the information, and classify for future reuse. Abstraction is one of the most powerful techniques known for solving complex problems. The process of abstraction allows us to selectively consider the relevant aspects of a problem while suppressing all other aspects that have no bearing on the problem or its solution. Abstraction is based on the principle of information hiding where a problem can be separated into a static interface and changeable implementation (Parnas, 1972). In all engineering disciplines, abstraction is used to deal effectively with complex problems in the domain. Relative to other well-established engineering disciplines, such as electrical engineering, software engineering is lagging based on the abstraction and artifacts available to designers.

Separation of concerns is another approach to complex design problems that can work well with abstraction. The observation that the structure of a program is as important as its correctness (Dijkstra, 1968) puts the role of design in perspective as the size and complexity of software continuously increases. One of the objectives in design is to reduce the complexity of the problem by separating it into a set of problems that are relatively independent of each other but which collectively provide a solution to the problem. Both, abstraction and separation of concerns play a critical role in any approach to design and implementation.

The effectiveness of a design methodology can be measured by the extent to which it supports the use of the above two principles. System developers take advantage of these principles by either using them directly or with tools, such as CASE. Having the

right tool available can shorten the development time and enhance the quality of the product. Tools are used at various levels of abstraction corresponding to the various phases of the project. Tools typically leverage the techniques of abstraction and separation of concerns to help the user of the tool in dealing with the difficulty of the problem.

For tools to be effective, they must be used in a supporting role to the development process. The development process specifies the tasks to be fulfilled along with the input and output of each task in order to develop the system. The process definition is based on a methodology that identifies the problem domain and justifies the applicability of the methodology for a given problem. Figure 2.2 shows the mutual influence that methodologies, processes, and tools can have on each other.

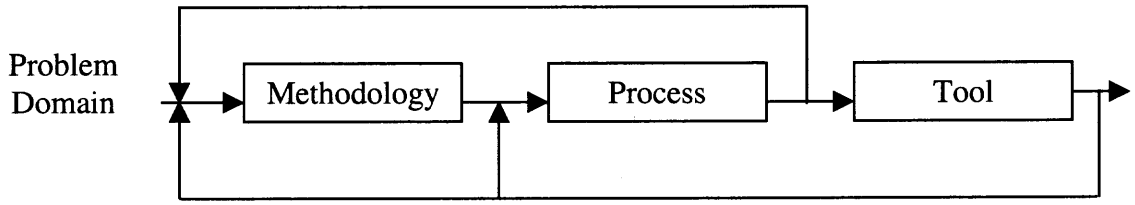


Figure 2.2 Tools in Support of Methodology and Process

2.2 Domain Analysis

Domain analysis is the process of identifying, capturing, and organizing information during software development for reusing it in the development of new systems (Prieto-Diaz, 1990). Jim Neighbors introduced the expression *domain analysis* to refer to "the activity of identifying the objects and operations of a class of similar systems in a particular problem domain" (Neighbors, 1984). The importance of the information

captured in domain analysis is the key to reusable software, with emphasis on reusing analysis and design over code (Neighbors, 1989; Prieto-Diaz, 1990; Arango, 1989). Software applications belong to particular domains and are developed to serve the needs of users within the domain. To describe the purpose of a software system, we often use terms that relates the system to the domain in which it is used, such as airline reservation system, inventory control, banking. We can view the problem space of all software applications as spheres of domain, with some having overlapping instances. Each domain consists of a set of applications that address different aspects of the domain with some applications belonging to more than one domain. Applications within a domain often need to interoperate in order to provide an integrated solution to a business process. Therefore, by using domain information, we can a more complete understanding of the context in which we will use the application and how it may be designed to interoperate with other applications in its environment. Domain analysis is the process that we use to identify, capture, and organize the information about the domain that can be used to develop new systems (Prieo-Diaz, 1990). By making domain information available to software developers, we have created a reusable software artifact that can be used in the development of systems across the domain. Not all domains are equally understood. Our understanding of the domain evolves over time and is enhanced with every system that we build. For domain information to be useful in system development, we must have sufficient experience in the domain (Agrawala et al., 1992).

2.3 Design and Architecture

2.3.1 Design Patterns

The complexity and the size of software systems are steadily increasing. Designing large and complex systems is continuously pushing software developers to the edge of their abilities. The object-oriented paradigm has helped us maintain some intellectual control over the increasing complexity of design by abstracting the problem domain into a set of cooperating objects. The design of the system can then be based on the discovery of these objects, assigning the responsibilities to each of the objects, and defining the interaction between these objects. However, as the number of objects continues to increase, with it increases the complexity of the system design. There is a need for higher-level abstractions that can help us maintain control over the design complexities. Designers within the object-oriented community have turned to design patterns as a way to scale up the abstraction based on objects to a new level. Instead of looking at the object-based design in terms of individual objects, we now have a way to group objects together in a meaningful way to form new abstractions. We can see the design in terms of cooperating clusters of objects. The cluster of objects is described by a design pattern that represents the implementation of functionality in the problem domain. Design patterns have been suggested as a way to capture implicit knowledge of object-oriented design gained from years of experience in object-oriented software development (Gamma et al., 1995). Design patterns are an artifact of design that allows us to characterize and capture an interaction between numbers of design elements (i.e. objects). From a design instance, we can extract an abstract design that can be used in similar situations. The motivation behind this process of documenting design patterns is to simplify the task of

the software designers. Designers routinely rely on their experiences in tackling new problems. For many system designs, the same problems tend to appear in different contexts but with a common applicable solution. A design pattern represents a solution to a recurring problem. Novices as well as advanced software designers can use design patterns to help them in the task of designing software. Patterns are micro-architectures (Gamma et al., 1995) that represent a form of reusability at the design level. The abstraction represented by a design pattern defines collaboration between a set of classes. Many authors have documented a growing number of design patterns (Coplien, 1995; Vlissides, 1996; Gamma et al., 1995; Bushman et al., 1996). Gamma suggests using the following elements in describing a design pattern (Gamma et al., 1995):

1. The pattern name is used to identify a pattern.
2. The problem describing the applicability of the pattern and any required conditions that must be met.
3. The solution describes the elements of the design and the relationship, responsibilities, and collaboration between them.
4. The consequences as a set of tradeoffs in time and space that results from using a given pattern.

Patterns can be organized and catalogued according to different criteria such as their granularity and level of abstraction. Gamma offers the following classification based on two criteria: purpose and scope, Table 2. The purpose helps us identify a pattern

according to whether it is creational, structural, or behavioral. The second criteria identify whether the pattern applies to a class or object.

Table 2 Classifications of Design Patterns (Gamma, 1995)

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsab. Command Iterator Mediator Memento Observer State Strategy Visitor

Design patterns have so far been described within the realm of objects. Objects are design artifacts with which we can describe the design of a system. Objects also appear in the implementation phase when we use object-oriented languages. This establishes a direct link between the two phases of the development: design and implementation. It is interesting to note that with objects, it was possible to define patterns, but the same has not happened with functions. For example, we do not have a catalogue of design patterns based on functions. From the point of view of implementation, we can argue that objects represent a higher-level abstraction than functions. As such, objects are better suited for expressing design. Design patterns are defined at an abstract level, independent of any programming environment or language. To make use of a pattern, the programmer first identifies the applicability of the pattern to a certain design problem at hand and proceeds to instantiate the design within the overall

design of the system. The implementation of the design is carried out using the implementation programming language.

Application frameworks take the idea of design patterns beyond a simple description. By focusing on a particular problem domain, it is possible to identify one or more abstract patterns that can provide the basis for a design for such applications. The patterns are instantiated to form a skeleton of a solution, with several classes either uninstantiated, or represented as templates requiring customization. The incomplete status of the design allows for the definition of a family of applications which share the basic structure but which can differ at the points where the framework has not completely been specified.

2.3.2 Object-Oriented Frameworks

Application framework based on the object-oriented model is an approach to software reusability where the goal is to apply reuse to two critical aspects of a system: the design that defines the structures of a system and the components that provide its functionality. A framework is associated with a domain of applications (Schmid, 1997) for which the framework can automate the generation of customized applications through reuse of design and software components (Demeyer et al., 1997). Using an object-oriented model allows us to define the framework as a set of cooperating classes of which some are left abstract. The points in the design where instantiation from abstract classes is required are referred to as hot spots (Schmid, 1996). The variability of the applications generated is achieved through the hot spots embedded within the framework. The hot spot extensions of the framework can be accomplished with a black-box approach, by a selection from a

set of supplied classes, or with a white-box approach through class inheritance of a framework class and the use of polymorphism.

Defining a framework requires a mature domain and a deep understanding of the applications within the domain. One of the successful domains for which frameworks were developed is the Graphical User Interface (GUI) where a programmer can combine visual components from a library to build the GUI of an application. The Microsoft Foundation Classes (MFC) is an example of a GUI framework that has become a standard for applications running on a PC. The CORBA specification is another successful example of a framework for distributed objects that is targeted towards the enterprise application domain (Fayad and Scmidt, 1997).

Experience has shown that, in general, it is difficult to learn frameworks (Fayad and Schmidt, 1997; Johnson and Foote, 1988; Schmid, 1997). It requires the user to understand the classes of the framework and their pattern of interaction. This can further be complicated with the white-box approach used by some frameworks. Due to the lack of interoperability between various object-oriented languages (Johnson, 1997), some frameworks have been implemented in two different languages to allow for interoperability (Doscher and Hodges, 1997). Approaches based on CORBA and COM are addressing this problem.

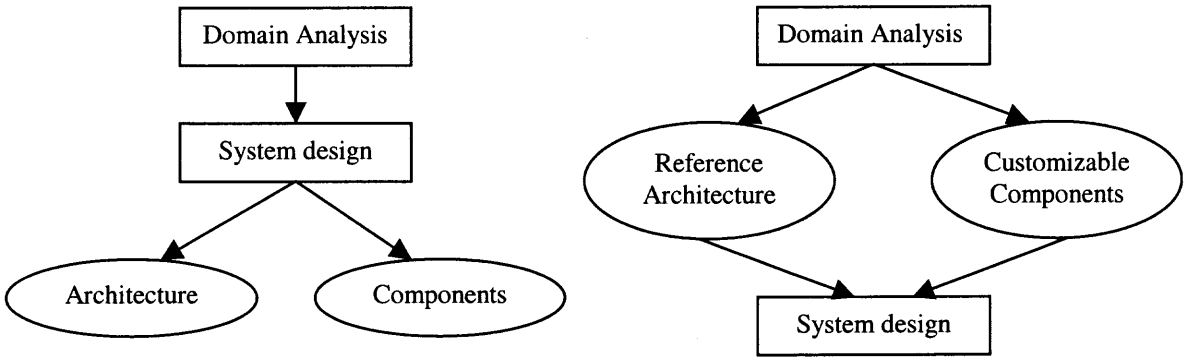
An application framework is an abstract application with incomplete parts that can be used to create customized applications conforming to a particular architecture within a specific domain (Fayad and Schmidt, 1997; Johnson and Foote, 1988). Object-oriented frameworks have become popular by capitalizing on the inherent benefits of the object-oriented model. The potential for reusability in frameworks is based on the

common characteristics shared by software systems within a domain. The criterion in defining application domains is the identification of the common salient features among a set of applications. Within domains, frameworks are developed to capitalize on the commonalties among the applications by carefully factoring out and packaging the various aspects of the system for reuse. Defining a framework requires a view that combines a top-down synthesis with a bottom-up analysis approach. In the top-down view, domain information is compiled to identify individual applications with a corresponding generic architecture. The late binding of some aspects of the application within the reference architecture allows the production of a variety of potential applications. In the bottom-up view, we categorize the applications within a domain based on our experience of existing applications, identifying the commonalties and potentials of reuse. From this information, it will be possible to define a reference architecture that describes a set of applications. In practice, it is more likely to combine the domain analysis information with the experience about existing system to help develop a framework.

2.3.3 Domain Specific Software Architecture

Domain-specific software architectures are used to describe a family of applications and promote reuse at the architectural level (Mettala and Graham, 1992; Tracz et al., 1993; DeBaud, 1996, 1998; Waite and Sloane, 1992). Developing architecture for a specific domain has many advantages and opportunities for reuse. To take advantage of reuse we need to first define software artifacts (Tracz et al., 1993). A domain-based approach to reusability provides a framework that allows for the definition of reusable components conforming to a reference architecture, that allow customization. Traditionally, domain

analysis techniques provided us with a model of the problem domain to serve as a basis for developing the system design. The approach taken by the domain-specific software development is to use the domain information to generate two distinct models describing the architecture and the components of the system (Tracz et al., 1993). In this case the design and implementation of the system is reduced to reuse and adaptation.



(a) Reuse Based on Domain Model

(b) Reuse Based on Reference Architecture and Components

Figure 2.3 Domain Analyses

DSSA was a five-year program consisting of six projects, each combining efforts from industrial research laboratories, academicians, and military laboratories. Four of the projects were targeted towards specific military domains and the other two projects investigated enabling technologies for use in domain specific architecture construction (Mettala and Graham, 1992). An objective of DSSA was to discover ways to transform software development from traditional monolithic systems to one in which systems can be built from building block components with appropriate tools for automation. The domain specific projects considered domain specific architectures in avionics navigation

(Coglianese et al., 1992), command and control (Braun at al., 1992), distributed intelligent control and management for vehicle management (Hayes-Roth et al., 1992), and intelligent guidance in navigation and control (Agrawala et al., 1992).

Of the three distinct approaches to architecture used in the DSSA program, we will consider the one based on domain modeling. In this approach, the process starts with a domain analysis of the target domain to capture a model that identifies the critical aspects of the domain, such as objects, operations, and relationships, as viewed by experts of the domain. The model is built to represent a family of applications. A software architecture is synthesized from the domain model. This is a reference architecture, rather than a specific architecture, that can be used as a basis to derive application-specific architectures belonging to the domain (Mettala and Graham, 1992). The domain model and the reference architecture become development artifact of the software process that enables reuse at a very high level.

The problems targeted by the DSSA program were very large and complex (Coglianese et al., 1992; Braun et al., 1992), but the solutions were well understood. The research capitalized on the fact that within a particular domain of problems, there were many more similarities than differences, many of which can be addressed by an adaptation technique (Coglianese et al., 1992). Therefore, it is up to the domain analysts to organize the domain information in a way that relates the features of the problems in the domain. The domain information can be used to describe the behavior of the components within the domain along with their interfaces according to an underlying architecture.

In the command and control project (Hatch et al., 1992) of the DSSA program, reusability was based on application generator technology. This allows software applications to be created by using high-level language specification targeted specifically to the particular domain. This is similar to fourth generation languages, which are usually targeted towards management information systems, where productivity improvements have been reported to be as high as 50 to 100 times more than traditional languages. Application generators are important whenever the application domain is rather narrow, as is the case with the command and control project. The development of application generators is a significant undertaking and the code they generate is far from optimal. Some accept annotations incrementally to provide tuning capabilities to the translator.

In the distributed intelligent control and management project (Hayes-Roth et al., 1992), the objective was to develop a framework for understanding the software architectures for control problems. The framework provides a generic but customizable model for controllers. The problem domain required an architecture that can accommodate a number of controllers forming a hierarchical structure based on the scope of behavior they address, the resources they control, and the time frame spanned by their decisions (Hayes-Roth, 1992). The DSSA architecture for this domain provided a specification for the structure of controllers and a blackboard system, augmented with expert system capabilities, to exploit opportunistic reasoning.

2.3.4 Megaprogramming

Megaprogramming was introduced by DARPA to address component-based engineering and life-cycle management associated with software (Tracz, 1991). The goal is to find

new approaches to software development in dealing with the increasing complexity of large systems. Instead of developing software by putting statements together one at a time, it should be possible to develop software by putting together one component at a time (Wiederhold, 1992). Megaprogramming is an effort to change the focus from programming in the small towards programming in the large (DeRemer and Kron, 1976; Wiederhold, 1992). It not only addresses the problem of largeness of size but also persistence, variability, and infrastructure (Wiederhold et al., 1992). Megaprogramming is a technology for programming with megamodules that captures the functionality of services provided by "large organizations like banks, airline reservation systems, and city transportation systems" (Wiederhold et al., 1992). The approach advanced by megamodules is to replace the tightly coupled structure of programs based on a set of procedures and functions with large components that have considerable degree of independence. Each megamodule encapsulate data, procedures, types, concurrency, and ontology, while providing an interface through which the services of the megamodule can be accessed. The coordination among the megamodules is handled at the megaprogram level. Megaprograms provide the context in which megamodules can interoperate. Megaprogram is a composition of megamodules supported by synchronous and asynchronous coordination schemes, decentralized data transfer, parallelism, and conditional execution (Wiederhold et al., 1992). Several benefits can be derived from the megamodule model:

Megaprograms encapsulate a higher level of abstraction than functions, procedures, or objects: The level of abstraction of an entity depends on the level of abstraction of its constituents. By defining an entity in terms of other abstractions, the

entity takes on an abstract level that is higher than the ones it includes. Functions encapsulate statements and expressions and objects encapsulate data and procedures. By contrast, megamodules go beyond objects to encapsulate behavior, knowledge, concurrency, and ontology. This gives megamodules an abstraction at the level of subsystems.

Megamodules define a basis for reusability: The high level abstraction of a megamodule as described above gives it a high degree of independence from its environment. A megamodule relies primarily on its own resources to perform its task. It interoperates with other megamodule in order to achieve the goals of the megaprogram of which it is a part. It is possible to specify and build megamodules based on domain analysis and according to needs. A pre-facto building of megamodules as components for composition makes them units of reuse.

Megamodules improve maintenance management: There is a high degree of cohesion within a megamodule and a low coupling between megamodules. This focuses the maintenance effort primarily at the megamodule level and to a much lesser extent on the megaprogram level. Because of the clear separation between the megamodules and megaprogram, the maintenance tasks are confined primarily to within individual megamodules. This improves the reliability of the megamodules and greatly enhances their maintenance.

Megaprogramming is not a silver bullet solution for improving software productivity and quality (Boehm and Schrelis, 1992). It is based on a product line approach to software development that is based on domain analysis as a basis for the development of software component and domain oriented software architecture.

Megaprogramming as a programming paradigm requires a disciplined approach to software design and a critical mass of software components (Tracz, 1991, 1993).

2.4 Infrastructure

2.4.1 Middleware

Middleware is the software that resides between the operating system and the application program to enhance distributability of applications, interoperability between applications, and portability of applications between systems (Colonna-Romano and Srite, 1995). Middleware supports standards-based interfaces, including APIs and networking protocols, to provide high-level transparent distributed services. The middleware services can be used as building blocks to provide distributed computing resources in developing enterprise-wide information systems. Middleware Services can be represented as taxonomy categorized according to the level of integration between the services and the relevance to a particular domain (Colonna-Romano and Srite, 1995). At the lowest level, a middleware service can be generally independent of other services providing a specific function to the application. At the next level, the services are regarded as an integrated set where a particular service takes advantage of other services to reduce complexity and provide higher-level functionality. For example, distributed files service, which uses a directory service and security service to provide remote and secure file access transparently. At the highest level, integration frameworks provide services that are domain specific that can be useful in building applications in that domain

The OSF Distributed Computing Environment (DCE)

The Distributed Computing Environment provides a set of integrated services that work across multiple systems and remain independent of any single system (Rosenberry et al., 1992). DCE supports the construction and integration of applications in heterogeneous distributed environments. The modular structure of DCE allows the various components to be installed on servers with appropriate resources. A decentralized control allows you to manage each component of DCE independently. The services offered by DCE include security, directory, time, remote procedure call (RPC), threads, and file services (Chappell, 1994). The first three services in the list are mandatory in defining a DCE cell, which is the basic unit of operation and administration. The boundaries of a DCE cell are influenced by four considerations: purpose, administration, security, and overhead. DCE was designed to support procedural programming. It does not support the object-oriented paradigm and hence there is no concept of class hierarchy, inheritance, late binding, or dynamic objects. The inherent synchronous nature of RPC does not support message queuing as in asynchronous programming. Similarly, RPC does not handle directly the implementation of transaction semantics, although such semantics can be built within the application. One of DCE's design goals is to conform to standards. For example, it is compatible with the X.500 Directory Services and DCE threads implementation is based on the POSIX standard.

The Object Management Architecture (OMA)

The Object Management Group (OMG) provides a specification for a standard object-oriented architecture for the development of distributed systems. It provides a common

framework for application development based on reusability, portability, and interoperability of object-based software in distributed heterogeneous environments. The OMG Object Management Architecture (OMA) consists of an Object Model and a Reference Model. The Object Model provides an organized presentation of object concepts and terminology while the Reference Model characterizes interactions between objects. The Object Model relies on the following concepts: abstraction, encapsulation, inheritance, and polymorphism. There are many benefits for combining the object model with distributed computing. The expressive power of objects to model the real world combined with reusability and extensibility of objects makes the development of distributed applications easier. The Reference Model consists of the following components:

- Object Request Broker (ORB)
- Object Services
- Common Facilities
- Application Objects

The Object Request Broker (ORB) is provides the communication bus for all objects in the system. The Common Object Request Broker (CORBA) (OMG, 1998) specification provides the details of the ORB component of OMA. The main components of CORBA 2.0 (Vinoski, 1997) are:

- The ORB core
- The Interface Definition Language (IDL)
- The Interface Depository

- The language Mappings
- The Stubs and Skeletons
- The Dynamic Invocation and Dispatch
- The Object Adapters
- The Inter-ORB Protocols

The IDL definition language is used to specify the interfaces of the objects and the data structures that will be shared between remote objects. The mapping of IDL into many diverse languages such as C, C++, Smalltalk, Java, Ada, COBOL, Modula-3, Perl, and Python allows applications to be developed in these languages to interoperate across heterogeneous distributed environments. The Object and Reference Models define the rules of interaction between objects independently from the underlying network protocols used. The CORBA specification allows for the integration and interoperation with components written in DCE or COM.

Comparing the CORBA and the DCE Distribution Models

The distribution model of DCE relies on RPC while in CORBA distribution is based on communicating objects. Since CORBA is based on the object model it can benefit from all the advantages of the object-oriented technology. CORBA provides a more flexible environment to develop distributed applications. By making all the communications go through an object request broker objects can be added to the running environment in a transparent way without requiring the modification of existing objects. DCE requires a more direct coupling between the distributed cooperating modules requiring direct low-

level knowledge. CORBA supports synchronous and asynchronous communications. DCE supports synchronous communication primarily and asynchronous communications through the use of multiple synchronous threads. This makes CORBA better suited for the integration of legacy systems that use asynchronous mode of communication.

Transaction Processing Monitors

Transaction Processing (TP) monitors (Bernstein, 1990) is a type of framework. Its main function is to coordinate the flow of requests between terminals and application programs that processes these requests (Bernstein, 1996). A TP monitor supports functions for transaction management, transactional inter-program communications, queuing, and forms and menu management (Bernstein, 1996). Transaction management includes support for start, commit, and abort operations on transactions. Transactional interprogram communication supports the propagation of a transaction's context when programs call each other. The services offered by a TP monitor may be integrated and accessible through a simplified and uniform API.

2.4.2 TAFIM

TAFIM (TAFIM, 1996) is a technical reference model that establishes a common vocabulary and defines a set of services and interfaces common to the Department of Defense (DoD) information systems. The reference model is not a specific system architecture, rather it defines the target technical environment for the acquisition, development, and support of DoD information systems. The model adopts the foundation work of the IEEE POSIX P100.3 working group. The basic elements of the model are shown in Figure 2.4. The application software entity represent the applications used by

the different services (Army, navy, Air Force, Marine Corps). The model promotes interoperability between applications and to be portable across various hardware and software platforms.

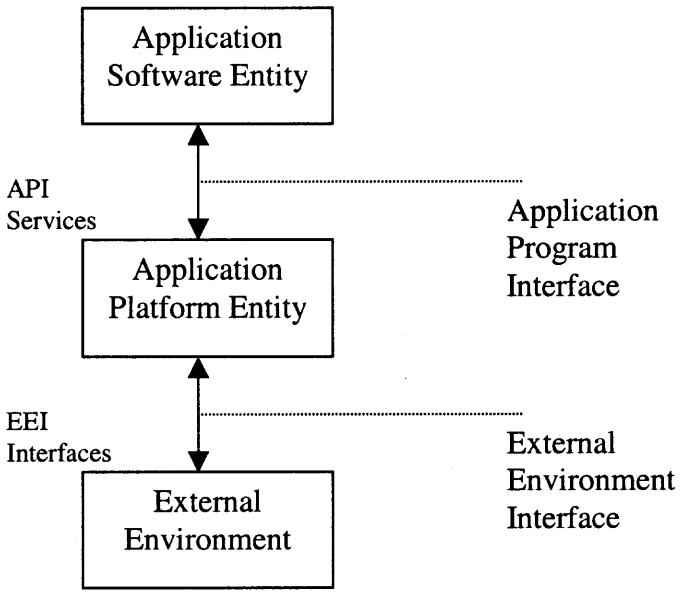


Figure 2.4 Generic DoD Technical Reference Model (From TAFIM)

The model promotes the goals of developing modular applications, software reuse, portability, and data sharing. The Application Program Interface (API) is the interface between the application software and the application platform across which all services are provided. The services specified by the API include system services API (for software engineering and operating system services), communication services API, information services API (for data management and data interchange), and human/computer interaction services API (for user interface and graphics services). The Application Platform Entity defines the set of resources on which the application will execute. All resources are accessed through service requests across the API. The

External Environment Services (EEI) is the interface between the application platform and the external environment across which information will be exchanged. Its purpose is to support system and application software interoperability. The EEI interface includes services for human/computer interactions, information, and communications. The External Environment contains the external entities with which the application platform exchanges information.

CHAPTER 3

A COMPONENT-BASED SOFTWARE DEVELOPMENT FRAMEWORK

3.1 Programming with Components

Distinguishing between levels of programming has been long recognized as necessary in order to separate and address different software development concerns at the most appropriate level (Wiederhold 1992; DeRemer and Kron 1976). The terms "programming-in-the-small" and "programming-in-the-large" have been used to describe two particular programming levels. The first addresses the creation of modules using traditional programming languages, while the latter addresses the integration of modules into larger software systems. Programming languages and methodologies have traditionally focused on the sub-module level by incorporating improved data and control structures to increase the expressive power of languages. More recently, a concern about inadequate attention to the architectural levels of software has lead to the emergence of a large variety of Architecture Description Languages (ADL) (Shaw, 1991). Establishing direct linkages and mappings to requirements engineering and programming will further enhance the importance of ADLs. In this chapter, we present a framework for architectural specification of software consisting of components. What differentiates our solution from previous work is in the modeling of components and the application of late binding techniques in assembling components.

3.1.1 Component Model

In Chapter 4 we present a comprehensive model for software components. While our component model has aspects that are shared by other component models, it is

semantically richer and targeted for manipulation by tools. There are two important aspects of software components that require mentioning:

1. Software components are binary executable units that encapsulate specific functionality accessible through well-defined interfaces. The functionality of a component is accessed through instantiation. A component can therefore be viewed as a software factory that creates instances of defined functionality on demand. For the practical reasons of simplifying our writing and enhancing readability of this document we will blur the difference between the distinct notions of a component and its instance in our discussions. Whenever the intension is clear the term component is used interchangeably to mean either a component or its instance.
2. Tool-driven mechanisms for probing and manipulating software components. The tools should be able to discover relevant aspects of the component interfaces and usages. Through the properties of introspection and reflection it is possible to communicate with a component and access its meta- information.

Two important implications may be drawn from the above definition of components:

1. While the source code of components may be available, it is assumed that the binary form of a component is not intended for modification through access to source code and recompilation. Components should be viewed as "plug-and-play" or "as-is" elements that may allow a certain degree of customization by design.

2. Tool-accessibility to components implies that the effort in building software from components can and should be primarily automated. In effect, this approach offers the potential to improve and speed-up the development of software by an order of magnitude. The tedious aspects of software plumbing will be finally relegated to the machines rather than humans.

3.1.2 Mediating Component Mismatch

To build software from components it is assumed that it will be possible to match components along their interfaces. Advocating components requires that we address the issue of how to make interfaces between cooperating components compatible. This is likely to be the most significant problem standing in the way of wide-scale use of components. The problem exists not because there are inherent difficulties in making components compatible, but because there is an absence of standardization forces that can address the semantic gap existing between components that are to be developed independently by different parties. The common answer to the mismatch problem is to mediate between the interfaces of mismatched components through the introduction of new software. This is typically handled through the use of adapters or wrappers:

- An adapter mediates the mismatch between two components by providing transformations that convert the communication content generated by one component to one that is recognizable by the other component. These adapters are sometimes called bridges or mediators (Figure 3.1).
- Similarly, a wrapper accomplishes mediation by encapsulating logically the access to one component and presenting an interface that is compatible with the other component. The composite component can be treated as a single

component with the interfaces of the inner component unseen and inaccessible from outside the wrapper component (Figure 3.2). An important use of wrappers is to encapsulate legacy systems so they may be treated uniformly as part of a larger system. This is a viable approach to deal with systems that may be unstructured and difficult to modify or prohibitive in cost to replace. Through the use of wrappers it becomes possible to streamline the representation of such systems.

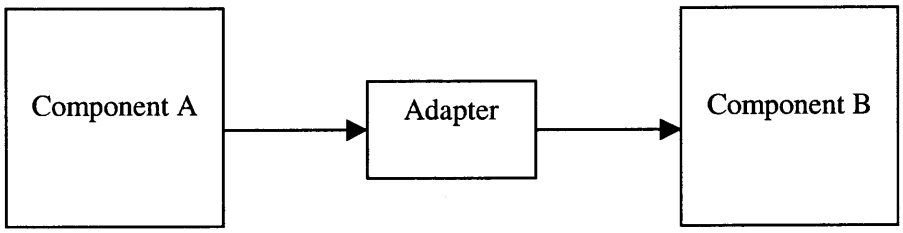


Figure 3.1 Adapter Between Components

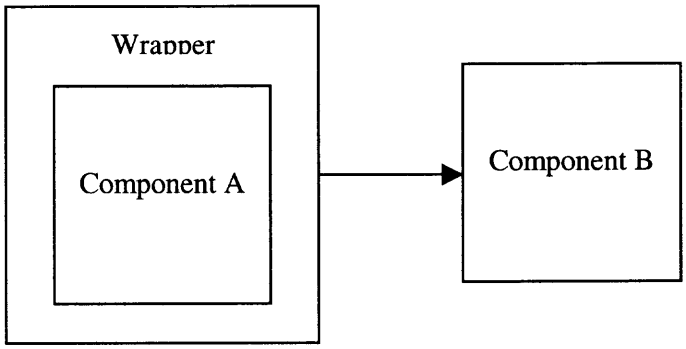


Figure 3.2 Wrapping a Component

We will focus on the problem of creating adapters between components. The process of creating wrappers for legacy systems requires human intervention and may be difficult, if not impossible, to automate completely.

3.1.3 Simple and Composite Adapters

An adapter, in general, is used to mediate differences between components. However, when we examine the causes for the mismatch we find that there are three distinct types related to the specification of function, control, and data between components. The use of adapters must, therefore, handle all types of mismatch, including the case when multiple mismatches are involved simultaneously. We use the term Simple Adapter (SA) to refer to the type of adapters that deal with one type of mismatch. The term Composite Adapter (CA) is used to refer to an adapter dealing with more than one type of mismatch.

3.1.4 Generating Adapters

Adapters may be complex and difficult to produce. The process involves a detailed examination and an understanding of the interfaces of the components involved. The cost of generating adapters can be divided in some proportion to the cost of the development and maintenance phases of software. As we attempt to build a software system from components there is a cost associated with the creation of any adapter deemed necessary to allow interoperability between components. Later on, as the system evolves and undergoes revisions it may be necessary to disassemble and reassemble certain components differently, or it may be necessary to add new components to expand the functionality of the system. In both cases there may be a great effort and cost involved in the creation of adapters. Perhaps the most prohibitive cost involved is in the time

required and reliability of the resulting system. Therefore, the conclusion tends to point towards automating the process of generating adapters. This can be made possible if we build the interfaces in such a way as to allow the tools to communicate with components enabling us to discover the type of interfaces presented. Only then suitable adapters may be formulated.

The graph below shows the cost of developing adapters manually in a component-based development environment. The cost does not only accrue initially at system development time, but extends to the entire life cycle of the system. The curvature of the graph suggests an increasing rate in the cost is due to the increased complexity in creating new adapters while the system may undergo frequent requirement modifications imposed on agile organizations that need to respond quickly and frequently to changing market conditions.

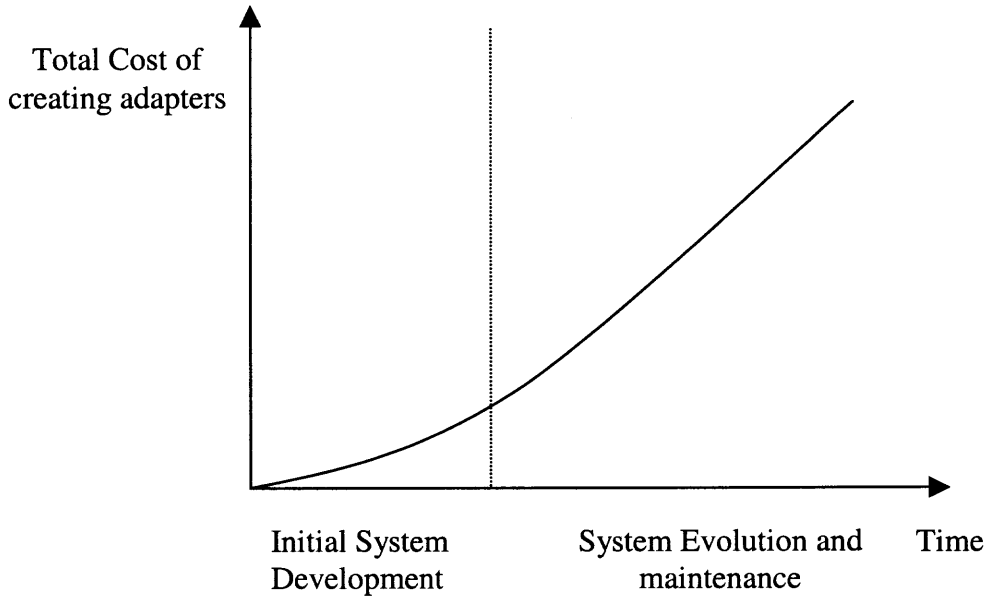


Figure 3.3 Cost of Developing Adapters

3.2 Component Binding Time

Assigning value to the attribute of an entity during software development is called attribute binding. The binding time refers to the time at which binding takes place. Early binding usually refers to the binding of attribute before run-time, while late binding refers to binding that takes place at run-time. We define static binding to include compiling- and linking time. Features in programming languages associated with static binding are more easily understood by programmers and less likely to be misused. Features with late binding are more difficult to understand but potentially can offer programmers many benefits. Some of the most important benefits of object-oriented programming languages have been derived from late binding. The following table shows some of these features with their binding time, as implemented in C++.

Table 3 Binding time of some object-oriented Features in C++

	Compile-Time	Run-Time
Templates	X	
Virtual Functions		X
Inheritance	X	
Event-Based		X

The Java programming language offers a more extensive set of features with late binding properties. By giving the programmer access at a meta-programming level, the structure and properties of classes can be modified dynamically.

3.2.1 Virtual Functions and Polymorphism

Virtual functions are a good example of the use of late binding that permits programmers to be concise and abstract in writing code without being ambiguous. A function is defined on an inheritance-based hierarchy of classes such that each class can define different implementation of the function while maintaining the same function signature. With a reference to an object instance, belonging to a class in the inheritance hierarchy, invoking the function on this object requires that the specific function implementation be identified correctly. The compiler generates code that determines the correct binding for each reference to virtual functions based on the objects' type, which can only be determined at run-time. The added benefits of virtual functions come from the code extensibility property: classes can be added to the inheritance hierarchy without having to make significant changes to the existing code.

3.2.2 Event-Based Programming

Event-based programming has been popularized by graphical user interface toolkits and object-oriented frameworks. This technique allows the programmer to establish function calls dynamically. In traditional programming, the programmer explicitly specifies within the code the point in the program from which a function will be called. Using the event model it is possible to determine at run-time whether a function should be called. Functions subscribe to events as needed, allowing for the decision to be made based on the programming logic.

3.2.3 Late Binding with Components

Late binding is an essential property to consider in building software by component assembly. The binary nature of components implies that either the components are designed and built knowing exactly how they will interface with each other, or that late binding will be used on potentially mismatched components. In the latter case the use of adapters would have to be used to mediate the interactions.

3.3 Software Deployment Environments

A Software Deployment Environment (SDE) is the environment in which components are first instantiated and their functionality added to the existing run-time environment of a system. This is different from both, development environments and run-time environments. In a development environment the major task performed is to develop new components. Many advanced development environments exist which offer developers an integrated set of services that include editing, compiling, execution, debugging, and testing. SDE is also different from the run-time environment where components are expected to have been already instantiated and running as part of an overall system. This definition places SDE as a phase between the initial development and eventual execution of components.

3.3.1 Automation in Component Deployment

Software components are deployable binary units that simultaneously offer services to their environment and expect the availability of certain services from their environment. Deployment requires that connections between the component and its environment be established. Our model of the run-time environment consists of a set of components that

interact with each other to implement a desired functionality. Therefore, the deployment of a component requires the establishment of the rules of engagement between the new component and the rest of the previously deployed components. This involves the establishment of links between the events generated by the new components and the methods of the components that wish to subscribe to this event. In addition, the new component may need to subscribe to events generated by the existing components. The relationship established for any new component defines the role it will have in the context of the existing run-time environment. A specification language can be defined with the purpose of using it to specify the interaction involving the new components within the context of their run-time environment. The specification should be executable to allow automation of the process of deploying components.

3.3.2 Processes to Automate

There are various tasks that can be automated in the deployment of components. At the most basic level we can specify the link between two components. This requires us to define a 4-tuple value consisting of an event along with the component responsible for triggering the event, a method, and the component containing the method. The deployment of a single, or group of, components require a set of links be defined. These link specifications could be grouped together to define a particular pattern or configuration of component interaction. This assumes that the interfaces involving the event and the method have been selected in such a way that no incompatibility exists.

3.4 Factors in Component Interaction

For components to communicate there has to be an agreement between them on the protocol they wish to use. At an implementation level the calling function must adhere to the signature of the function being called. The signature of a function consists of the following:

- Reference or name of the function
- The type and number of parameters
- The type of return value

When a function needs to be called, the function name or a reference to it identifies the function. The type and number of parameters are normally checked at compile time in statically typed languages, otherwise, the run-time environment dynamically checks if the actual and formal parameters are compatible. These requirements imposed by the language and enforced at either compile- or run-time forces programs to be a priori bound to specifics. While this may be a desirable feature from a strongly typed language point of view, it does not take advantage of the benefits derived from dynamic binding. In the following sections we address the issue of how late binding can be utilized in the context of software components to allow components to interconnect and avoid early binding. In particular, there are three aspects of component interaction that can be implemented with late binding: control, data, and function.

3.4.1 Binding Control

Control is the mechanism by which functions can be invoked. Having control over the execution of a function implies that a reference to the function is available and it can be utilized to invoke the function. In the push model (described in Chapter 5), functions

being are invoked explicitly. Instead, the invoker subscribes with an event that pushes the information to all subscribers of the event. The push model does not require early binding between the invoker and the function since the subscription mechanism allows the invoker to subscribe and unsubscribe to a function at any time during execution. The actual implementation of the link between the invoker and the function (or between the components to which the event and function belong) may require the creation of an adapter to mediate mismatch in the interfaces. The component in which the method is defined, creates for each event a method for subscribing and another method for unsubscribe.

3.4.2 Binding Data

Functions communicate with each other by passing parameters. The invoker of the function must match the parameters required by the called function both, in number and type. This requires an early binding between the two functions in order to conform to the type checking rules. This can severely limit the connectivity between components, not because the two components in question are defined on different semantic basis, but because syntactically they do not conform to the type checking rules. To remove this restriction, components have to agree to exchange data on a meta-semantic level. Rather than have to agree on the data representation in terms of the data types provided by the programming language, components have to agree on the meta-data. This allows data to be passed between components based on a common semantic understanding. The actual data can be encoded and represented as a string of characters. The encoding of the data also carries with it a description or meta-information about the data being sent. On the

receiving end, the string can be parsed according to the meta-information to extract the semantic contents of the passed message.

3.4.3 Binding Service

Components offer services to their environment and receive services from their environment. To request a service from a component directly, the requestor must normally be able to reference the provider of the service. This forces the requestor to be bound to the name of the service. To allow components a greater degree of independence, it is necessary to decouple the service-requesting component from the service-providing component while allowing the two components to interact when it is needed to do so at run-time. To accomplish this task it should be possible for the service-requesting component to discover and obtain a reference at run time to:

- The component providing the service
- The service within the component

Discovering the service-providing component can be accomplished through a broker service that may be available to components at run-time. This is similar to the broker service defined in CORBA. Alternatively, a directory service similar to LDAP can provide the necessary information at run-time. Once a reference to the service-providing component has been obtained, an exchange between the two components based on an established protocol will allow the service-providing component to inform the requestor of the service about the availability and extent of the service.

3.5 Meta-Semantics Programming

In order to decouple components and prevent unnecessary dependencies due to early binding it is necessary to specify the interaction between components at a meta-semantic level. At this level we are not describing the information shared by interacting components, but providing a description of the properties of the information. Therefore, programming required at this level is more abstract than in the case where programs explicitly specify the control, data, and functions to be used in the interaction between the components.

3.5.1 Function Meta-Semantics

A component provides its services through its functions or methods. At an abstract level it is neither desirable nor possible for components to be aware a priori about the precise information on the function to call within a given component. A protocol based on the semantics is established between components that wish to interact with each other. The service-requesting components use this protocol to initiate the type of request needed and the service provider component replies by specifying the meta-information on the service requested. From this information it will be possible to find out how the service can be requested. For example, the service-requesting component is able now to subscribe to the event associated with the service and thereafter will be notified.

The suggested common semantics shared by the components must be established on domain level. For each domain a set of vocabulary is defined which identifies the relevant terms in the domain. If we assume that a service offered by a component is based on information flow between the component and its environment, then sending a

request represented by a domain term will allow components to understand what information is being requested (Figure 3.4).

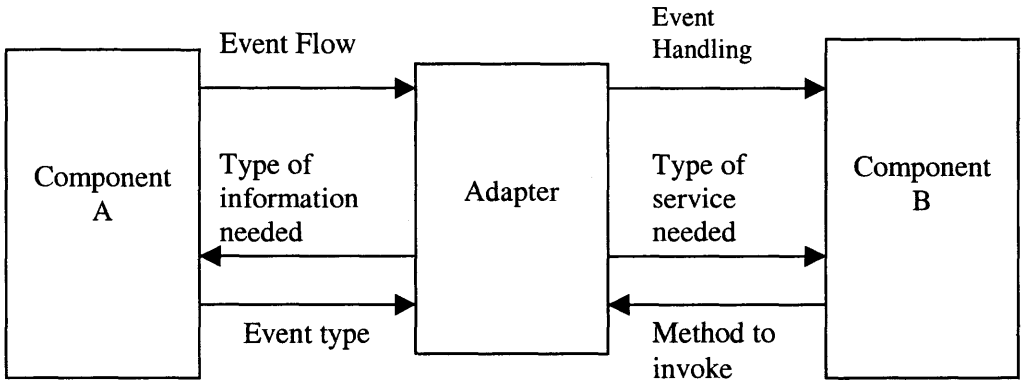


Figure 3.4 Adapter for unspecified event and handler

3.5.2 Data Meta-Semantics

Data meta-semantics refers to the meaning of the data semantics being communicated between the components. We make a distinction between the types of the data, or the semantic content of the message, and the type associated with the event being communicated between the components. The semantic content of the data is encoded using the domain vocabulary and we assume that cooperating components have a common understanding and agreement of the domain vocabulary. On the other hand, the type of the event sent might not be compatible with the type of event expected by the receiving component. This is not necessarily an undesirable situation since it imposes fewer restrictions on the design of the components. The objective is to have a balance between absolute conformity and no standardization. The first ensures compatibility, but will be difficult to enforce. The second will lead to mismatches between components that might be too difficult and costly to overcome.

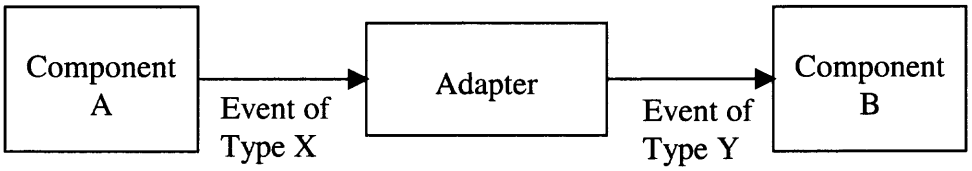


Figure 3.5 Adapters for Event Mismatch

3.5.3 Control Meta-Semantics

For two components to interact, it is expected that one component will register with the other component for receiving event notifications. But for a component to register it must implement a specific interface required by the event-producing component. Since no assumption is made about the usage of these components, the event-receiving component could not have been built with the required interface. The adapter can mediate the mismatch by implementing the required interface to make the adapter a registered recipient of the event. The adapter implementation of the method receiving the event is to forward it to the intended target component.

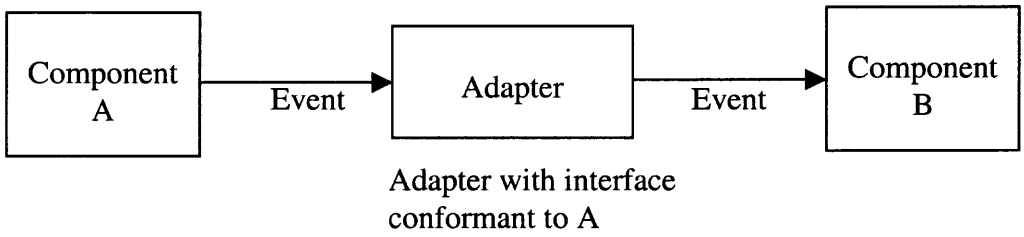


Figure 3.6 Adapter for Control Mismatch

3.6 Language Specification

A meta-semantic language allows the specification of a system at an abstract level. Using late binding with function, data, and control, it will be possible to describe the system at an architectural level by abstractly specifying the required components and the way they will be interconnected.

3.6.1 Elements of the Language

There are several important entities to describe in this language:

- Deployment environment: The deployment environment consists of the environment in which components are instantiated. The main task of the deployment environment is to communicate with the run-time environment of the deployed component about the availability of the component. The deployment environment communicates primarily with a cluster component of which the component will be part.
- Cluster: A cluster is a component whose task is to manage the configuration of a set of components belonging to the same logical group. The main reason for grouping components into clusters is to have a single coarse-grained representation of the group, which allows a centralized management of the component. Typically, components within a cluster have high degree of coupling and cohesion between them.
- Adapter: An adapter is a software component that is dynamically created to mediate between mismatched components. Adapters are created to specifications based on the interfaces of the components that need to interact. The cluster

component described above is responsible for creating the adapters needed for the components within the cluster to communicate.

- Components: Components represent the basic functional units that provide services to other components. It is assumed that components cannot be modified, but do allow customization consistent with their functional specifications. A component implements one or more interfaces, where each interface consists of a set of event and methods.
- Event: Events are associated with components and clusters. An event is a source of information to which other components can subscribe when they need to receive the information.
- Method: A method is a function defined within a component, which can be accessed by other entities through the interface of the component.

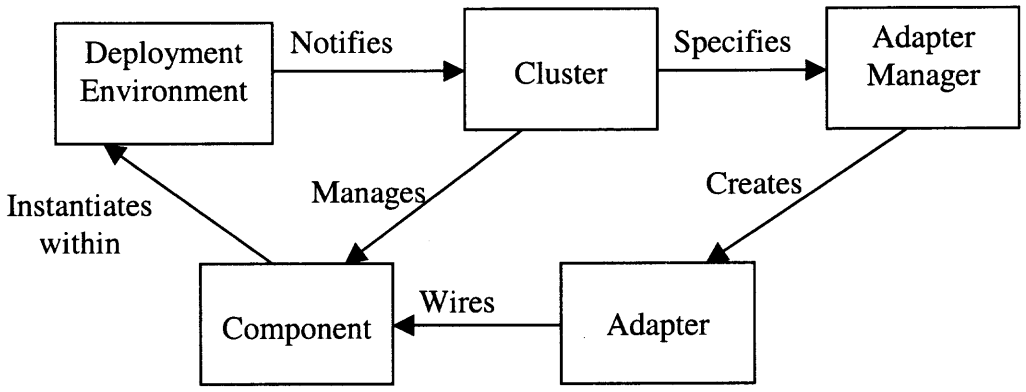


Figure 3.7 Run-time Activities

3.6.2 The Semantics of Clusters

The language is designed to allow the user to convey the manner in which components will interact with each other. This is different from specifying "how" the components are

to be interconnected. The "how" involves an understanding of the interfaces at a programming language level and executing specific instructions that allow two components to communicate with each other. This normally requires tedious work at a low-level coding and is a labor-intensive activity. A cluster is responsible for generating adapters at deployment-time. However, clusters are implemented as components with one or more interfaces allowing them to interact with other components (including other clusters) and to be included in a hierarchical structure of clusters.

3.6.2.1 Promoting Events and Methods

The definition and functionality of a cluster is derived from the functionality of the components within the cluster. A cluster may define its own set of events and methods within its declared interfaces, or it may make the events and methods of its components visible through its interfaces. Taking an object-oriented approach, components interacting with a cluster need not be aware of the internal structure of a cluster. The events and methods "promoted" from the inner component interfaces to the cluster interface appear as the cluster's own event and methods. In the case of event promotion, the cluster must create an appropriate interface to allow external components to register and un-register with the event. The request for registration are not handled by the cluster, but passed on to the component on which the event is declared (Figure 3.8). Promoting a method from the interface of an internal component to the interface of the cluster requires the cluster to pass all invocation requests it receives on the function to the component implementing the function

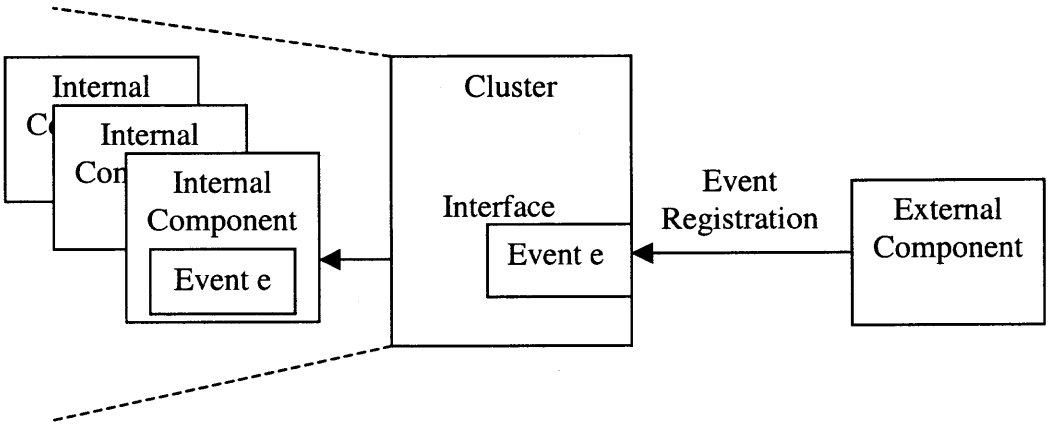


Figure 3.8 Registering with a Cluster Event

Cluster components follow a simple procedure in wiring the component members. As each component member is instantiated the cluster receives a notification from the deployment environment containing a reference to the component. The cluster maintains a "blue-print" of the way its components are to be interconnected. When all of its components have been instantiated, the cluster creates an appropriate adapter and uses the reference to the components to perform the wiring.

3.6.3 Adapter Manager

The role of the adapter manager is to receive the specification from a cluster component about the interconnections required among a set of components. Using introspecting the adapter manager first probes into the involved interfaces of the components and carefully constructs the adapter as a class. The code of the adapter is generated and written to a temporary file. The file is compiled and the class is loaded to the run-time environment, from which an object adapter is instantiated. The adapter manager returns to the cluster component a reference to the adapter. The cluster component uses the reference to the

newly created adapter to wire the components to the adapter. The following describes some of the major steps taken to create an adapter:

- Include type declarations of any external entity that is used within the adapter. This includes class declaration of all components that are part of the interaction being handled by this adapter.
- Include class type declaration of all parameters used in the methods that are declared within the adapter. This includes the class type of all event messages communicated between the components.
- The adapter must implement the interface of each event-generating component. This allows the adapter to subscribe to all the events that are of concern. The adapter is then able to receive the events and forward them to the appropriate components.
- Include methods to initialize the references to all components that are the target of events. These initialization methods are later used by the cluster component to make the initializations as soon as the components needed for the initializations have been instantiated.

3.6.4 Event Management in Adapters

The specifications may require that a certain type of events from one component be delivered to another component. The following are some of the possible situations that may be encountered concerning the compatibility of the message type between the sender and the receiver component:

- If the method handling the event in the receiving component does not expect the event to contain any message or information then any message content of the

event is ignored. This case applies when the intent of the interaction is to simply send a notification to a component.

- If the event-receiving method expects a specific type of event with enclosed information, the event-generating component has to send an event with compatible type. Except for the case described next, failing to agree on an event type will result in failure to generate an adapter.
- When the type of event clearly identifies the semantic content of the message accompanying the event, the adapter manager will construct a new event based on the type expected by the receiving component. The semantic content of the original event is extracted and transferred to the new event. This is an accepted solution, given the assumption that the content of the message will include meta-information from which the receiving component will be able to extract the intended information.
- In some situations a component requires a reference to another component as soon as the latter one is instantiated. We refer to this type of event as the "null-event". This type of event is specific and requires that the receiving component implement an event-handling method that accepts this type of event. This will allow the adapter to be constructed in such a way that it can generate these types of events. The component whose instantiation will trigger this event is not involved in this interaction and will normally be unaware of the event that the adapter fires on its behalf.

3.7 Generating Adapters from Abstract Specifications

Automating the process of creating simple adapters may be technically challenging but does not provide a complete solution to the problem of component integration. The process involves having at the component design time knowledge about the interfaces of the components with which interaction will be expected. This is not a likely or desirable situation since it relies on a great deal of standardization, which will prohibit competitiveness in a component market and limit creativeness in the design of new components. However, establishing guidelines and generic frameworks for the structure of components and the messages exchanged by components can provide a basis for the generating "smart" adapters from high-level and abstract specifications.

3.7.1 Standardizing Event Structures

Events serve the purpose of delivering notifications to interested components about the occurrence of certain events. Often times an event carries information with it to inform the receiving components relevant details about the event. In the absence of any guideline about the structure, including the semantic content, of the event it may be very difficult to find basis for mediating between two communicating components. A reasonable guideline is to embed the semantic content of the event as a standard field with no visible structures from the language-typing point of view, such as a string of characters. The string may have arbitrary structures embedded within it that can be discovered through parsing of the string. It is of no importance what is the exact type declaration of the event or whether the two components agree on that type. The adapter will be able to detect the type mismatch and handle the situation in a straightforward manner. This can be done by

extracting the semantic content of the received message and embed it in a newly created event of the type expected by the receiving component (Figure 3.9).

3.7.2 Standardization of Components

Currently, component models are not sufficient to support an automated process of creating smart adapters. Interconnecting component requires specific information about the interfaces of the components. It is unlikely that at the design time of a component we will know the detailed interface specifications of all components with which our component will interact. Components make their functionality available to other components through a set of services defined in their interfaces. While it may be reasonable to assume that components of certain functionalities will exist, the interfaces to these components may not be known, except for a general abstract knowledge about certain services that they provide.

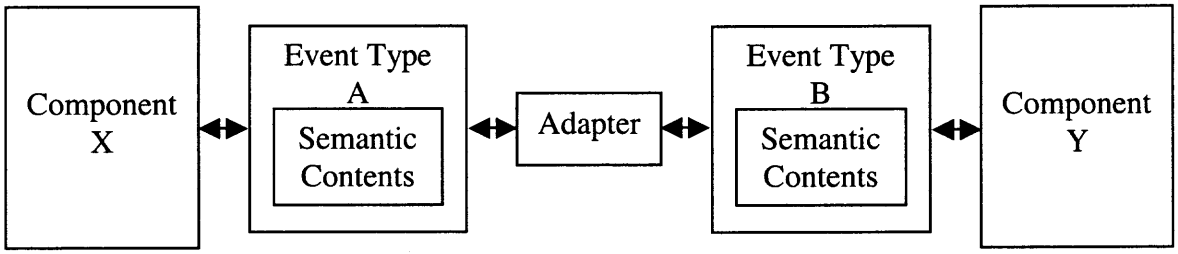


Figure 3.9 Adapter for Mismatched Event

This suggests that there has to be semantic level standards that can serve as a basis for communication between components. More specifically, a domain vocabulary has to be established such that components developed for that domain can agree on the

vocabulary and associated semantics. For example, in a business domain, the term "invoice" should carry the same semantics for the component "sales" as it does for the component "accounting". Even when these components are to be developed independently by different organizations, the term "invoice" should still be agreed upon. This implies that when we query these components about invoices, they should each understand what is being requested. In the case of the "sales" component, we might be interested in sending all the generated invoices to the "accounting" component for processing. It should be possible to query the "sales" component for an event to which other components can subscribe in order to receive the invoices. Based on the result of the query the adapter can be partially constructed. A similar reasoning applies to the "accounting" component if the method that handles invoices was not known. A query to the "Accounting" component should give us the signature of the method we need to invoke and pass the invoice events to it for processing. The result of the query can be used to complete the definition of the adapter for establishing communication between these two components.

3.7.3 Support for Higher-Level Processes

The relatively high level of abstraction with which we are able to specify the interconnection between components (as discussed above) suggests that we may be able to move the level of specification from the software development domain to the user's problem domain. It should be possible to define an abstraction layer that deals with the tasks and processes that the user needs to accomplish to meet certain objectives within the organization. These tasks and processes can be already mapped into software components that provide implementation for them. The user can describe the composition

of task and processes that are required using a domain-based specification language. The specification can then be translated into a set of components and adapters that can be put automatically together to generate the system that the user had requested.

CHAPTER 4

A MODEL FOR SOFTWARE COMPONENTS

In this chapter we define a model for software components. We begin by providing an understanding of the object model since software components can be viewed as an evolutionary step in the development of objects. We describe the benefits and limitations of the object-oriented paradigm in the software development process, particularly in the design phase. While the concept of objects provide a strong context for software reusability, systematic object reuse has not materialized, as evidenced by the limited number of object libraries in existence today. Software components, on the other hand, overcome the limitations of objects by expanding on the object model to include a tool-supported development environment that is capable of manipulating components at design time. This will simplify the assembly and integration of components and remove the labor-intensive and error-prone manipulations that are often required to achieve reuse with objects.

4.1 The Object-Oriented Model

The object-oriented paradigm defines objects and classes as its basic elements. Classes can be used to implement the concept of an Abstract Data Type (ADT). Although the definition of object varies among researchers, the following properties are commonly stated: encapsulation, data hiding, inheritance, polymorphism, and dynamic binding. The type of an object is described by it class definition. A class provides the blueprints for instantiation of objects and determines the properties and behavior of these objects. Each object has a unique identity and can be characterized by its state at run time. The methods

defined on an object can alter the state of the object when invoked. Objects use the message paradigm to describe the invocation of methods. To use the services of an object we send it a message, where a message is equivalent to a function defined on the object. It is possible to send values as parameters when sending a message to an object. Data encapsulation refers to the syntactical grouping of the data and functions defining the object into a single structure. Data hiding refers to the controlled access to the elements of an object (attributes and functions) to enforce the concept of an ADT. Data hiding serves to separate the internal implementation of the object from its public interface. Inheritance is one of the major contributions of the object model. It allows the definition of a new class of objects to be based on the definition of an existing class of objects. The inheritance relationship defines the derived class to have all of its attributes and methods of the base class. Any of the inherited attributes may be overridden arbitrarily. Additional attributes and methods can be defined on the derived class as needed. The type of the derived class becomes compatible with its base class, allowing objects of the derived class to appear anywhere objects of the base class appear. Inheritance plays an important role in the reusability aspect of objects. Polymorphism is a property that allows the reference to be associated with different objects at different times during execution. Dynamic binding allows the name of a method to be bound to one of several functions. Many of the object model benefits can be derived from the combination of inheritance and dynamic binding.

4.1.1 A Unifying Model with Objects

One of the major impacts of the object-orientated paradigm on the software development process is in providing a unifying model in all phases of development. Object-orientation

introduced a major change in software development methodologies that requires shifting the focus from a functional mindset towards one based on objects. This happened at a time when the limitations of the waterfall model were becoming increasingly apparent. Alternatives, such as the spiral model, have been suggested to include iterative steps into the process. One of the major drawbacks of traditional software development has been the lack of a unifying model that integrates the various phases of the software development life cycle (Korson and McGregor, 1990). Objects, as the primary elements of representation, are used to express the analysis and deign of the system. The objects that appear in the analysis phases are likely to appear again during design. This continuity helps designers in making the transition between these two phases. This is in striking contrast to traditional software development where, for example, the outcome of a data flow analysis is to be used as a basis for a functional decomposition during system design. This common semantics framework includes the implementation phase since objects can be directly implemented into some programming languages. Object-oriented development methodologies (Coad and Yourdon 1990; Yourdon, 1994; Booch, 1994) have all capitalized on this semantics-based integration to simplify the development process.

4.1.2 Reusability of Objects

One of the advantages of the object model over the functional model in software development is in reusability. Functions are not good artifacts for reusability in general. With the exception of mathematical libraries and other few specific domains, there have not been general-purpose libraries. Objects on the other hand have a semantic basis for reusability. Objects represent real world entities in the problem domain and are more

likely to be used across software projects. A good measure of reusability in objects can also be realized with abstract classes and class templates. An abstract class represents the definition of an incomplete class that can be used as a basis for inheritance in order to define a customized version of the class. A class template is a parameterized definition of a class with respect to one or more types. It allows the creation of a new customized class through type specification.

4.1.3 Object Support for Good Design

A design describes the decomposition process of a problem. The decomposition of the problem is based on abstractions represented by design artifacts. The procedural abstraction has been widely used in decomposing a problem into a set of hierarchical tasks. A design based on the object abstraction decomposes the problem into a set of cooperating classes and objects. In both cases, the designer needs to identify the set of elements, functions and objects, which are able to solve the problem. In the case of objects, the designer identifies the relevant entities in the problem domain and seeks to represent them as objects in the solution domain. There is a one-to-one correspondence between the problem entities and the objects of the design. This is one of the desirable properties of the object-oriented design paradigm since it establishes a way to track the user requirements into the design phase and vice versa. In the procedural-based approach, the relationship between the requirements and the design artifacts is much more difficult to establish. Most object-oriented design methodology is based on the following basic criteria:

- Identify the entities in the problem domain and represent them as objects in the domain model and design

- Establish the relationship between the objects. This includes all types of relationship, collaboration and inheritance
- Iterate over the set objects and relationships so as to obtain a good model of the problem

Object-oriented design offers many good design properties (Korson and McGregor, 1990) such as:

1. Abstraction: An object is an abstract representation of an entity in the problem domain. Objects hide many of their details so that the designer only has to be concerned with the appropriate level of abstraction.
2. Modularity: Objects provide a simple criterion for modularization based on a single or a set of collaborating classes.
3. Weak coupling between objects: By definition, an object is self contained capable of maintaining its state and can manage its resources internally. Therefore, collaboration between objects is specified along a client/server model where an invocation on an object is based on a request of service. This tends to minimize the amount interaction between objects.
4. Strong cohesion within an object: this follows from the same argument given for the weak coupling between objects. Most of the actions performed are contained within each object.

4.1.4 Design Limitations with Objects

The object-oriented model imposes several limitations, as the size of the system becomes very large. The same semantic properties that made objects useful as a basis for design

start to break down as the number of objects and their relationships increase within the design of a given system. Below are some of the important drawbacks associated with large system designs that use objects:

1. The inheritance property of objects is an important aspect of the object-oriented model that makes the relationship between objects of the design explicit and which promotes reusability of code and design. The inheritance relationships between the classes of a design define a set of class hierarchies to which classes can belong. Each hierarchy defines a set of dependencies that relates a class to all its ancestors within the hierarchy. Changes made to a particular class will affect the definition of all the descendent classes. The complexity associated with keeping track of changes can be overwhelming to maintain and can present a serious challenge to the system (Ousterhout, 1998).
2. Inheritance violates the concept of information hiding. When a class inherits from another class, some of the hidden elements of the base class become visible to the derived class. The dependency on the base class that is based on implementation violates one of the essential properties of classes in support of information hiding.
3. The complexities associated with a design that has a large number of objects can be overwhelming and present some of the same problems associated with unstructured code. An object-oriented design representation shows the objects and classes along with the relationships between them. The lack of structure above the class level to organize the classes in a hierarchy, which can make it easy to understand, is a serious drawback to the usefulness of the design. In this regard, the design methodologies based on the procedural paradigm can offer a better solution to the problem of size.

The procedures representing the system tasks are hierarchically organized by the design and present a more manageable challenge.

4. The granularity of objects is often low and therefore does not scale up to the architectural level of abstraction for a large system design. One of the attractions to objects is in their ability to represent real world entities and thereby relating the various phases of system development on a semantic basis. The object model has no inherent limitations, which prevents us from defining coarse-grained object entities. However, the semantic association of objects to real world entities and the data-centered approach to object definition tends to keep the granularity of the objects small and therefore unsuitable for architectural specifications.
5. The behavior of objects is defined using the functional model. The relationship between objects is often implemented as a function call from the object requesting a service towards the object that is providing it. The explicit reference made to the service provider object from within the client object creates a source code dependency for establishing the relationship. This can limit our ability to use objects in a compiled binary form that is suitable for programming by object composition.

4.2 The Software Component Paradigm

As the size and complexity of software systems increases, there is a growing need to find software methodologies and design artifacts that can be applied to the development of such systems. The traditional approaches based on the procedural paradigm are limited in providing reuse artifacts (Korson and McGregor, 1990). Functions and procedures specifications are too dependent on detailed information to be easily reused. With the exception of some mathematical libraries and low-level functions, there has not been

wide usage of functional libraries. The object-oriented paradigm starts to show weaknesses when applied to large systems. As the number of classes and their interrelationship increases so does the complexity of the design. The lack of higher-level structures to group objects diminishes our capability for using objects at the architectural level of design. Software components have evolved from the object-oriented view of system development. Components address many of the shortcomings of object and retain some of the advantages of the procedural abstraction.

4.2.1 The Role of Components in Design

Components are coarse-grained functional units that can be used at the architectural level of design. There are several reasons for the emergence of components as the next viable technology to take us beyond objects. Some of the reasons given for the relevance of components in software development are (Clements, 1995; Brown, 1996; Szyperski, 1998; Krieger and Adler 1998):

1. Components can be used as architectural elements from which systems can be designed. We can design a system by specifying its components in terms of the functionality and interface that each component will have. A small number of these large-size components can provide an initial basis for describing the architecture of a system.
2. The architecture of a system must take into account the distributed nature of applications and the heterogeneous environment in which applications are increasingly being deployed. The client-server paradigm model has become popular as networks and computing have increased in speeds and reliability. The distributed nature of many organizations with distributed resources can no longer rely solely on

the mainframe approach to run the organization. The client-server architecture reflects the distributed aspect of many of today's corporations and takes advantage of many of the benefits of distributed systems, such as reliability, performance/cost, scalability, modularity, etc. (Mullender, 1989).

3. The rapid advances in technological innovations play an important role in the competitiveness of organizations. A monolithic software system can have limited capacity to be able to assimilate new technologies. A component-based system allows us to take advantage of any new technology to upgrade or improve a system's capability without requiring an extensive effort or a major reengineering process.
4. Interoperability between systems is becoming an essential requirement for an organization. There is much to gain from integrating the systems-based tasks within an organization to reflect an integrated business process. Interoperability is not only important within an organization but also to link system along the supply chain to the customers and suppliers of an organization (Chesbrough, 1997; Baldwin and Clark, 1997; Jololian, 1998). Heterogeneity is the rule rather than the exception, as many organizations have diversified their computing environments. Components can isolate and hide the platform dependencies between the various parts of the system.
5. There is a growing demand for shorter time-to-market, higher reliability, and lower cost for software development. Traditional software development cannot meet these challenges without an evolutionary step that can improve by an order of magnitude our ability to develop increasingly larger and more complex systems.

4.2.2 A Model for Components

Software components are an evolving technology that is currently lacking widely accepted standards. Various models for software components have been suggested. In this section, we present a set of requirements and constraints for software components that will motivate our model for components. The following are the requirements basic requirements on software component deployed (Brown, 1996; Clements et al., 1995; Whitehead et al., 1996; Krieger and Adler, 1998):

1. Components are executable units of composition that can be independently deployed. This is an essential requirement that expects components to be distributed in binary form and to be assembled without the need to have access to source code. This requirement implies certain constraints on the architecture of components, which will be described in the following section.
2. Components are reusable elements developed by third parties. A significant benefit of components is in our ability to reuse them in developing new systems by composition. Domain information is an important aspect of the semantic definition of components. We assume the availability of a market for components where it will be more economical to purchase a component, conforming to domain and technical standards, rather than developing it.

There are two important implications that we can derive from the above requirements if components are to be successful in meeting our requirements (Szyperski, 1998). First, there must be a technical standard for wiring components so that we are able to connect two components without extra adaptations. Second, There must a domain standard for

defining the semantics of components, so that the development of components is free from any architectural mismatch (Garlan et al., 1995). Of the previous two conditions, only the first one has been addressed. The current wiring standards include Sun's JavaBeans, the Object Management Group's (OMG) CORBA, and Microsoft's COM. Although these standards are not directly interoperable, it is possible by using adapters to allow two heterogeneous components to communicate. For example, with CORBA, JavaBeans and COM components can interoperate within the same system (Krieger and Adler, 1998).

Given the above requirements on software components, A set of tools and development environment is needed to help system developers in meeting their objectives: An integrated development environment (IDE) in which users can develop their applications (Krieger and Adler, 1998), a browser for allowing developers to select components of interest, and component customization editors to allow the user access to the properties of the component so they may be edited according to needs.

4.2.3 Design Elements of Components

The following paragraphs identify the major elements of a component: interface, properties, methods, and events.

4.2.3.1 Interface

A component interface defines the accessible characteristics of a component. The services provided by a component are accessible through one or more of its interfaces. The services are usually implemented as functions, which can be invoked directly by the client through the function calling mechanism or indirectly through the registration mechanism associated with events. The interface of a component can also provide

mechanism that allows the developer to access the customizable properties of a component. The properties are a set of values that a component makes publicly accessible to other components.

4.2.3.2 Properties and Methods

The properties and events defined on an interface of a component provide us with the means to access the functionality of a component. The methods are used to define the services provided by the component. The properties define the accessible elements of a component that describes its state partially. Customization of the component can then be achieved through these properties. Properties can be implemented as variables of various types, which may be single or multi-valued. Properties may be changed through functions that may be invoked to set or alter the values associated with the property.

4.2.3.3 Events

The invocation of methods within objects is based on the function call model. This is not the only way function invocation can take place. Event-based callback is another way a function can be invoked. This is the approach used by many graphical user interface libraries, such as the X library and the Motif widget set. A function that needs to respond to a user action on a particular user interface element, such as a button, is registered with the element. When the user interacts with the graphical element, the handler function is called to respond to the user's action. This event-based invocation of functions could be used with objects in general. However, events have not been used as part of the object model and most object libraries do not include them. With the exception of the graphical user interface builders, such as Visual Basic, object-oriented code uses the forward function call. Judging by their relative success, GUI builders have demonstrated that

combining the event-based callback model with objects is a powerful concept that can make objects reusable.

Relying on function calls alone without the callback mechanism, leads to a model in which building programs from existing objects requires source code manipulation. The following example demonstrates this point with two object classes OrderEntry and Billing. An OrderEntry object implements an interface that allows users to input their purchase order. A Billing object computes the billing information based on the purchase order it receives. The function in OrderEntry that performs this task is submitOrder(). Assuming the two classes exist, the code in OrderEntry must be modified to include an explicit call to the submitOrder function of Billing.

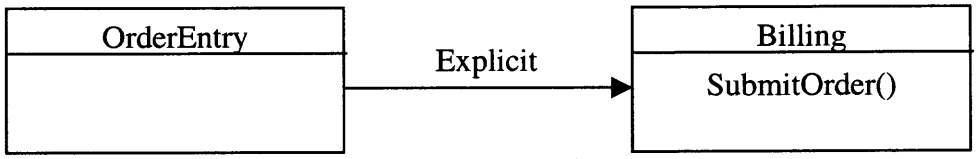


Figure 4.1 Forward Calling

By contrast, using events and callbacks the definition of the object OrderEntry is modified to include an event orderComplete which is triggered when the user completes the purchase order. The class OrderEntry also allows other classes to become listeners of this event by registering a calllback function with this event. In this case neither class needs to be modified, instead a code external to both classes can perform the necessary registration of the callback.

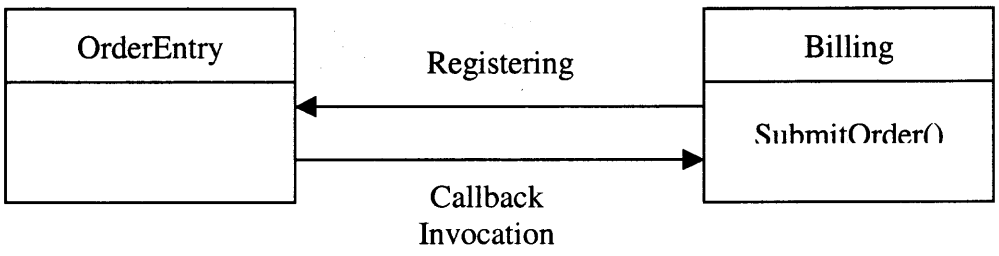


Figure 4.2 Callbacks

4.2 Summary

In this chapter we have presented two models: object-oriented and software components. These models, although different, share many characteristics. The benefits of the object-oriented model in the design of software are clearly related to the notions of Abstract Data Types (ADT) and inheritance. However, as we have explained the object-oriented model as it has been implemented in languages such as C++ offers only limited potential for software reuse. The most obvious limitation is in the difficulty involved in object composition. Software components build on the notion of abstract objects by limiting all interactions to the services provided by the interface(s). In addition, components are executable units that can be deployed into run-time environments. The challenge to component designers is to find a model that allows component composition without requiring changes in the component implementation. Consequently tools have a significant role in automating the assembly of components.

CHAPTER 5

CLUSTERS

A cluster is a configuration of components that interact with each other in a specified way so as to provide a desirable functionality to its environment. Components within a cluster have no a priori knowledge of the particular configuration constraints that will be imposed on them by being part of the cluster. Clusters allow us to define higher-level abstraction based on the composition of the member components. In this chapter we show the potential use of clusters in the design of systems by providing the developer with a powerful and yet simple tool for creating abstractions based on design patterns.

5.1 Component Representation of Channels

5.1.1 The Push/Pull Model

Programming is based on a set of abstractions implemented by a programming language. These abstractions define a model with which design solutions can be formulated. Traditional programming is based on function calling where the caller initiates the invocation of a function. This is referred to as the pull model where the client function actively requests an event from the server function. The caller transfers control to the callee to allow it to execute a task. When the callee ends its execution, it transfers control back to the caller. The standard client/server model uses this approach for communication. The model has also been used for communication between objects, inside a single program or in a distributed environment as defined by CORBA for example. This type of communication between functions or methods of objects is

considered a synchronous form of communication. A different model for programming can be based on event communication. This approach can be integrated within the object or component models. It is referred to as the push model where the occurrence of an event in the server object triggers an invocation of a method within a client object that had previously requested this notification. This form of communication is often used in applications involving user interactions where users' actions arrive asynchronously.

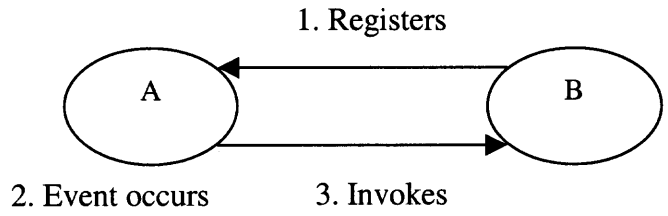


Figure 5.1 Event-based Interaction

5.1.2 Channel-Based Communication

CORBA supports the event and notification model and implements the concept using an architectural element called the event channel. The event channel becomes the intermediary between a set of objects that are event producers and another set of objects that are event consumers. Event consumer objects register their interest with the event channel in receiving events. Producers send the events they generate to the event channel, which will then be distributed to all registered consumer objects. The role of the channel is to decouple the direct interaction between the producers and consumers. The channel acts as a single consumer to all event producer objects while acting as a single producer to all consumer objects.

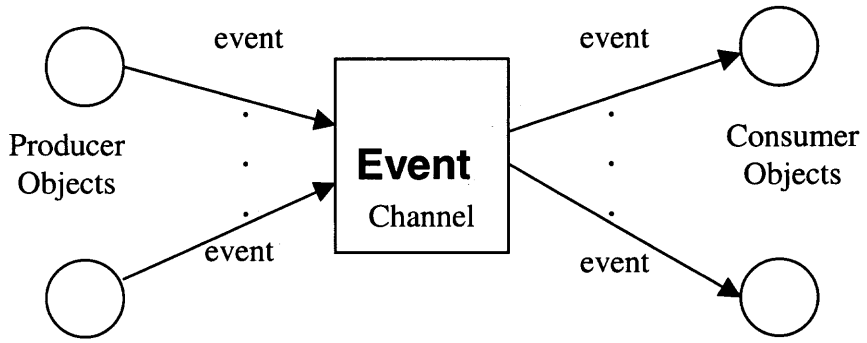


Figure 5.2 A Channel

An example of the event channel model is the display of data graphically within a document. The data used by the graph-producing object depends on various external sources. When any of the external values change a notification is sent to the graph handling code to revise the presentation of the graph. The event received by the graph object can also be of interest to other parts of the document creating a multiple-producer/ multiple-consumer situation.

5.1.3 Extension of the Channel Model

The event-based model as described above provides a basic functionality and serves as a basis for implementation. The model can be extended to provide added functionality beyond event notification and offers a set of services for controlling and monitoring the actions within the channel. The control mechanism acts to customize the transmission of the events according to the programmers' specific needs. The monitoring of the events that go through the channel can provide a mechanism to observe the performance of the system as measured from the channel point of view. CORBA implements a similar

model through its notification service. Orbix is an implementation of CORBA that extends the concept with their OrbixNotification to provide two main features consisting of event filtering and Quality of Service (QoS). The basic notification model delivers all events arriving to the channel to all consumers registered with the channel. We can choose to apply a filtering mechanism that allows only certain types of events to go through the channel while rejecting other events that do not meet particular criteria. Other mechanisms are available to control the delivery characteristics of event messages. We can use properties such as reliability to control the delivery policies of messages. In distributed environments where the delivery of message through physical networks is not guaranteed, mechanisms in software can be introduced to implement any one of a number of policies such as:

- At least once
- At most once
- Exactly once
- Best effort

In a best effort case, no promise is made that the event will be delivered to every consumer. A consumer may even receive the event more than once. Other services can also be used to specify properties of event delivery such as:

- An expiration time that determines the time period within which the delivery of the event will be allowed.

- The earliest time to deliver the event. If the event arrives at the channel at an earlier time it will be held until the time specified before it is delivered to the consumers of the channel.
- The order of the event delivery can be specified so those events can be delivered in an order other than their order of arrival. A priority scheme can be specified on the events to create an order of delivery.

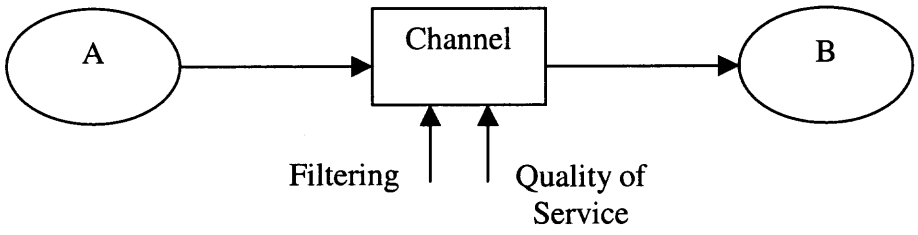


Figure 5.3 Filtering and QoS on Channels

The decoupling of the communication between two components based on an event channel is a starting point to control the communication between the two components. The focus shifts from either a caller-oriented or callee-oriented model to one where the channel is the primary element describing the interaction between components. In the absence of a channel, the filtering of events and the quality of service described above could still be used by directly applying it to either or both ends of the communication. This would mean that the properties of the communication we choose to use between any two components has to be integrated within the component, putting more constraints and variability on the definition of the component and limiting its reusability potential. By removing the properties of the communication from the components and describing it at the channel level, we can apply any combination of

properties on the communication between the components, as it may be dictated by the situation in which the components are deployed.

5.1.4 A Component Representation of a Channel

A channel captures the communication between a set of components. In the above case we considered a simple pattern of interaction that allows components, by registering with the channel, to receive notifications when events arrive to the channel from the event generating components. This type of interaction is also described by the observer pattern (Gamma et al., 1995). The interaction diagrams below show the interaction between the observer and the subject for the two cases: observer pattern and channel.

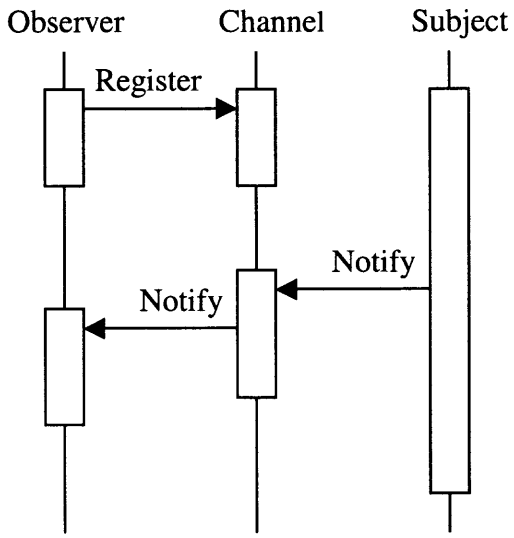


Figure 5.4 Interaction Diagram for a Channel

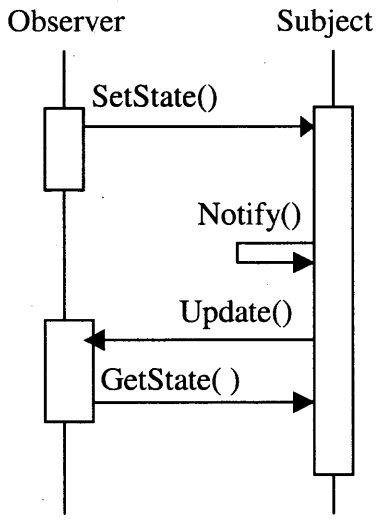


Figure 5.5 Interaction Diagram for the Observer Pattern

There is a major difference in the pattern of interaction between the two models. In the observer design pattern as described by (Gamma et al., 1995) and shown above, a notification does not carry any information beyond the fact that a particular event has occurred. There may be information that the observer would need along with the event notification. This is implemented in the observer pattern by having the observer explicitly invoke a method on the subject that triggered the event requesting the state of the subject be passed. The observer is then able to act on the event. In the Channel model we assume that the event notification carries with it the necessary information that the observer would require.

We can consider the channel as a controller for the interaction between an arbitrary set of components. The pattern of interaction is abstracted from the components so that the components are no longer dependent on each other directly. All of the interactions that each of the component will have is now restricted to the channel only. The channel has an internal representation of the interaction between the components. The behavior

of the system is no longer determined by the wiring of the components, but instead is determined solely by the channel. The components are each wired to the channel and identified by the functionality they provide. As an example, consider the pattern of interaction in the Model-View-Controller design pattern. To simplify the model we can assume the interaction between the three components to be limited to the following:

- When the Controller component receives input from the user that requires a change in the model, it fires an event that results in the Model component being notified about the change. Because of this event, the Model component modifies its internal representation.
- When the internal representation of the Model component changes, it fires an event, which is received by the Viewer component. The event signifies a change in the model requiring a corresponding change in the view.

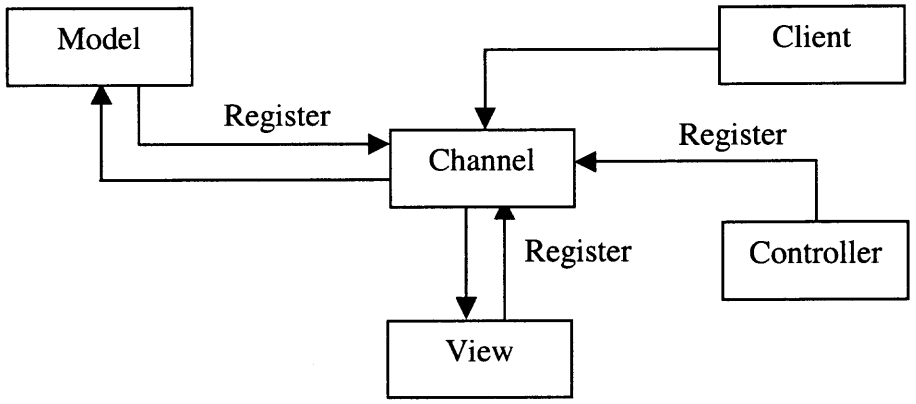


Figure 5.6 Channel Representation of the Model/View Controller Pattern

The above diagram (Figure 5.6) shows the relationship of the individual components of the Model-View-Controller to the channel.

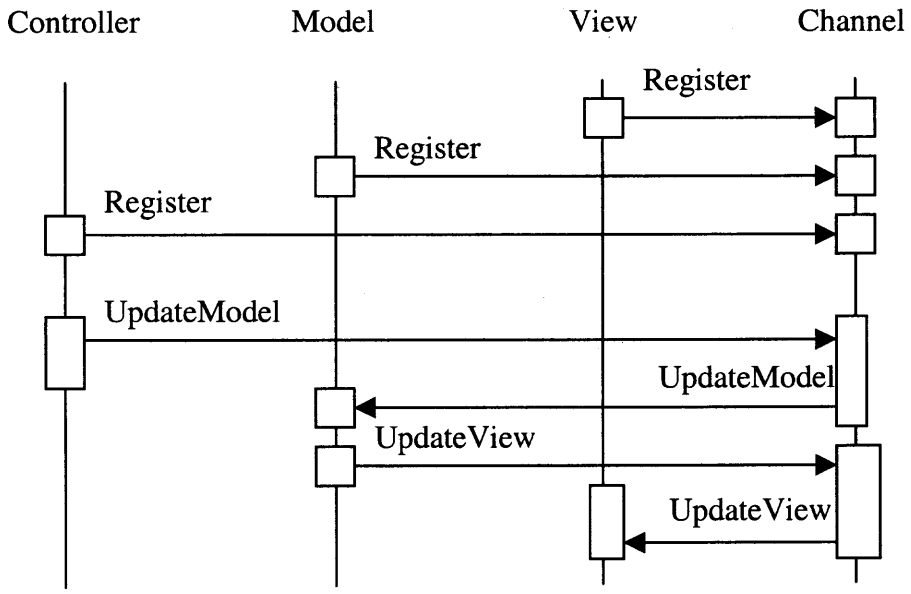


Figure 5.7 Interaction Diagram of the Model/View/Controller Pattern

5.2 Design Environment

The design of a system consists of a decomposition process that transforms a problem into a set of cooperating components. The level of complexity of the problem often requires that each of the components be further decomposed in an iterative fashion to yield components whose implementation are relatively easier to perform. The structured analysis and design techniques follow a similar pattern of problem decomposition. Each level of decomposition represents an abstract layer of the system. The decomposition process is often applied to processes, similar to what is done in the procedural paradigm. At the highest level of abstraction, the system is represented by a single task. However, through iterative decomposition we can replace each task by an equivalent sub-process that specifies how the task can be performed. Similar decomposition can be applied to other design artifacts to yield comparable simplifications based on abstraction layering.

For example, objects can be grouped to represent different abstractions of the system. The grouping of the objects can be based on the level of interactions between these objects. Thus, several objects may cooperate to provide a service or functionality of the system. DeChampeaux has investigated the identification of such cooperation between objects, which he refers to as ensembles, during system analysis to define the object clusters that can provide a layered model of the system (Wirfs-Brock and Johnson, 1990).

The components of a cluster are at a lower level of abstraction than the cluster itself. The cooperation among the components within the cluster gives it a higher-level semantics even when the cluster does not provide any additional functionality beyond the ones offered by the constituent components of the cluster. However, such added semantics is possible and can further extend the functionality of the cluster. Once we identify a set of components for the definition of a cluster, our task is to give a representation for this cluster that will enable it to interact with other elements of the design. The representation of the cluster should be selected in such a way that it does not introduce new semantics that can increase the complexity of the design. A uniform representation of the cluster allows it to blend into the design and combine with other components in a consistent manner. The uniform representation of the cluster implies that the cluster must be treated as a component with an interface that allows other components to use its services. The cluster must also be able to call the services of other components with which it can cooperate. The services that the cluster provides are offered through its interface and define its semantics. These services of the cluster define the functional behavior of the cluster with respect to the overall design. Non-functional attributes, such as performance and reliability can also be defined on the cluster as a

whole to meet similar goals on the overall design of the system. Both of the functional and non-functional requirements have to be achieved by the function and characteristics of the individual components. The services of the cluster are mapped to the services of the individual components. There are at least two ways that we can define the services of the cluster:

- We can map a specific service of the cluster directly to a specific service of one of the constituent clusters. Any request of service that the cluster receives will be forwarded to the underlying component that provides the service without the participation of any of the other components of the cluster.
- We can define a new service that is not yet provided by any of the constituent component interfaces. This new service is identified with the cluster and no single component can provide it. The implementation of the service requires the cooperation of more than a single component. The role of the cluster in this case is to coordinate the interaction between the components and ensure the proper delivery of the service.

The cluster can be characterized in the same way, as are components through defining its properties, events, and methods. The cluster can interact with a component or with another cluster at a level that abstracts all the constituent components of the cluster. Because of the multiple components defined within a cluster, it is possible to have multiple threads of execution within the cluster as long as the cluster can manage the coordination of these threads.

5.3 Properties of Clusters

5.3.1 Centralized vs. Distributed Control

A distributed system provides many design advantages. A distributed design reflects the distributed nature of the resources and capitalizes on the local processing of the distributed elements. The design of a distributed system can be much more complicated than a centralized system. The simplicity of the centralized system comes from the centralized control over all the elements of the system. By contrast, the control mechanism within a distributed system is weak since the distributed elements of the system run in different environments. The distributed elements have a degree of autonomy that makes coordination between the elements a difficult task. The advantages of the distributed over centralized systems include scalability and maintenance. A combination of the benefits of the approaches can give system qualities that are not achievable by either model alone. We want to start with a distributed design that reflects the reality of the situation where the resources are distributed but at the same time we like to have a control mechanism over the distributed elements that is comparable to a centralized system. The approach we take is to decouple some of the design decisions from the components and centralize it within a separate unit that serves to control the interaction between the distributed elements. The distributed elements are no longer connected to each other but instead are connected directly through the control unit. Any interaction between the elements of the system is passed through this control unit. The centralization of the control allows us to manage the interaction between the elements from a single point. In a traditional distributed system, any change in the configuration of the system requires that changes be made in the configuration of each distributed

element. In the case of the distributed system with a centralized control it is possible to modify the configuration of the system by modifying the configuration within the control unit.

5.3.2 Race Conditions

A distributed system is susceptible to many complications that are not present in a centralized system. In this section, we discuss one such problem that is often hard to deal with in a distributed system. A race condition is a problem that is often associated with circuit design, where the order of arrival of two separate signals to a given destination is not specified in the design and can occur in an unpredictable way during the operation of the circuit. This is a highly undesirable condition in a design since it can lead to unpredictable behavior on the part of the system. A designer of a system will avoid having a race condition by introducing new elements in the design to insure a desired order of event will always occur in a predictable way. However, race conditions are difficult to deal with and even more difficult to detect in a complex circuit. The problem can also occur in the design of distributed software systems. When there are multiple threads of execution within the system running in parallel, the order of the events can have an unpredictable behavior on the system. Detecting a race condition within a software system can be as difficult as it is in the circuit design. The lack of a centralized control mechanism within a distributed system makes the problem difficult to detect and remove. In a centralized control, similar to the one suggested in the previous section, it is possible to prevent a race condition by including assertions at the control level that insure that only a certain sequence of events are allowed in certain situations. The centralized

control mechanism also makes the detection of the race condition easier since the pattern of interaction between the elements is specified in a single location.

5.3.3 Deadlocks

A deadlock within a system signifies the inability of a system to carry out its function due to a condition that forces it into an infinite waiting state. Without an external intervention, the system will be unable to break out of this state. Algorithms can deal with the problem of deadlock within a system. The designers of operating systems often have to deal with this problem to ensure that the operation of the system will not be adversely affected by the presence of this condition. In general, the problem of deadlocks is characterized by two different sets representing the resources of a system and the users of the resources. The resources are finite and cannot meet the simultaneous need of all the users. The approach taken in the case of limited resources is to provide mechanisms to share the resources among all the users in such a way that we can eventually satisfy all the users, even if we require that users wait before acquiring a resource. A queue of users can be maintained with every resource to keep track of the requestors of the resource. Various algorithms have been defined to deal with the problem of deadlock at various levels of the problem. There are three major categories of algorithms for dealing with the deadlock problem:

- Deadlock avoidance
- Deadlock prevention
- Deadlock detection and recovery

The first two categories of deadlock algorithm will not allow the deadlock to occur, either by implicitly making it part of the system design or by monitoring the events of the system and denying any request that can potentially put the system in a deadlock situation. The third category is not concerned in stooping the deadlock condition from occurring. Instead, the system is allowed to allocate its resource freely in an unrestricted way to users whenever requests are made. As this does not stop the deadlock from occurring, the algorithm simply spends its time monitoring the system for any deadlock condition. Once a deadlock has been detected, the system must decide on how to get out of this condition by preempting one or more users from the resource that it is holding. There are tradeoffs among the different approaches that can force the designer of the system to choose between performance penalties or allowing deadlocks to occur. Deadlocks can also occur in a distributed system, however, the problem is much more complicated to address. The algorithms outlined above work rather efficiently n a centralized system, but become very expensive when applied to the distributed case. This is another example of how a problem within a centralized system can be addressed more easily than the distributed case. The suggested configuration in the above sections can also help in addressing this problem. We are able to apply a centralized algorithm on a centralized control situation even when the components are distributed.

5.3.4 Dynamic Configuration Management Control

The management configuration of a distributed system can be much more complicated than in a centralized system. The distributed aspect of the components makes their configuration difficult to implement. The problem can be further complicated if we need to dynamically change the configuration of the system. The required coordination of the

autonomous parts of the system makes configuration changes difficult. However, changing configuration allows the system to have more flexibility and gives it a more dynamic aspect. The dynamic aspect of the system can be included in the design of the system. However, the design of a dynamic system is difficult since the relationship between the components of the system is usually static. One of the advantages of components is in the dynamic aspect of their configuration, where a designer can assemble the existing component in a way that meets the desired objectives. The suggested model for distributed components with a centralized control can address the problem of dynamic configuration management by the fact that the control is centralized. The user of the system can alter the configuration of the system by making changes to the control unit of the system. This allows the users to respond to changing requirements dynamically by controlling the system from a centralized location.

5.4 Design Pattern Components

5.4.1 Language Support for Design Artifacts

The limited support for high-level design artifacts in programming languages limits the ability of software developers to express directly aspects of their design using appropriate language constructs. As a result the initial design phase of software development is typically followed by a low-level design that maps the high-level aspects of the design into the low-level abstractions that can be mapped directly into the constructs of the implementation language, such as functions, procedures, and objects. While the low-level design specification simplifies the implementation phase of the system, it becomes more difficult to see the high-level design in the final implementation. The high-level design

becomes implicit and not easily detectable or traceable. This can have serious effects on the maintenance and evolution of software. When modification to the system becomes necessary, as a normal part of software evolution, there will be a need to go back and review the design documents in order to get an understanding of the structure and the implicit aspects of the design. Failure to understand the underlying design leads developers to inadvertently violate the architectural integrity of the system.

5.4.2 Design Patterns for Components

Design patterns in the context of objects and classes and the benefits associated with documenting them have been discussed in an earlier chapter. Design patterns can also be discussed in the context of software components. The concept of components is more general than that of a class in the object-oriented paradigm since a class is only one way of implementing components, i.e. components do not have to be implemented exclusively as classes even if this happens to be the most intuitive way of doing it. Therefore, similar to the way that design patterns can be identified and defined on classes it is be possible to define design patterns on components as well.

5.4.3 Component Representation for Designs

In the context of the component model used in this work we limit the specification of design patterns primarily to interactions between components that is based on events and methods. We will define a link to be a single interaction pattern between two components. A link associates an event in one component to a method in another component. This is equivalent to having a component subscribe to the event produced by another components. It is possible to define any number of links on a set of components.

We refer to a set of associated links as constituting a design pattern. While this definition allows the arbitrary grouping of links, design patterns are intended to group interactions between components that collectively achieve a certain objective.

5.4.4 The Structure of Design Pattern Components

Design patterns can be grouped together to form even larger patterns. Figure 5.8 demonstrates the additive property of design patterns. The pattern on the left describes the interaction between components A, B, and C, while the pattern on the left describes the interaction between components C and D. It is assumed that the two patterns have been defined separately.

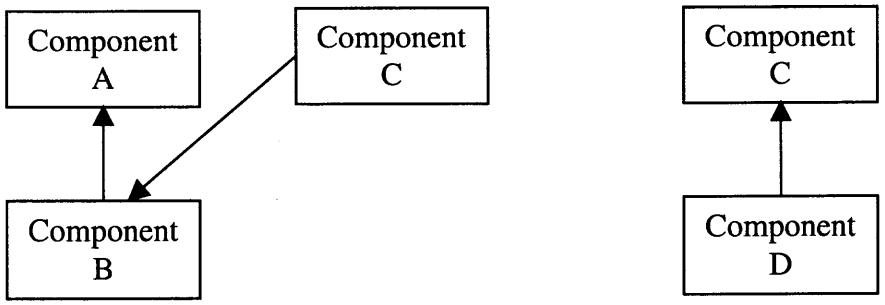


Figure 5.8 Design Patterns Independently Defined

It is possible to combine these two patterns into a single pattern that includes the all the interactions defined in both patterns. The result of combining these two patterns is shown in the following diagram.

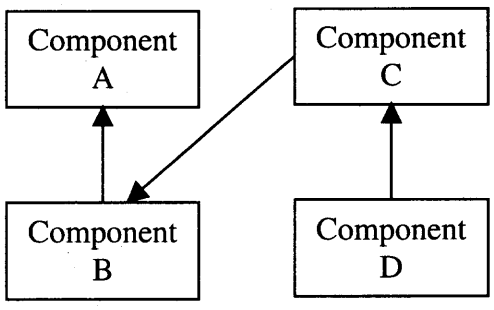


Figure 5.9 Combining Two Design Patterns

Design patterns can also be nested to define hierarchical relationship relationships between components. We can show this property intuitively by considering that components can be nested to produce new components at various levels of granularity. A design pattern defined on a set of components and acting as a single component can be part of another design pattern that includes a different set of components. Figure 5.10 shows a composite component with a design pattern applied to its member components.

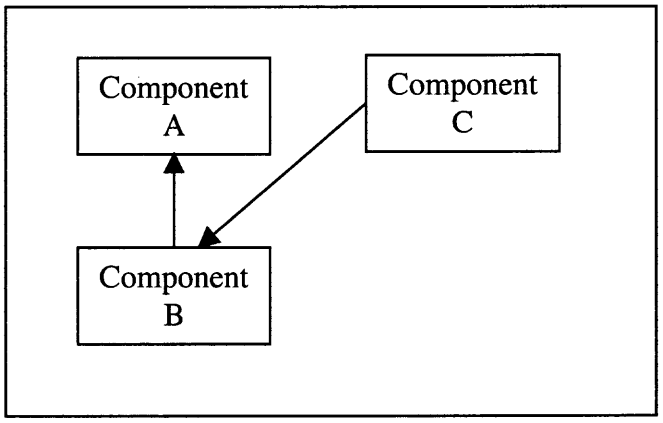


Figure 5.10 A Composite Component

We assume that a composite component has an interface that allows it to interact with other components in its environment without explicit reference to any of the

enclosed components from outside components. Figure 5.11 shows how this pattern can now be nested in another pattern.

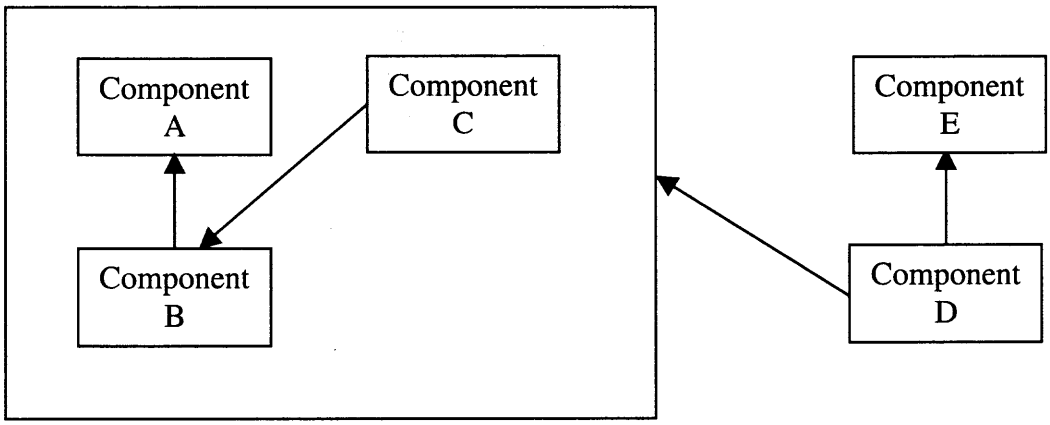


Figure 5.11 Nesting of Design Patterns

5.5 Implementation of Selected Design Patterns

In this section we show the application of documented object-oriented design patterns to components. The selected patterns have been classified in the literature as belonging to different classes of patterns. The adapter and façade patterns belong to the class of structural patterns.

5.5.1 The *Adapter* Pattern

Definition: An adapter can be used to overcome the mismatch between the interface that the client is expecting from the server and the actual interface provided by the server. In this case an Adapter pattern can be used to convert the request sent by the client into one that is compatible with the server component interface.

When components are to be assembled to form a subsystem the incompatibility of the interfaces requires the developer to either modify the code of one or both components or, if code modification is not an option, the user must introduce new code to perform the function of an adapter. A common situation with wiring components occurs when a *client* component is to be the recipient of events generated by a *server* component. The client must implement an interface that includes the methods that can be invoked on the client when the server fires the event. While the client does have a method to handle the event, it may not be implementing the interface. The solution to the problem lies in creating dynamically an adapter component that implements the interface and which can act as the recipient of the server-generated events. The adapter in turn will invoke the method of the client every time it receives the event. Any message contained in the event, which the adapter receives from the server, is passed on to the client component.

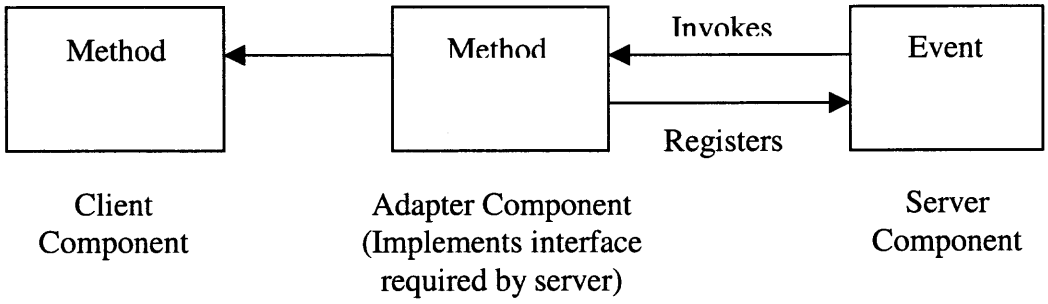


Figure 5.12 The Adapter Pattern

5.5.2 The *Facade* Pattern

Definition: When components are assembled as a cluster to represent a subsystem for example, we often need to provide a common interface that represents the higher-level

services provided by the subsystem. The new interface offers a simplified way to use the subsystem by abstracting the interfaces of the constituent components into a single logical interface.

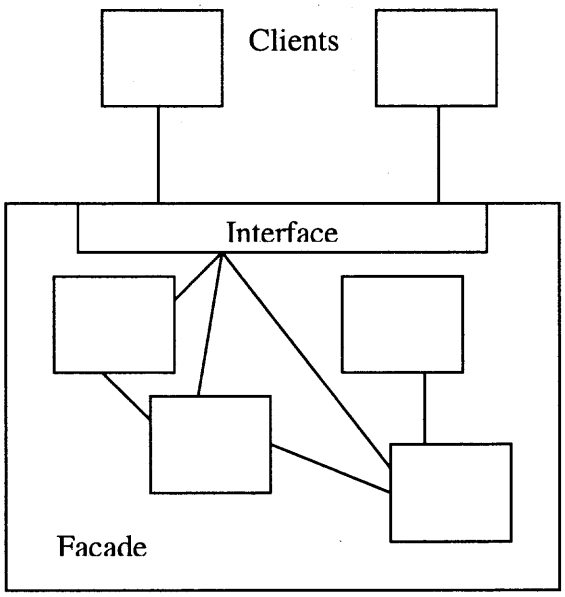


Figure 5.13 The Façade Pattern

The functionality of the cluster is defined in terms of the functionality of the components within the cluster. This can be achieved by selectively "promoting" some of the elements defined in the interfaces of the components to act as the element of the cluster interface. The cluster can be treated as a component, where the interface defined on it is the only accessible part while the components encapsulated within the cluster and hidden from other components outside the cluster.

5.5.3 The *Decorator* Pattern

Definition: A component may require an added service that is not part of its implementation. Using this pattern allows the developer to define a cluster that extends

the functionality of the component by enclosing it within the cluster and adding the missing functionality to the cluster so as to create the overall needed component.

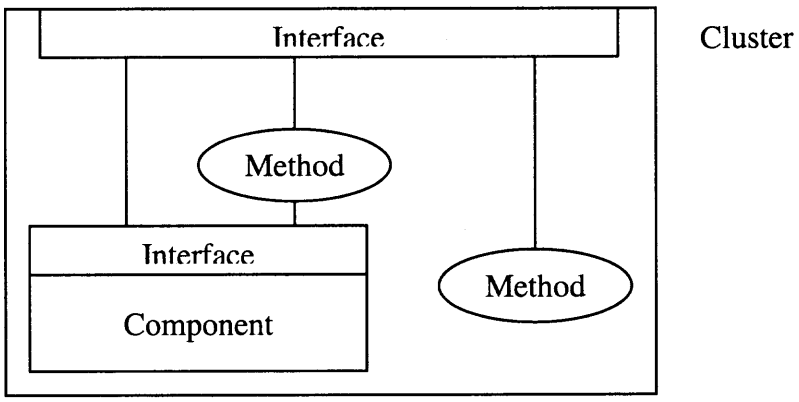


Figure 5.14 The Decorator Pattern

The interfaces of the component are passed to the cluster making the cluster behave the same as the component. In addition, a new functionality is added to the cluster beyond what is defined on the component in terms of one or more methods or the introduction of one or more events. The added methods and events are defined externally to the component but included in the interface of the cluster. For example, a new event is defined on the cluster that can only fire only if a number of events are fired by the component. Additionally, a method can be introduced that can be the target of some event and/or which may a new source of events.

CHAPTER 6

COMPILE: A COMPONENT INTERCONNECTION LANGUAGE

In this chapter we present a COMPonent Interconnection LanguagE called COMPILE. It is an object-based language that can be used by software designers and architects to give high-level declarative specifications of the system requirements in terms of components, interconnections, and constraints. The software specification is translated into executable code that can be executed in a run-time environment. The inclusion of architectural elements and the ability to create high-level abstractions in the COMPILE language allows the designer to quickly and reliably develop prototypes for software systems.

6.1 Elements of the Language

COMPILE is a Very High Level (VHL) language for expressing the required patterns of interactions between the components of a system. Compared to object-oriented languages, such as Java, COMPILE offers higher-level abstractions that can be used to describe architectural specifications of a system. COMPILE works at the component level, while Java-like languages can be targeted for the sub-component-level to develop individual components. The motivation behind COMPILE is that general-purpose programming languages lack the features and support to address the higher-level design issues of software systems. By making the architectural design explicit in the code, it will become possible to get a better understanding of the software structure, which in turn can improve verification and maintenance of the software. Conversely, The lack of support for high-level design features in programming languages makes the

design decisions implicit in the code and prevents developers from understanding the full implications of any modifications to the code.

In the following subsections, we will describe the important aspects of the COMPILE language, emphasizing the features with support for high-level design.

6.1.1 Components

Components represent the basic computational units in the COMPILE language. Components can be simple or composite. There are no means within the language to access the internals of a simple component beyond what is offered through its interfaces. The interface of a component specifies the events it can fire and the methods that can be invoked on it. We can define more than one interface on a component. A composite component consists of a set of one or more components, which are embedded within a single component. Within the COMPILE language it is possible to create composite components, called clusters. Composite components have the same characteristics as simple components and can interact with other components through their own unique interfaces. Clusters are discussed separately later in this chapter. In a complete specification, one component is designated as a system component, which encapsulates all other components. A system component represents the root in the hierarchy of components making up the software. The following is a specification of a component. The first statement declares a component and gives a reference to it within the name scope of the program. The second statement declares an interface for the component. The interface is referenced when interconnection between components is specified. The following two statements declare an event and a method, which are then added to the interface. We assume that the events and methods have been implemented

within the actual component and that these declarations merely create a reference to them. The last statement associates the interface with the component.

```
COMPONENT C = new COMPONENT("myComponent");
INTERFACE I = new INTERFACE("myInterface");
EVENT E = new EVENT("myEvent");
METHOD M = new METHOD("myMethod");
I.addEvent(E);
I.addMethod(M);
C.addInterface(I);
```

Figure 6.1 Specification of a Component

6.1.2 Clusters

A cluster is an abstraction that allows developers to define a set of interacting components and have a single component representation for the entire set. As a result coarse-grained components can be defined through the use of clusters. The definition of a cluster is based on three aspects: (1) the set of components belonging to the cluster, (2) the set of interfaces (one or more) defined on the cluster, and (3) a defined set of interaction patterns between the components. The first two aspects will be discussed in this section, while interaction patterns is discussed in the following section. A cluster has all the properties of a component. However, the cluster has properties that allow it to define its interfaces based on the definition of the component interfaces embedded in it. An event (or a method) belonging to the interface of a component within the cluster can be "promoted" to become an event (or a method) on the interface of a cluster. The implied semantics of promoting events is that components that wish to receive event notifications can register with the cluster and do not need to know the origins of the

event. The translation of the specification involving such actions will map the interactions directly with the real component whose event is being referenced. At run-time the cluster will not be involved in the interaction. Similarly, a specification in COMPILE can have components invoking the promoted method of a cluster. However, at run-time, the component will be wired in such a way to have the method invoked directly on the component offering it. The following code shows the declaration of cluster. In the first statement a cluster is declared. The string is used as the name of the component generated to represent the cluster. In the second statement the interface to the cluster is declared. The following two statements show a method and an event, previously defined on components that are part of the cluster, are getting promoted to the cluster's interface. The last statement assigns the interface to the cluster.

```
CLUSTER CL = new CLUSTER("myCluster");
INTERFACE CLI = new INTERFACE("myClusterInterface");
CLI.addEvent(E);
CLI.addmethod(M);
CL.addInterface(CLI);
```

Figure 6.2 Specification of a Cluster

6.1.3 Design Patterns

The interaction between components is defined in terms of links. A link associates the event defined on the interface of one component to the method defined on the interface of another component. The semantics of a link imply that whenever the event is fired in the first component the associated method in the link is invoked. Any information carried by the event will be delivered to the method. Links can be grouped together to form a pattern of interaction, referred to as *design pattern* in COMPILE. Each link in a design pattern

can be defined on the same or different components. We can use several design patterns to describe all the interactions between the components of a cluster. In the following code we define two links. The first link associates event E1 of interface I1 with method M1 of interface I2. It is assumed that the event, method, and interfaces have been previously declared. Similarly, the next statement declares a second link. The third statement declares a design pattern. The last two statements add the two links to the design pattern.

```
LINK LK1 = new LINK("myLink1", E1, I1, M1, I2);
LINK LK2 = new LINK("myLink2", E2, I3, M2, I4);
DPATTERN DP = new DPATTERN("myPattern");
DP.addLink(LK1);
DP.addLink(LK2);
```

Figure 6.3 Specification of a Design Pattern

The design pattern can now be added to a cluster using the method addDpattern of cluster. Currently there are three types of links implemented in COMPILE: explicit, implicit, and null. An explicit link specifies explicitly the entities on both ends of a link. This requires knowledge of the exact names for the event and the method defined by the link. An implicit link has either one or both ends of the link implicitly specified. This requires the cluster to have access at run-time to the meta-information of the component and discover the name of the event or method implicitly referenced in the link. A null link refers to a link that only specifies a method. The unspecified event is assumed to be the run-time instantiation of a component. This allows us to specify that whenever a particular component is instantiated an event should be sent to the component specified in the link. This is useful when a reference to a component is to be maintained by another

component for the purpose of possibly invoking a service on the component at a future time.

6.1.3.1 Applying Rules within a Design Pattern

A design pattern consists of multiple links defined on a set of components. In the normal case it is assumed that despite the grouping, each link acts independently of the other link in the design pattern. The grouping of links, however, gives us an opportunity to impose additional rules and create dependencies between the links. This in turn allows us to have additional control over the behavior of the interacting components. For example, we might impose an ordering on the links that implies a strict sequence in the delivery of the events. As a result, mechanisms have to be created at run-time that allows the delivery of the events to follow the order stated in the specifications. The ability to specify such rules has not been implemented, but there is little doubt about its usefulness in specific situations.

6.1.3.2 Applying Rules between Design Patterns

Following a similar argument to the one given in the previous subsection, there might be situations that necessitate establishing a linkage between the design patterns of a cluster. We can again justify the need for such control mechanism by stating that we might require a particular behavior from a cluster of components, which may only be possible if we impose rules binding two design patterns. For example, a certain type of event from the first design pattern should be fired before any event can be fired from the second design pattern.

6.2 Properties of the Language

COMPILE is implemented as a superset of the Java language. The abstractions discussed above that gives COMPILE its ability to express architectural specifications, are implemented as an added library to the Java compiler. To compile a program written in COMPILE you only need to invoke the java compiler. The specification will be translated into Java byte code and can then be executed on a Java virtual machine. The result of executing a COMPILE program is a number of Java classes organized in directories and conforming to the JavaBean component model.

6.2.1 Identifiers, Scoping Rules, and Control Structures

The structure of a COMPILE program consists of one or more Java classes. Developers can write regular Java code and use any of the Java language features in their code. Instructions implemented by COMPILE can be used freely with the regular Java code. The scoping rules of COMPILE are the same as in Java. The definition of a cluster can be made within a function or distributed over many functions. All abstractions introduced by COMPILE can be passed as parameters between cooperating functions and classes. The naming and scope of identifiers in COMPILE is subject to the same rules as in the Java language.

Programs written in COMPILE use the same selection and iterative control structures as the ones defined in Java. COMPILE does not introduce any new control structures. Algorithmically, COMPILE is a complete language that can be used to express any algorithm. This allows developers to use the computational power of a full programming language to develop the logic needed to produce architectural specifications.

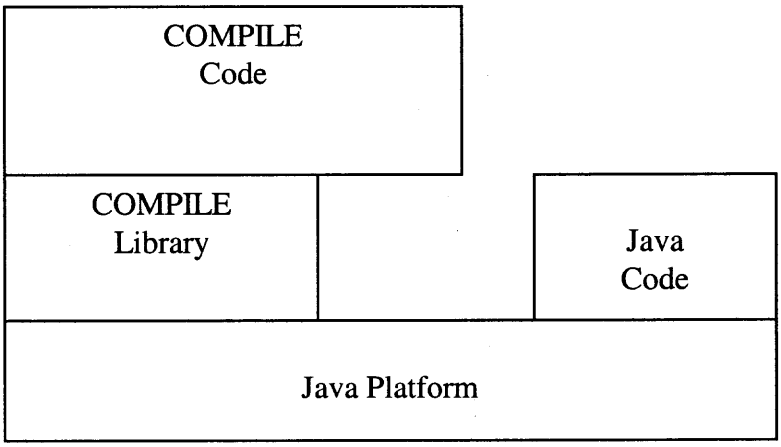


Figure 6.5 Platform Dependencies

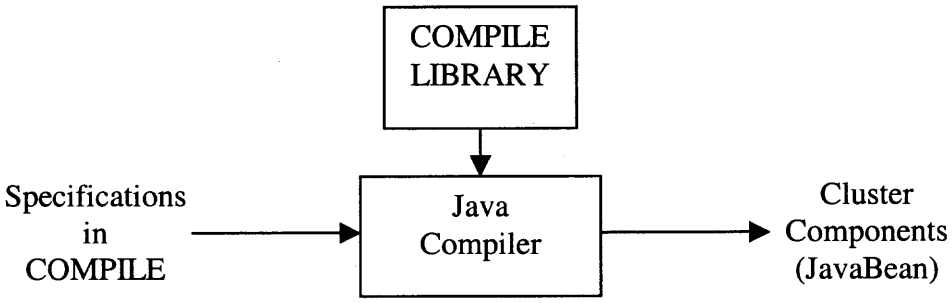


Figure 6.4 Compiling the COMPILE Code

CHAPTER 7

A COMPILER EXAMPLE

7.1 The Multiphase Compiler

The multi-phase compiler is one of the well-known and understood applications. A compiler is characterized by the input and output languages for which it provides translation. The input presented to the compiler is written in the source language of the compiler. The task of the compiler is to perform translation on its input so as to produce an output in the target language that is equivalent to the input. In this chapter we use the example of a compiler to demonstrate the application of the software development framework. This is a non-trivial example, though we have deliberately simplified the features of the source language so as to increase the clarity of the presentation.

7.1.1 A Multiphase Compiler Model

The most common implementation of a compiler consists of breaking up the translation process into discrete steps that perform the following associated tasks: lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and code generation. The first three phases can be grouped together into an analysis stage that is primarily concerned with the decomposition of the input and verification of the syntactical and semantic correctness. The combination of these phases can be referred to as the front end of the compiler. The combination of the last three phases makes up the synthesis stage that yields the required translation. The following diagram shows the various phases of the compiler. We will refer to the above six tasks performed by the compiler by the following names: scanner, parser, semantor, intermediate code generator,

optimizer, and code generator. The diagram also shows that there is a symbol table used whose purpose is to be a shared repository of information accessed by the various phases throughout the compilation process.

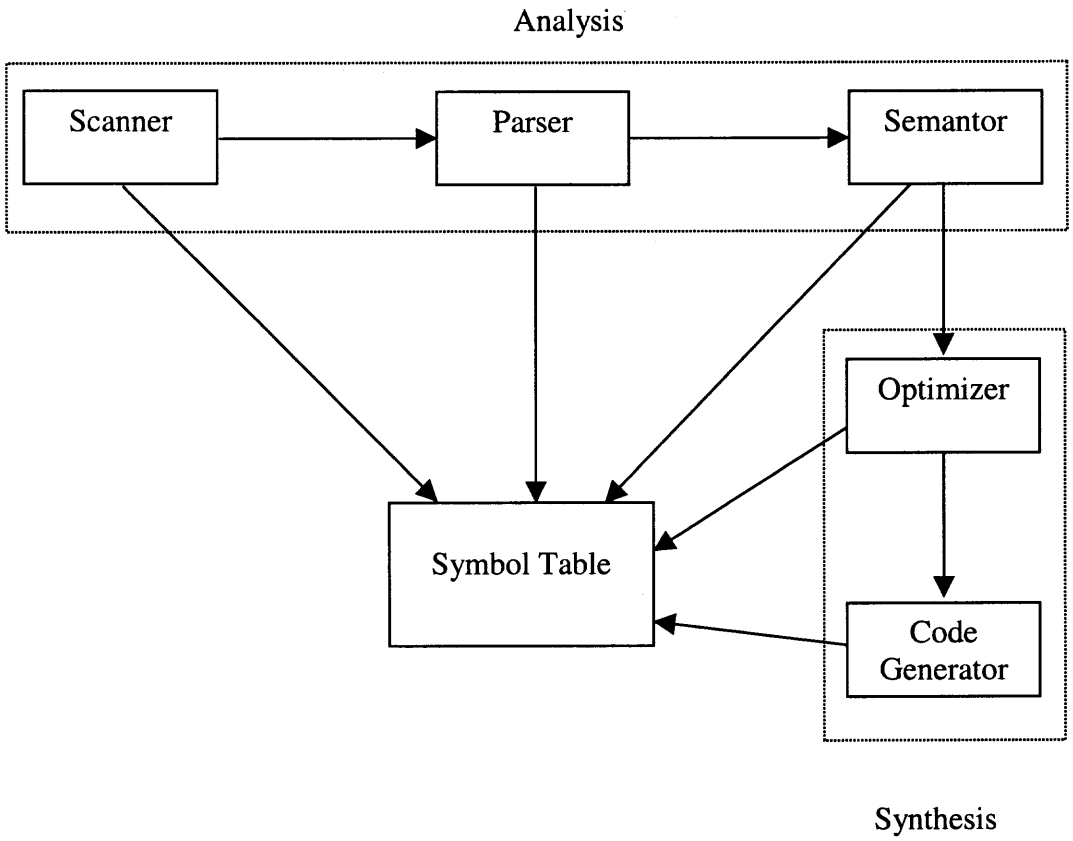


Figure 7.1 The Multiphase Compiler

In the following section we use a generic approach to identify the components required for building the compiler. We consider compilers to represent an application domain, which we analyze on the basis of functionality, data, and control.

7.2 A Domain Specification for Compilers

As we have outlined previously we will use three different models to represent the domain characteristics of compilers. The purpose of the models is to synthesize

specifications for building a component market for this specific domain. In the following paragraphs we will describe the functional, data, and control model.

7.2.1 Functional Model

The functional model gives a description of the processing elements in the domain. This model emerges as a result of a domain analysis by experts in the domain.

Table 4 Component Functionality

Components	Methods
Scanner	Start Scanning Synchronize
Parser	Parse Structure Synchronize
Semantor	Check Semantics Synchronize
Optimizer	Optimize Code Synchronize
Code Generator	Generate Code Synchronize

The maturity level of the domain works in favor of producing a stable model. In this case we can identify the functional or processing elements discussed above which would be mapped into specific components. The following are the specifications associated with each of the components.

7.2.1.1 The Scanner

The basic functionality of the scanner is defined by the processing required in reading the input source as a stream of characters and grouping the characters into meaningful tokens

within the context of the language. For the purposes of this example we can simplify the functionality of the scanner to consist of two functions: *Start Scanning* and *Synchronize*. The function *Start Scanning* is invoked to start the scanning process by accessing the input file and performing some internal initializations. The function *Synchronize* makes the scanner go into a state from which it can resume the scanning after an error has been detected.

7.2.1.2 The Parser

The basic functionality of the parser is to group the tokens into syntactical structures that conform to the grammar of the language. For the purposes of this example we can simplify the functionality of the parser to consist of two functions: *Parse Structure* and Synchronize. The function *Parse Structure* consumes enough tokens to verify the correctness of a structure as defined by the grammar of the language. The function Synchronize makes the parser go into a state from which it can resume parsing after an error has been detected.

7.2.1.3 The Semantor

The basic functionality of the semantor is to associate a meaning with a syntactical structure. The associated meaning may result in learning new facts about the entities in the user code and may result in some intermediate being generated. As a simplification, we assume that there are two functions defined on the semantor: *Check Semantics* and *Synchronize*. The function Check Semantics performs the semantic analysis on a structure. The function Synchronize makes the semantor go into a state from which it can resume semantic checking after an error has been detected.

7.2.1.4 The Optimizer

The basic functionality of the optimizer is to improve the efficiency of the code wherever possible, either in space or time, while maintaining the semantics of the code intact. The result of the optimization is in the same intermediate code format that is used by the semantor. As a simplification, we assume that there are two functions defined on the optimizer: Optimize Code and Synchronize. The function *Optimize Code* performs the optimization on the intermediate code. The function *Synchronize* allows the optimizer to go into a state from which it can resume its function after an error has been detected.

7.2.1.5 The Code Generator

The code generator receives intermediate code and generates an equivalent translation in the target language. We assume there are two functions defined on this component: *Generate Code* and *Synchronize*. The function Generate Code is called to translate a quadruple into the target language. The function Synchronize allows the code generator component to go into a state from which it can resume the translation after an error has been detected.

7.2.2 The Data Model

The data model describes the data that may be required as an input or that is produced as an output from any of the processing elements described in the functional model. The data model does not include any data that may be used strictly by a functional element but not outside of it. There are three types of data that will appear in this example: tokens, syntactical structures, intermediate code, and target code. The description of the data is platform independent.

7.2.2.1 The Tokens

Tokens are a type of data that is shared between components. Tokens are generated by the scanner and could be passed to any component that is interested in receiving it. Tokens can be represented by a single value, as is the case with keywords or special symbols, or it may be necessary to associate additional values with it, as in the case of identifiers and numerical literals. Figure 7.2 shows the encoding of the tokens for the compiler example in XML.

```
<?xml version="1.0"?>
<!DOCTYPE token
[<!ELEMENT token (TNUM | TID | TLPAREN | TRPAREN |
TEOF | TTIMES | TDIVIDE | TPLUS | TMINUS |
TEQUAL | TSEMI | TBEGIN | TEND)>
<!ELEMENT TNUM (#PCDATA)>
<!ELEMENT TID (#PCDATA)>
<!ELEMENT TLPAREN EMPTY>
<!ELEMENT TRPAREN EMPTY>
<!ELEMENT TEOF EMPTY>
<!ELEMENT TTIMES EMPTY>
<!ELEMENT TDIVIDE EMPTY>
<!ELEMENT TPLUS EMPTY>
<!ELEMENT TMINUS EMPTY>
<!ELEMENT TEQUAL EMPTY>
<!ELEMENT TSEMI EMPTY>
<!ELEMENT TBEGIN EMPTY>
<!ELEMENT TEND EMPTY>] >
```

Figure 7.2 Token Encoding

In the case of the source language for which we are building the compiler, the token include numerical literals, identifiers, the four basic arithmetic operators, the two keywords *begin* and *end*, few special symbols, and the end-of-file symbol.

7.2.2.2 The Syntactical Structures

Syntactical structures are described by a multi-value structure consisting of the *sequence*, *rule*, and zero or more arguments. *Sequence* refers to the sequence number of the rule being applied. *Rule* refers to the number of the rule being applied, where the rules in the grammar have been assigned a unique number. Both, the parser and the semantor, know the grammar rules, and there is an agreement on the rule number assignment. Figure 7.3 shows the encoding format of the syntactical structures.

```
<?xml version="1.0"?>
<!DOCTYPE struct [
<!ELEMENT struct (seq, rule, argument*)>
<!ELEMENT seq (#PCDATA)>
<!ELEMENT rule (#PCDATA)>
<!ELEMENT argument (num | seq | id)>
<!ELEMENT num (#PCDATA)>
<!ELEMENT seq (#PCDATA)>
<!ELEMENT id (#PCDATA)>]>
```

Figure 7.3 Syntactical Structure Encoding

7.2.2.3 The Intermediate Code

Quadruples are used to represent the intermediate code generated by the semantor. Each quadruple instruction is a 4-tuple consisting of an operator, two operands, and a target. In some cases there may be only one operand when the operator is unary. The target represents where the result of the operation will be stored. The semantor may create temporary variables to hold the partial results of complex statements. These temporary variables will appear in the quadruple instructions. The quadruples are generated by the semantor as a form of representing the semantics of the source code. Figure 7.4 shows the encoding of quadruples.

```
<?xml version="1.0"?>
<!DOCTYPE quadruple [
<!ELEMENT quadruple (operator, operand1, operand2, result)>
<!ELEMENT operator (plus | minus |times | divide)>
<!ELEMENT operand1 (#PCDATA)>
<!ELEMENT operand2 (#PCDATA)>
<!ELEMENT result (#PCDATA)> ] >
```

Figure 7.4 Intermediate Code Encoding

7.2.2.4 The Target Code

The target code is the final representation produced by the compiler. For this example, we use a fictitious assembly language into which we translate the quadruples generated by the optimizer.

```
<?xml version="1.0"?>
<!DOCTYPE targetCode [
<!ELEMENT targetCode (instruction, operand1, operand2)>
<!ELEMENT instruction (load, store, add, subtract, divide, multiply)>
<!ELEMENT load (#PCDATA)>
<!ELEMENT store (#PCDATA)>
<!ELEMENT add (#PCDATA)>
<!ELEMENT subtract (#PCDATA)>
<!ELEMENT divide (#PCDATA)>
<!ELEMENT multiply (#PCDATA)>
<!ELEMENT operand1 (#PCDATA)>
<!ELEMENT operand2 (#PCDATA)> ]>
```

Figure 7.5 Target Code Encoding

This assembly language consists of a load and store instructions to move data from a memory location to a register, and from a register to a memory location, respectively. It also has instructions to add, subtract, divide, and multiply to perform mathematical operations on the data contained in its registers. We assume there are two registers, R1 and R2. Figure 7.5 shows the encoding of the target code.

7.2.3 The Control Model

The control model specifies the events that can be fired by each of the functional units, described in the functional model. Components can subscribe to these events and specify which method to invoke upon firing the event. The event can carry information, which will be sent to the method of the receiving component. Table 5 lists the events with their associated functional elements.

Table 5 Control Table

Components	Events
Scanner	Token Scanned Scan Completed Token Error
Parser	Structure Parsed Parse Completed Syntax Error
Semantor	Structure Represented Representation Completed Semantic Error
Optimizer	Code Optimized Optimization Completed Optimization Error
Code Generator	Code Translated Translation Completed Translation Aborted

7.2.3.1 The Scanner

The scanner can generate three types of events: *Token Scanned*, *Scan Completed*, and *Token Error*. Each time a token has been identified a *Token Scanned* event is fired. The event will carry with it the type of token. When the scanner reaches the end of the input source and no tokens will be further encountered, it fires the event *Scan Completed*. This event does not carry any information with it. When the scanner encounters an error

during scanning it fires the event Token Error to notify the subscribed components of the error. This event can carry with it some information related to the nature of the error encountered.

7.2.3.2 The Parser

The parser can fire three types of events: *Structure Parsed*, *Parse Completed*, and *Syntax Error*. The event *Structure Parsed* is fired when the parser successfully applies a grammar rule. The information carried by the event was described in the data model. The event *Parse Completed* is fired when the parser finishes parsing all the tokens. This occurs when the parser receives an end-of-input token. The event *Syntax Error* occurs when the parser encounters a syntax error during parsing. Relevant information associated with the event is passed to the appropriate method.

7.2.3.3 The Semantor

The semantor fires three types of events: *Structure Represented*, *Representation Completed*, and *Semantic Error*. The event *Structure Represented* is fired when the semantor has successfully assigned a meaning to the parsed structure. The information associated with this event is the quadruple generated. The event *Representation Completed* is fired when the semantor expects that there will be no more structures to translate. The event *Semantic Error* is fired when the semantor discovers an error related to the translation it is doing. Relevant information is passes with the event.

7.2.3.4 The Optimizer

The Optimizer generates three types of events: *Code Optimized*, *Optimization Completed*, and *Optimization Error*. The optimizer fires the event *Code Optimized* when it has concluded the optimization of a quadruple. It may not always be possible to optimize each instruction. The event *Optimization Completed* is fired when the optimizer finishes

the optimization process of all the intermediate code. The event *Optimization Error* is fired when the optimizer encounters an error at any time during the optimization process.

7.2.3.5 The Code Generator

The code generator fires three types of events: *Code Translated*, *Translation Completed*, and *Translation Aborted.* The code generator fires the event *Code Translated* after it has translated a quadruple into the target language. The event *Translation Completed* is fired when after the last quadruple has been translated into the target language. The event *Translation Aborted* is fired when an error has been encountered at any time during the translation process.

7.2.4 Combining the Three Models

The three models described above can be combined in order to complete the specification of the components required in this domain (see Table 6).

7.3 Specifying the Compiler

In the previous section we defined the components required for the compiler domain. In this section we assume that the components are built according to the specifications outlined in the previous section. Our task is to specify how these components should interact with each other to implement the functionality of a compiler. In the following subsection we look at selected parts of the specification. The full specification is given in Appendix C.

Table 6 The Combined Model

Events And Methods	Component	Data
Events: Token Scanned Scan Completed Token Error Methods: Start Scan Synchronize	Scanner	Tokens
Events: Structure Parsed Parse Complete Syntax Error Methods: Parse Structure Synchronize	Parser	Structures
Events: Structure trans. No more structures Semantic Error Methods: Translate Structures Synchronize	Semantor	Intermediate Code
Events: Structure Optimized Optimization completed Optimization Error Methods: Optimize Structure Synchronize	Optimizer	Optimized Intermediate Code
Events: Code Translated Translation completed Translation Aborted Methods: Optimize Structure Synchronize	Code Generator	Target Code

7.3.1 Specification of a Component

The code shown in Figure 7.6 represents how components are introduced into the specifications. The interface and the component are declared first. Two events, "<token>" and "noMoreTokenEvent" are defined and included in the interface. Finally, the interface is associated with the component.

```
// Specification of the scanner component
INTERFACE I1 = new INTERFACE("scanInterface");
COMPONENT C1 = new COMPONENT("myCompiler.scanner.Scanner");
EVENT e1 = new EVENT("<token>");
EVENT e1a = new EVENT("noMoreTokenEvent");
I1.addEvent(e1);
I1.addEvent(e1a);
C1.addInterface(I1);
```

Figure 7.6 The Scanner Component Specification

The defined component is specified using the string "myCompiler.scanner.Scanner". This name corresponds to the address of the JavaBean package where the actual component can be found. The events in this code are introduced differently. The event "noMoreTokenEvent" is a name that can be mapped directly into the name of an event specified in the component. However, the event "<token"> is a reference to an event whose name is only understood within the semantics of the domain. The corresponding name of the event in the scanner component is different. It is the task of the cluster component to probe the meta-information of the component and get the actual name of the event that corresponds to the domain name. Once the name of the event has been discovered, the cluster component is able to create an appropriate adapter for the scanner component.

Similarly, we introduce the code corresponding to the parser component in Figure 7.7. There are several methods and events defined on the interface of the parser component. In general, if the actual name of the method defined on the interface is known, it can be used directly in the specification. This is the case with the method "ProcessNoMoreToken" where the method name is mapped directly to a method in the component interface.

```
// Specification of the parser component
INTERFACE I3 = new INTERFACE("parserInterface");
COMPONENT C3 = new COMPONENT("myCompiler.parser.Parser");
EVENT e3 = new EVENT("structEvent");
EVENT e3a = new EVENT("noMoreStructEvent");
METHOD m1 = new METHOD("<token>");
METHOD m1a = new METHOD("processNoMoreToken");
METHOD m3 = new METHOD("SetSymbolTable");
I3.addEvent(e3);
I3.addEvent(e3a);
I3.addMethod(m1);
I3.addMethod(m1a);
I3.addMethod(m3);
C3.addInterface(I3);
```

Figure 7.7 The Parser Component Specification

By contrast, the method specified by "<token>" is only a reference to a method specified in the semantics of the compiler domain. In order for a cluster to create an adapter involving this method, it must probe the parser component for the actual name of the method.

7.3.2 Specification of a Cluster

After introducing the components into the specifications, we need to define how the components should interact with each other. This is done in two steps: first we define a design pattern that describes the pattern of interactions between the components. Second,

we define the cluster by specifying the components it includes, the design patterns it uses, and the interfaces it presents to other components in its environment. Figure 7.8 shows the definition of two design patterns for cluster1. The definition is based on specifying the links that should be established between components. Each link associates an event within an interface to a method within another interface. One or more links can be grouped to form a design pattern. One or more design patterns can be used to describe the interactions between the components of the cluster.

```
// Specification of the design pattern for cluster1
DPATTERN dp1 = new DPATTERN("DPCa");
LINK L1 = new LINK("l1", e1, I1, m1, I3);
LINK L9 = new LINK("l3", e9, I9, m3, I3);
dp1.addLink(L1);
dp1.addLink(L9);
DPATTERN dp1a = new DPATTERN("DPCaa");
LINK L1a = new LINK("l1a", e1a, I1, m1a, I3);
dp1a.addLink(L1a);
```

Figure 7.8 Design Pattern Specification

The specification of cluster1 includes the above design patterns, three components, and an interface. The interface of the cluster specifies the elements that can be referenced to allow the cluster to interact with external components. The interface of cluster1 consists of two events, e3 and e3a, which had been previously defined on the interface of the parser component. This allows the cluster to become a source of these two events to which other components outside the cluster can register. We refer to these as "promoted" events since in reality the cluster does not produce these events. Whenever in the specification a reference is made to the promoted event of the cluster, the cluster treats as a reference to the component that is generating the event.

```
// Specification of cluster1
INTERFACE I6 = new INTERFACE("cluster1Interface");
CLUSTER CL1 = new CLUSTER("myCompiler.dpca.DPCa");
I6.addEvent(e3);
I6.addEvent(e3a);
CL1.addComponent(C1);
CL1.addComponent(C3);
CL1.addDpattern(dp1);
CL1.addDpattern(dp1a);
CL1.addInterface(I6);
CL1.addImport("myCompiler.shared.*");
CL1.setDebugMode();
CL1.show();
```

Figure 7.9 A Cluster Specification

Similarly, we can define a "promoted method" to be a method that has been declared on the interface of a cluster, but whose implementation is provided by a component that belongs to the cluster. Whenever a reference to the promoted method is made in the specification, the cluster treats it as a reference to the method of the underlying component.

7.3.3 Specification of Cluster Hierarchies

Clusters can be assembled to form hierarchies. In this example we defined two clusters, cluster1 and cluster2, that define separate parts of the compiler. Each cluster contains a set of components, which need not be visible from outside the cluster. These two clusters interact with each other according to the interfaces defined on them. In this case we define a third cluster to capture the pattern of interactions between these two clusters. The events and methods defined in the interfaces of the two clusters have all been promoted from components internal to the cluster.

```
// Specification of the design pattern for cluster3
DPATTERN dp3 = new DPATTERN("DPCc");
LINK L4 = new LINK("l4", e3, I6, m4, I7);
dp3.addLink(L4);
DPATTERN dp3a = new DPATTERN("DPCca");
LINK L4a = new LINK("l4a", e3a, I6, m4a, I7);
dp3a.addLink(L4a);

// Specification for cluster3
INTERFACE I8 = new INTERFACE("cluster3Interface");
CLUSTER CL3 = new CLUSTER("myCompiler.dpcc.DPCc");
I8.addEvent(e3);
CL3.addComponent(CL1);
CL3.addComponent(CL2);
CL3.addDpattern(dp3);
CL3.addDpattern(dp3a);
CL3.addDpattern(dp3b);
CL3.addInterface(I8);
CL3.addImport("myCompiler.shared.*");
CL3.setDebugMode();
CL3.show();
```

Figure 7.10 A Hierarchical Cluster

CHAPTER 8

CONCLUDING REMARKS AND FUTURE WORK

8.1 Summary

In this dissertation we developed a new approach to deal with the problem of developing software from components. Software components have been gaining wide acceptance by the software development community because of the potential benefits that can be drawn to the development process. A market for components can greatly reduce the cost of developing software by benefiting from the economies of scale. In addition, the reliability and robustness of a component is likely to be better when compared to code developed specifically for a given project, since component usage over time is likely to eliminate most, if not all, of the bugs in the code. The difficulty with components is not due to our inability to produce them. In fact, developing a component with a well-defined functionality and interfaces should not be any more difficult than writing the code for an object or module in general. The major difficulty presented by software components is in making them interoperate with each other. Most attempts in building software systems from components cite the problem of interface mismatch between components.

8.1.1 Lessons Learned

Components are still not widely used across domains. The most notable success of components is in the area of Graphical User Interface (GUI). The existence of many powerful and yet easy to use component-based frameworks designed for the rapid development of GUIs provides ample evidence of the potential benefits of components.

The apparent success of GUI frameworks is in their ability to capture a rather complete set of functionality required by developers in the construction of user interfaces. The functionality of the GUI domain has been carefully studied over time and resulted in the definition of GUI elements that are complete, orthogonal, complementary, and customizable. A de facto standardization of the functionality required from the basic set of elementary GUI elements allows component developers to produce components with wide acceptance. However, agreeing on the functionality of components within a certain domain is not sufficient by itself. A component model is also required to allow interaction between the components without having to deal with the problem of interface mismatch between components.

8.1.2 Applying What We Learned

By looking at the example of GUI frameworks we can make the following generalizations: (1) an analysis on the domain of application is required to define the set of components and associated semantics, (2) A component model that allows direct interoperation between components by eliminating potential mismatches with interfaces, (3) A development environment that allows developers to specify how components should be assembled together to meet the users' requirements. The work presented in this dissertation addresses the above issues. We believe that a domain model that can guide the development of components for the domain and provide a basis for interaction between components on a semantic basis is needed to achieve success with components on a wide scale and not just in few narrow domains. The development environment should provide a high-level interface that can allow an end user, and not just the developer, to perform the assembly of components. The development environment need

not be necessarily graphical, although graphical environments do have an intuitive appeal that makes them desirable by the users of the components.

8.1.3 Results Achieved

In this dissertation we have proposed an original solution to the problem of system development through the assembly of software components. Our approach is based on the separation of the syntactic features from the semantic contents within the definition of a component. This separation allows us to address the implementation issues involved in software integration from the application concerns of semantic integration. We base our results on the following two observations:

- Post-facto integration: The post-facto integration of components to produce systems is an engineering challenge that often leads to sub-optimal results, and
- Design for integration: Addressing the problems of constructing increasingly *large and complex* software systems requires an approach based on integration of executable software components.

We have identified three problems of integration that require to be isolated and separated within the definition of components: control, data, and function. These problems are described below:

- Problem 1: Establishing links to a set of non-conforming components, subject to constraints (Figure 8.1).
- Problem 2: Associating data-generating components with data-processing components in the absence of information about either interface (Figure 8.2).

- Problem 3: Translation of data exchanged between components without violating data types compatibility rules (Figure 8.3).

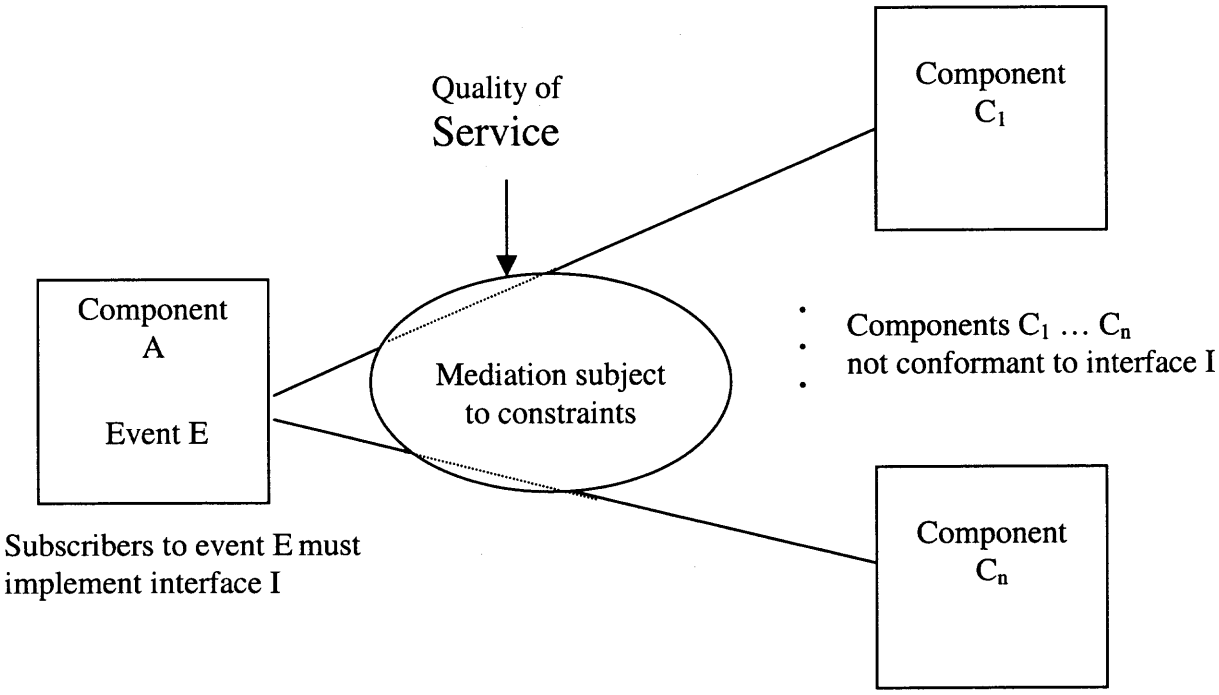


Figure 8.1 Separation of Control

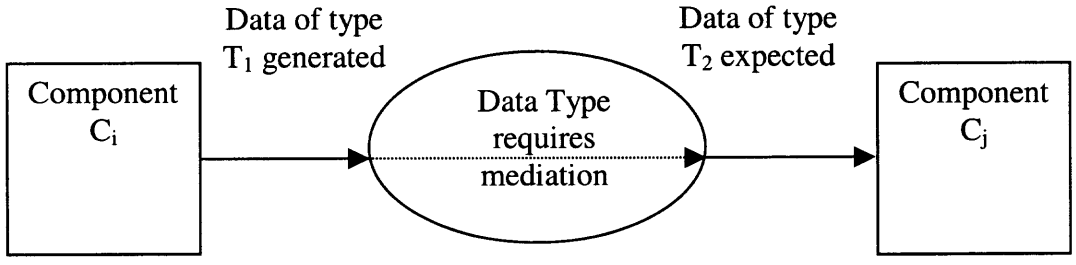


Figure 8.2 Separation of Data

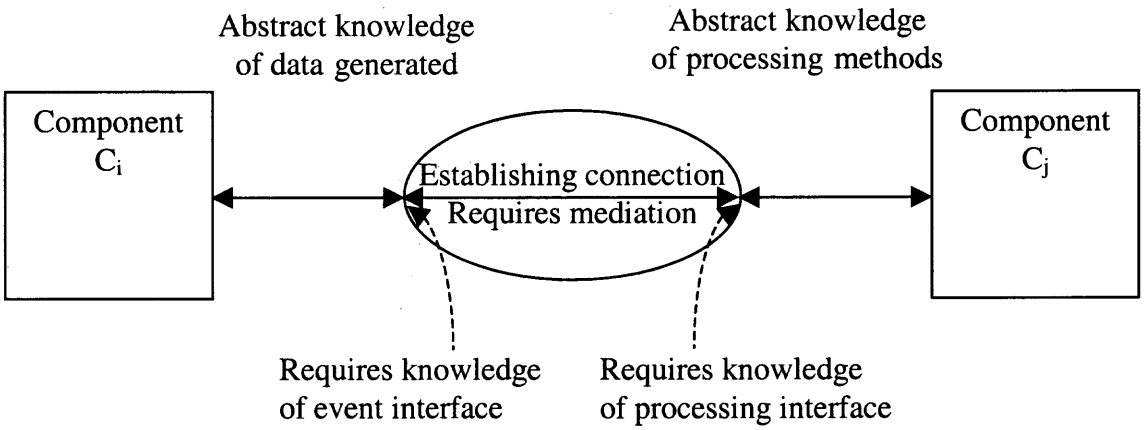


Figure 8.3 Separation of function

The solution to these three problems has been addressed throughout the dissertation and is based on the following approaches:

- Isolating Semantics: Our approach is based on the observation that isolation of semantic concerns is necessary for providing a basis for integration.
- Linguistic Approach: An interconnection language that separates the syntactic dependencies of component representation from the semantic issues of component integration is needed.
- Support Environment: A framework for a component architecture that supports third party development, incremental growth, and subassemblies.

These ideas have been incorporated into the design of an interconnection language for component composition. The language allows a developer to specify interaction between components based on the semantics of data described in the analysis of the domain. The specification allows the developer to request that certain type of data produced by one

component should be received by a second component that is capable of processing the information. The link between the components is specified without a reference to the event in the source component with which the information is associated, or a reference to the method within the target component with which the processing of the information is associated. The interaction between components is mediated through the use of adapters that are automatically generated at run-time and bound to the components to allow them to communicate. A late binding technique is used to allow maximum flexibility while specifying interaction between components with mismatched interfaces.

8.2 Future Work

There are several aspects of this research that can be explored. The model we have used does not preclude that the components of a system may be deployed in a distributed fashion, not only within the computing environment of an organization, but also over wide area networks, such as the Internet. However, in this work we did not address some of the specific issues that may be present in such distributed environments. For example, the use of directory services should be explored for maintaining information about the availability and characteristics of deployable components. A second direction to follow from this research is to explore the interaction between heterogeneous component models that currently exists, such as OMG's CORBA and Microsoft's COM. How will the interconnection language map to the various component models. A third direction could explore the ways in which the analysis and representation of the domain semantics can be carried out.

Clearly, the above questions and many other ones deserve some attention. The field of software components is still evolving. While there is still much to be accomplished in

this area, components do represent an evolutionary step in software development that promises to solve many of the current limitations we face.

8.3 Contribution

This dissertation includes a comprehensive study of software components. The architecture of components and the design of smart adapters for mediating between cooperating components has been explored in depth. Our conceptual results have been used as a basis for developing an interconnection language that is capable of describing the high-level design aspects of software systems. We have addressed the problem of how to mediate incompatibilities between prefabricated components when assembled into a system. The approach is based on the separation of the syntactical elements from the semantic properties of components. This separation allows developers to design a system from components by focusing on the semantic concerns of the system. The typing requirements that often limit the assembly of components due to mismatches in the interface of components are isolated and treated separately. The proposed component architecture allows a late binding of the control, data, and function between components through the use of callbacks, domain-based data model, and component meta-information, respectively.

This dissertation makes contribution in the areas of: component-based software development, component architecture based on semantic integration, and design specification languages for component integration. These results are to the best of our knowledge unique and original.

APPENDIX A

CONTEXT-FREE GRAMMAR REPRESENTATION

The syntax of the COMPILE specification language discussed in chapter 6 and the compiler example discussed in chapter 7 are both described using Context-Free Grammar (CFG). A CFG is denoted by G = (V, T, P, S), where V and T are finite sets of variables (or non-terminals) and terminals, respectively. P is a finite set of production rules of the form A ::= x, where A is a variable and x is a string of symbols from (V $\cup$ T)*. Finally, S is the start symbol. The following table summarizes the meta-symbols used in the grammar.

Meta-Symbol	Meaning
::=	Is defined as
|	Alternatives: e.g., X | Y means either X or Y
*	Repetition: zero or more
+	Repetition: one or more
[...]	Optional

In addition, the following convention is used:

1. Non-terminals symbols start with an upper case letter. The remaining characters are in lower case. When a non-terminal is made up of multiple words, the first letter of each word is in upper case.
2. Terminal symbols are in upper case letters and italic.
3. Meta-symbols are in bold characters.

APPENDIX B

CONTEXT-FREE GRAMMAR FOR THE COMPILE LANGUAGE

Context-Free Grammar for COMPILE:

Program	::= (Declaration | Statement) *
Declaration	::= SystemDeclaration
	| ClusterDeclaration
	| ComponentDeclaration
	| InterfaceDeclaration
	| DesignPatternDeclaration
	| LinkDeclaration
	| EventDeclaration
	| MethodDeclaration
SystemDeclaration	::= *SYSTEM* Sident [= ***new*** *SYSTEM (string)*]
ClusterDeclaration	::= *CLUSTER* Cident [= *new CLUSTER (string)*]
ComponentDeclaration	::= *COMPONENT* Pident [=*new COMPONENT (string)*]
InterfaceDeclaration	::= *INTERFACE* Iident [= *new INTERFACE (string)*]
DesignPatternDeclaration	::= *DPATTERN* Dident [= *n ew DPATTERN (string)*]
LinkDeclaration	::= *LINK* Lident [= new LINK (string , LinkSpec)]
EventDeclaration	::= *EVENT* Eident [= *new EVENT (string)*]
MethodDeclaration	::= *METHOD* Mident [= *new METHOD (string)*]
Statement	::= SystemStatement
	| ClusterStatement
	| ComponentStatement

	| InterfaceStatement
	| DpatternStatement
SystemStatement	::= Sident = ***new SYSTEM (string)***
	| Sident . *addCluster* (Cident)
	| Sident . *show* ( )
	| Sident . *SetDebugMode* ( )
ClusterStatement	::= Cident = *NEW CLUSTER (string)*
	| Cident . *addComponent* (Pident)
	| Cident . *addEvent* (Eident)
	| Cident . *addMethod* (Mident)
	| Cident . *addDpattern* (Dident)
ComponentStatement	::= Pident = *new COMPONENT (string)*
	| Pident . *addInterface* (Iident)
InterfaceStatement	::= Iident = *new INTERFACE (string)*
	| Ident . *addEvent* (Eident)
	| Iident . *addMethod* (Mident)
DPatternStatement	::= Dident = *new DPATTERN (string)*
	| Dident . *addLink* (Lident)
LinkSpec	::= Eident , Iident , Mident , Iident
Sident	::= *IDENTIFIER*
Cident	::= *IDENTIFIER*
Pident	::= ***IDENTIFIER***
Iident	::= ***IDENTIFIER***

Dident ::= ***IDENTIFIER***

Lident ::= ***IDENTIFIER***

Eident ::= ***IDENTIFIER***

Mident ::= ***IDENTIFIER***

APPENDIX C

CASE STUDY: THE COMPILER EXAMPLE

Context Free Grammar for The Compiler example

Input	::=	*BEGIN* StatementLlist *END EOF*
StatementList	::=	StatementList Statement
	|	Statement
Statement	::=	*IDENTIFIER* = Expression *;*
Expression	::=	Expression + Term
	|	Expression - Term
	|	Term
Term	::=	Term * Factor
	|	Term / Factor
	|	Factor
Factor	::=	(Expression)
	|	NUMBER
	|	IDENTIFIER
IDENTIFIER	::=	letter (letter | digit) *
NUMBER	::=	digit +
KEYWORDS	::=	(B | b) (E | e) (G | g) (I | i) (N | n)
	|	(E | e) (N | n) (D |d)

The Scanner Messages

The input to the scanner component is a stream of characters. The output produced by the scanner is a stream of tokens as described above. The tokens are embedded in messages that are expressed in XML format according to the following Data Type Definition (DTD):

```
<?xml version="1.0"?>
<!DOCTYPE token
[<!ELEMENT token (TNUM | TID | TLPAREN | TRPAREN |
TEOF | TTIMES | TDIVIDE | TPLUS | TMINUS |
TEQUAL | TSEMI | TBEGIN | TEND)>
<!ELEMENT TNUM (#PCDATA)>
<!ELEMENT TID (#PCDATA)>
<!ELEMENT TLPAREN EMPTY>
<!ELEMENT TRPAREN EMPTY>
<!ELEMENT TEOF EMPTY>
<!ELEMENT TTIMES EMPTY>
<!ELEMENT TDIVIDE EMPTY>
<!ELEMENT TPLUS EMPTY>
<!ELEMENT TMINUS EMPTY>
<!ELEMENT TEQUAL EMPTY>
<!ELEMENT TSEMI EMPTY>
<!ELEMENT TBEGIN EMPTY>
<!ELEMENT TEND EMPTY>] >
```

The Parsing Messages

The input to the parser component is a stream of tokens according to he description of the tokens above. The output generated by the parser is a stream of messages that encapsulate the parsing action and includes the semantic values associated with it. These messages are expressed in XML syntax according to the following Data Type Definition:

```
<?xml version="1.0"?>
<!DOCTYPE struct [
<!ELEMENT struct (seq, rule, argument*)>
<!ELEMENT seq (#PCDATA)>
<!ELEMENT rule (#PCDATA)>
<!ELEMENT argument (num | seq | id)>
<!ELEMENT num (#PCDATA)>
<!ELEMENT seq (#PCDATA)>
<!ELEMENT id (#PCDATA)>]>
```

The Semantor Messages

The semantor component accepts a stream of parsing actions and processes these actions for their semantic contents. The actions of the semantor may result in generating some intermediate code in the form of quadruples. The output of the compiler is a stream of messages representing translated quadruples expressed in XML syntax according to the following Data Type Definition.

```
<?xml version="1.0"?>
<!DOCTYPE quadruple [
<!ELEMENT quadruple (operator, operand1, operand2, result)>
<!ELEMENT operator (plus | minus |times | divide)>
<!ELEMENT operand1 (#PCDATA)>
<!ELEMENT operand2 (#PCDATA)>
<!ELEMENT result (#PCDATA)> ] >
```

The Optimizer Messages

The optimizer component receives a stream of quadruples to which it applies a set of optimization techniques. The output produced by the optimizer is identical in format to its input. The quadruples generated by the optimizer follow the same XML format described in the semantor section above.

APPENDIX D

SPECIFICATION OF THE COMPILER IN COMPILE

```
public class Spec {
  public static void main(String arg[]) {

        // Specification of the scanner component
        INTERFACE I1 = new INTERFACE("scanInterface");
        COMPONENT C1 = new COMPONENT("myCompiler.scanner.Scanner");
        EVENT e1 = new EVENT("<token>");
        EVENT e1a = new EVENT("noMoreTokenEvent");
        I1.addEvent(e1);
        I1.addEvent(e1a);
        C1.addInterface(I1);

        // Specification of the parser component
        INTERFACE I3 = new INTERFACE("parserInterface");
        COMPONENT C3 = new COMPONENT("myCompiler.parser.Parser");
        EVENT e3 = new EVENT("structEvent");
        EVENT e3a = new EVENT("noMoreStructEvent");
        METHOD m1 = new METHOD("<token>");
        METHOD m1a = new METHOD("processNoMoreToken");
        METHOD m3 = new METHOD("SetSymbolTable");
        I3.addEvent(e3);
        I3.addEvent(e3a);
        I3.addMethod(m1);
        I3.addMethod(m1a);
        I3.addMethod(m3);
        C3.addInterface(I3);

        // Specification of the semantor component
        INTERFACE I4 = new INTERFACE("semantorInterface");
        COMPONENT C4 = new COMPONENT("myCompiler.semantor.Semantor");
        EVENT e4 = new EVENT("quadEvent");
        EVENT e4a = new EVENT("noMoreQuadEvent");
        METHOD m4 = new METHOD("processStruct");
        METHOD m4a = new METHOD("processNoMoreStruct");
        METHOD m7 = new METHOD("SetSymbolTable");
        I4.addEvent(e4);
        I4.addEvent(e4a);
        I4.addMethod(m4);
        I4.addMethod(m4a);
        I4.addMethod(m7);
        C4.addInterface(I4);
```

```
// Specification of the optimizer component
INTERFACE I5 = new INTERFACE("optimizerInterface");
COMPONENT C5 = new COMPONENT("myCompiler.optimizer.Optimizer");
METHOD m5 = new METHOD("processQuad");
METHOD m5a = new METHOD("processNoMoreQuad");
METHOD m6 = new METHOD("SetSymbolTable");
I5.addMethod(m5);
I5.addMethod(m5a);
I5.addMethod(m6);
C5.addInterface(I5);

// Specification of the symbol table component
INTERFACE I9 = new INTERFACE("SymbolTableInterface");
COMPONENT C9 = new COMPONENT("myCompiler.symbol.SymbolTable");
EVENT e9 = new EVENT("null");
I9.addEvent(e9);
C9.addInterface(I9);

// Specification of the design pattern for cluster1
DPATTERN dp1 = new DPATTERN("DPCa");
LINK L1 = new LINK("l1", e1, I1, m1, I3);
LINK L9 = new LINK("l3", e9, I9, m3, I3);
dp1.addLink(L1);
dp1.addLink(L9);
DPATTERN dp1a = new DPATTERN("DPCaa");
LINK L1a = new LINK("l1a", e1a, I1, m1a, I3);
dp1a.addLink(L1a);

// Specification of cluster1
INTERFACE I6 = new INTERFACE("cluster1Interface");
CLUSTER CL1 = new CLUSTER("myCompiler.dpca.DPCa");
I6.addEvent(e3);
I6.addEvent(e3a);
CL1.addComponent(C1);
CL1.addComponent(C3);
CL1.addDpattern(dp1);
CL1.addDpattern(dp1a);
CL1.addInterface(I6);
CL1.addImport("myCompiler.shared.*");
CL1.setDebugMode();
CL1.show();

// Specification of the design pattern for cluster2
DPATTERN dp2 = new DPATTERN("DPCb");
LINK L3 = new LINK("l3", e4, I4, m5, I5);
```

```
        LINK L8 = new LINK("l8", e9, I9, m6, I5);
        LINK L10 = new LINK("l10", e9, I9, m7, I4);
        dp2.addLink(L3);
        dp2.addLink(L8);
        dp2.addLink(L10);
        DPATTERN dp2a = new DPATTERN("DPCba");
        LINK L3a = new LINK("l3a", e4a, I4, m5a, I5);
        dp2a.addLink(L3a);

        // Specification of cluster2
        INTERFACE I7 = new INTERFACE("cluster2Interface");
        CLUSTER CL2 = new CLUSTER("myCompiler.dpcb.DPCb");
        I7.addMethod(m4);
        I7.addMethod(m4a);
        I7.addEvent(e4a);
        CL2.addComponent(C4);
        CL2.addComponent(C5);
        CL2.addDpattern(dp2);
        CL2.addDpattern(dp2a);
        CL2.addInterface(I7);
        CL2.addImport("myCompiler.shared.*");
        CL2.setDebugMode();
        CL2.show();

        // Specification of the design pattern for cluster3
        DPATTERN dp3 = new DPATTERN("DPCc");
        LINK L4 = new LINK("l4", e3, I6, m4, I7);
        dp3.addLink(L4);
        DPATTERN dp3a = new DPATTERN("DPCca");
        LINK L4a = new LINK("l4a", e3a, I6, m4a, I7);
        dp3a.addLink(L4a);

        // Specification for cluster3
        INTERFACE I8 = new INTERFACE("cluster3Interface");
        CLUSTER CL3 = new CLUSTER("myCompiler.dpcc.DPCc");
        I8.addEvent(e3);
        CL3.addComponent(CL1);
        CL3.addComponent(CL2);
        CL3.addDpattern(dp3);
        CL3.addDpattern(dp3a);
        CL3.addInterface(I8);
        CL3.addImport("myCompiler.shared.*");
        CL3.show();
    }

}
```

REFERENCES

1. Agrawala, A., Jackson, M., and Vestal, S., "Domain-Specific Software Architectures for Intelligent Guidance, Navigation & Control," Proceedings of the DARPA Software Technology Conference, Los Angeles, California, April 28-30, 1992, pp. 223-226.

2. Arango, G., "Domain Analysis – From Art to Engineering Discipline," Fifth International Workshop on Software Specification and Design," 1989, pp. 152-159.

3. Baldwin, C.Y. and Clark, K.B., "Managing in an Age of Modularity," Harvard Business Review, Vol. 75, No. 5, September-October 1997, pp. 84-93.

4. Balzer, R., Cheatham, T.E., and Green, C., "Software Technology in the 1990's: Using a New Paradigm," Computer, Nov. 1983, pp.39-45.

5. Bancroft, N.H., Seip, H., and Sprengel, A., "Implementing SAP R/3," Manning Publications Co., Greenwitch, Connecticut, 2nd edition, 1998.

6. Bernstein, P.A., "Transaction Processing Monitors," Communications of the ACM," Vol. 33, No. 11, November 1990, pp. 75-86.

7. Bernstein, P.A., "Middleware: A Model for Distributed System Services," Communications of the ACM," Vol. 39, No. 2, February 1996, pp. 86-98.

8. Biggerstaff, T.J. and Perlis, A.J., "Introduction" in Software Reusability (Vol. I), Concepts and Models, ACM Press, 1989, pp. xv-xxv.

9. Boehm, B., "A spiral Model for Software Development and Enhancement," ACM SIGSOFT Software Engineering Notes, Vol. 11, No. 4, August 1986, pp.14-24.

10. Boehm, B.M. and Scherlis, W.L., "Megaprogramming," Proceedings of the DARPA Software Technology Conference, Los Angeles, California, April 28-30, 1992, pp. 63-82.

11. Bohrer, K.A., "Architecture of the San Francisco Frameworks," IBM Systems Journal, Vol. 37, No. 2, 1998, pp. 156-169.

12. Bohrer, K., Johnson, V., Nilsson, A., and Rubin, B., "Business Process Components for Distributed Object Applications," CACM, Vol. 41, No. 6, June 1998, pp.43-48.

13. Bohrer, K., Johnson, V., Nilsson, A., and Rubin, B., "Business Process Components for Distributed Object Applications," Communications of the ACM, Vol. 41, No. 6, June 1998, pp. 43-48.

14. Booch, G., "Object-Oriented Analysis and Design," The Benjamin/Cummings Publishing Company, 2nd edition, 1994.

15. Brown, A.W., "Foundations for Component-Based Software Engineering," in Component-Based Software Engineering, IEEE Computer Society Press, Edited by Alan W. Brown, 1996, pp. vii-x.

16. Brown, A.W., "Engineering of Component-Based Systems," in Component-Based Software Engineering, IEEE Computer Society Press, Edited by Alan W. Brown, 1996, pp. 7-15.

17. Clements, P.C., "From Subroutines to Subsystems: Component-Based Software Development," American Programmer, Vol. 8, No. 11, Nov. 1995.

18. Brooks, F.P., "No Silver Bullet – Essence and Accidents of Software Engineering," IEEE Computer, Vol. 20, No. 4, April 1987, pp. 10-19.

19. Braun, C., Hatch, W., et al., "Domain Specific Software Architectures – Command and Control," Proceedings of the DARPA Software Technology Conference, Los Angeles, California, April 28-30, 1992, pp. 215-222.

20. Bronsard, F., Bryan, D., et al, "Toward Software Plug-and-Play," ACM 1997.

21. Bushmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., "Pattern-Oriented SoftwareArchitecture", J. Wiley & Sons, 1996.

22. Card, D. and Comer, E., "Why do so Many Reuse programs Fail," IEEE Software, Vol. 11, No. 5, September 1994, pp. 114-115.

23. Chappell, D., "The OSF Distributed Computing Environment (DCE)," appeared in Distributed Computing, pp. 175-199, Raman Khanna editor, Prentice Hall PTR, Englewood Cliffs, NJ, 1994.

24. Chen, Y.T., Bayraktar, I., and Tanik, M.M., "Techniques for Software Reuse in Design and Specification," Control and Dynamic Systems, Academic Press, 1994, Vol. 61, pp. 329-361.

25. Chesbrough, H.W. and Teece, D.J., "When is Virtual Virtuous? Organizing for Innovation," in "Seeing Differently: Insights on Innovation," Harvard Business School Publishing, edited by J.S. Brown, 1997, pp. 105-119.

26. Christiansen, M.G., Delcambre, S.N., Demirors, E., Demirors, O., and Tanik, M.M., "Software Development with Transformable Components," Hawaii International Conference on System Sciences, IEEE Press, 1992, pp. 558-559.

27. Coad, P. and Yourdon, E., "Object-Oriented Analysis," Yourdon Press, 1990.

28. Coglianese, L., Goodwin, M., et al, "An Avionics Domain-Specific Software Architecture," Proceedings of the DARPA Software Technology Conference, Los Angeles, California, April 28-30, 1992, pp. 211-214].

29. Colonna-Romano, J, and Srite, P., "The Middleware Source Book," Digital Press, 1995.

30. Comer E.R., "Alternative Software Life Cycle Models," In Software Engineering, edited by Dorfman, M. and Thayer, R.H., IEEE Computer Society Press, 1997, pp. 404-414.

31. Coplien, J.O. and Shmidt, D.C., "Pattern Languages of Program Design," Addison-Wesley, Reading, MA, 1995.

32. Coplien, J.O., "Idioms and Patterns as Architectural Literature," IEEE Software, Volume 14, Number 1, 1997, p. 36-42.

33. Cox, B.J., "Object-Oriented Programming," Addison-Wesley, Reading, Mass., 1986.

34. DeBaud, J.M., "Viewing a DSSA in Context: Problems versus Solutions," SIGSOFT 96 Workshop, San Francisco, CA, 1996, pp. 19-23.

35. DeBaud, J.M., Flege, O., and Knauber, P., "PuLSE-DSSA – A Method for the Development of Software Reference Architectures," ISA W3, Orlando, Florida, 1998, pp. 25-28.

36. DeRemer, F. and Kron, H.H., " Programming-in-the-Large Versus Programming-in-the-Small," IEEE Transactions on Software Engineering, Vol. SE-2, No. 2, June 1976, pp. 80-86.

37. Dijkstra, E.W., "The structure of the T.H.E. multiprogramming system," CACM, Vol. 11, No. 5, May 1968, pp. 453-457.

38. Demeyer, S. Meijler, T.D., Nierstrasz, O., and Steyaert, P., "Design Guidelines for Tailorable Frameworks," Communications of the ACM, Volume 40, Number 10, October 1997, p. 60-64.

39. Doscher, D. and Hodges, R., "SEMATECH's Experience with the CIM Framework," Communications of the ACM, Volume 40, Number 10, October 1997, p. 82-84.

40. D'Souza, D.F. and Wills, A.C., "Catalysis: Practical Rigor and Refinement," In Fusion in Real World, edited by Coleman, D., et al., Englewood Cliffs, N.J., Prentice-Hall, 1995.

41. D'Souza, D.F. and Wills, A.C., "Objects, Components, and Frameworks with UML – The Catalysis Approach," Addison-Wesley, 1999.

42. Fafchamps, D. "Organizational Factors and Reuse," IEEE Software, Vol. 11, No. 5, September 1994, pp. 31-41.

43. Fayad, M.E. and Schmidt, D.C., "Object-Oriented Application Frameworks," Communications of the ACM, Volume 40, Number 10, October 1997, p. 32-38.

44. Gamma, E., E.,Helm, R., Johnson, R., and Vlissides, J., "Design Patterns:Abstraction and Reuse oObject-Oriented Design," ECOOP '93 proceedings.

45. Gamma, E.,Helm, R., Johnson, R., and Vlissides, J., "Design Patterns: Elements of Reusable Object-oriented Software," Addison-Wesley, 1995.

46. Garlan, D., Allen, R., and Ockerbloom, J., "Architectural Mismatch: Why is it hard to build systems out of Existing Parts," Proceedings of the International Conference on Software Engineering, Seatle, April 1995.

47. Gibbs, W.W., "Software's Chronic Crisis," Scientific American, September 1994, pp. 86-95.

48. Glass, R.L. and Vessey, I., "Contemporary Application-Domain Taxonomies," IEEE Software, Vol. 12, No. 4, July 1995, pp. 63-76.

49. Gomaa, H. and Scott, D., "Prototyping as a Tool in the Specification of User Requirements," Proc. 5th IEEE International Conference on Software Engineering, 1981, pp. 333-342.

50. Graham, I., Henderson-Sellers, B., and Younessi, H., "The OPEN process Specification," Addison-Wesley, 1997.

51. Hayes-Roth, F., Erman, L.D., Terry, A., and Hayes-Roth, B., "Domain-Specific Software Architectures: Distributed Intelligent Control and Management (DICAM) Applications and Development Support Environment," Proceedings of the DARPA Software Technology Conference, Los Angeles, California, April 28-30, 1992, pp. 227-230.

52. Hayes-Roth, R. and Tracz, W., "DSSA Tool Requirements for Key Process Functions," HTTP://,Version 2, October 24, 1994.

53. Henderson-Sellers, B. and Edwards, J.M., "The Object-Oriented Systems Life-Cycle," CACM, Vol. 3, No. 9, September 1990, pp. 143-159.

54. Henninger, S. "Using Iterative refinement to Find Reusable Software," IEEE Software, Vol. 11, No. 5, September 1994, pp. 48-59.

55. Jacobson, I., Booch, G., and Rumbaugh, J., "The Unified Software development Process," Rational Software Corporation, Addison-Wesley, 1999.

56. Johnson, R.E. and Foote, B., "Designing Reusable Classes," Journal of Object-Oriented Programming, Volume 1, Number 5, June/July 1988, p. 22-35.

57. Johnson, R.E., "Frameworks=(Components + Patterns)," Communications of the ACM, Volume 40, Number 10, October 1997, p. 39-42.

58. Jololian, L. and Smith, M.F., "Architectural Framework for Virtual Enterprises," The Third Biennial World Conference on Integrated Design and Process Technology, Berlin, Germany, July 5-9, 1998, pp. 347-350.

59. Jones, T.C., "Reusability in Programming: A Survey of the State of the Art," IEEE Transactions on Software Engineering, Vol. SE-10, Sept. 1984, pp. 488-494.

60. Joos, R., "Software Reuse at Motorola," IEEE Software, Vol. 11, No. 5, September 1994, pp. 42-47.

61. Korson, T.K., McGregor, J.D., "Understanding Object-Oriented: A Unifying Paradigm," CACM, Vol. 3, No. 9, September 1990, pp. 40-60.

62. Krieger, D. and Adler, R.M., "The Emergence of Distributed Component Platforms," IEEE Computer, Vol. 31, No. 3, March 1998, pp. 43-53.

63. Lewandowski, S.M., "Frameworks for Component-Based Client/Server Computing," ACM Computing Surveys, Vol. 30, No. 1, March 1998, pp.3-27.

64. Mettala, E., and Graham, M., "The Domain Specific Software Architecture Program," Proceedings of the DARPA Software Technology Conference, Los Angeles, California, April 28-30, 1992, pp. 204-210.

65. Mittermeir R.T. and Kofler, E., "Layered Specifications to Support Reusability and Integrability," Journal of Systems Integration, Kluwer Academic Publishers, Boston, Vol. 3, 1993, pp. 273-302.

66. Mullender, S., "Distributed Systems," Addison-Wesleyand ACM Press, edited by Sape Mullender, 1989.

67. Neighbors, J., "Software Construction Using Components," Ph.D. Thesis, Department of Information and Computer Science, University of California, Irvine, 1981.

68. Neighbors, J., "The Draco Approach to Constructing Software from Reusable Components," IEEETransactions on Software Engineering, SE-10, pp. 564-573, September 1984.

69. Neighbors, J.M., "DRACO: A Method for Engineering Reusable Software Systems," in Software Reusability (Vol. I), Concepts and Models, ACM Press, 1989, pp. 295-319.

70. OMG, "Common Object request Broker: Architecture and Specifications," Version 2.2, Object Management Group, Inc., February 1998.

71. Ousterhout, J.K., "Scripting: Higher-level Programming for the 21st Century," IEEE Computer, Vol. 31, No. 3, March 1998, pp.23-30.

72. Parnas, D., "On the criteria for decomposing systems into modules," CACM, Vol. 15, No. 12,December 1972, pp. 1053-1058.

73. Parnas, D.L., Clements, P.C., Weiss, D.M., "Enhancing Reusability with Information Hiding," in Software Reusability (Vol. I), Concepts and Models, ACM Press, 1989, pp.141-157.

74. Polan, M.G., "Technical Note – Using the San Francisco Frameworks with VisualAge for Java," IBM Systems Journal, Vol. 37, No. 2, 1998, pp. 215-225.

75. Pree, W., "Designing Patterns for Object-Oriented Software Development," Addison-Wesley, 1995.

76. Pressman, R.S., "A Manager's Guide to Software Engineering," McGraw-Hill, New York, N.Y., 1993.

77. Pressman, R.S., "Software Engineering: A Practitioner's Approach," McGraw-Hill, 4th edition, 1997.

78. Prieto-Diaz, R., "Classification of Reusable Modules," in Software Reusability (Vol. I), Concepts and Models, ACM Press, 1989, pp. 99-123.

79. Prieto-Diaz, R., "Domain Analysis: An Introduction," ACM SIGSOFT, Software Engineering Notes, Vol. 15, No. 2, April 1990, pp. 47-54.

80. Prieto-Diaz, R., "Status Report: Software Reusability," IEEE Software, May 1993, pp. 61-66.

81. Rosenberry, W., Kenney, D., and Fischer, G., "Understanding DCE," O'Reilly & Associates, Inc., 1992.

82. Royce, W.W., "Managing the Development of Large Software Systems: Concepts and Techniques," WESCON Technical Papers, Vol. 14, Western Electronic Show and Convention, 1970.

83. Schmid, H.A., "Design Patterns for Constructing the Hot Spot of a Manufacturing Framework," Journal of Object-Oriented Programming, 9, 3, June 1996, p. 25-37.

84. Schmid, H.A., "Systematic Framework Design by Generalization," Communications of the ACM, Volume 40, Number 10, October 1997, p. 48-51.

85. Shaw, M., "Heterogeneous Design Idioms for Software Architecture," IEEE Workshop on Software Specification and Design, October 25-26, 1991, pp.158-165.

86. Shaw, M., "Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging," Proceedings of the 17th International Conference on Software Engineering on Symposium on Software Reusability, 1995, pp. 3-6.

87. Sun Microsystems, "Enterprise JavaBeans – Frequently Asked Questions," http://java.sun.com.

88. Szyperski, C., Component Software – Beyond Object-Oriented Programming," Addison-Wesley and ACM Press Books, 1998.

89. "TAFIM - Technical Architecture Framework for Information Management," Defense Information Systems Agency Center for Standards, Version 3, April 30, 1996.

90. Tang, Y., "A Methodology for Component-Based System Integration," Ph.D. Dissertation, Computer and Information Science Department, New Jersey Institute of Technology, 1999.

91. Tanik, M.M. and Chen, E.S., "Fundamentals of Computing for Software Engineers," Van Nostrand Reinhold Press, 1991.

92. Tanik, M.M., Ertas, A., and Dogru, A.H., "Techniques in Abstract Design Methods in Engineering Design Development," Control and Dynamic Systems, Vol. 61, Academic Press, 1994, pp. 285-328.

93. Thomas, A. and Seybold, P., "Enterprise JavaBeans – Server Component Model for Java," December 1997, http://java.sun.com.

94. Tracz, W., "A Conceptual Model for Megaprogramming," Software Engineering Notes, Vol. 16, No. 3, July 1991, pp. 36-45.

95. Tracz, W., Coglianese, L., and Young, P., "A Domain-Specific Software Architecture Engineering Process Outline," ACM SIGSOFT, Software Engineering Notes, Vol. 18, No. 2, April 1993, pp.40-50.

96. Tracz, W., "Megaprogramming and Domain Engineering," ICSE 15th International Conference on Software Engineering," Tutorial Notes, Baltimore, Maryland, May 17-21, 1993.

97. Tracz, W., "DSSA Frequently Asked Questions," ACM Software Engineering Notes, 19, 2, April 1994, p. 52-56.

98. Vanhelsuwe, L., "Mastering JavaBeans," Sybex Publishing, 1997.

99. Vinoski, S., "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," IEEE Communications, vol. 35, no. 2, pp. 46-55, 1997.

100. Vlissides, J.M., Coplien, J.O., and Kerth, N.L., "Pattern Languages of Program design 2," Addison-Wesley, Reading, MA, 1996.

101. Waite, W.M., and Sloane, A.M., "Software Synthesis via Domain-Specific Software Architectures," Technical report CU-CS-611-92, Department of Computer Science, University of Colorado, September 1992.

102. Whitehead, E.J., Robbins, J.E., Medvidovic, N., and Taylor, R.N., "Software Architecture: Foundation of a Software Component Marketplace," Dept. of Information and Computer Science, University of California, Irvine.

103. Wiederhold, G., "Model-Free Optimization," Proceedings of the DARPA Software Technology Conference, Los Angeles, California, April 28-30, 1992, pp. 83-96.

104. Wiederhold, G., Wegner, P., Ceri, S., "Toward Megaprogramming," Communications of the ACM, Volume 35, No. 11, November 1992, pp. 89-99.

105. Wirfs-Brock, R.J. and Johnson, R.E., "Surveying Current Research in Object-Oriented Design," CACM, Vol. 3, No. 9, September 1990, pp. 104-124.

106. Yourdon, E., "Object-Oriented Systems Design: An Integrated Approach," Prentice-Hall, 1994.

107. Yin, W., Tanik, M.M., Yun, D.Y.Y., Lee, T.J., and Dale, A.G., "Software Reusability: A Survey and a Reusability Experiment," Fall Joint Computer Conference (FJCC), IEEE Press, 1987, pp. 65-72.

108. Yin, W.P., Tanik, M.M., and Yun, D.Y.Y., "Software Design Representation: Design Object Descriptive Attribute," Hawaii International Conference on System Sciences, IEEE Press, 1988, pp. 431-435.