2001

An Adaptive Integration Architecture for Software Reuse

Denver Robert Edward Williams
University of Central Florida, denverrwilliams@gmail.com

AN ADAPTIVE INTEGRATION ARCHITECTURE FOR SOFTWARE REUSE

By
Denver Robert Edward Williams

2001

UCF

AN ADAPTIVE INTEGRATION ARCHITECTURE FOR SOFTWARE REUSE

by

DENVER ROBERT EDWARD WILLIAMS
M.S. University of Central Florida, 1993
B.Sc. (Special) University of the West Indies, 1986

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2001

Major Professor: Dr. Ali Orooji

ABSTRACT

The problem of building large, reliable software systems in a controlled, cost-effective way, the so-called software crisis problem, is one of computer science's great challenges. From the very outset of computing as science, software reuse has been touted as a means to overcome the software crisis issue. Over three decades later, the software community is still grappling with the problem of building large reliable software systems in a controlled, cost effective way; the software crisis problem is alive and well. Today, many computer scientists still regard software reuse as a very powerful vehicle to improve the practice of software engineering. The advantage of amortizing software development cost through reuse continues to be a major objective in the art of building software, even though the tools, methods, languages, and overall understanding of software engineering have changed significantly over the years.

Our work is primarily focused on the development of an Adaptive Application Integration Architecture Framework. Without good integration tools and techniques, reuse is difficult and will probably not happen to any significant degree. In the development of the adaptive integration architecture framework, the primary enabling concept is object-oriented design supported by the unified modeling language. The concepts of software architecture, design patterns, and abstract data views are used in a structured and disciplined manner to established a generic framework. This framework is

applied to solve the Enterprise Application Integration (EAI) problem in the telecommunications operations support system (OSS) enterprise marketplace.

The proposed adaptive application integration architecture framework facilitates application reusability and flexible business process re-engineering. The architecture addresses the need for modern businesses to continuously redefine themselves to address changing market conditions in an increasingly competitive environment. We have developed a number of Enterprise Application Integration design patterns to enable the implementation of an EAI framework in a definite and repeatable manner. The design patterns allow for integration of commercial off-the-shelf applications into a unified enterprise framework facilitating true application portfolio interoperability. The notion of treating application services as infrastructure services and using business processes to combine them arbitrarily provides a natural way of thinking about adaptable and reusable software systems.

We present a mathematical formalism for the specification of design patterns. This specification constitutes an extension of the basic concepts from many-sorted algebra. In particular, the notion of signature is extended to that of a vector, consisting of a set of linearly independent signatures. The approach can be used to reason about various properties including efforts for component reuse and to facilitate complex large-scale software development by providing the developer with design alternatives and support for automatic program verification.

I dedicate this dissertation to my family: my wife, Michelle and sons Nicoli and Stefan.

ACKNOWLEDGEMENTS

I would like to thank my doctoral committee Dr. Ali Orooji, Dr. Charles Hughes, Dr. Sheau-Dong Lang, Dr. Rebecca Parsons, and Dr. Harley Myler for their help, critical comments, direction, support, and encouragements during the different phases of my doctoral study program. I appreciate the time, energy, and valuable insights they have provided so that this work could be accomplished.

Dr. Ali Orooji, my advisor and committee chair, deserves special thanks for editing the work and the countless hours we spend discussing various issues. His guidance and tutelage over the years contributed significantly to my academic maturity, for which I am profoundly grateful and indebted.

Dr. Rebecca Parsons deserves a special thank you for the significant insights she provided in the areas of algebraic structures and their semantics.

I would like to thank Dr. Charles Hughes and Dr. James Rogers (Dr. Rogers is a former faculty in the Department of Computer Science) for helping to cultivate in me a love for theoretical computer science.

My Mom and Dad deserve a very special thank you for their love, support, and encouragement throughout all my life. I would like to thank them for the discipline and love of learning that they have instilled in me in my early childhood. Without their efforts, none of this would have been possible.

I would like to thank my brother and sisters Lawrence, Jacqueline, Carol, and Racquel for their love, support, and encouragement throughout these years.

Finally and significantly, I am indebted to my family: my wife Michelle and our two sons Nicoli and Stefan, for their patience and understanding, their love, support and encouragement throughout these years. I thank you guys for believing in me.

TABLE OF CONTENTS

List of Acronyms xiii

List of Figures xvi

Chapter 1: Introduction 1

1.1 Expansive View of Software Reuse 2

1.2 Module Interface and Software Reuse 5

1.3 User Interface and Reuse 8

1.4 Our Contribution 9

1.5 Outline of Dissertation 12

Chapter 2: Outline of Previous Work 13

2.1 Abstractions 13

2.1.1 Abstractions in Software Development 14

2.1.2 Abstractions in Software Reuse 18

2.1.3 Cognitive Distance 19

2.2 Classification of Reusable Models 20

2.2.1 Software Components 22

2.2.2 Software Components Reuse Model 24

2.2.3 Classification Principles 26

2.2.4 Software Classification 28

2.2.5 Conceptual Closeness 31

2.2.6 Domain Analysis 32

2.3 Types of Reusable Software Systems 34

2.3.1 Passive Systems 35

2.3.2 Active Systems 39
2.4 Reuse, Design Pattern and the Object-Oriented Paradigm 44
2.4.1 Program to an Interface, not an Implementation 45
2.4.2 Object Composition 46
2.4.3 Delegation 48
2.5 Current Trends 49
2.5.1 Challenges in System Development 50
2.5.2 The Common Request Broker Architecture (CORBA) 51

Chapter 3: Abstract Data Views, Design Patterns, and Software Architecture 53
3.1 Abstract Data View 54
3.1.1 ADV and Software Reuse 56
3.2 Design Patterns 58
3.2.1 Abstraction and Design Pattern 59
3.3 Architecture Approach to Software Construction 62
3.3.1 Software Architecture and Abstraction 63
3.3.2 Benefits of Architectural Approach to Software Construction ... 64

Chapter 4: Outline of Our Work 67
4.1 What is the Enterprise Application Integration Problem? 68
4.2 Solution to the Enterprise Application Integration Problem 69
4.3 Generic Adaptive Application Integration Architecture Model 71
4.3.1 Domain Applications 74
4.3.2 Domain Application Adapters 74
4.3.3 Asynchronous Distributed Object Framework and Infrastructure Services 75
4.3.4 Mediation Services 76
4.3.5 Automated Mapping 78
4.3.6 Presentation Services 79
4.3.7 Thin Client Applications 79

4.4 Frameworks and Patterns of Interaction 80
4.4.1 Coordination Pattern 80
4.4.2 Configuration Pattern 82
4.4.3 Model Pattern 83

Chapter 5: Adaptive Orthogonal N-Tier Integration Architecture 86
5.1 The Need for Application Portfolio Integration 86
5.2 Traditional Approaches to Enterprise Application Integration 88
5.3 N-Tier Orthogonal Application Integration Architecture 91
5.4 Implementation and Protocol of the Enterprise Mediation Layers 94
5.4.1 Component Construction 96
5.4.2 Component Interaction 98

Chapter 6: The Adaptive EAI Framework 102
6.1 Distributed Object Framework 104
6.2 Domain Application Adapters 109
6.2.1 Domain Application Adapter Design Pattern 110
6.3 Application Adapter Mediation 112
6.3.1 Application Adapter Mediation Pattern 113
6.4 Event Mediation 113
6.4.1 Event Mediation Pattern 114
6.5 Package Mediation 115
6.6 Flexible Business Process 115
6.7 Putting it Together 117

Chapter 7: OSS Integration in the Telecommunications Industry 119
7.1 Key Industry Standards 121
7.2 Solution to the Telecom OSS Integration Problem – a business process centric approach 122
7.3 Information Architecture: Static Domain Model 130

7.3.1 Customers and Order 130
7.3.2 Service Enrollment Simplified 131
7.3.3 Order Operations 132
7.3.4 Offerings and Offering Instances 134
7.3.5 Offerings 135
7.3.6 Customer and Service Locations 137
7.3.7 Customers and Service Enrollments 139
7.3.8 The Order World 140
7.3.9 The Customer World 142
7.3.10 Simplified Telco Organization Structure 143
7.3.11 Telco Organization in Detail 144
7.3.12 Instances of Business Activity Flows 145
7.3.13 Orders in the Flow of Business Activity 147
7.3.14 Worklists 148
7.3.15 Combined Business Activity Flow 149
7.4 Example: Get Customer Record for Viewing 151
7.5 Summary 155

Chapter 8: UML Model Based Component development Framework 157
8.1 Model Based Software Construction 157
8.2 Meta-Object Information Repository 162

Chapter 9: A mathematical Formalism for Specifying Design Pattern 165
9.1 Definitions and Concepts 166
9.2 Semantics of Design Patterns and their Specification Constructors 188
9.3 Closure of Design Patterns Under Composition 191
9.4 Examples Illustrating the Use of the Formalism Presented Above 193
9.4.1 The Transformation Process of Building the Document Framework Pattern 199
9.5 Applicability to Reuse 201

Chapter 10: Concluding Remarks and Future Work 203

10.1 Summary 204

10.2 Future Work 206

References 208

LIST OF ACRONYMS

ACID	Atomicity, Consistency, Isolation, Durability
ADO	Abstract Data Object
ADT	Abstract Data Type
ADV	Abstract Data View
ALV	Abstraction-Link View
API	Application Program Interface
ASL	Action Semantic Language
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off The Shelf
CRM	Customer Relationship Management
DOD	Department of Defense
DOM	Distributed Object Management
EAI	Enterprise Application Integration
ER	Entity Relationship
EUML	Extended UML
FSA	Finite State Automaton
HLA	High Level Architecture
HLL	High-Level Language
IDE	Integrated Development Environment

IDL	Interface Definition Language
IIOP	Internet Inter Operable Protocol
ISP	Internet Service Provider
IT	Information Technology
ITU	International Telecommunications Union
LA	Lexical Affinities
LIFO	Last-In-First-Out
MBCD	Model-Based Component Development
MIR	Meta-Object Information Repository
MOF	Meta Object Facility
MPP	Massively Parallel Processing
MVC	Model View Controller
NUI	Non-User Interface
OA&D	Object Analysis and Design
OMA	Object Management Architecture
OMG	Object Management Group
OODBMS	Object-Oriented Database Management System
ORB	Object Request Broker
OSS	Operations Support System
RDBMS	Relational Database Management System
RMI	Remote Method Invocation
SE	Service Enrollments
SI	Systems Integrators

SMP	Symmetric Multi-Processing
TMN	Telecommunications Management Network
UDL	Universal Design Language
UI	User Interface
UID	User Interface Design
UIDS	User Interface Design System
UML	Unified Modeling Language
VAR	Value Added Reseller
VHLL	Very High-Level Language
VLSR	Very Large-Scale Software Reuse
XML	Extensible Markup Language

LIST OF FIGURES

2.1 Two-level Abstraction Hierarchy 15

2.2 Mapping from a variable abstraction specification 17

2.3 Reuse System Genealogy 40

2.4 A Window class delegates its Area operation to a Rectangle instance 49

3.1 The Abstract Data View Model 56

3.2 The Modularization Theorem of Reuse of ADTs interpreted through ADVs 57

3.3 Generic two-level abstraction hierarchy for design patterns 60

3.4a Structure and participants of the Reactor design pattern 60

3.4b Structure and participants of the Factory Method design pattern 61

4.1 Generic Adaptive EAI Architecture Model 73

4.2 Distributed Object Framework 75

4.3 The Sub-Layers in the Mediation Services Layer 76

4.4 Example Coordination Interaction 81

4.5 Example Configuration Interaction 83

4.6 Generic Model Pattern 84

5.1 Traditional N-Tier Application Integration Architecture Model 90

5.2 Mediation Service Layers Implementation and Protocol 95

6.1 Adaptive EAI Architecture Framework 103

6.2 Persistent Object Service Components 105

6.3 Event Service Objects 106
6.4 Domain Application Adapter Design Pattern 111
6.5 API Specific Wrapper 112
6.6 Application Adapter Mediation Pattern 113
6.7 Event Mediation Pattern 114
6.8: ADV Representation of Business Processes 117
7.1 ITU Standard TMN Information Model 121
7.2 Generic OSS Integration Architecture 125
7.3 EAI Context Diagram 126
7.4 Application Mediation Server 127
7.5 Billing Application Adapter 128
7.6 CRM Application Adapter 129
7.7 Customers and Orders 130
7.8 Service Enrollment Simplified 131
7.9 Order Operations 133
7.10 Offering and Offering Instances 134
7.11 Offerings 136
7.12 Customer and Service Locations 138
7.13 Customer and Service Enrollments 139
7.14 The Order World 141
7.15 The Customer World 142
7.16 Simplified Telco Organization Structure 143
7.17 Detail Telco Organization Structure 145

7.18 Instances of Business Activity Flows 146

7.19 Orders in the Flow of Business Activities 147

7.20 Worklists 148

7.21 Combined Business Activity Flow 150

7.22 Customer Business Object 151

7.23 Get Customer from CRM Operation 152

7.24 Get Contact Information from CRM State Diagram 153

7.25 Get Contact Action Semantic Language 154

7.26 Retrieve Customer Data for Viewing Operation 155

7.27 Retrieve Customer Data Activity Diagram 156

8.1 Model-Based Component Development Framework 158

9.1 Commuting Diagram illustrating the homomorphism condition of the homomorphism h: $A \rightarrow B$ for the operation $w = (n : s_1 \times \ldots \times s_k \rightarrow s)$, $k \geq 0$ 170

9.2 Commuting diagram illustrating the vector homomorphism condition 173

9.3 A Design Pattern Fragment 176

9.4 Graphical Representation of a Module Signature 177

9.5 Schematic Representation of Design Patterns 180

9.6 Schematic Derivation Tree for Vector Algebra B being derived from Vector Algebra A 187

9.7 A General Schema for an ADV 189

9.8 A General Schema for an ADT 190

9.9 A General Schema Showing Inclusion of an ADT in an ADV 191

9.10 Generic Structure of the Factory Method Design Pattern 193

9.11 Instance of Factory Method Design Pattern 194

9.12 Factory Method Design Pattern with Interface 196

Chapter 1

Introduction

The problem of building large, reliable software systems in a controlled, cost-effective way, the so-called software crisis problem, is one of computer science's great challenges. From the very outset of computing as science, software reuse has been touted as a means to overcome the software crisis issue. At the 1968 NATO conference McIlroy presented the seminal paper on software reuse, Mass Produced Software Components [McIlroy 1968]. In this paper, he proposed the notion of a library of reusable software components and automated techniques for customizing the components to different degrees of precision and robustness. McIlroy envisioned that software component libraries could be effectively used for numerical computation, I/O conversion, text processing, and dynamic storage allocation.

Three decades later, the software community is still grappling with the problem of building large reliable software systems in a controlled, cost effective way; the software crisis problem is alive and well. Today, many computer scientists still regard software reuse as a very powerful vehicle to improve the practice of software engineering. The advantage of amortizing software development cost through reuse continues to be a major objective in the art of building software, even though the tools, methods, languages, and overall understanding of software engineering have changed significantly over the years.

In spite of its potential benefits, reuse has failed to become a reality in software development, in that the efficiency of software construction has not improved by an order of magnitude. In light of this failure, the computer science community has renewed its interest in understanding how and where reuse can be effective and why it has proven so difficult to bring the seemingly simple idea of software reuse to the forefront of software development technologies [Krueger 1992].

1.1 Expansive View of Software Reuse

Software reuse is the reapplication of a variety of existing knowledge during the construction of a new system to reduce the effort of development and maintenance of the new system. This reused knowledge includes artifacts such as domain knowledge, development experience, design decisions, architectural structures, module-level implementation structures, specifications, transformations, requirements, designs, code, documentation, etc. This expansive view of reuse is necessary because the more narrowly defined views of reuse, in general, have shown very little return on investment. The more narrowly defined views of software reuse include the following: "Reuse is the reapplication of code," "Reuse is the use of subroutine or object libraries," or "Reuse is the use of C++ classes" [Gamma 1996]. These views are all centered on the reapplication of code components. Source code languages induce a high degree of specificity on the reusability of software components and hence, the most highly reusable components tend to be small. Building systems out of small components leaves a lot of work to be done in building the architectural superstructure that binds the components into a whole system.

The cost to build this superstructure is typically much larger than the savings afforded by reusing a set of small components [Biggerstaff 1989].

One possible improvement is to make the code components larger. Unfortunately, this approach has a corresponding set of problems. As the software code components increase in size, the probability of reuse decreases. Their specificity reduces the likelihood that exactly the same set of requirements will arise again. Therefore, while the potential payoff for any single reuse may be high, it is mitigated both by the low likelihood of reuse and the significant effort that may be required to understand and adapt large components to the new system. This is the crux of what has been dubbed as the Very Large-Scale Reuse (VLSR) problem.

Thus, code-oriented reuse is not sufficient to unlock the full potential of software reuse. Code-oriented reuse is expected as a matter of course, but if we are to realize the full potential of reuse, we must look beyond code-oriented reuse to Very Large Scale Reuse.

VLSR introduces a whole new set of research problems centered around the issue of making the component representation sufficiently general to allow reuse over a broad range of target systems, and possibly across multiple domains. That is, VLSR mandates that we eliminate some of the specificity necessitated by a source code-oriented specification. We must determine representations that allow the large-grain components structure to be described precisely while leaving many of the small, relatively unimportant details uncommitted. Such representations must allow a broader range of information to be specified than source code can accommodate, e.g., design structures, domain knowledge, design decisions, etc.

There is great diversity in the software engineering technologies that involve some form of software reuse. However, there are commonalties among the techniques used. For example, software component libraries, application generators, source code compilers, and generic software templates all involve abstracting, selecting, specializing, and integrating software artifacts [Krueger 1992]. Software engineering technologies can be analyzed and contrasted in terms of their idiomatic reuse techniques along four metrics:

a. Abstraction

All approaches to software reuse use some form of abstraction for software artifacts. Abstraction is the essential feature in any reuse technique. Without abstractions, software developers would be forced to sift through a collection of reusable artifacts trying to figure out what each artifact did, when it could be reused, and how to reuse it.

b. Selection

Most reuse approaches help developers locate, compare, and select software artifacts. For example, classification and cataloging schemes can be used to organize a library of reusable artifacts and to guide software developers as they search for artifacts in the library.

c. Specialization

With many reuse technologies, similar artifacts are merged into a single generalized (or generic) artifact. After selecting a generalized artifact for reuse, the software developer specializes it through parameters, transformations, constraints, or some other form of refinement. For example, a reusable stack

implementation might be parameterized for the maximum stack depth. A programmer using this generalized stack would specialize or adopt it by providing a value for this parameter.

d. Integration

Reuse technologies typically have an integration framework. A software developer uses this framework to combine a collection of selected software components and specialized artifacts into a complete system. A module interaction language is an example of an integration framework [Prieto-Diaz 1986]. With a module interaction language, functions are exported from modules that implement them and imported into modules that use them. Modules are assembled into a system by interconnecting modules with the appropriate exports and imports.

1.2 Module Interface and Software Reuse

A major limiting factor to the reuse of designs and implementations of software objects and modules is the fact that they internalize knowledge about their surrounding environments. It is customary for a module or object of an application to know about its user interfaces, specifically details of how its data structures will be displayed, how the user will interact with the application, or what objects on the screen correspond to activation of components of the module. In addition, a module may know too much about the services offered by other modules. For example, a module may know too much about the naming conventions in a file system, or about the names of modules or functions from which it acquires services.

Such specific knowledge is counter to the notion of software reuse as well as to good software engineering practice. For instance, there are many ways that a data structure can be displayed, and since this is not an intrinsic property it should not be attached to the data structure. Input has a similar property. There are many ways that a user can interact with an application and so the application should not be aware of the mode of interaction. A module should know it requires services and specify that fact, but it should not specify how those services are supplied. That is, a component should not be aware of the syntactic or semantic structure of a component from which it acquires services. It follows that a disciplined approach to naming among components is a prerequisite to reuse of component specifications or implementations.

A module should be separated from user interactions or from the services supplied by another module or object. This requirement can be accomplished by using a specialized interface that isolates a module's interactions from knowledge of the interacting entities. The interface should be aware of the requirements of the module or object, but the module or object should not be aware of the interface. This approach to defining an interface implies a clear separation of concerns. Such a problem is often addressed in mechanical systems where a linkage "interface" joins two components, one of which supplies a service.

The literature is littered with treatises on various architectural models and programming approaches that have been proposed; these clearly separate the user interface from its corresponding application [Carneiro 1993; Olsen 1983; Green 1983; Bass 1991; Coutaz 1991; DEC 1991; Hill 1992; Hartson 1989; Krasner 1988; McCormack 1988; Hill 1986; Myers 1991]. However, in these architectural models, little

guidance is given to designing a program to have a reasonable level of assurance that the architecture will be followed. The model view controller (MVC) [Krasner 1988] and abstraction-link view (ALV) [Hill 1992] are specific implementation techniques that rely on contemporary programming models. For example, the MVC was originally introduced in Smalltalk and ALV used constraint programming in a LISP environment. These are excellent implementation strategies, but they are very difficult to map into other programming paradigms.

Windows toolkits such as X Windows [McCormack 1988] or Motif [OSF 1990] offered another approach to the module interface separation issue. These systems expose window components as objects that can be accessed by the application. Although it is possible to use these toolkits and maintain a high degree of separation, there is no well-defined approach as to how to achieve this goal. In addition, most window toolkits do not appear to support an appropriate level of abstraction for user interfaces. The toolkits tend to expose details such as the event dispatcher that places the control with the application resulting in asynchronous calls to the user interface components or a spaghetti of callbacks. Cowan, Lucena, and Stepien [Cowan 1993], [Cowan 1993a] espouse the view that control should reside with the user interface and not the application, and since this approach simplifies communication the toolkit should support that view. Systems such as Visual Basic [MSC 1991;WIC 1993] and Tk/Tcl [Ousterhoust 1994] conform very closely to this view of user interface.

1.3 User Interface and Reuse

Cowan and Lucena performed exhaustive examination of the problem of separation of concerns and reuse of designs. This led them to propose a new formal design model for both user interfaces and general module interfaces. There are some key requirements that they think the model should satisfy. The model should have the structure and operators to guide the designer into clearly separating the interface from the application and encourage the programmer to maintain that separation during the implementation. The model should also allow the designer to reason about the complete design and its various substructures. Furthermore, the model should be independent of a specific programming environment. They have created a design model called the abstract data view (ADV) [Cowan 1993; Cowan 1993a; Cowan 1993b] that makes significant progress in satisfying the above stated properties.

Using pairs of objects to represent application components and their interfaces in reusable designs provided the original motivation for the concept of abstract data views [Cowan 1993; Cowan 1993a]. The specific types of application components and interface components are called, respectively, abstract data objects (ADO's) and abstract data views (ADV's). An ADV is used as an interface (in a very broad sense) for ADO's in designs and provides a "view" of an ADO. Specification constructors are used to combine ADV's and ADO's to produce more complex designs, and this process has been validated by proof of concept architectures. The approach can be seen as a way of providing language support for the specification and abstraction of inter-object behavior [Helm 1990].

The ADV approach has been validated in a number of research prototypes. ADV's have been used to support user interface for games and a graph editor [Cowan 1992], to interconnect modules in a user interface design system (UIDS) [Cowan 1992], and to support concurrency in a cooperative drawing tool. In addition, it has been used to design and implement both a ray-tracer in a distributed environment [Lucena 1993] and a scientific visualization system for the Riemann problem.

1.4 Our Contribution

The contributions of this thesis are as follows:

1. Our work is primarily focused on the development of an Adaptive N-Tier Orthogonal Enterprise Application Integration (EAI) Architecture Framework [Linthicum 1999]. Software reuse and software integration are very closely related concepts since integration is the combination of two or more existing components. Without good integration tools and techniques, reuse is difficult and will probably not happen to any significant degree. In the development of the EAI architecture framework, the primary enabling concept is object-oriented design support by the unified modeling language (UML) [Harman 1997; Derr 1997]. The concepts of software architecture, design patterns, and abstract data views are used in a structured and disciplined manner in establishing a generic EAI framework. This framework is applied to solve the EAI problem in the telecommunications operations support system (OSS) marketplace.

   We used the concepts of design patterns [Gamma 1996; Fowler 1999], abstract data views [Cowan 1992; Alencar 1994], and software architecture

[Cowan 1993a; Booch 1999; Orfali 1998] to develop the EAI framework. Design patterns allow us to solve various pieces of the overall problem. For example, we developed a number of EAI design patterns that are used to integrate legacy third party applications into the EAI framework. These design patterns allow us to develop a very definite and repeatable process for integrating legacy as well as newly developed applications into a unified framework. The abstract data view approach with its compositional capability is used to aggregate and build up the overall solution by combining smaller macro components.

In addressing the EAI problem in a generic manner, our architecture centric approach presents solutions for the following broad problematic areas:

a. Facilitate the integration and interoperability of stove pipe legacy applications
b. Cater for a clear separation between the business models and machine models
c. Facilitate the development of adaptive business process re-engineering
d. Facilitate the use of the Internet as a business platform across the entire enterprise

The problem areas indicated by (a), (b), and (c) have been around for a long time and notoriously regarded as almost intractable problems in the sphere of the business community. Solving these problems will present a whole new way of looking at how we develop business software systems of the future.

2. We used the adaptive orthogonal EAI framework to develop a solution to the operations support system (OSS) problem in the telecommunications industry. The approach presents an adaptive business process integration framework where

business processes acts as collaboration agents between objects from lower levels of the architecture.

3. We present a model-based software development approach. This is an approach to raise the abstraction level at which application developers work and to automate the process of translation from an application model to its corresponding distributable runtime component. The basic thesis here is that we can effectively reverse the effort role in the software development process in which about 80% of the effort goes into the development of infrastructure services and 20% into the development of application logic.
4. We present a mathematical formalism for the specification of design patterns. This specification constitutes an extension of the basic concepts from many-sorted algebra [Zilles 1974; Enderton 1972]. In particular, the notion of signature is extended to that of a vector, consisting of a set of linearly independent signatures. The linearly independence property is necessary to satisfy non-interference that is essential for compositional based construction. This is of fundamental concern in the building of large-scale software systems where we have the composition of smaller components to form larger components. The approach can be used to determine efforts for component reuse and facilitate program verification. The approach has the potential to be able to aid complex software development by providing the developer with design alternatives and automatic program verification capabilities.

1.5 Outline of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 presents an outline of the previous work. Chapter 3 extends the discussion on previous work by presenting an overview of the central concepts of design pattern, abstract data views, and software architecture. These concepts form the foundation of our work. Chapter 4 presents an outline of our work. Chapter 5 provides the key concepts of the adaptive orthogonal n-tier integration framework. Chapter 6 presents the adaptive EAI architecture framework. Chapter 7 provides the solution to the telecom OSS integration problem. The solution incorporates a detailed domain analysis of the telecommunications domain. Chapter 8 presents the model based software development framework. Chapter 9 provides a mathematical formalism for the specification of design patterns. The formalism is an extension of the relevant many-sorted algebraic concepts. In chapter 10 we present our conclusion and future work.

Chapter 2

Outline of Previous Work

This chapter provides an outline of the major concepts that have influenced the general thinking in the area of software reuse. Some fundamental concepts such as *abstraction* and *classification* and the role they play with respect to software reuse are examined. We also present various models of software reusability.

2.1 Abstraction

Abstraction is an essential part of any software reuse system and as such can be viewed as a unifying theme for software reuse. This notion reflects the view that successful application of a reuse technique to a software engineering technology is inexorably tied to raising the level of abstraction for that technology. Raising the abstraction levels for software engineering technologies has proven to be extremely difficult, thus the relation between abstraction and reuse provides us with the first clue as to why there are so few successful reuse systems.

The relationship between software reuse and abstraction has been noted in the literature [Booch 1987; Parnas et al. 1989; Wegner 1983]. Wegner states that "abstraction and reusability are two sides of the same coin." He states that every abstraction describes a related collection of reusable entities and that every related collection of reusable entities determines an abstraction.

2.1.1 Abstraction in Software Development

Abstraction is a tool that is used by software practitioners and computer scientists to help manage the intellectual complexity of developing very large software systems [Shaw 1984]. An abstraction for a software artifact is a succinct description that suppresses the details that are unimportant to the software developer and emphasizes the information that is important. For example, the abstraction that is provided by a high level programming language allows a programmer to construct algorithms without having to worry about the details of hardware register allocation.

Software typically consists of several layers of abstraction built on top of the raw computer hardware. The lowest level software abstraction is object code, or machine code. Assembly language is a layer of abstraction above object code. A high-level programming language, like C, is a layer of abstraction above the assembly language level. In object-oriented languages such as C++, the class specification can serve as a layer of abstraction above the implementation details.

These examples demonstrate that every software abstraction has two levels. The higher of the two levels is referred to as the abstraction specification. The lower, more detailed level is called the abstraction realization. When abstractions are layered, the abstraction specification at one layer is the abstraction realization at the next higher layer. Figure 2.1 shows a hierarchy with two abstractions, L and M. Rep 1, Rep 2, and Rep 3 are three representations of the same software artifacts, where Rep 1 is the most detailed (lowest level) representation. For abstraction L, Rep 2 is the abstraction specification, and Rep 1 is the abstraction realization. From the point of view of abstraction M, Rep 3 is the abstraction specification, and Rep 2 is the abstraction realization.

An abstraction is composed of three sections: a hidden part, a variable part, and a fixed part. The hidden part consists of details in the abstraction realization that are not visible in the abstraction specification. The variable part and the fixed part are visible in the abstraction specification. The variable part represents the variant characteristics in the abstraction realization, whereas the fixed part represents the invariant characteristics in the abstraction realization. Therefore, an abstraction specification with a variable part corresponds to a collection of alternate realizations. The variable part of an abstraction specification maps into the collection of possible realizations. Figure 2.2 illustrates the mapping between abstraction specifications and realizations.

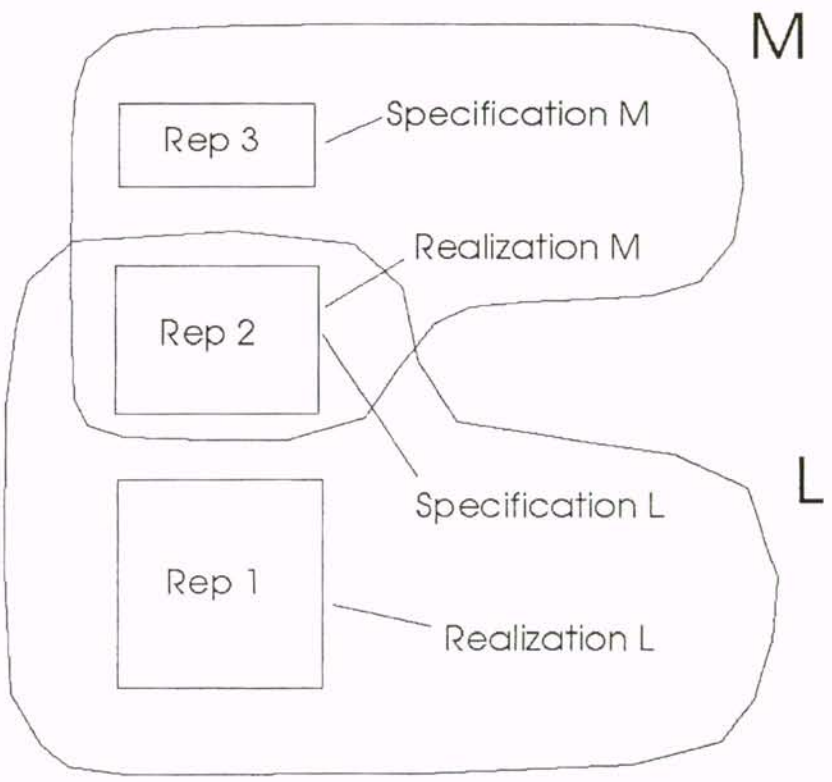


Figure 2.1: Two-level Abstraction Hierarchy

To illustrate this notion, consider the canonical stack example. The fixed part of the abstraction specification expresses the invariant characteristics for all stack realizations, such as the last-in-first-out (LIFO) semantics. The invariant stack behavior does not depend on the type of elements stored in the stack. Hence, the element type can be considered to be a constituent of the variable part of the abstraction specification. Different element types therefore correspond to different stack realizations.

This view is consistent with the capabilities of traditional high level programming languages such as C. In this model, support for each element type would have to be explicitly programmed. This model contrasts significantly with that offered by the modern object-oriented model in which we have languages such as C++ that offer support for parameterized classes or generic template classes. In this model, we would have a single implementation of stack, as a template class, that supports different element types passed in as a parameter.

The partitioning of an abstraction into variable, fixed, and hidden parts is not an innate property of the abstraction but rather an arbitrary decision made by the creator of the abstraction. The creator decides what information will be useful to users of the abstraction and puts it in the abstraction specification. In addition, the creator may also decide which properties of the abstraction the user might want to vary and places them in the variable part of the abstraction specification. Continuing with the stack example, the value of the maximum stack depth can be placed in either the variable, fixed, or hidden part of the stack abstraction. If it is placed in the variable part, the user has the ability to choose the maximum stack depth. If the maximum stack depth is placed in the fixed part,

the user knows the predefined value of the maximum stack depth but cannot change it. If placed in the hidden part, the stack depth is totally removed from the concerns of the user.

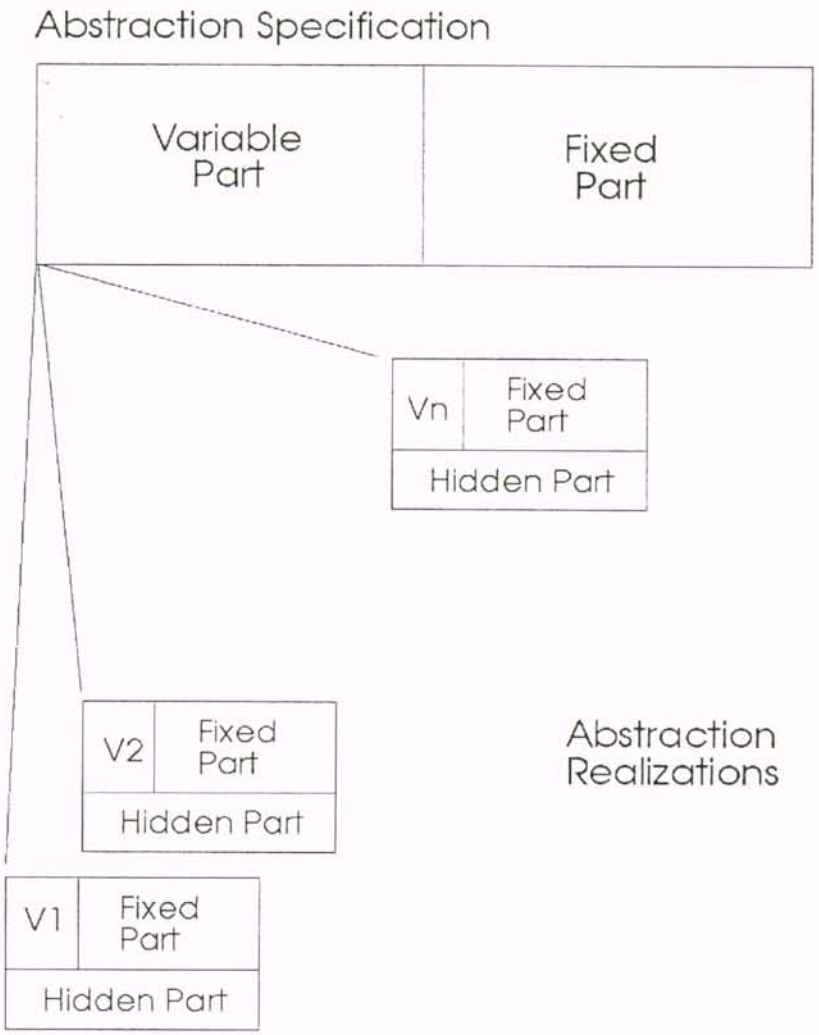


Figure 2.2: Mapping from a variable abstraction specification

Abstraction specifications and realizations can take on many forms. They can be formal or informal, explicit or implicit. Once again, consider the stack example written as a generic C++ template. The abstraction realization corresponds to an instantiation of the generic package with a particular stack element type. The abstraction specification, on the other hand, must be a combination of different descriptions due primarily to C++ limited expressiveness. The generic template class will provide the syntactic specification for operations of the stack abstraction, but the semantic specification must be expressed outside of the C++ language. One possibility is to use a formal notation such as Hoare axioms [Hoare 1969; Sun 1996]. Another is to use an informal description such as English text.

In summary, an abstraction expresses a high-level, succinct, natural, and useful specification that corresponds to a less perspicuous realization level of representation. The abstraction specification describes "what" the abstraction does, whereas the abstraction realization describes "how" it is done. For an abstraction to be effective, its specification must express all of the information that is needed by the person who uses it. This may include space/time complexity characteristics, precision statistics, scalability limits, and other information not normally associated with specification techniques.

2.1.2 Abstraction in Software Reuse

Abstraction plays a central and often limiting role in each of the other facets of software reuse:

- Selection

Reusable artifacts must have concise abstractions so users can efficiently locate, understand, compare, and select the appropriate artifacts from a collection.

- Specialization

  A generalized reusable artifact is in fact an abstraction with a variable part. Specialization of a generalized artifact corresponds to choosing an abstraction realization from the variable part of an abstraction specification. The object-oriented paradigm somewhat extends this notion. Inheritance, one of the key ideas of the object paradigm, allows for the abstraction realization to be implemented as a specialization derived from a previously defined parent class. This is the generalization or Is-a relationship between super-class and sub-classes.

- Integration

  To effectively integrate a reusable artifact into a software system, the user must clearly understand the artifact's interface (i.e., those properties of the artifact that interact with other artifacts or the integration framework). An artifact interface is an abstraction in which the internal details of the artifact are suppressed.

2.1.3 Cognitive Distance

Cognitive distance is defined as the amount of intellectual effort that must be expended by software developers to take a software system from one stage of development to another [Kruger 1992]. From this definition, it is clear that cognitive distance is not a formal metric that can be expressed with numbers and units. Rather, it is an informal notion that relies on intuition about the relative effort required to accomplish various software development tasks.

The effectiveness of abstractions in a software reuse technique can be evaluated in terms of the intellectual effort required to use them. Better abstractions means that less effort is required from the user.

The creator of a software reuse technique should strive to minimize cognitive distance by (1) using fixed and variable abstractions that are both succinct and expressive, (2) maximizing the hidden part of the abstractions, and (3) using automated mappings from abstraction specification to abstraction realization (e.g., compilers). This can be summarized in an important truism about software reuse [Kruger 1992]:

> *For a software reuse technique to be effective, it must reduce the cognitive distance between the initial concept of a system and its final executable implementation.*

This truism, along with others in the software reuse literature, are obvious and seemingly simple requirements on software reuse techniques that have proven very difficult to satisfy in practice.

2.2 Classification of Reusable Modules

The capability to classify and store as well as to identify and locate software components, is an increasingly important activity in software development environments where the notion of reuse is taking on added significance. Classification schemes are essential for setting up and maintaining a software library. A software library is a changing and growing collection of modules that have been certified as reusable components.

For code reuse to be attractive, it must require less effort than the creation of new code. Code reuse involves three steps: (1) accessing the existing code, (2) understanding it, and (3) adapting it [Kruger 1992]. A classification scheme is central to code

accessibility. Code understanding depends on both the reuser experience and program characteristics such as size, complexity, documentation, and programming language. Code adaptation depends on the differences between requirement and the features offered by the existing components and on the skills of the reuser.

Classification of a collection is central to making code reusability an attractive approach to software development. A collection organized by attributes related to software development will reduce the probability of retrieving non-relevant components. A search-and-retrieval mechanism is necessary for a classified collection to be of value. An effective retrieval system must have a well-defined classification structure embedded within. In addition, the classification and retrieval system must be able to differentiate between very similar components in the collection, thus allowing the user to select the component that requires the least adaptation effort. A proper classification must be based on an integrated solution: a classification scheme embedded in a retrieval system and supported by an evaluation mechanism.

A classification scheme that caters to reusability must be designed with the features of expandability, adaptability, and consistency as integral to its operation. Expandability allows new classes to be added to the collection with minimum disturbance, i.e., with little or no reclassification of the components. An adaptable classification scheme can be customized for different environments. Consistency allows components from different collections in the same class to share the same attributes. Hence, this feature permits different organizations to share their collections.

2.2.1 Software components

This section attempts to shed some light on the creation of software components as a result of a reclamation process based on the dissection and decomposition of existing software systems. It also examines the use of software components through interfacing and decomposition.

Megaprogramming is the term commonly used in reference to the construction and engineering of software systems from existing components, as contrasted with software development by coding one instruction at a time. The analogy is obviously to industrial mass production techniques. The main goal is to reduce time-to-market and improve the reliability and maintainability of the final product. The economics of scale indicate, if not dictate, that megaprogramming is indeed the future of the software market place.

There are two main dimensions to the notion of megaprogramming. First is the notion of a brokerage that supervises overall development of product line and releases the product to end users (black-box reuse). Second is a component library system that users interact with and can extend by using existing components as a template for constructing new ones.

A conceptual framework is defined that distinguishes among three aspects of software component [Marciniak 1994]:

- The concept or abstraction specification that the component represents,
- The content or the abstraction realization of the component, and
- The context under which the component is defined or what is needed to complete the definition of a concept or content within a certain environment.

The concept represented by a reusable software component is an abstract description of "what" the component does. Concepts are identified through requirement analysis or domain modeling and provide the desired functionality for some aspects of a system. An interface specification and a description of the semantics associated with each operation realize a concept. The content represented by a reusable software component is an implementation of the concept or "how" a component does "what" it is supposed to do. It assumes that each reusable software component may have several implementations that obey the semantics of its concept. The context represented by a reusable software component depends on understanding and expectations based on familiarity with previous implementations.

With the objective being the development of useful, adaptable, and reliable software modules from which new applications can be built, the following three requirements [Marciniak 1994] should be addressed by a component-centered model of a system:

1. Components must be *useful*, i.e., they must meet the high-level requirements of at least one concept necessary to design and implement a new software application.
2. Components must be *adaptable*, i.e., they must provide a mechanism such that modules can be easily tailored to the unique requirements of an application. The inheritance principle of object-oriented software design supported by the C++ language provides an approach to facilitate the adaptability requirement.

3. Components must be reliable, i.e., they must accurately implement the concept that they define.

Each component is basically made up of code plus interface specifications. The problem of code development is generally more tractable than the problem of providing precise, unambiguous and generalized interface specification. This is an alternative way of stating the known fact that raising the level of abstraction for a particular domain is a very ***hard*** problem. The software industry is in the process of specifying and developing some aspects of the requisite technologies to define formalisms for interfaces, so that software components could inter-operate smoothly. The Common Object Request Broker Architecture (CORBA) developed by the Object Management Group (OMG) facilitates distributed object communication [OMG 1997]. The High Level Architecture (HLA) proposed and sponsored by the Department of Defense (DOD) is another example of an effort to facilitate distributed object interoperability [Carothers 1997; Dahmann 1997]. HLA is primarily focused on distributed simulation.

2.2.2 A Software Component Reuse Model

Reuse is the use of previously acquired concepts or objects in a new situation. Reusability is a measure of the ease with which one can use those previous concepts or objects in the new situation. This very general view assumes that knowledge has been coded at different levels of abstraction and stored for future reuse [Freeman 1983].

Models of reuse are operational in well-established disciplines such as civil or electrical engineering. In these domains, the number of alternatives is usually large and

several combinations of components may give feasible solutions, thus creating a selection problem. It is customary to acquire components rather than to create them. Components are described by standard attributes that capture their functional characteristics.

A model for software component reusability is based on the above observations and on the assumption that available components usually do not match the requirements perfectly, making adaptation the rule rather than the exception. The general approach is to provide an environment that assists in the finding of components and estimates the adaptation and conversion effort necessary to effect reuse. The process is as follows:

- A set of functional specifications is given. The user then searches a component library to find the candidates that satisfy the specification. This step can take several iterations, with each progressively narrowing the search space.
- If a component satisfying all the specification is available, then reusing it becomes trivial.
- The more typical scenario is one in which several candidates exist, each satisfying some of the specifications. In this situation, the problem is transformed into one of selecting and ranking the available candidates based on how well they match the requirements and on the effort required to adapt the non-matching specification.
- Once an ordered list of similar candidates is available, the reuser selects the easiest to reuse and adapts it.

Selecting similar components is a classification problem. The degree of similarity depends on how the collection is organized. Closely related components may be grouped

by carefully selecting relevant attributes and meaningfully organizing them. The classification scheme is a central component in the software component reuse process.

2.2.3 Classification Principles

A classification principle describes how to classify components so that they can be located for reuse. Classification makes explicit the relationship among things and among classes of things. The result of a classification is a structure that details the relationships between objects and classes of objects. A classification scheme is a tool for the production of systematic order based on a controlled and structured index vocabulary called the classification schedule. The classification schedule consists of a set of names representing concepts or classes, listed in a systematic order to display the relationship between the classes [Buchanan 1979].

A classification scheme must be able to express both the hierarchical and syntactical relationships. Hierarchical relations employ the principle of subordination or inclusion in which a universe is successively divided into its component classes. On the other hand, syntactical relationships relate two or more classes from different hierarchies. In practice, classification schemes are hierarchical in nature, with syntactical relationships being manifested as compound classes. For example, the compound class "respiration of birds" relates the term *respiration* from the class "processes" with the term *birds* from the class "taxonomy".

Classification schemes can be either enumerative or faceted. The enumerative method postulates a universe of knowledge divided into successively narrower classes that include all the possible compounded classes. These are then arranged to display their

hierarchical relationships. The Dewey decimal classification [Dewey 1979] is a typical example of an enumerative hierarchy, where all possible classes are predefined.

The faceted classification scheme, used in library science, relies on the building up or synthesizing of compound classes from the subject statements of the particular documents, as opposed to the decomposition of a universe used in the enumerative schemes. In this approach, subject statements are analyzed and their component elemental classes determined. These classes are then listed in the classification schedule. The generic relationships of the elemental classes are the only relationships displayed. Compound classes are expressed by assembling their elemental components. This process of constructing a compound class from its elemental components is called synthesis. The arranged groups of elemental classes that make up the scheme are the facets. The elements or classes that make up a facet are called terms [Prieto-Diaz 1985; Prieto-Diaz 1991a].

Facets are considered as perspectives, viewpoints, or dimensions of a particular domain. This is because the characteristics of the facets are determined by the nature of the application. Different kinds of applications will have different perspective of a particular domain and this will in turn determine the existence of relationships (grouping) between the elemental classes.

Both enumerative and faceted schemes can be used to express the same number of classes. The difference is that in the enumerative scheme, classes with more than one elemental component are listed ready-made, while with the faceted scheme the classifier will have to make multi-element classes by synthesis. A problem typical of enumerative schemes is traversing the hierarchical tree to find the most appropriate class. Implicit to

the use of this scheme is the expertise of the librarian in both the classification scheme and the subject matter or domain which guide him to determine the most appropriate class. This is usually a difficult task because more than one class may be applicable. Cross-references are usually established to compensate for ambiguities in the class selection process. This is a cumbersome and error-prone process.

In the faceted scheme, both facets and terms are derived from analysis of a representative sample of the collection to be classified. The synthesis process used in the construction of compound classes tailors each class to a perfect fit. This makes the faceted approach very attractive for classifying reusable software components. The ordering of a facet's characteristics coupled with the fact that facets can be ordered by their relevance to the users of the collection is termed *citation ordering*. Citation ordering enhances search and retrieval performance when used to organize a database. Terms within a facet can be arranged based on how closely they relate to each other (conceptual closeness). This feature provides a way for locating similar components in a collection – an essential feature for software reusability.

2.2.4 Software Classification

Any reasonable software classification scheme must make the following assumptions about the collection of reusable software components: (1) that the number of components are very large and growing continuously and (2) that there are large groups of similar components – even in very specific classes [Prieto-Diaz 1985]. Software components can be described by the function they perform, the way they perform it, and their implementation details, among other things. These descriptors can be mapped directly

into facets that may be ordered by their relevance to reusability. A component specification is thus reduced to a tuple of terms where each term is an attribute value of a selected facet. Prieto-Diaz and Freeman [Prieto-Diaz 1987] suggested that a characterization of the functionality (what it does) and the environment (where it does it) of a software component would suffice for classification.

If the description of a software component is to be used as both a classification code and a retrieval key, it must be brief, succinct, and semantically rich. That is, it must consolidate in a single descriptor the "what," "where," and "how" of the component.

With modern object-oriented development approaches and techniques such as parameterizing or template classes in C++, the impact of the external environment can be reduced considerably. Template classes can be used to develop generic software components that can operate on any object type. This could eliminate the need for low level algorithmic adaptation of software components when moving between domains. The high-level abstraction specification would truly capture the semantics of the software component. Under these circumstances the reuser would only have to focus on user interface issues that are relevant to the particular domain.

A faceted scheme can be developed using the facets from the functional and environmental characterizations. The citation order is based on relevance to users and assuming that the typical users of the collection are software engineers designing and building new systems from components, the following citation order can be adapted: function, objects, medium, system type, functional area, and setting [Prieto-Diaz 1987]. Classifying a component consists of selecting the sextuple that best describes the component. Some examples follows:

<add, integers, array, matrix-inverter, modeling, aircraft-manufacturer>
<compress, files, disk, file-header, DB-management, catalog-sales>
<compare, descriptors, stack, assembler, programming, software-shop>

The Prieto-Diaz and Freeman classification method employs a *controlled vocabulary* technique for indexing software components. They have used this approach to avoid duplicate and ambiguous descriptors of software components arising from synonyms. Describing code using controlled vocabulary is not problem prone for any audience. A term thesaurus can be used to gather all synonyms under a single concept. The term that best expresses the concept would be chosen as the representative term [Prieto-Diaz 1989]. The thesaurus is used primarily for vocabulary control and for broadening the index vocabulary. These uses also enhance recall performance. A thesaurus can also be used to control the size of schedules. This can be done by increasing the number of terms assigned to a particular group or by breaking up groups into terms. Ambiguities between the term lists can be resolved be selecting a number of contexts.

Prieto-Diaz notes that keyword-based retrieval is good for books and journal articles because of the large amount of free text [Prieto-Diaz 1991a]. Software's characteristics make it a candidate of controlled vocabulary retrieval approach. A predefined set of keywords is used for indexing as described in the faceted approach. Software is a good candidate for faceted classification. First, software has a low amount of free text. Second, the programmers establish software keyword conventions. Last what the components do and how they do it is uncertain from their free text.

Guru parses the natural language documentation of the component source code for classification [Maarek 1991]. Other library systems only parse the comments or free text of the component. In addition, Guru uses the concept of lexical affinities (LA), as opposed to single terms typically used in other reuse libraries. LAs are "lexical affinity, ..., between two units of a language stands for a correlation of their common appearance in the utterances of the language." Other research has shown that such word relationships are separated by at most five words. These LAs are used in the creation of Guru's component indices. The indices are used to locate functions that match a user query. The indices are organized in a hierarchical format with the description and function being similar between siblings. The format is very similar to the hierarchical organization of classes in object-oriented languages.

2.2.5 Conceptual Closeness

This is a measure of closeness among terms in a facet [Prieto-Diaz 1985; Prieto-Diaz 1987]. In situations where a reuser cannot find an exact match to his search criteria, any reasonable software reuse system should present him with list of "likely" components ordered from most likely to least likely matched. The notion of conceptual closeness is to present a mechanism for the determination of similarity of software components within a software reuse system. A conceptual graph can be used to measure closeness among terms in a facet. It is defined as an acyclic directed graph in which the leaves are terms and the internal nodes are supertypes that denote general concepts relating two or more terms [Prieto-Diaz 1987]. The user assigns weights in the edges of the graph. The smaller the value of the weight, the closer is the perceived relationship of a term to a supertype.

The concept of closeness measurement could be utilized during the component retrieval process. In cases where the query for a term cannot match any descriptor, a retrieval system can check the nearby terms for related items. It is time-consuming to construct a conceptual graph with more than a few terms. However, the basic graph structure doesn't change much during the expansion of the collection of software components, and it also tends to remain stable. Conceptual graph construction can be considered a substantial but one-time effort. Regardless, once constructed, a conceptual graph would need tuning as users provide feedback on retrieval performance.

2.2.6 Domain Analysis

To make the faceted classification scheme a more efficient method for a software component reusability, the *domain analysis* methodology is recommended. This section provides an introduction to domain analysis and its application to classification and software reuse. According to Arango: "domain analysis is a knowledge intensive activity for which no methodology or any kind of formalism is yet available" [Arango 1988].

Domain analysis is an activity that happens even before the system analysis phase of the software development life cycle, and creates a domain model to support the system analysis. This information/model can be used in the subsequent phases of the software development process. In the domain analysis process "information used in developing a software system is identified, captured, and organized with the purpose of making it reusable when creating a new system" [Prieto-Diaz 1989]. Domain analysis can play an active role in the creation and organization of reusable software artifacts. Matsumoto

[Matsumoto 1987] reported the successful application of domain analysis in the development of software factories.

The domain analysis process can be incorporated into the software development process. A simplified three-step domain analysis procedure to advance reuse is:

1. Identification of reusable entities
2. Abstraction or generalization of those entities
3. Classification and cataloging for further reuse

Based on the above procedure, Prieto-Diaz proposed a procedural model for domain analysis [Prieto-Diaz 1989]. Using the faceted classification schemes, his methodology is "to create and structure a controlled vocabulary that is standard not only for classifying but also for describing titles in a domain specific collection" [Prieto-Diaz 1998].

In the context of domain analysis, Arango [Arango 1988] sees reuse as a learning system. In his proposed model, software development is a self-improving process which draws from a knowledge source that is named *reuse infrastructure*, and is integrated with the software development process. Reuse infrastructure consists of domain-specific reusable resources (i.e., components in particular and assets in general) and their descriptions. In Arango's reuse environment, by employing the reuse infrastructure and utilizing the specification of the software to be built, an implementation of the desired software is constructed. Then, the software thus proposed is compared against the input of the system (i.e., the specification of the system).

There are three particular functions that are crucial for reuse infrastructure. These functions [Prieto-Diaz 1989] are the abstractions of the duties of:

1. A librarian (making assets accessible to potential reusers)

2. An Asset Manager (controlling asset quality)

3. A reuse manager (facilitating the collection of domain analysis relevant data and coordinating all reuse operations)

Assets are those entities (documents, deliverables, and components) in the software development life cycle that is potentially reusable.

The typical process resulting from the integration of conventional software development and domain analysis is as follows:

1. Reusable resources are identified and added to the system.

2. Reuse data is gathered and fed back to the domain analysis process for tuning the domain models and updating the resource library.

The newly developed system can then be used to refine the reuse infrastructure [Prieto-Diaz 1998].

2.3 Types of Reusable Software Systems

There are two main types of reusable software systems: active and passive. Active systems have components that generate the final system. These systems are tailored to specific user needs. Passive systems are libraries of components such as the standard C library. These systems require knowledge of the components and how to use the components. The advantage of this type of library system is that the existing software can be easily added to the library. Therefore, the reuse components can be quickly incorporated into the development cycle.

2.3.1 Passive Systems

Passive libraries, such as the standard C++ or Java Class library, require the user to have some level of knowledge about their components without direct assistance from the library. Most passive libraries provide a written manual explaining each of the library's components. But, how does a user know which components will match his software needs without reading the entire manual?

Passive libraries weren't designed to be easily extendible. Typically, enhancements are only available with periodic releases of the library. This fosters numerous similar components to be developed between releases. Developers cannot wait for the next needed functionality. The relatively long time periods between releases do not support the responsiveness demand of software producers and consumers. The typical passive systems [Arnold 1988] are described in the following sections.

High-Level Languages

The reusable artifacts in a high-level language are assembly language patterns. High-level language constructs serve as abstraction specifications for low-level assembly language patterns.

High-level languages are often the lowest level of abstraction used by software developers. However, it is not widely recognized that high-level languages are examples of software reuse. Nor is it recognized that, in many ways, high-level language technology is a paragon of software reuse that researchers currently can only hope to emulate. For example, discovery of a new reuse technology that routinely offered a factor

of 5 speedup in software development would be among the most significant software engineering achievements of the decade.

The primary limitation of high-level languages as a reuse technology is the large amount of system design effort required prior to coding. Thus, there is a large cognitive distance between the informal requirements for a software system and its implementation in a high-level language.

Design and Code Scavenging

The reusable artifacts in scavenging are source code fragments. The abstractions for these artifacts are informal concepts that a software developer has learned from design and programming experience. When a programmer recognizes that part of a new application is similar to one previously written, a search for existing code may lead to code fragments that can be scavenged.

In ideal cases of scavenging, the software developer is able to find large fragments of high-quality source code quickly that can be reused without significant modification. In these cases, the developer goes directly from an informal abstraction of a design to a fully implemented source code fragment. In this situation, the cognitive distance between the initial concept of a design and its final executable implementation is small.

In practice, the overall effectiveness of code scavenging is severely restricted by its informality. A programmer can only scavenge those code fragments he or she remembers or knows how to find. In the worst case, a software developer spends more

time locating, understanding, modifying, and debugging a scavenged code fragment than the time required to develop the equivalent software from scratch.

These limitations lead to another truism of software reuse [Krueger 1992]:

> *For a software reuse technique to be effective, it must be easier to reuse the artifacts than it is to develop the software from scratch.*

Source Code Components

McIlroy's "Mass Produced Software Components" introduced the notion of software reuse by proposing an industry of off-the-shelf source code components. These components were to serve as building blocks in the construction of larger systems. Given a large enough collection of these components, software developers could ask the question "What mechanism shall we *use*?" rather than "What mechanism shall we *build*?"

Compared to code scavenging, reusable component libraries can be considerably more effective since components are written, collected, and organized specifically for the purpose of reuse. The most successful reusable component systems, such as the IMSL math library [Betts 1990], rely on concise abstractions from a particular application domain. One-word abstraction specifications such as *sine* often allow a software developer to go directly from an informal requirement to a fully implemented and tested source code component. Thus, the cognitive distance between the informal concept and its final executable implementation is very small.

For components that do not have simple abstractions, more general specification techniques are required. These descriptions can often be as difficult to understand as source code, thereby increasing the cognitive distance.

Creating a relatively complete and practical library of reusable components is a formidable challenge. Library implementers must have the theory, foresight, and means to produce a collection of components from which software developers can select, specialize, and integrate to satisfy all possible software development requirements. This is currently possible to a limited degree for specific application domains that have a rich and thorough theoretical body of knowledge, such as statistical analysis. General-purpose libraries, however, remain elusive for at least two reasons: (1) the implementation characteristics and tradeoffs for data structures and computations are widely variable, and (2) library size grows rapidly with respect to general-purpose component size.

Software Schema

Software schemas are a formal extension to reusable software components. Reusable components often rely on ad hoc extensions to programming languages to implement reuse techniques such as specification, parameterization, classification, and verification. With software schemas, however, these mechanisms are an integral part of the technology.

Compared to reusable source code components, reusable schemas place a greater emphasis on the abstract specification of algorithms and data structures and place less emphasis on the source code implementation. This shift in emphasis helps reduce the cognitive distance or separation between the informal requirements of a system and its

executable implementation by isolating the software developer from the source-code-level details.

Unfortunately, with software systems we do not have many universal abstractions above the stack, list, tree, etc. Therefore, the semantics of higher level abstractions are often expressed with logic formalisms and specification languages. Formal specification for schema abstractions can be large and complex. Even with automated tools it can be difficult for software developers to locate, understand, and use schemas. This complexity serves to increase the cognitive distance, thereby offsetting some of the advantages of using higher level abstractions. Hence, the challenge for implementers of a schema technology is to find abstraction formalisms that are natural, succinct, and expressive.

2.3.2 Active Systems

Since the late 1980s researchers recognized the failings of passive libraries and began proposing solutions. All solutions share two characteristics in common. The systems actively assist users in locating components that matched their needs. In addition, these systems take an active role in promoting the development of reusable software components. Therefore, these systems include component cataloging and retrieval functionality.

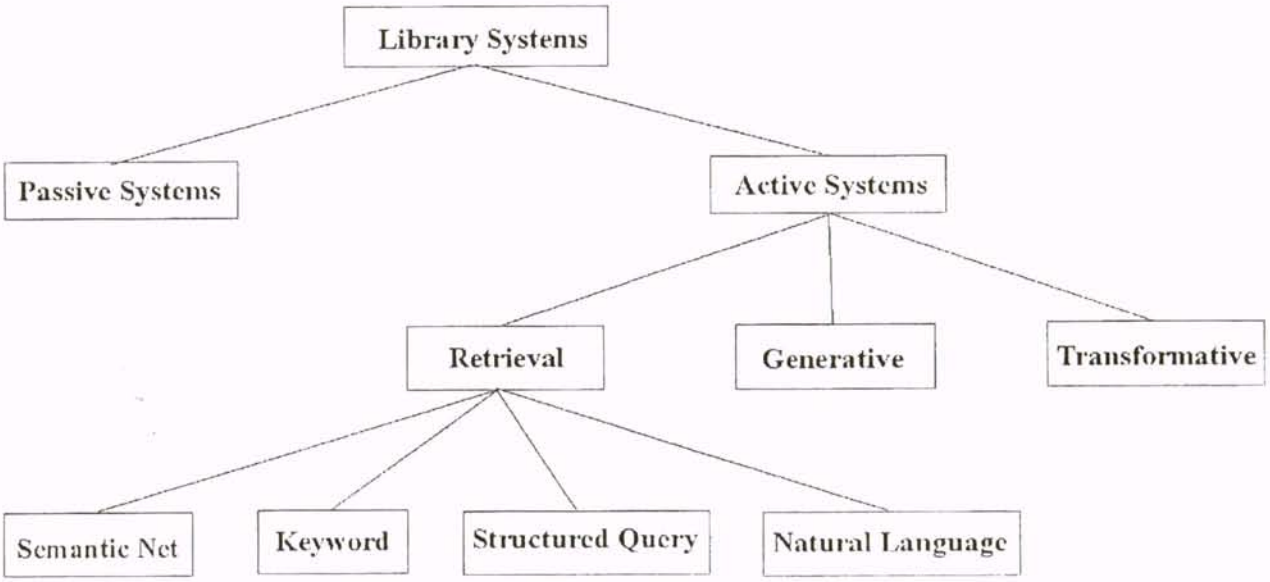


Figure 2.3: Reuse System Genealogy

Active systems can be grouped into several classes (Figure 2.3). These classifications include generative, transformative, and retrieval based systems. Within the retrieval classification, many types of retrieval mechanisms exist. These include semantic net, keyword, structured query, and natural language. The following sections present various systems as examples of active library types. In addition, a solution using object-oriented design is presented.

Application Generators

Application generators operate like programming language compilers; input specifications are automatically translated into executable programs [Cleaveland 1988], [Neighbors 1989]. Application generators differ from the traditional compilers in that the input specifications are typically very high-level, special-purpose abstractions from a

very narrow application domain [Levy 1986]. Application generators are appropriate in application domains where

- Many similar software systems are written,
- One software system is modified or rewritten many times during its lifetime, or
- Many prototypes of a system are necessary to converge on a usable product.

In these cases, the systems have significant source code overlap. Application generators generalize and embody the commonalities, so they are implemented once when the application generator is built and then reused each time a software system is built using the generator.

Application generators are specialized by writing an input specification for the generator. Due to the diversity in application abstractions, the techniques used for specialization are also widely varied. Examples include grammars, regular expressions, finite state machines, graphical languages, templates, interactive dialog, problem-solving methods, and constraints.

Very High-Level Languages

Very high-level languages (VHLLs) are an attempt at improving on the successes of conventional high-level languages (HLLs). Developing software with VHLLs is very much like developing software with HLLs. Both VHLLs and HLLs provide a syntax and semantics for expressing general-purpose computation.

VHLLs use high-level mathematical abstractions suitable for general-purpose software development. The goal of VHLL implementers is to find abstractions that are more natural and expressive than the abstractions in HLLs. As a result, VHLL programs

can be an order of magnitude more succinct than corresponding HLL programs. VHLLs are not, however, as powerful as application generators since application generators use domain-specific abstractions, which can be at a much higher level of abstraction.

The distinction between VHLLs and application generators exemplifies the tradeoff between *generality* and *leverage* in software reuse technologies [Biggerstaff and Richter 1989]. Typically, the more general a reuse technology is, the more effort is required to implement systems with it. The goal of VHLL research is to maximize the leverage offered by higher levels of specification without sacrificing computational generality.

Transformation Systems

Transformation systems are used to develop software in two phases:

1. Software developers describe the semantic behavior of a software system using a high-level specification language.
2. Software developers then apply *transformations* to the high-level specifications. The transformations are meant to enhance the efficiency of execution without changing the semantic behavior.

The two phases make a clear distinction between specifying what a software system does and the implementation issues of how it will be done [Zave 1984].

The first phase is equivalent to using a VHLL. Software developers create an executable system in a language that has a relatively small cognitive distance from the developer's informal requirements for the system [Balzer 1989]. The second phase in the transformational approach is essentially a human-guided compilation. The goal in this

phase is to produce an executable system that satisfies the high-level specification and that also exhibits performance comparable to an implementation in a conventional HLL. The transformation phase can be thought of as an interactive, human-guided compilation. Human intervention is necessary because issues such as automatic algorithm and data structure selection are beyond the current computer technology. By involving the software developer in the compilation process, transformational systems increase the cognitive distance in order to achieve better run-time performance.

Software Architectures

Reusable software architectures are large-grain software frameworks and subsystems that capture the global structure of a software system design. This large-scale global structure represents a significant leverage in the development of software. The leverage offered by software architectures comes from the small cognitive distance between informal concepts in an application domain and executable implementations. The mapping from abstraction specification to abstraction realization is mostly automated and this isolates the software developer from the hidden and realization parts of the abstraction.

Software architectures are analogous to very large-scale software schemas. Software architectures, however, focus on subsystems and their interaction rather than data structures and algorithms. Software architectures are also analogous to application generators in that large-scale system designs are reused. Application generators, however, are typically standalone systems with implicit architectures, whereas software architectures can often be explicitly specialized and integrated with other architectures to create many different composite architectures.

Draco is an example software architecture technology [Freeman 1987; Neighbors 1984, 1989]. Draco encapsulates software architectures in application generators. The output from the architecture generators can be used as building blocks for higher-level architecture generators, making Draco an architecture generator *generator*.

In Draco, each software architecture has a *domain language* and a set of *components* that implement the domain language. The domain language corresponds to the abstraction specification for an architecture. It captures the relevant abstractions for a software architecture in a particular domain.

2.4 Reuse, Design Patterns and the Object-Oriented Paradigm

The object-oriented approach to software development has emerged as one of the primary vehicles for the realization of software reuse. The features of inheritance, dynamic binding, and polymorphism offered by this paradigm provide an extremely powerful and elegant approach to software reuse, which differs fundamentally from other mechanisms. There are a number of design methodologies that exploit its basic structuring concepts to impose a discipline on the use of languages such as C++ and Java. These languages fully support the object-oriented approach to developing reusable software.

This section examines some of the more important principles and techniques that design patterns employ in solving design problems. Some of these are well-entrenched practices in the object-oriented software development community and are expressed as principles of reusable object-oriented design.

2.4.1 Program to an Interface, Not to an Implementation

An object's class defines how the object is implemented. The class defines the object's internal states and the implementation of its operations. In contrast, an object's type only refers to its interface (set of signatures) – the set of requests to which it can respond. An object can have many types, and objects of different classes can have the same type. Class inheritance defines an object's implementation in terms of another object's implementation. Hence, it's just a mechanism for code and representation sharing. In contrast, interface inheritance (or sub-typing) describes when an object can be used in place of another. In languages such as C++, inheritance means both interface and implementation inheritance. Pure interface inheritance can be approximated in C++ by inheriting publicly from pure abstract classes. Pure implementation or class inheritance can be approximated with private inheritance.

Although most programming languages don't support the distinction between interface and implementation inheritance, many of the design patterns depend on this distinction. For example, objects in a Chain of Responsibility must have a common type, but usually they don't share a common implementation. In the Composite pattern [Gamma 1996], Component defines a common interface, but Composite often defines a common implementation. Command, Observer, State, and Strategy are often implemented with abstract classes that are pure interfaces.

Class inheritance is basically a mechanism for extending an application's functionality by reusing functionality in parent classes. It lets you define a new kind of object rapidly in terms of an old one. It lets you get new implementations almost for free, inheriting most of what you want for free, inheriting most of what you need from exiting

classes. However, implementation reuse is not the end. Inheritance's ability to define families of objects with identical interfaces (by inheriting from an abstract class) is very important. This is because polymorphism depends on it.

There are two benefits to manipulating objects solely in terms of the interface defined by abstract classes:

1. Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.
2. Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface.

This greatly reduces implementation dependencies between subsystems that leads to the following principle of reusable object-oriented design [Gamma 1996]:

Program to an interface, not an implementation

The Creational patterns Abstract factory, Builder, Factory Method, Prototype, and Singleton let you instantiate concrete classes [Gamma 1996; Schmidt 1999]. By abstracting the process of object creation, these patterns give you different ways to associate an interface with its implementation transparently at instantiation. Creational patterns ensure that your system is written in terms of interfaces, not implementations.

2.4.2 Object Composition

Class inheritance and object composition are the two most common techniques for reusing functionality in object-oriented systems [Biggerstaff 1989; Blair 1989; Gamma 1996; Fowler 1999]. With the class inheritance approach, you define the implementation of subclasses in terms of parent or super classes. This is normally referred to as "white-

box" reuse because the internals of the super classes are often visible to the subclasses. With composition, new functionality is obtained by assembling or composing objects to get more complex functionality. This approach to reuse is called "black-box" reuse, because no internal details of objects are visible.

Class inheritance has the distinct advantage of being defined statically at compile-time and is therefore straightforward to use. This also makes it easier to modify the implementation being reused. On the other hand, class inheritance has some disadvantages. First, you cannot change the implementations inherited from parent classes at run-time. Second, and more limiting, parent classes often define at least part of their subclasses' physical representation and inheritance exposes a subclass to the details of its parent's implementation. Thus, the notion of "inheritance breaking encapsulation" [Sny86]. The implementation of a subclass becomes so bound up with the implementation of its parent class that any change in the parent's implementation will force the subclass to change.

Object composition, on the other hand, is defined dynamically at run-time through objects acquiring references to other objects. Composition requires objects to respect each other's interface, which in turn requires carefully designed interfaces. This approach has the very powerful benefit of not breaking encapsulation, because objects are accessed solely through their interfaces.

A design based on object composition has the following advantages: (a) it helps you keep each class encapsulated and focus on one task and (b) the classes and class hierarchies will remain small and will be less likely to grow into an unmanageable conundrum. This leads to another principle of object-oriented design [Gamma 1996]:

Favor object composition over class inheritance.

In practice, the set of reusable components is never rich enough to facilitate a purely compositional approach to software construction. Reuse by inheritance makes it easier to make new components that can be composed with old ones. Inheritance and composition thus complement each other.

2.4.3 Delegation

Delegation is a way of making composition as powerful for reuse as inheritance [Lieberman 1986; Johnson 1991]. In delegation, two objects are involved in handling a request: a receiving object that delegates operations to its delegate. This is analogous to subclasses deferring requests to parent classes. But with inheritance, an inherited operation can always refer to the receiving object, "this member" variable in C++ and "self" in Smalltalk. To achieve the same effect with delegation, the receiver passes itself to the delegate to let the delegated operation refer to the receiver.

For example, instead of making class Window a subclass of Rectangle, the Window class could reuse the behavior of Rectangle by keeping a Rectangle instance variable and delegating Rectangle-specific behavior to it. Figure 2.4 depicts a Window class delegating its Area operation to a Rectangle instance.

Delegation has a disadvantage it shares with other techniques that makes software more flexible through object composition: dynamic, highly parameterized software is harder to understand than more static software. There are also run-time inefficiencies, but the human inefficiencies are more important in the long run. Because of these disadvantages, delegation works best when it's used in highly stylized ways such as

standard patterns. The State, Strategy, and Visitor design patterns [Gamma 1996] make extensive use of delegation. Delegation is an extreme example of object composition. It shows that you can always replace inheritance with object composition as a mechanism for code reuse.

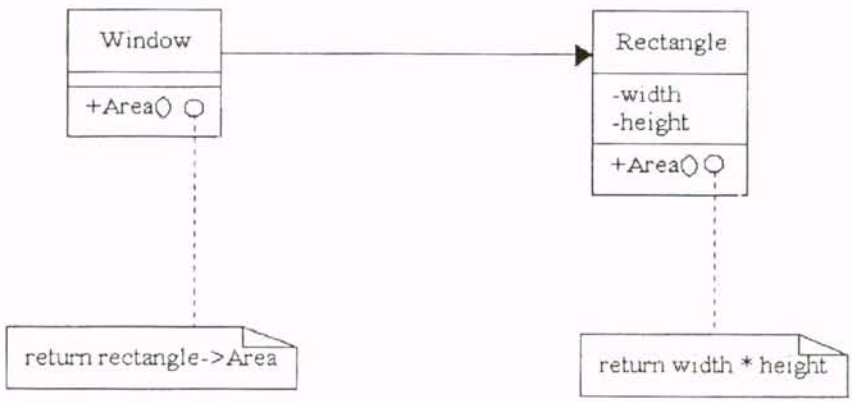


Figure 2.4: A Window class delegating its Area operation to a Rectangle instance

2.5 Current Trends

This section briefly examines some of the recent trends in the software development industry. The analysis was performed with the intention of determining how the object-oriented phenomena have been influencing the evolution of software construction and what role reuse has played in this process.

2.5.1 Challenges in System Development

System development today is about rapid change and responding to the realities of the business environment [Bigus 1998]. The key to successful system development is how well an enterprise can (1) perform system integration, (2) manage the future, and (3) find suitable supporting technology [Mowbray 1997].

Information systems development has changed from a reliance on unconstrained design and development to an increasing reliance on software integration methods in which new systems or applications are created by connecting components [OMG 1997]. Traditionally, integration has been viewed as simpler than new software development. However, this notion has proven to be incorrect within the current context of the software industry. Integration has not resulted in the deployment of new capabilities at either a faster or cheaper rate. This is due primarily to customization efforts to achieve interoperability among components that were not originally designed to work together. The investment to develop the interface can easily exceed the effort required to develop the code for the functions themselves.

Traditionally, software system development has been primarily focused on the development of monolithic single-ended software systems. The client server software paradigm is a notable exception to this theme. However, it still falls within a broader definition of monolithic systems with de-coupled client and server components. The focus on systems integration brings to light a void in the software development process, a model of developing truly distributed software.

2.5.2 The Common Object Request Broker Architecture (CORBA)

In recognition of this technology vacuum, the Object Management Group (OMG) was created in 1989 [DEC 1990; Mowbray 1998; OMG 1995]. The primary mandate was to develop a specification for defining interoperability of software components [OMG 1997]. The CORBA specification addresses two of the most prominent problems faced in the software industry: (1) the difficulty of developing client server applications and (2) how to rapidly integrate legacy systems, off-the-shelf applications, and new development.

CORBA is a specification for an emerging technology known as distributed object management (DOM) [OMG 1995; OMG 1997; Orfali 1998]. DOM technology provides a higher-level, object-oriented interface on top of the basic distributed computing services. At the most basic, CORBA defines a standard framework from which a software developer can easily and quickly integrate network-resident software modules and applications to create new, more-powerful applications. It combines object technology with a client server model to provide a uniform view of an enterprise's computing system - everything on the network is an object.

The twin concepts of software reuse and software integration are closely related since integration is the combination of two or more existing components. Without good integration tools and techniques, reuse is difficult and will probably not happen to any significant degree because, without a back plane or broker, custom interfaces must be defined for each interaction between components. With a broker, however, each interface is defined just once and the broker handles subsequent interactions. The CORBA Interface Definition Language (IDL) [Orfali 1998; OMG 1995] is used to define

interfaces in a standardized, platform-independent fashion. This offers a significant reduction in complexity to the software developer.

Summary

Software reuse is the process of creating software systems from existing software artifacts rather than redeveloping every facet of the new software system from scratch. This simple but powerful vision has failed to become a standard software engineering practice. This chapter surveyed the different approaches to software reuse found in the literature. Abstraction plays a central role in software reuse. Concise and expressive abstractions are essential if software artifacts are to be effectively reused. The effectiveness of a reuse technique can be evaluated in terms of cognitive distance – an intuitive gauge of the intellectual effort required to use the technique. Cognitive distance is reduced in two ways: (1) higher level abstractions and (2) automation. We have proposed design patterns as a way of raising the abstraction level.

The next chapter gives an overview of the important concepts of design patterns, abstract data views, and software architecture. The concepts are fundamental to the adaptive application integration architecture framework that we have developed.

Chapter 3

Abstract Data Views, Design Patterns, and Software Architecture

This chapter gives an overview of the central concepts of abstract data views (ADVs) [Cowan 1992; Alencar 1994], design patterns [Gamma 1996; Fowler 1999], and software architecture [Cowan 1993a; Booch 1999; Orfali 1998]. These concepts form the foundation of our work by providing a platform from which we developed a new approach to building large-scale reusable software systems. The abstract data view and its companion abstract data object (ADO) concepts are design constructs created with the notions of separation of concerns and reuse as important considerations. The term abstract data object (ADO) is very similar to that of an abstract data type (ADT), but is distinguished from an ADT by having the property of state. Design patterns are macro software design artifacts that express the static and dynamic structures and collaborations of components in a software architecture. An architectural approach to software development enables the imposition of an overarching structure that rationalizes, arranges, and connects components to produce the desired functionality.

These three concepts provide a very powerful approach to addressing the very large-scale software reuse (VLSR) problem from both a development and integration perspective. As pointed out in Chapter 2, any reasonable approach to the VLSR problem must provide an inherent mechanism for raising abstraction. The architectural approach

enforces a disciplined approach to decomposition, specification, and separation of functional modules or layers within a software system. Thus, software architecture searches to uncover abstractions and make them explicit. Design patterns are macro constructs that facilitate the reuse of various kinds of knowledge in the software construction process. The knowledge being reused is independent of the implementation technology. ADV allows us to partition a module or layer in the architecture by explicitly separating the specification part from the realization part of the module. The ADV concept also facilitates the building of new components or extensions of module functionality from existing ones using composition. Hence, these three conceptual approaches focus heavily on the use of the principle of abstraction in realizing their core functionality.

3.1 Abstract Data View

Abstract data views and abstract data objects are design constructs that divide the specification of software modules into two distinct types of components: the *interface* that is represented by the ADV and the *implementation* that is represented by the ADO. Both ADV and ADO are instances of abstract data types (ADTs). An ADV can be used to provide an interface between two ADTs or between an ADT and another medium such as a user or a network. In addition, the ADV concept facilitates the construction of larger component from smaller components using the principle of composition. Thus, an ADV is both a specification of an ADO and a method of interacting with the ADO. The ADV approach has been used in a number of prototype applications [Cowan 1992; Lucena 1992; Potengy 1993].

The formulation of the ADV model was motivated by work in composition technology [Fiadeiro 1993], and as a method of separating the specification of the user interface (UI) from the non-user-interface (NUI) components of an interactive system [Cowan 92] and was intended to promote design reuse. In the ADV concept, ADOs or ADTs are completely independent of the interface and have their requirements for data translated by one or more ADVs. In the context of user interfaces this implies that ADTs or ADOs do not have direct access to input or output. An ADV approach to software design describes all design decisions that relate to information exchange between the user and the UI application, among other ADVs, and between the NUI and UI applications. Figure 3.1 shows the relationship between input and output devices, ADVs, and ADTs. The relationship between ADV and ADT, shown by an arrow, associates the public interface of the ADT with the corresponding specification in the ADV. The name of the corresponding ADT in the ADV specification is represented by a variable called *owner*, which provides the connection between the UI and NUI parts of the system. The relationship between an ADV and an ADT is not symmetric.

In the abstract data view approach to software design, a consistency property needs to be satisfied. This is due to the fact that several ADV instances could be associated with the same ADT instance in order to provide different views or different control functionality for the same ADT.

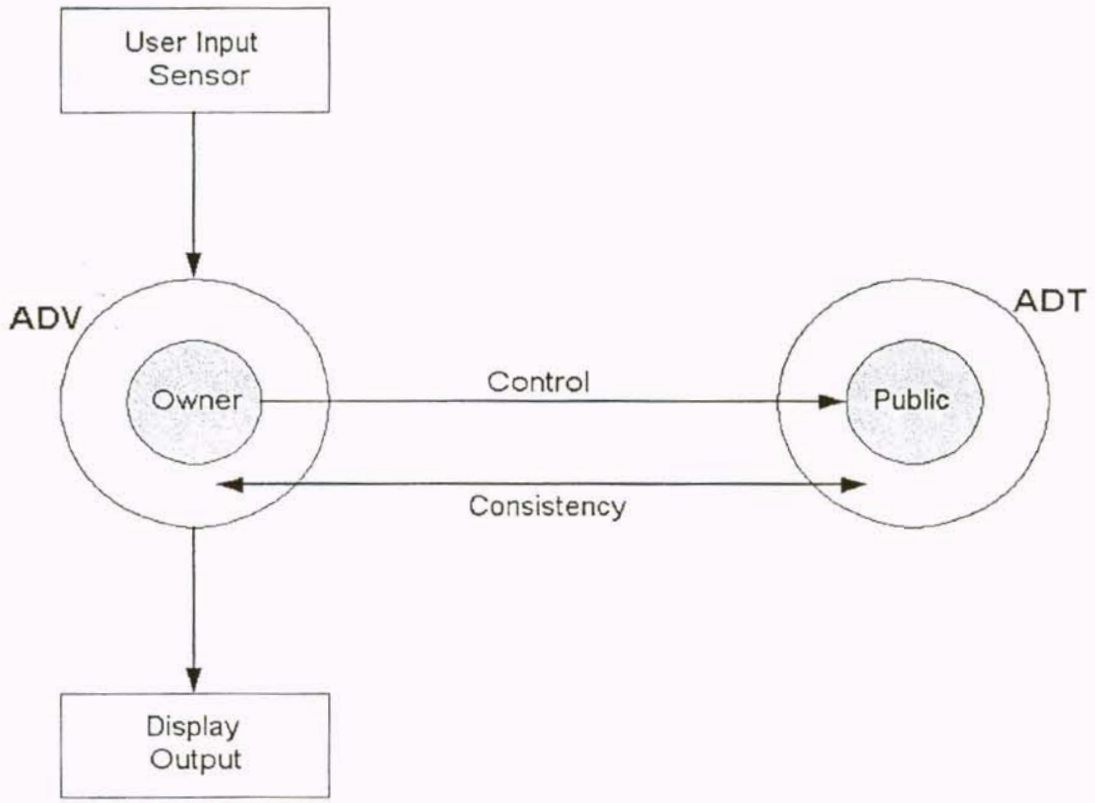
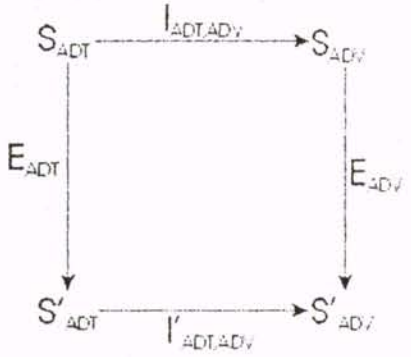


Figure 3.1: The Abstract Data View Model

3.1.1 ADV and Software Reuse

ADV design concept promotes reuse of interface specification through the principle of composition because it allows a complex interface to be built from simpler interface components. A composite module specification must guarantee syntactic non-interference and semantic context independence [Dennis 1973] among the different modules. Specification constructors are used to perform the combination of specifications and these include simple composition with locality, set and sequences of component

types, and inheritance of specification [Jones 1990] that are used for both ADV and ADO specifications.



S_{ADT} - ADT Specification
S_{ADV} - ADV Specification
I - Interpretation of A_{ADT} in terms of S_{ADV}
I' - Interpretation of A'_{ADT} in terms of S'_{ADV}
E_{ADT} - Extension of S_{ADT} by means of reusable ADTs using specification constructors
E_{ADV} - Extension of S_{ADV} by means of reusable ADVs associated with ADTs using specification constructors

Figure 3.2: The Modularization Theorem of Reuse of ADTs interpreted through ADVs

Within the context of ADV, the central property of very large-scale software reuse can be phrased in a style similar to that adopted by Gaudel [Gaudel 1986]: "An ADT specification extended by other reusable ADT specifications through the use of a given set of specification constructors can be interpreted (or viewed) as an equivalent ADV specification provided that the original ADT can be interpreted (or viewed) by an associated ADV that is extended by the application of the same given set of specification constructors applied to the ADVs associated with the reused ADTs" [Alencar 94]. That is, we need to show that the operators in Figure 3.2 commute. Using a suitable terminology [Turski 1987], we can say that we have a "modularization theorem" for the reuse-in-the-large of ADTs interpreted as ADVs.

3.2 Design Patterns

Design patterns [Gamma 1996] are a promising technique for capturing and articulating proven techniques for developing extensible large-scale software systems, which are invariably distributed in nature. Design patterns express the static and dynamic structures and collaborations of components in software architectures. Patterns aid the development of extensible distributed system components and frameworks by expressing the structure and collaboration of participants in software architectures at a level higher than (1) source code components or (2) object-oriented design models that focus on individual objects and classes [Schmidt 1996].

Experienced object-oriented designers do not solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, it becomes a component in their arsenal of tools for reuse. Consequently, you will find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented design more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience.

The unified modeling language (UML) graphical-based notation is an important and useful aspect of describing design patterns [Booch 1999; Derr 1995]. However, it is not sufficient because it simply captures the end product of the design process as relationship between classes and objects. To reuse the design, we must record the decisions, alternatives, and trade-off that led to it. In addition, concrete examples are a very important dimension to describing design patterns, because they help you see the designs in action.

Design patterns can be classified along two dimensions so that we can refer to families of patterns [Gamma 1996]. The first criterion, called *purpose*, reflects what a pattern does. Patterns can have either creational, structural, or behavioral purpose. The second criterion, called *scope*, specifies whether the pattern applies primarily to classes or objects. Class patterns have static relationships that are established through inheritance and fixed at compile time. Object patterns have more dynamic relationships that can be changed at run-time.

3.2.1 Abstraction and Design Pattern

The general structure of a design pattern can be modeled into a two-level abstraction hierarchy. The top level represents application independent structure, while the lower level represents application specific structure. Figure 3.3 illustrates the general abstraction hierarchy in design patterns. The structure and participants in the Reactor [Schmidt 1995b] and Factory Method [Gamma 1996] design patterns (Figures 3.4a and 3.4b) illustrate this notion.

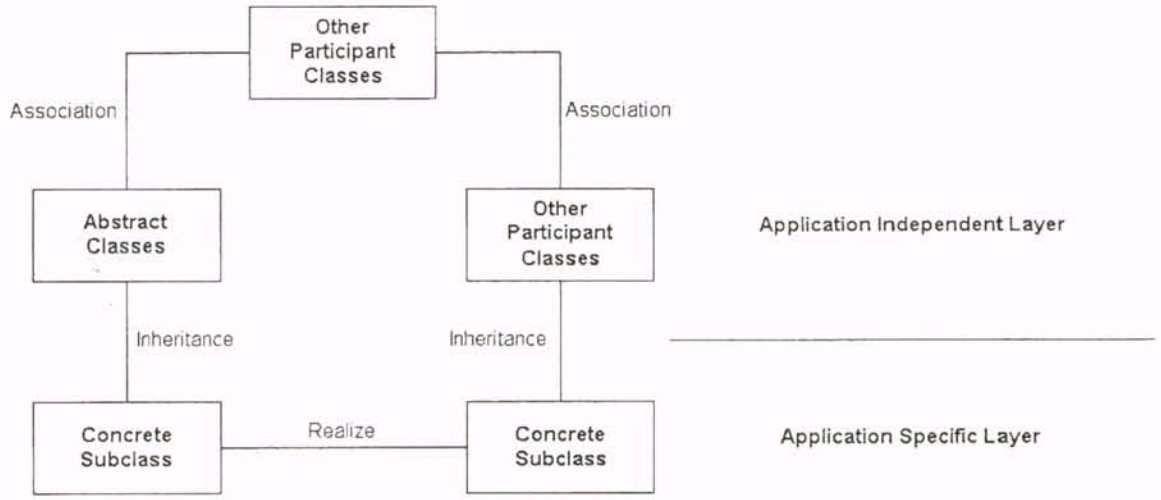


Figure 3.3: Generic two-level abstraction hierarchy for design patterns

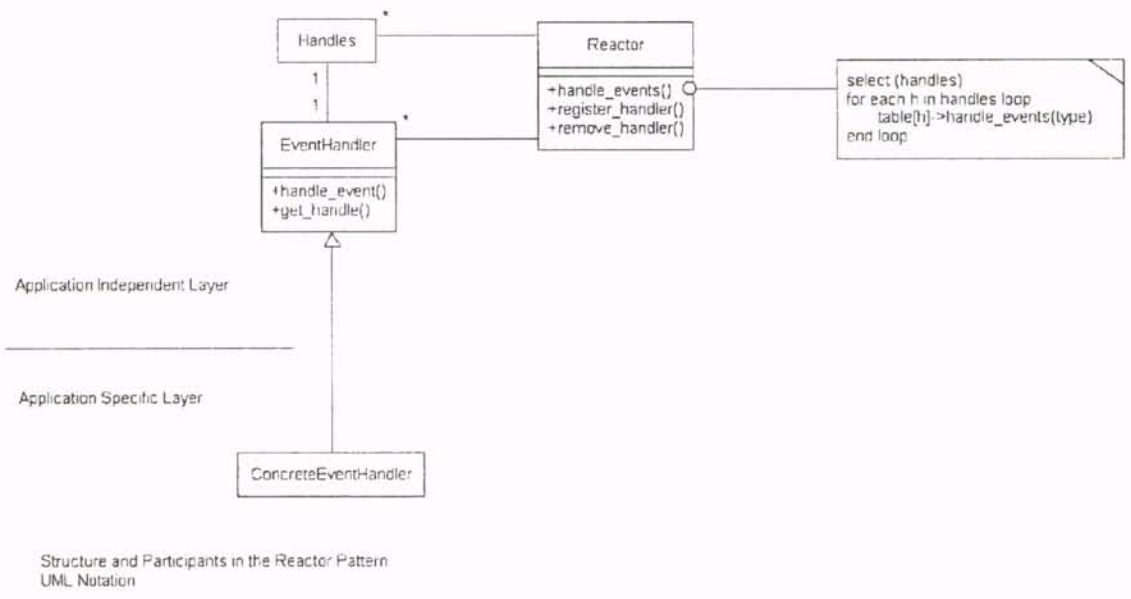
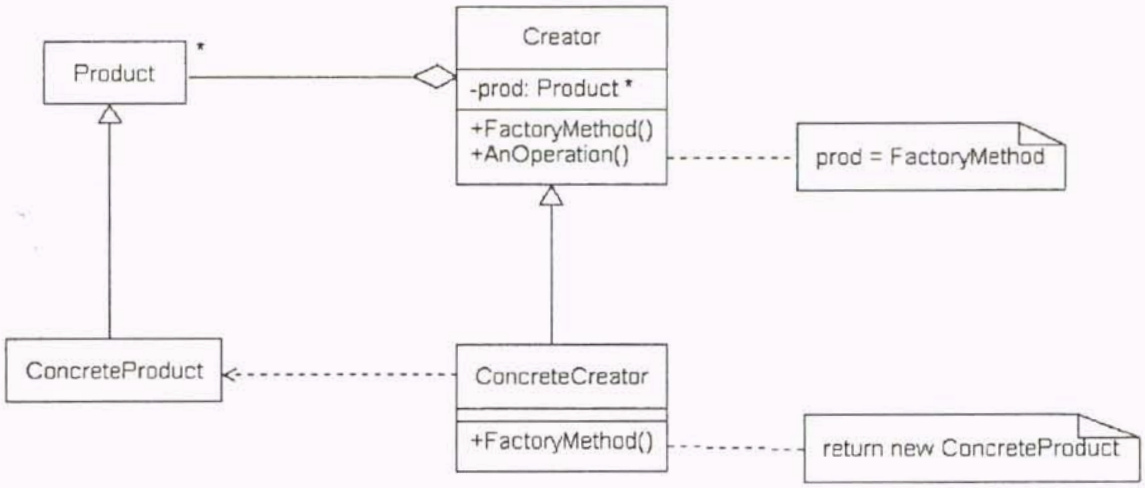


Figure 3.4a: Structure and participants of the Reactor design pattern



Structure and Participants in the Factory Method
UML Notation

Figure 3.4b: Structure and participants of the Factory Method design pattern

The application independent layer of a design pattern can be defined in such a way as to capture the majority of the behavioral functionality being implemented by the pattern. The application specific layer implements the concrete behavioral functionality. In other words, the application independent layer captures the requisite interface and relationships while the application specific layer implements the concrete methods and classes to effect the required functionality.

A natural progress follows in which the super-structure of an application can be designed within the context of the application independent layer using structural design patterns and the corresponding constructs. The concrete classes can be implemented in

from the *business model*. This architectural separation allows business strategies to drive technology decisions, and isolates the business application from evolving technology.

3.3.1 Software Architecture and Abstraction

Software architecture is akin to general systems theory in the natural sciences. On this topic, Gerald Weinberg quotes James G. Miller: "At each level there are scientists who apply systems theory in their investigations. They are systems theorists but not necessarily general systems theorists. They are general systems theorists only if they accept the more daring and controversial position that – though every living system and every level is obviously unique – there are important formal identities of large generality across levels" [Weinberg 1988]. The realm of software architecture is to find these formal generalizations, or abstractions, and apply them in solving the programming problems encountered in applications development. Again, according to Weinberg: "The more general problem is often easier to solve and, in programming, the more general design may be easier to implement, may operate faster on the machine, and can actually be better understood by the user. In addition, it will almost certainly be easier to maintain, for often we won't have to change anything at all to meet changing circumstances that fall within its range of generality" [Weinberg 1988].

After decades of building and delivery of distributed, client/server, and object systems with unpredictable results, a critical success factor has become apparent. That factor is a complete, consistent, and well-understood architecture. Grady Booth observes: "Two traits that are common to virtually all of the successful object-oriented systems we have encountered, and noticeably absent from the ones that we count as failures. These

traits are: (1) The existence of a strong architectural vision, and (2) The application of a well-managed iterative and incremental development cycle" [Booch 1996].

3.3.2 Benefits of Architectural Approach to Software Construction

A complete architecture is the first step in successfully building large-scale software systems. The benefits of a complete architecture include:

- Architecture enables embedded reuse through the implementation of frameworks that encapsulate shared functionality. It also enables service reuse through well-defined interfaces between applications that encapsulate business functions.
- Architecture allows improved time to market of applications because of parallel development opportunities. The architecture will facilitate partitioning of the problem into self-contained levels of abstraction and business services.
- Partitioning the levels of the architecture with clear specifications allows the selection of commercial off-the-shelf (COTS) products. This is because architecture modularizes the solution, providing modules with clearly defined interfaces.
- A complete and robust architecture forces the separation of the business model from the machine model. This separation will allow both models to evolve independently to support business and technology changes. Model separation will lead to an adaptive architecture system that will quickly satisfy the business objectives of the future.
- Proper technical isolation will allow the ability to separate and change the implementation. Proper levels of abstraction in the technical and software

architectures provide isolation from specific technologies, reducing the cost of changing the implementation and maximizing the flexibility of the solution.

- The architecture ensures consistency and integrity of information. Well-defined interfaces between the semantic boundaries of the architecture will allow encapsulation of functionality and ensure the consistency and integrity of information in the layers. Building the elements of the architecture as context-insensitive components that maintain their own state, will allow the components to be reused without losing consistency and integrity.
- The implementation of the architecture will allow for operational performance improvements through parallelism and asynchronous processing while reducing the developers' need to understand the technical details associated with the implementation. Partitioning the architecture into fine grained objects and using asynchronous messaging between components will position the system to take advantage of mulit-threaded operating systems, symmetric multi-processing (SMP), and massively parallel processing (MPP) hardware. The modular software will also allow performance optimization to occur at multiple levels of the architecture.

Summary

Software-based architecture is currently a vaporous silver bullet. Reuse, the Holy Grail of software engineering, can only be predictable and reliably achieved with an architectural foundation. The ability to drive reuse to higher levels of abstraction such as design patterns and artifacts, analysis artifacts, business models, frameworks, and requirements

requires a solid architectural underpinning. Software development processes and lifecycles are frustrating and ultimately ineffective exercises in managing the unmanageable, if implemented without an architecture. Metrics, software quality assurance, software process improvement, and even quantitative and qualitative estimation and project management are ineffective abstractions without an architectural framework within which one can analyze, compare, and reason about them. It is apparent that unlocking the mystery of software architecture is essential to gaining insight into the issues of raising the abstraction levels for a software developer to be most productive.

The need for and the goal of an architecture framework is to manage complexity, minimize the impact of change, incorporate and leverage existing components and, *maintain an overall perspective of the system*. If accomplished, the framework accelerates the systems development process, reduces system costs, and improves systems quality.

It is our belief that *software architecture, abstraction, and software reuse are orthogonal views of the same concept*. However, software architecture provides a natural approach to gain insight into uncovering abstractions in the software system. In addition, it has the added benefit of being able to be mapped to a methodology and is therefore repeatable. In the next chapter, we present a detailed outline of our contribution with major focus on the adaptive enterprise application integration framework.

Chapter 4

Outline of Our Work

This chapter presents a detailed outline of our contribution. Our work is primarily focused on the development of an Adaptive Application Integration Architecture Framework. Software reuse and software integration are very closely related concepts since integration is the combination of two or more existing components. Without good integration tools and techniques, reuse is difficult and will probably not happen to any significant degree. In the development of the adaptive architecture framework, the primary enabling concept is object-oriented design support by the unified modeling language (UML). The concepts of software architecture, design patterns, and abstract data views are used in a structured and disciplined manner in establishing a generic adaptive integration framework. To illustrate our approach, the proposed framework is applied to solve the Enterprise Application Integration (EAI) problem in the telecommunications operations support system (OSS) marketplace (Chapter 7).

The need for and the goal of an architecture framework is to manage complexity, minimize the impact of change, incorporate and leverage existing components, and *maintain an overall perspective of the system.* Design patterns allow us to solve various pieces of the overall problem. For example, we have developed a number of EAI design patterns that are used to integrate legacy third party applications into the architecture

framework. These design patterns allow us to develop a very definite and repeatable process for integrating legacy as well as newly developed applications into a unified framework. The abstract data view approach with its compositional capability is used to aggregate and build up the overall solution by combining smaller macro components.

4.1 What is the Enterprise Application Integration Problem?

The enterprise application integration (EAI) problem is the inability of an enterprise to leverage its enterprise domain silo software applications from a unified platform and to use these software systems to gain advantages in an intensely competitive marketplace. Businesses in general are in a process of continuous re-definition. Any reasonable solution to addressing large-scale software development must facilitate the notion of continuous business process re-definition or re-engineering. That is, the software system must be adaptive allowing the enterprise to adjust its business models to address changing market conditions. The problems associated with an enterprise inability to perform application integration, adaptation, and business function interoperability at the enterprise level are further exacerbated in the telecommunications industry where introduction of new frame breaking technologies and services are the norm. The telecommunications industry is therefore in a heightened state of awareness with respect to the problem of enterprise application integration.

In addressing the EAI problem in a generic manner, our architecture centric approach presents solutions for the following broad problematic areas:

1. Facilitate the integration and interoperability of stove pipe legacy applications
2. Cater for a clear separation between the business models and machine models
3. Facilitate the development of adaptive business process re-engineering

4. Facilitate the use of the Internet as a business platform across the entire enterprise

The problem areas indicated by (1), (2), and (3) have been around for a long time and notoriously regarded as almost intractable problems in the sphere of the business community. Solving these problems will present a whole new way of looking at how we develop business software systems of the future.

4.2 Solution to the Enterprise Application Integration Problem

Our approach to the enterprise application integration (EAI) problem is to develop an overarching view of the entire enterprise integration problem using software architecture. To this end, we developed the n-tier orthogonal application integration architecture model that is based on our proposed n-tier orthogonal architecture model. The n-tier orthogonal architecture model examines adaptive business centric software development from a single application context. The n-tier orthogonal application integration architecture model looks at the notion of adaptive business centric software systems from an enterprise perspective, which results in a close examination of the problem of integration at the enterprise level. The principle of architectural layering is applied in the development of the adaptive application architecture model. The subsequent paragraphs outline this approach.

The object-oriented approach provides useful levels of abstraction for addressing the complexity of modern business problems. Problems are decomposed recursively. At each level of decomposition, the prevalent vocabulary and concepts are used to describe that part of the domain. At the highest level, concepts are centered on overall enterprise procedures and workflows. Following these are the business objects supporting the

enterprise-level procedures and workflows. Finally, key concepts that are not particularly business or industry-specific are described.

The application architecture deals specifically with the business problems and functionality. It describes the visible portion of the application – the presentation. The application architecture is also distinct from the technical implementation details. With object orientation, technology-based details can be suppressed, allowing more focus on the problem to solve and on the business process to engineer. This results in the description of at least three layers of system architecture – the presentation, the application layers, and the technology-based details.

Each of the layers can be further refined to contain sub-layers or components. Each layer and sub-layer is designed so that it arranges and connects layers and components to produce the desired functionality. The sub-layers within the application map to the levels of the problem domain described above, and the components of the application should represent processes and concepts that exist in the problem domain. This structure provides the following benefits:

- Integrity is enhanced because components share a common conceptual structure.
- The system is extensible because components and connections interact through well-defined interfaces. Also, the implementation details become hidden from the rest of the system, allowing components to change to take advantage of new technology or to address new business needs without affecting other system parts.

4.3 Generic Adaptive Application Integration Architecture Model

The object management group's (OMG) object management architecture (OMA) [OMG 1997] goes a long way in addressing the core requirements of a distributed object-oriented application framework. OMA is the canonical n-tier architecture model. However, it does not address the concerns of integrating enterprise silo applications into a unified framework that allows the enterprise to leverage its data across the different domain applications. It has been estimated that in excess of 95% of all enterprise data resides in legacy applications. These applications were not developed to facilitate interoperability and play nicely in a unified distributed framework. Hence, there is tremendous need to provide a framework for integrating legacy applications in large enterprise.

The generic adaptive application integration architecture model describes a highly modular approach to integrating enterprise domain silo applications using standard client/server relationships. While this model is built on traditional concepts of clients and servers, the distinction between client and server is of a logical nature, resulting in peer-to-peer relationships among components. The components are small and functionally specialized so they can be easily reused.

Figure 4.1 shows a schematic representation of the generic adaptive enterprise application integration (EAI) architecture model. This model contributed significantly to the development of the adaptive EAI framework pattern presented in Chapter 6. The adaptive EAI architecture model can be defined as a layered model, with each layer providing a specific function in the overall scope of the resultant software system. This

architecture model enables the overall software system to be partitioned into small-grained services. The major layers of this architecture model are as follows:

1. Domain Applications
2. Domain Application Adapters
3. Asynchronous Distributed Object Framework and Infrastructure Services
4. Mediation Services, equivalent to the Application Architecture in the n-tier Architecture model and consisting of the following layers:
    i. Package Mediation
    ii. Intrinsic Objects
    iii. Domain Objects
    iv. Business Objects and Business Object Managers
    v. Business Processes
5. Presentation Services
6. Thin Client Application

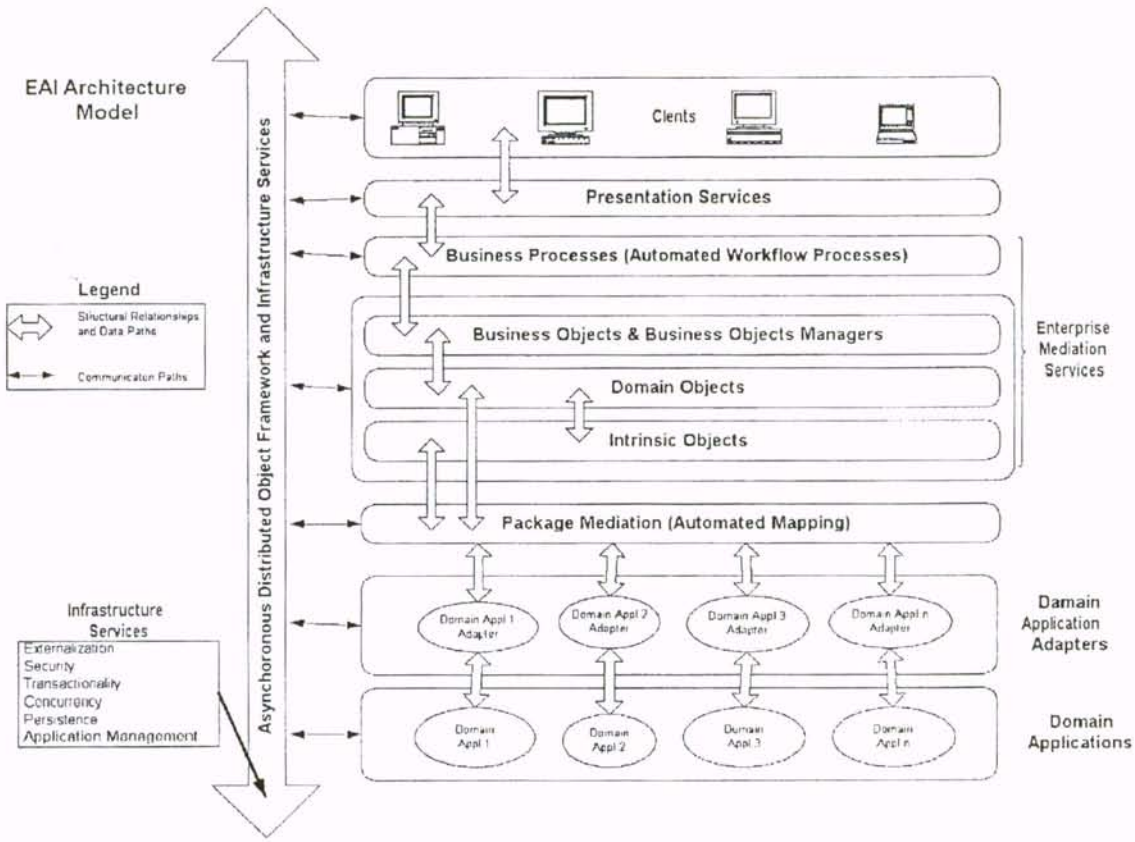


Figure 4.1: Generic Adaptive EAI Architecture Model

The benefits of using the EAI architecture model, which enables application partitioning, are:

- Adaptable business process, continuous business process re-engineering
- Framework facilitating plug-and-play capability for best of breed domain applications
- Framework for integrating the domain silo applications into a unified view
- Separation of business rules from presentation services and data access mechanisms

- Increased application performance
- Increased application scalability
- Isolate security and critical business processes
- Reuse of not just software but entire applications (service reuse)
- Macro software component reuse
- Macro software pattern reuse

The following sections describe the layers of the adaptive EAI architecture model.

4.3.1 Domain Applications

These are the enterprise silo applications that form the core of the information technology (IT) and enterprise business automation that companies rely upon for management and successful execution of day-to-day operations to satisfy their customers' needs. Legacy applications are a special class of domain applications specifically when it comes to integration and business function interoperability. These applications were not developed with the intention to facilitate integration and business function interoperation. There are a host of interesting problems associated with the integration of legacy applications. These include problems relating to differences in technology, design methodology, implementation strategies, etc.

4.3.2 Domain Application Adapters

These serve to externalize the information model, application services, and data models of the respective enterprise silo applications. By so doing the domain application adapter principle facilitates wholesale reuse of specific classes of domain applications. Domain

applications can be plugged and played without impacting the enterprise information model. Likewise the enterprise information model can be modified without impacting the domain silo applications.

4.3.3 Asynchronous Distributed Object Framework and Infrastructure Services

In the adaptive EAI architecture model, application services are available on the network and are accessed through an object-oriented programming interface. The application services are distributed on machines throughout the network. The Object Request Broker (ORB) technology [DEC1991] is used to facilitate the transparent cooperation of remote objects. The ORB provides a very rich set of distributed middleware services. The ORB lets objects discover each other at run time and invoke each other's services whether remotely or locally located. Figure 4.2 shows a sample distributed object framework.

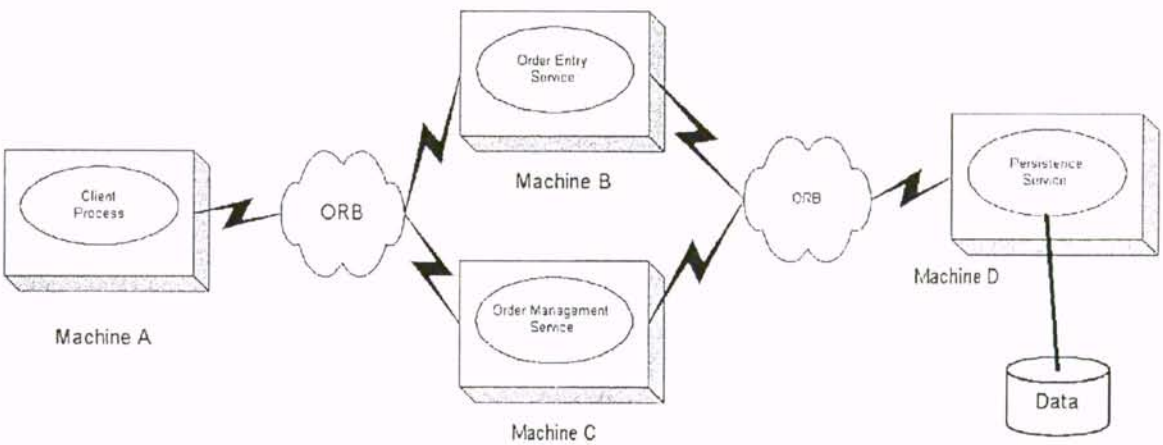


Figure 4.2: Distributed Object Framework

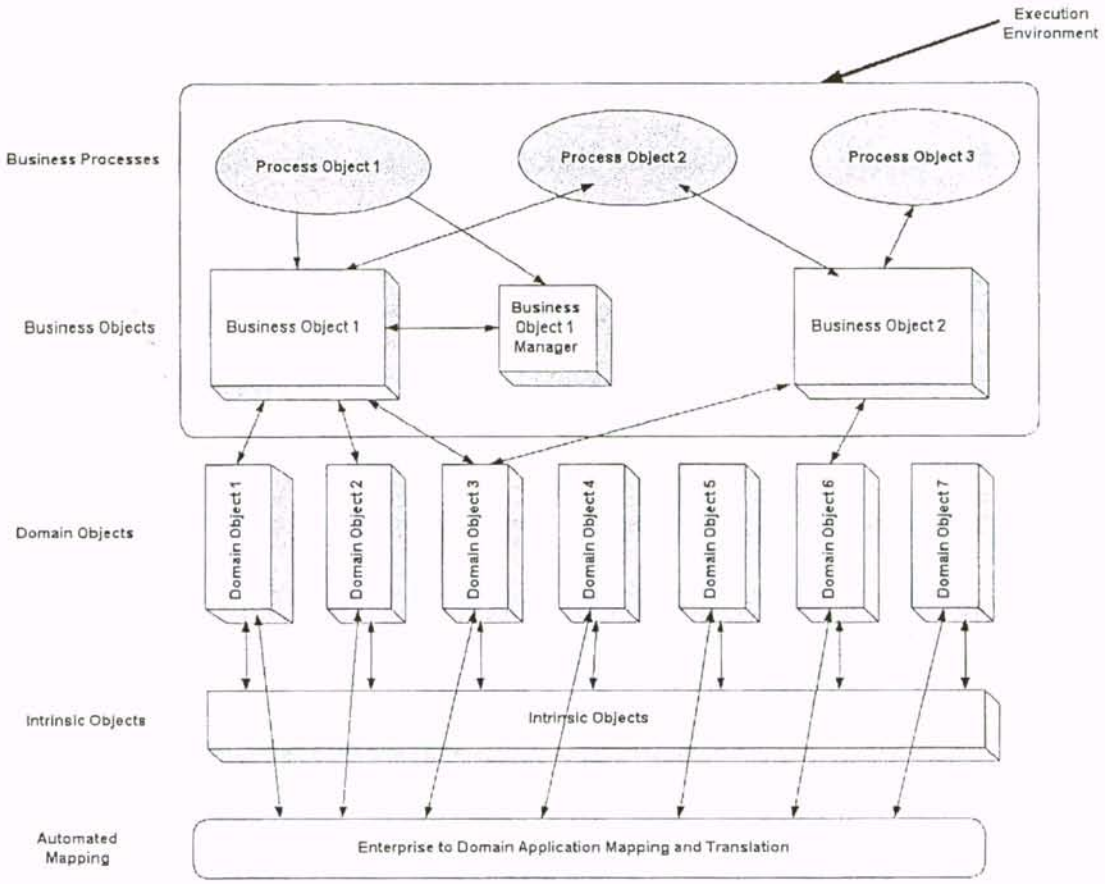


Figure 4.3: The Sub-Layers in the Mediation Services Layer

4.3.4 Mediation Services

The mediation services layer is a distinct layer in the adaptive EAI architecture. This layer has a number of sub-layers that combine to realize an orthogonal integration framework for integrating legacy and newly developed software applications. Figure 4.3 provides a description of the mediation services sub-layers. The sub-layers form the framework for building a unified enterprise information model, application services, and data models.

Process Objects

Process objects represent business processes, sequence of events, business rules knowledge, and concepts that span specific business objects. These objects manage runtime coordination and cross-validation of business objects. For example, the order entry process is not tied to a specific business object. Instead, it involves customers, the location of customers, and the details of what has been ordered. A detailed pattern of interaction among business objects constitutes the correct way to describe the placement of an order and the appropriate information on an order. To put this knowledge in a business object would violate the object-oriented principle of encapsulation. Instead, this knowledge should be encoded in an order entry process object. Behavioral rules should be contained within process objects.

Process objects are fundamental to the notion of adaptive business process re-engineering. The process objects decouple the specification of the business functions from the business objects that are used in the implementation of the business functions.

Business Objects

Business objects represent business concepts. This includes items such as information about an organization, customers, and orders. A business object marries the basic object with specific behavior, information, and structural business rules. A business object should not bind to other business objects – this is the responsibility of the process objects. Any cross-business object functionality should be pushed up to a process object.

Domain Objects

Domain objects represent key business and industry concepts and are independent of a particular application. They may include basic information regarding a customer or order, but they may also include items such as products and contacts.

Intrinsic Objects

Intrinsic objects are foundation objects. These objects are not tied to an industry or domain. They include items such as addresses, names, dates, and times. Intrinsic objects are primarily used in the construction/specification of domain objects. The primary relationship between intrinsic and domain objects is of a structural nature. Intrinsic objects are normally aggregated into domain objects.

4.3.5 Automated Mapping

The automated mapping layer is responsible for performing enterprise to domain application mapping and translation. The mapping is needed because domain silo applications are developed by different corporations and the fact that there is no universal domain application standards to specify in an unambiguous manner things such as component interfaces, information models, data models, collaboration sequences, and other issues that are critical to the development of a software system. Invariably we have the situation in which individual software vendors specify their own information models, data models, and interfaces. This is further exacerbated by each enterprise developing its own enterprise information and data models, etc. All this coupled with the need to have

interoperation between the various domain applications mandates the need to have some form of mapping and translation mechanism.

In our adaptive application integration architecture model, mapping and translation is restricted to occur between the enterprise business and domain objects and the domain applications and between the intrinsic objects and the domain applications. This mapping mechanism is key to decoupling the enterprise view from the domain application view and is the core technological framework in giving this architecture its best-of-breed plug-and-play capability. This module is transparently invoked whenever there is any data transfer between the enterprise view and any of the domain silo applications.

4.3.6 Presentation Services

Presentation services are server components that give a client application access to the enterprise services from the mediation services layer. This layer is inherently distributed in nature and the components are deployable across different machines. This layer can be implemented using the model pattern or the typical peer-to-peer client server model. Models are usually stateless and therefore a model may interact simultaneously with many clients.

4.3.7 Thin Client Applications

This layer's primary responsibility is to give a visual presentation of the information and/or data models projected from the presentation services. This implementation approach creates a clear separation between the enterprise services provided by the

mediation layer and presentation or view provided by the client layer. The client should not be aware of the semantics of the information or data models of the enterprise mediation layer. Implementation technologies such as XML over RMI or IIOP can be used to further isolate the client application from the presentation and mediation layers. The clients are called thin clients because they implement very little or no business function capabilities.

4.4 Frameworks and Patterns of Interaction

The components of a distributed system interact to fulfill the overall requirements. These interactions are termed collaborations and represent requests from one component to another. By collaborating, seemingly disparate sub-systems and components can be connected to perform the system responsibilities. Components can collaborate between layers or across the same layer depending on the type of function being performed. Rules of visibility must be established for each component to maintain consistency for the types of collaboration allowed across layers and between layers. Typical collaboration patterns include the coordination pattern and the configuration pattern. These Patterns are described below.

4.4.1 Coordination Pattern

The coordination pattern of interaction represents tasks such as propose and confirm or validation of data items. A client might enter data into a field on the screen. Validation would be performed on the data and errors would be sent back to the client. Figure 4.4 shows one example.

To accomplish data validation, values entered by the client are sent from the order user interface to the order server application service. A policy component may be responsible for enforcing the business rules associated with the data item. The order server would request the order policy server to validate the data item. The status of the validation is sent back through the reverse path.

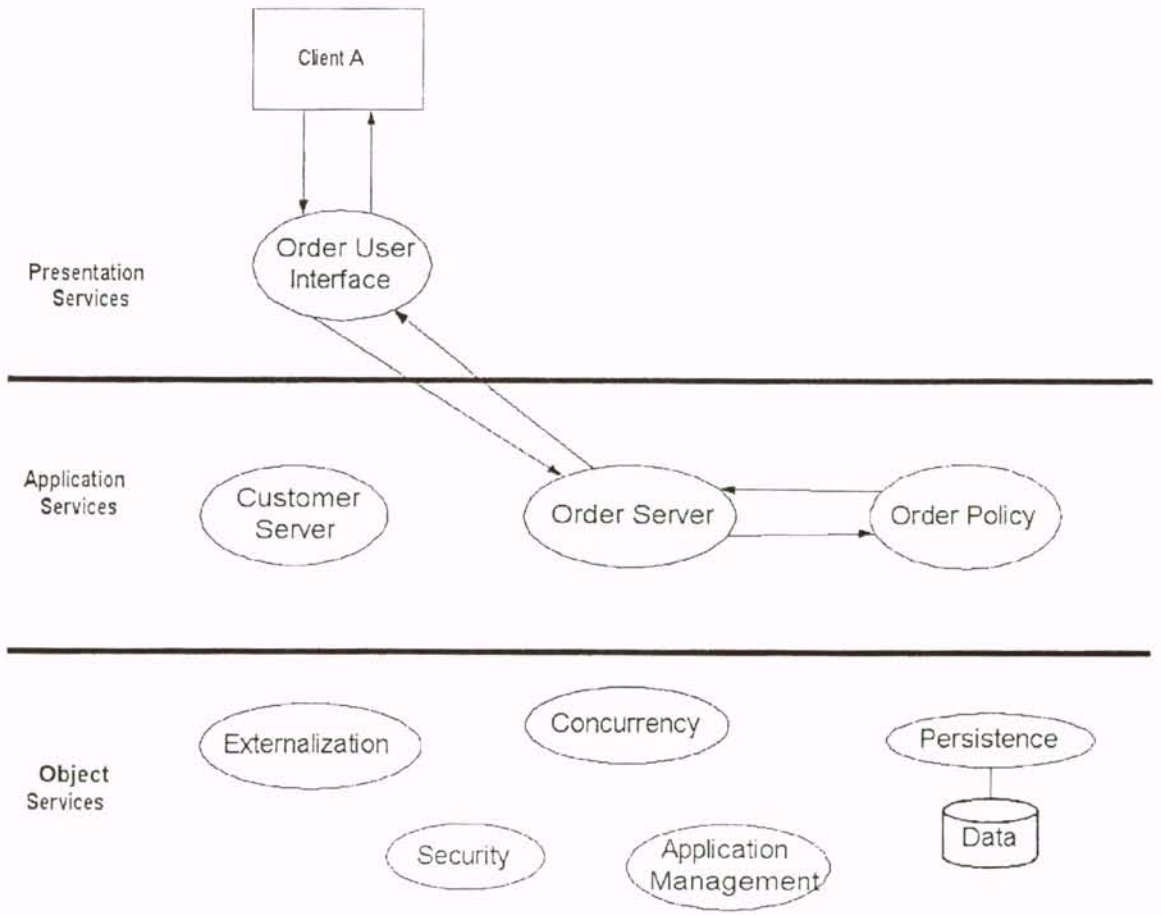


Figure 4.4: Example Coordination Interaction

4.4.2 Configuration Pattern

The configuration pattern is another common type of server collaboration. Figure 4.5 shows an example of a configuration interaction. The client can alter application behavior by setting configuration parameters or properties. This pattern of interaction traverses all the layers of the architecture. An application server would receive the configuration request from a presentation server and validate it with a policy server. The policy server would enforce rules such as range checking, user privileges, and conflicts with other settings. Once validated, all of the data storage servers associated with this change would be updated. This pattern gives the user control over presentation layer components. In addition, system behavior such as workflow steps and error message routing should be controllable by the user. In the application of this framework some tradeoffs must be considered along the dimensions of Time to Code (time to market), Execution speed, and Level of Effort to adapt.

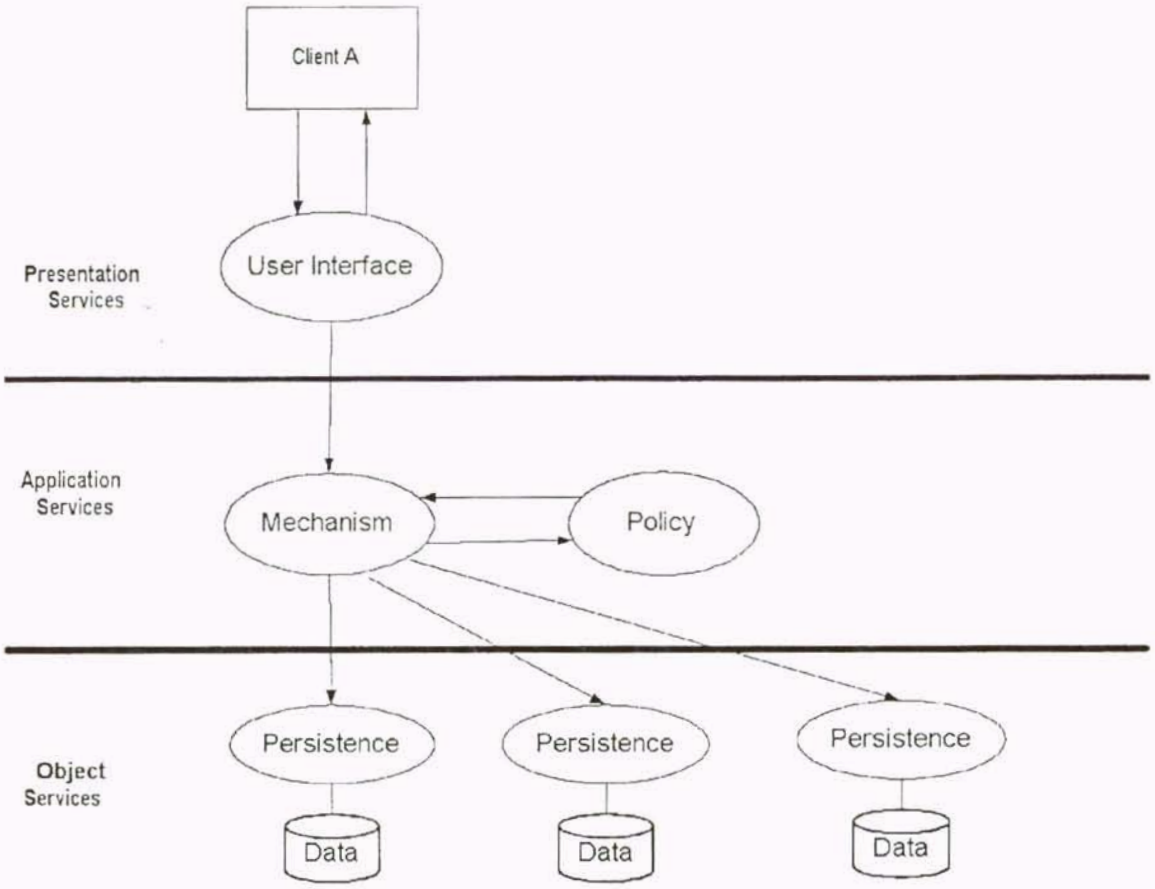


Figure 4.5: Example Configuration Interaction

4.4.3 Model Pattern

The model pattern is an object interaction framework that implements the collaborations between the presentation and application layers. Figure 4.6 shows a generic example of a model pattern.

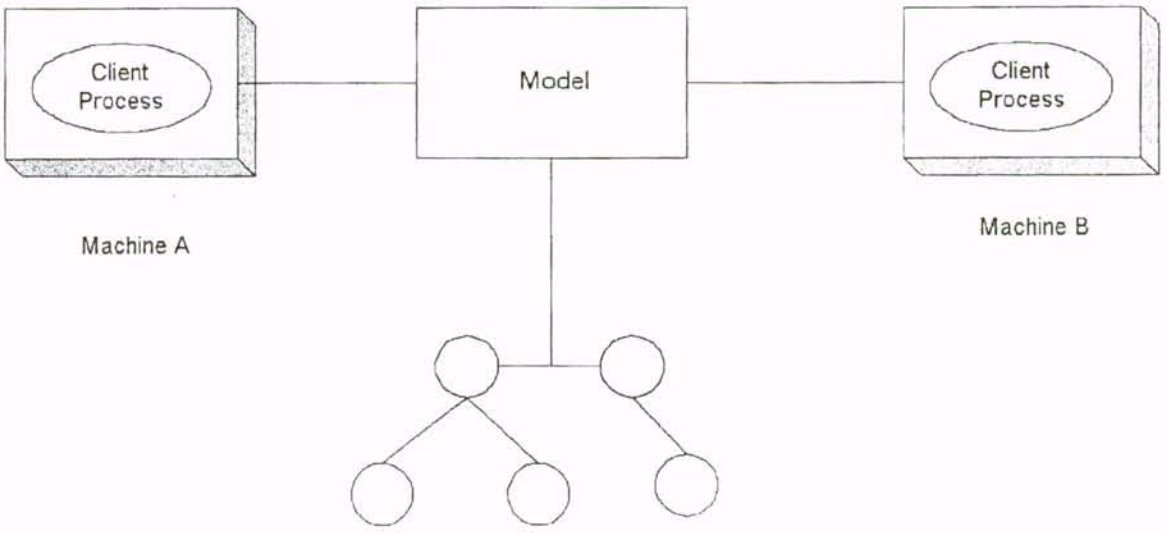


Figure 4.6: Generic Model Pattern

In the model pattern, clients access the business objects through a common *model* object. This object hides the details of the business objects by receiving requests for business activity or data, and coordinating the actions that must occur to meet those requests. A model object is normally focused around one specific domain or activity, such as order taking. Multiple models may exist throughout the system, each accomplishing a different task. Models should be stateless, letting multiple clients access them simultaneously.

In the following three chapters (Chapters 5-7) we will describe our proposed approach in detail along with a complete example.

In Chapter 8 we will present the model based software development approach. This is an approach to raise the abstraction level at which application developers work and to automate the process of translation from an application model to its corresponding

distributable runtime component. The basic thesis here is that we can effectively transform the effort deployment in the software development process in which about 80% of the development effort goes into the development of infrastructure services and 20% into the development of application logic [Eeles 1998].

In Chapter 9 we will present a mathematical formalism for the specification of design patterns. The formalism is the basis of a general-purpose approach for the specification of software systems and components. This formalism is based on many-sorted algebra. The approach thus provides a solid theoretical foundation for describing and reasoning about software artifacts.

Chapter 5

Adaptive Orthogonal N-Tier Integration Architecture

Traditional legacy applications have been developed along a synchronous push-oriented transaction model. In this architecture, the client initiates a transaction on the server by posting a request; the client blocks and waits for the server to service the request, and the server eventually delivers a response to the client. At this point, the client receives the response and continues with its tasks, possibly posting another request to the server. Because of the inherent lack of asynchronous capability in this type of application architecture, integrating it into a bus framework such as CORBA is fairly difficult. Most attempts at integration result in a peer-to-peer integration model over the communication bus. In addition, these kinds of applications suffer from a lack of clear demarcation of functionality between the application sub-layers and therefore embedding aspects of the business processes into the exposed application program interface (API).

5.1 The Need for Application Portfolio Integration

The intensely competitive nature of today's business market place mandates that businesses must have the ability to perform very flexible business process reengineering. Modern businesses, in general, are continually redefining their business models in an effort to differentiate themselves from their competitors and to ward off competition.

In this market environment, high availability and customer care management is essential to establishing market acceptability and high customer retention. These are essential ingredients to successfully operate a business in the Internet driven economy. These requirements coupled with the necessity of flexible business process reengineering have driven us to reevaluate the approaches that have been taken to address the large-scale (or enterprise) application integration (EAI) problem. The above requirements mandate a business process driven integration framework that allows individual business processes to be represented, monitored, and integrated with existing systems and users across the enterprise. It also requires the ability to dynamically reconfigure active business processes (software fault tolerant and hot swappable capabilities), allowing users to continuously adapt to rapidly changing market conditions. This process-driven infrastructure provides businesses the ability to adjust and alter their operational systems to changing market conditions without any down time.

Application portfolio integration is mandatory in order to support larger enterprises' organizational goals. These goals include operational efficiency via process flow-through and customer intimacy to enhance customer satisfaction. Examples of this include knowing what a customer has ordered across multiple products, what problems he or she has experienced, and his or her billing and payment history. Addressing this integration challenge requires a comprehensive application portfolio assembly approach that can exchange information among multiple application architectures each with different data and process models and with different data exchange models.

The challenge is to create a means of integration at the business process level. An information broker can create generalized event and object models to normalize the flow

of information between domain silo applications. We can create adapters to the various domain applications. This serves the purpose of migrating the architecture from a multi-point, spaghetti architecture into a much more manageable hub-and-spoke arrangement. However, the adapter approach is just the starting point because it does not allow for a flexible business architecture. To create an architecture that enables best-of-breed third party application selection, while not sacrificing integration and data sharing, requires another layer of "business aware" software that runs above the information broker.

With such a layer in place the business processes can change without affecting the underlying applications. And conversely, IT should be able to change applications without affecting the business processes.

5.2 Traditional Approaches to Enterprise Application Integration

Typical approaches to address the need for enterprise application integration involve building point-to-point interfaces between applications [OMG 1997; Linthicum 1999; Mowbray 1998]. This is an order n-squared problem and is therefore very expensive. In addition, it does nothing to address the adaptive requirements of most modern businesses. The business models and corresponding business processes are constantly undergoing changes to address the competitive nature of today's marketplaces.

The next prevalent approach to addressing the EAI problem is to use a hub-and-spoke architecture in which an application is chosen as the master and all the other applications are logically integrated through this master application [Mowbray 1998]. Normally, a bus framework is used and thus the n-square interface problem is eliminated. This type of integration architecture is very application specific. It does not facilitate

plug-and-play of best-of-breed or commercial off-the-shelf (COTS) software products. This kind of application integration continues to suffer from shortcomings resulting from the push transaction model. The new enterprise business processes are tightly coupled within the APIs of the master application and are therefore not adaptable. Figure 5.1 depicts such an application integration architecture model.

The mediation layer makes an attempt to capture the definition of business concepts independent of the different applications being integrated. However, the business rules and business processes are defined fully within the context of the master application. The master application also serves as the entry point of the system and therefore drives the overall system interaction.

The close coupling of the business processes with the inherent limitations of the master application architectural model severely limits the ability of such a system to adapt to changing market conditions. The rapid delivery of new products and services is a mandatory requirement in today's business environment for an enterprise to remain a viable entity. Hence, applications must have, as a core requirement, the ability to be rapidly adaptive to changing market conditions. A business analysis should be able to create an enhanced version of a product or service offering and deploy it in the runtime environment in hours or days, but certainly not weeks or months. Time to market responsiveness is absolutely critical for the survival of business in industries such as telecommunications that experience intense competition to acquire and retain customers through differentiated services.

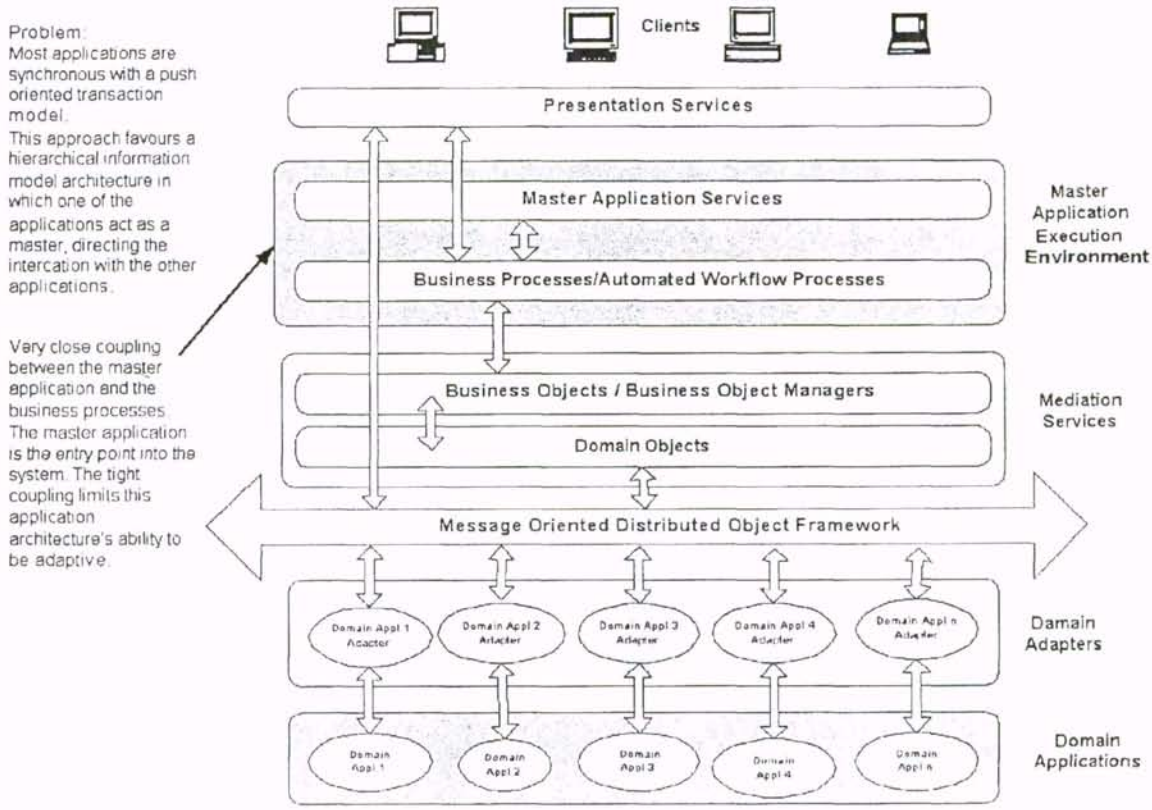


Figure 5.1: Traditional N-Tier Application Integration Architecture Model

The domain adapters are used to expose the enterprise silo application data model and application services. They may also be used to perform data mapping to and from the application domain. The domain applications often have application specific data and information models, making it necessary to have some form of data mapping. This functionality of the domain adapters further limits their reusability because they are closely tied to object specification in the master application. The master application, which is the entry point for the enterprise, hosts the enterprise object models. The

Enterprise object models vary from organization to organization and hence the adapter must be modified to reflect this in the functionality of the data mapping.

To address the shortcomings of the above-mentioned approach, we propose a new adaptive orthogonal integration architecture framework.

5.3 N-Tier Orthogonal Application Integration Architecture

Modern business is by definition evolutionary. An enterprise must continuously redefine itself to remain competitive and thus a viable business entity. To accomplish this fundamental business requirement, the business processes representing enterprises' business models must be evolutionary by nature. That is, the business processes must be adaptable and thus give the enterprise the ability to be responsive to changing market conditions and competitive market pressures. Inherent in the adaptability requirement is the fact that business processes must be elevated to the status of first class entities, complete with their own execution environments. From a distributed computing perspective, business processes can be viewed as deployable distributed components. These components should be developed as *complete software agents* with respect to their ability to interact, acquire, and use the services provided by the run-time distributed object framework and infrastructure services.

The Generic Adaptive Integration Application Architecture Model of Figure 4.1 (presented in Section 4.3) forms the basis of the Adaptive N-Tier Orthogonal Application Integration Architecture that we propose. The concept of the Adaptive N-Tier Orthogonal Integration Application framework is explored in Chapter 6. There we take a distinct

implementation perspective, identifying the major component and technologies required for the approach.

From a business perspective, a business process represents a business function. That is, a functional *use-case* of the application that is used to represent a business's functional requirement. Thus, in the finest granularity, there is a one-to-one correspondence between business processes and business functions. A change in business model is manifested as a change in business function. This should be ultimately reflected as a modification or enhancement to the business processes implementing the business functions. In the runtime environment, this could be achieved by deploying a new business process component that implements the new requirements.

The orthogonality of this architectural approach is accomplished by removal of the master application concept. Each application has the same level of importance with respect to peer relationship. This approach effectively destroys the hierarchical master-slave relationship between the master application and the other subordinate applications. In addition, this approach also addresses a more sinister and difficult problem. The hierarchical master-slave relationship imposes a hierarchical information model in the integration architecture. This is reminiscent of the problems resulting from strict functional decomposition that typifies the traditional software construction process. Inherent to software constructed using functional decomposition is the fact that the higher layers require knowledge of the lower layers. Knowledge percolates or flows upward in this kind of architecture and makes it inflexible and therefore resistant to change. Within the context of this kind of architecture, business functions or business models are also knowledge and have to be encoded. Thus, it is fair to state that, the traditional software

architecture and implementation approaches result in business models being encoded in the APIs of these kinds of applications.

The fact that the business models and resulting business processes are embedded in the APIs in traditional software applications is reinforced by the fact that these applications are extremely resistant to change. Changing them to address new business directives means reprogramming the application. To address this problem, most large enterprises have substantial Information Technology (IT) resources dedicated to address this problem. Again, it is fair to assume, for example, that banks are in the banking business and not information technology. If the software they use allows them to adapt to changing market conditions and facilitate growth, then they would not have to invest the current level of resources into their in-house IT departments. Thus, in order to tackle the problem of developing flexible business process objects to facilitate adaptable software systems, we must effectively develop a new architecture, one that destroys the notion of a hierarchical information model to handle interfacing between the layers of the application architecture. Hence, the adaptive orthogonal integration architecture.

Domain application orthogonality results in a simplification that can be characterized by the application services being viewed as extension of the infrastructure services. This is consistent with the work of the OMG in their effort to develop domain specific standard services. The logical extension is that these services become evolvable distributed components that can be deployed directly into the execution run-time environments.

The enterprise application architecture can now be developed using an object-oriented n-tier architecture model. The application and infrastructure services are the

foundations on which the intrinsic and domain objects are built. The process objects implement the enterprise business processes. Process objects are deployable distributed components. Process objects implement logic to facilitate collaboration between two or more business objects in addressing a business function. Integration between the various domain silo applications is manifested as collaborations between the business objects within the context of a process object.

This is a form of *dynamic integration*. This kind of integration is expressed as the collaboration logic between business objects within the context of a process object. The business objects represent domain application services. These services are implemented within the domain silo applications.

5.4 Implementation and Protocol of the Enterprise Mediation Layers

As described in Chapter 4, the mediation services layer is a distinct layer of the adaptive enterprise application integration (EAI) architecture model and is composed of several conceptual layers. This section describes the implementation of the mediation's sub-layers and the protocol between those layers. Figure 5.2 provides a graphical description of the implementation layers and protocols between the layers.

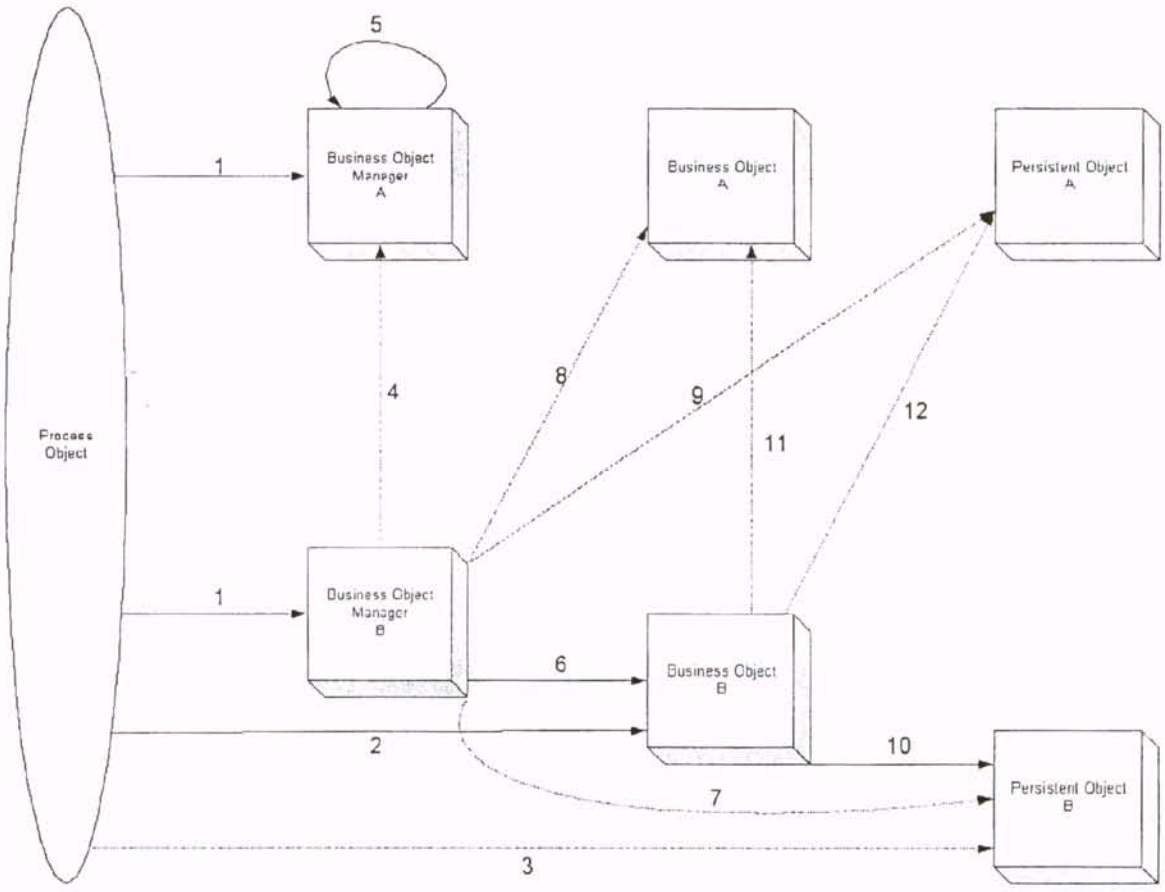


Figure 5.2: Mediation Services Layer Implementation and Protocol

Figure 5.2 shows the three major components in the mediation layer. The first is the process object. As described in Section 4.3.4, the process object can span a particular business object or concept. The second and third components are application components. These two components are represented by A and B in Figure 5.2. Either component could represent order, customer, affiliate, or any other component of an integrated telecommunication management application; however, the particular components should be viewed as patterns of interaction between components rather than details of a particular component. Each application component comprises a Business

Object Manager, a Business Object, and a Persistent Object. Whereas Section 4.3 describes the basic capabilities and function of each layer of the application, Section 5.2 describes some aspects of implementation relating to how the layers are constructed and how they interact.

5.4.1 Component Construction

This section describes the structural aspects of the mediation layer components.

Process Objects

A process object contains functionality that spans business object components. Process objects contain no state. Instead, they contain only functions that require a particular sequencing or cross-reference between other objects. These functions contain the procedural/behavioral knowledge of the application. Process objects may therefore be replicated for performance and scalability as needed. Process objects should be used to:

- Coalesce lists and queries that cross class boundaries
- Provide convenience functions for a user interface client
- Gather a subset of Business Objects to invoke the same function on each
- Provide validation of state at the model level (cross-object)
- Force a sequence of activities.

Business Object Managers

The Business object managers provide lifecycle and location services for a particular class of business object. Unlike process objects, the business objects managers

contain state. Each business object manager contains a transient list of business objects for the class it represents. The lists may be separated for performance reasons within a particular business object manager to contain, for instance, a list of those business objects that are active and those that are inactive, determined by the state of the business object itself. Other separation schemes may also be possible.

A business object manager can be designed to handle some larger-grained read-only queries. When such requests are made, the business object manager initiates a transaction and makes calls directly to private methods of the business object (the business object manager of a particular class can be implemented as a C++ friend of the same class of business object or in the same Java package). This implementation approach could be done to increase performance by handling the query in one transaction.

Because this layer of the application contains state, replication is not simple. To replicate this layer, an event mechanism must be implemented so that multiple business objects managers will be aware of the changes to the transient lists of business objects that other managers are making.

Business Objects

The business objects contain no state. They serve as gatekeeper to the persistent objects. A business object contains the transaction logic to access the persistent layer of the application for write transactions. Although this layer contains no state, the business objects may not be replicated because there is a distinct tie between an instance of a business object and its corresponding persistent object. In other words, the business

Some interactions are considered illegal within the context of this architecture and are not allowed. They are represented in Figure 5.2 by dash directed lines. These interactions primarily violate encapsulation and include, but are not limited, to the following: Process Object to Persistent Object of any class, Business Object Manager to Business Object Manager of different class, Business Object Manager to Business Object of different class, Business Object Manager to Persistent Object of different class, and Business Object Manager to Persistent Object of same class.

Process Object to Business Object Manager of Any Class

Process object to business object manager of any class invocations (Line 1 in Figure 5.2) may be performed to get a subset of the list of business objects contained in the business object manager. This is not completed in a transaction. To the extent possible, the business object manager provides convenience functions to narrow the list of business objects returned in a query, thus reducing the internal knowledge of a business object that a process object must contain.

When the process object receives a transient list of business objects, a question arises as to the integrity of the process object's function that may be addressed by the process object subscribing temporarily to event notification of update to the transient list of business objects. This will call for a case-by-case analysis of the process object function to determine how the event will be handled, and may include updating the transient list, breaking a transaction lock (if one was initiated) and forcing the process object to retry, or ignoring the event.

Process Object to Business Object of Any Class

Process object to business object of any class invocations (Line 2 in Figure 5.2) are performed when the process object has already invoked a function on the business object manager to get a subset of the particular business objects (narrowing the list of business objects through a query). The process object then either returns the list of business objects to the client that invoked a function on the process object or invoked a specific function on each of the returned business objects.

Business Object Manager to Business Object Manager of Same Class

Business object manager to business object manager of same class invocations (Line 5 in Figure 5.2) are required because of the transient list that exist in the business object manager. This is not a business object manager invocation of a function on itself, but rather is the invocation of a function on other instances of the same business object manager class. This could be a simple event notification mechanism to update the transient list.

Business Object Manager to Business Object of Same Class

Business object manager to business object of same class (Line 6 in Figure 5.2) supports create, read, update, and delete functions. Write transactions are done on a transaction-per-object basis to ensure persistent object integrity. The read transactions are done on a transaction-per-business object manager basis, so that one transaction may be used to collect the entire list of business objects.

Business Object to Persistent Object of Same Class

Business object to persistent object of same class (Line 10 in Figure 5.2) are performed for create, read, update, and delete functions. The business object public functions always access the persistent object through a transaction. The business object contains private functions that do not have a transaction that make the actual call to the persistent object. The business object public function (with the transaction) calls its own private function to carry out the public function.

Summary

In this chapter we compared and contrasted our proposed adaptive orthogonal integration architecture with the traditional integration approaches and showed how our approach avoids the issues relating to hierarchical information models and related problems resulting from functional decomposition. In addition, we showed how our architecture lends itself to the concept of adaptive business process by taking advantage of dynamic integration. This is essential to facilitate application portfolio interoperability.

The next chapter presents the adaptive application integration architecture. The presentation in that chapter is from an implementation perspective.

Chapter 6

The Adaptive EAI Architecture Framework

The central theme of the adaptive enterprise application integration (EAI) architecture framework is to provide an enterprise infrastructure for sharing objects and processes, making them accessible to applications at the enterprise level and thus facilitating application integration. Figure 6.1 gives an illustration of the adaptive EAI architecture framework. This is effectively a high level pattern corresponding to the Adaptive Orthogonal Integration architecture model presented in Chapter 4. The core component is the distributed object framework, such as CORBA, that acts as the essential glue for distributed object interoperability and fault-tolerant architecture. The major components in the adaptive EAI architecture are as follows:

- Distributed object framework
- Domain application adapters
- Application Mediation core
- Event mediation module
- Event Handlers
- Enterprise application architecture
- Business processes
- Package mediation

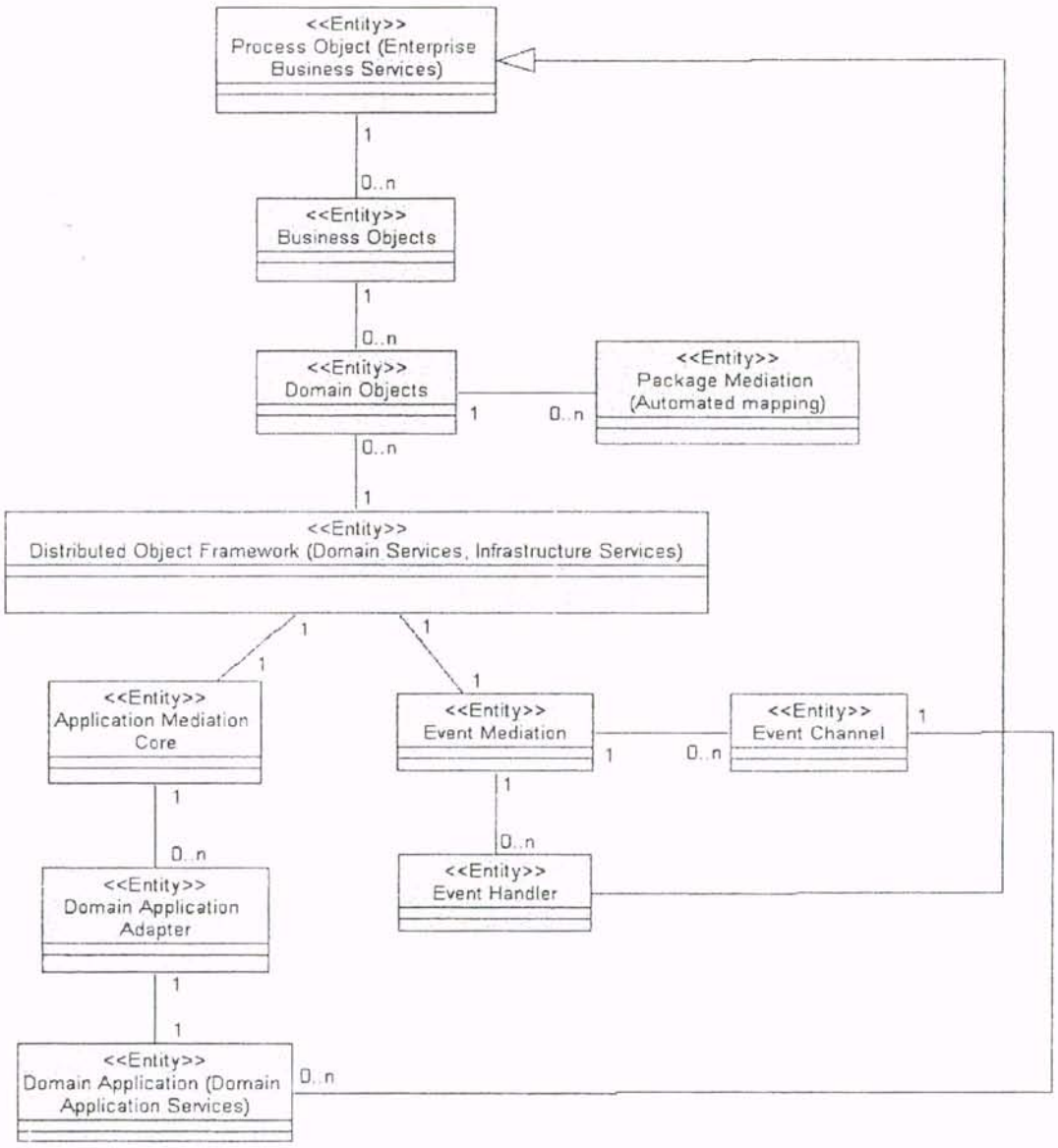


Figure 6.1: Adaptive EAI Architecture Framework

The EAI architecture framework facilitates object as well as design reuse. Object reuse results from the domain and infrastructure services being incorporated into the solution. Design reuse is a consequence of the design patterns and architecture principles

employed. A UML model-based translation development process can support the overall software component creation process. The component being developed can be specified as a UML model, complete with behavioral specification done using UML extended with an action semantic language (ASL) [Mosses 1992; Mosses 1996; Doh 1994; Even 1990].

6.1 Distributed Object Framework

The distributed object framework is the infrastructure mechanisms standardized by CORBA and can be implemented using a standard off-the-shelf object request broker (ORB) such as IONA Orbix [Iona 1999]. The role of the ORB is to unify access to application services, which it does by providing a common object-oriented, remote procedure call mechanism. The CORBA Interface Definition Language (IDL), an essential component of the family of standards that define the CORBA architecture, provides a language-neutral and location-neutral messaging interface for component interaction. CORBA provides a number of standard infrastructure services inclusive of the following:

- **Externalization Service**

  Externalization is the process of taking program data structure and other object states and converting that information into a form that can be stored or transmitted. This process involves removing pointers and converting binary data into flat representations so that the information can be considered to be a stream of bytes without additional internal structure. Externalization plays a very crucial role in object location transparency. We can think of this as representing an object graph as a flat stream by doing a graph traversal.

- **Persistent Object Service**

The Persistent object service provides the ability to store the state information and data of objects into a relational database management system (RDBMS) or an object-oriented database management system (OODBMS). The Persistent object service provides for the replacement of the persistence protocols used within the service. A persistence protocol is a particular set of interfaces used by a persistent object to store its persistent state. Figure 6.2 provides the components of the persistent object service.

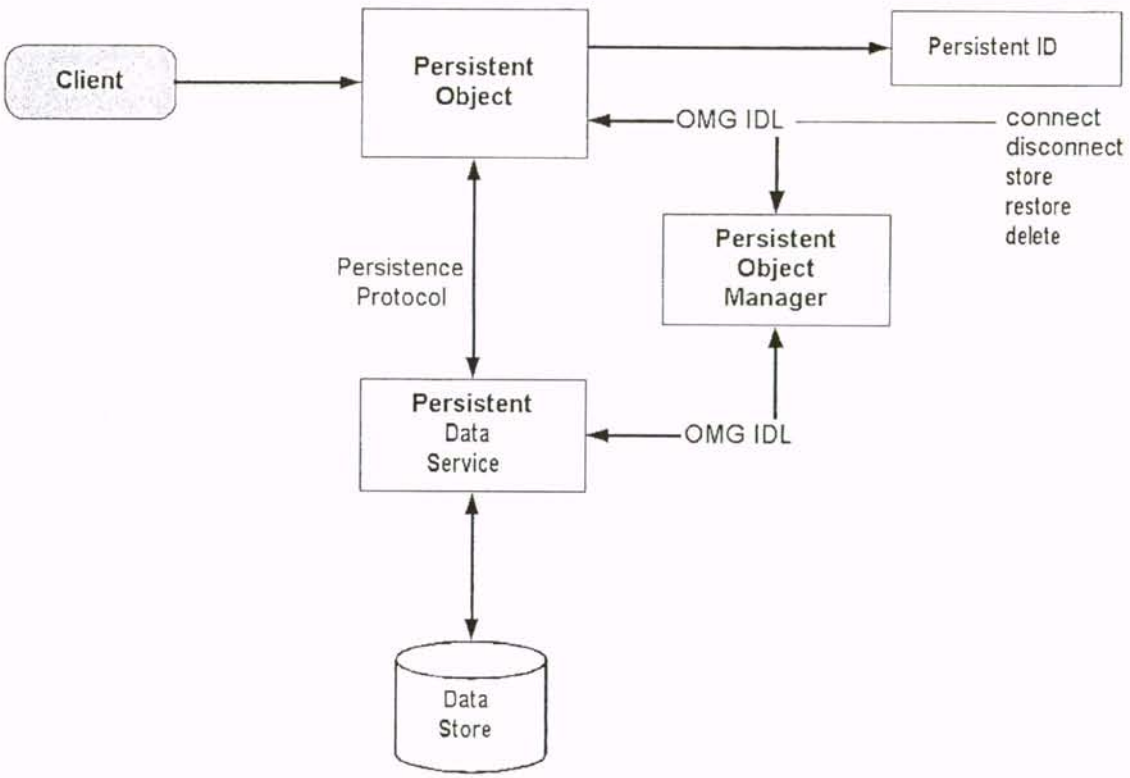


Figure 6.2: Persistent Object Service Components

- **Event Service**

The Event service defines generic interfaces for passing event information among multiple sources and multiple event consumers. It allows for decoupling of the generators and receivers of events and for a large number of receivers that are managed by the service and not by the event sources. Event notification is one way of using this service. This service can also be used as a multicast capability. The service provides a general set of mechanisms for allowing recipients of event information to register their interest in events. This also allows the source of a multicast message to post the message once and have it conveyed to multiple recipients without direct knowledge between the event's source and the recipients or direct connections between the supplier and consumer objects. Figure 6.3 provides the components of the event service.

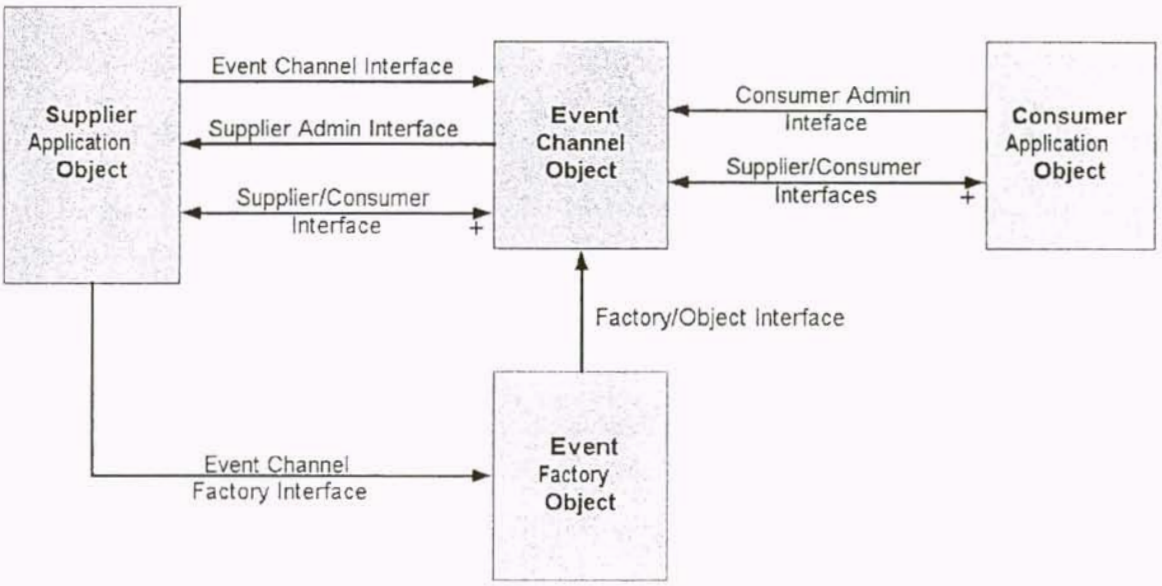


Figure 6.3: Event Service Objects

- **The Concurrency Service**

  The Concurrency Service is a general-purpose service for ensuring atomic access to distributed objects. The Concurrency Service provides synchronization across distributed environments and allows the locking of individual objects or several objects to provide atomic access when changing state information. This allows applications an enabling capability for assuring coherent state information in distributed systems. Previous capabilities for concurrency control, which are operating system and language dependent, do not extend easily to distributed systems. The Concurrency Service provides the advantage of portability and the effective use of concurrency across multiple operating system platforms and languages in a distributed environment.

  The Concurrency Service works with the Transaction Service in a closely coordinated manner. Regardless, it is likely that the Concurrency Service would be one of the key services used during transaction processing. When the Concurrency Service completes a transaction, either by committing the transaction or aborting the transaction, the combined services are responsible for releasing any concurrency locks that were put in place during the transaction. The locks are reset to their unlocked state. This is an important part of the clean-up on termination of transactions.

- **The Transaction Service**

  The Transaction Service is a general-purpose set of interfaces that can be used to encapsulate a variety of existing technologies and provide standard interfaces across all implementations of transaction monitors. For example, the Transaction Service is

designed to be layered over monitors that are compliant with the X/Open distributed transaction protocol [OG 1994]: monitors that use the Tuxedo protocols, and object-oriented database conformant with the OBMG-93 standards. The Transaction Service is a general capability that allows the manipulation of the state of multiple objects in a distributed environment. It builds on the capability of the Concurrency Service for controlling access to individual objects. The Transaction Service allows modification of the state of multiple objects to be viewed in a reliable and highly consistent way. The Transaction Service supports the ACID properties of transactions (atomicity, consistency, isolation, and durability).

- **Transparent Transactionality**

  All processing can be performed within the context of a transaction that ensures application consistency and full transparent recoverability.

- **Asynchronous Processing Model**

  This processing model minimizes synchronization points within the application for high throughput and traffic peak absorption.

- **Application Recoverability**

  The CORBA architecture provides recovery services for applications that are both transactional and non-transactional, enabling customers to integrate legacy data and process sources into the recoverable application model.

- **Reliable Queuing**

  Application developers can build models on different nodes and communicate via distributed asynchronous transactions.

- **Kernel Level Threading**

  Kernel threads are lightweight processes. The CORBA architecture is designed to take advantage of kernel mode threads, which minimizes context-switching overhead that reduces latency and improves performance.

- **Transparent Scalability**

  The CORBA architecture framework scales transparently by supporting single and distributed scaling mechanisms, providing flexibility for the application designer to make trade-offs among cost, manageability, and redundancy – as required by existing application and business models.

6.2 Domain Application Adapters

The domain adapters form a very powerful framework for incorporating domain applications into the overall enterprise framework. The adapter framework can be extended to include interfaces for integrating protocols such as SNMP, and provide support for CORBA, persistent resource managers, etc. We have developed a generic design pattern that can be used to implement this capability. This pattern is part of the set of EAI design pattern that we present in Chapter 8.

The domain adapters are used to expose the information and data models of the legacy applications. This exposition allows us to look at the application services provided by the legacy application as extension of the infrastructure services provided by the distributed object framework. Again, this approach is consistent with the work of the OMG in their specification of domain services.

6.2.1 Domain Application Adapter Design Pattern

The domain application adapter pattern provides a consistent and repeatable manner for thinking about and integrating domain silo applications into a unified framework. Figure 6.4 presents schematics of the adapter pattern. The pattern provides three main functional areas: (1) the communication transport, (2) the application interface, and (3) the adaptor, which is a container for the application functionality. The *ConcreteApplicationTransaction* class is the main class interacting with the application functionality via the application program interface (API). The API may have to be enveloped in a special purpose wrapper to handle platform specific data type conversion. This is shown in Figure 6.5.

The *ConcreteAdapter* class acts as a container for the concrete application transaction object. It effectively overrides the *GetTransaction* method to return the relevant transaction object. This is an instance of the Factory Method design pattern at work. The *TransactionMgr* object associates an *ApplTransaction* object with a communication object and returns it to the Adapter. Thus, we can have multiple concurrent transactions occurring at the same time. Hence, this pattern is inherently scalable and therefore imposes no limit on the performance of the legacy application it is adapting.

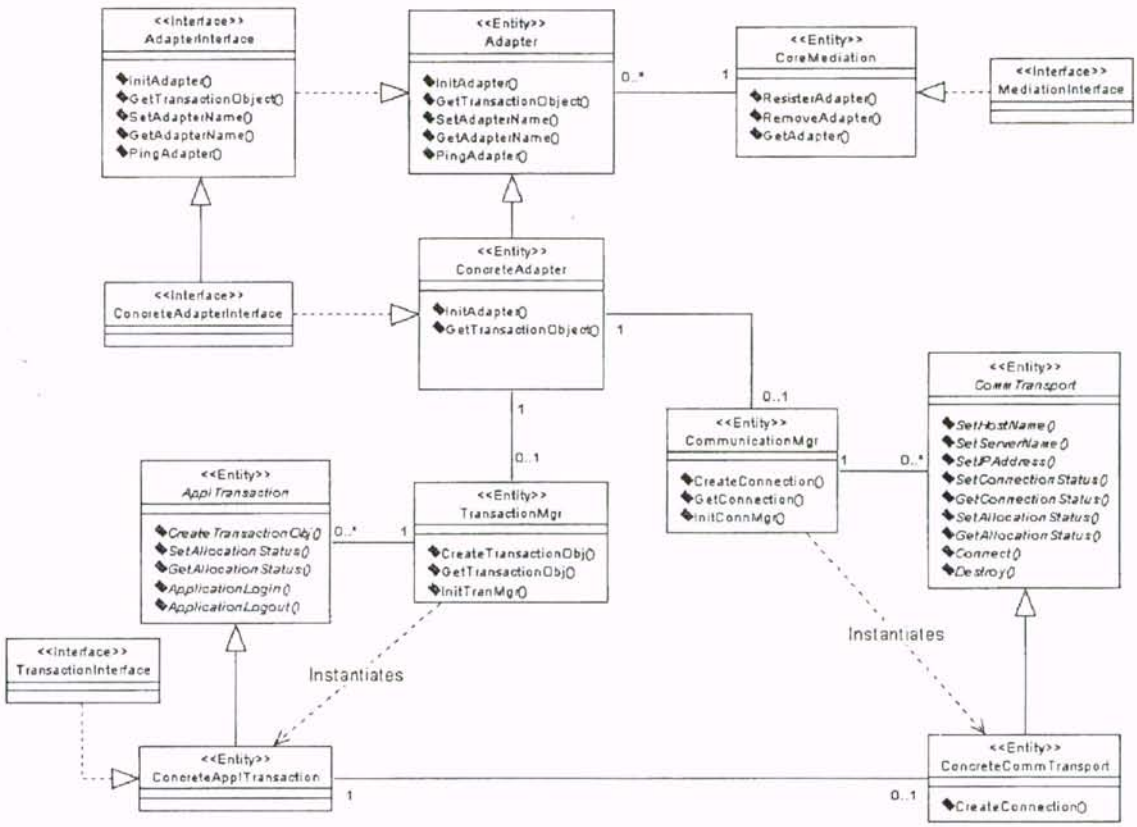


Figure 6.4: Domain Application Adapter Design Pattern

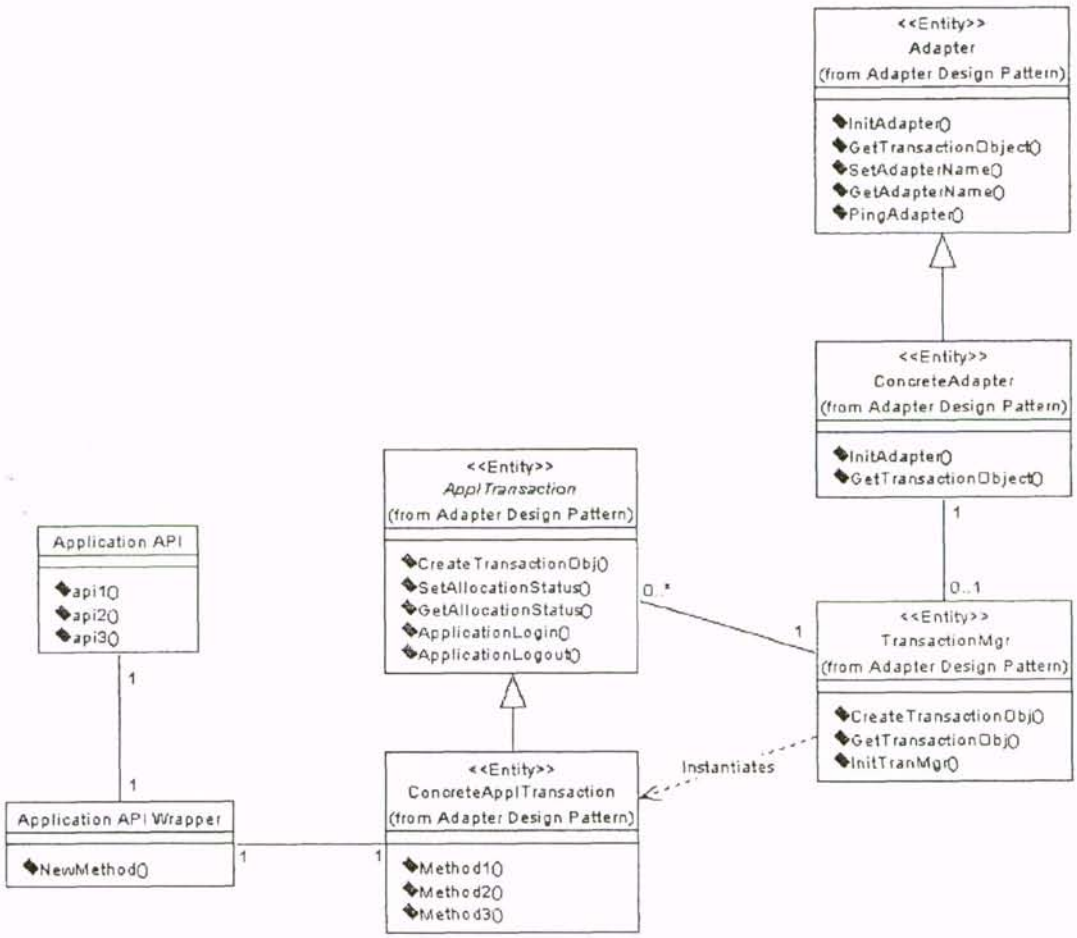


Figure 6.5: API Specific Wrapper

6.3 Application Adapter Mediation

The application adapter mediation is a central component that provides the actual hooks to anchor the domain application via the domain adapters to the core distributed object framework. We have developed a design pattern to accomplish this functionality. This pattern is part of the set of EAI design patterns. The application adapter mediation pattern is presented in the next section.

6.3.1 Application Adapter Mediation Pattern

The mediation pattern, Figure 6.6, provides the framework for the adapters to register themselves and make their functionality available. It uses the principle of delegation to present the functionality of the respective application via the adapter to interested parties.

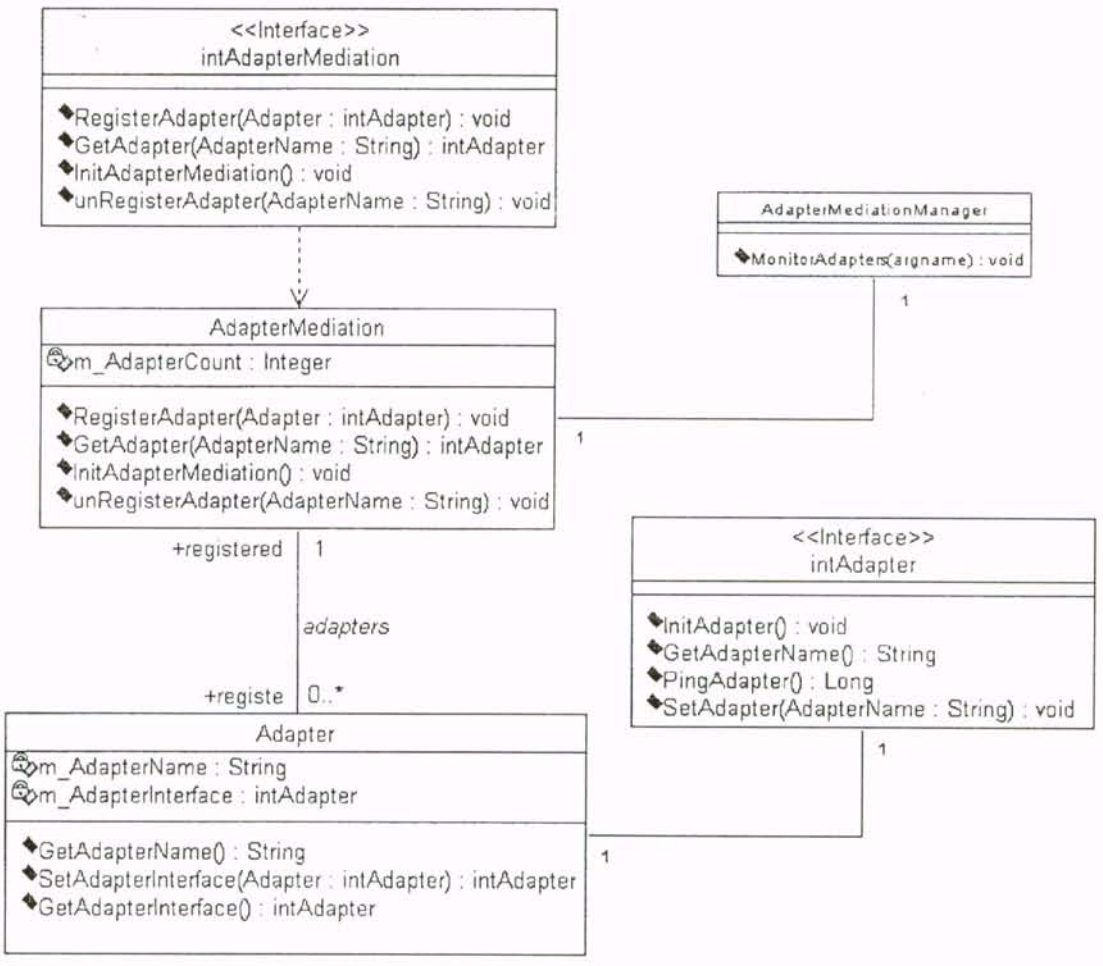


Figure 6.6: Application Adapter Mediation Pattern

6.4 Event Mediation

Most legacy applications have no concept of asynchronous processing. This is a consequence of the push oriented transaction model that typifies software applications

developed using a functional decomposition methodology. The event mediation module provides a mechanism to retrofit legacy applications with asynchronous notification capability. This is in effect a standardized layer that is wrapped around other rudimentary mechanisms (such as polling) that can be retrofitted into the legacy applications.

6.4.1 Event Mediation Pattern

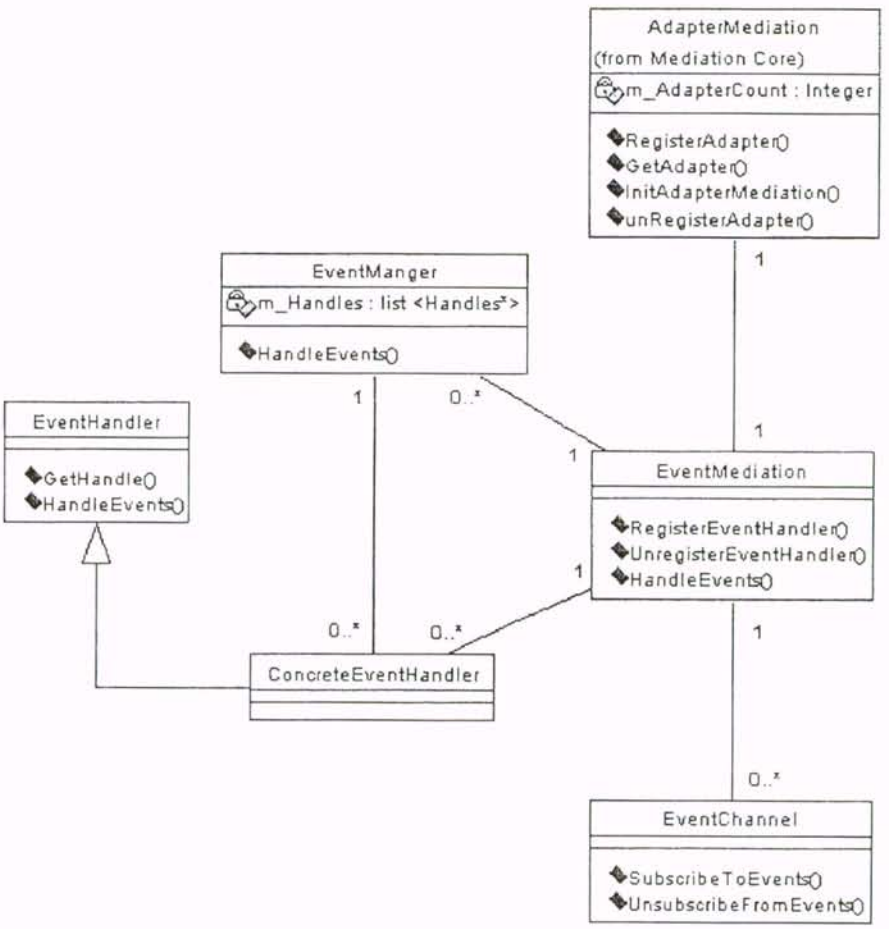


Figure 6.7: Event Mediation Pattern

The event mediation pattern, Figure 6.7, is an extension of the Subject Observer pattern. Concrete event handlers are registered with the event mediation object to receive specific event notifications. The respective event manager will invoke the event handler to process the delivered event. The event handler is able to activate process objects via the event mediation and adapter mediation objects.

6.5 Package Mediation

This module performs data mapping between the enterprise domain, enterprise business objects, and the silo domain application object representation. This module is transparently invoked whenever there is any data transfer between the enterprise view and any of the domain silo applications.

6.6 Flexible Business Processes

Business processes are the mechanism by which an enterprise implements its business models. Thus, these act as mini-workflow processes that coordinate and collaborate the interaction between business objects and business object managers in the N-Tier Orthogonal architecture. The business model, the critical intellectual property of the enterprise, is programmed into the business processes. Depending on the technology employed in the implementation, business process objects can be developed as deployable runtime executable components. Since business process objects are *objects* in the object-oriented sense, all the properties inherent to the OO paradigm apply to them. Thus, one can use the principle of inheritance to form a new business process that can be further refined to address changes in market conditions. The new business object is

effectively a new version. This is the manner in which this architecture achieves its adaptable capability.

An interesting consequence of the collaboration mechanism employed by the business process objects is that it presents a new way of looking at integration. A finite state automaton (FSA) can be used to represent the collaboration sequence of the business process objects. This coupled with the notion of versioning mentioned above forms a very powerful way of looking at integration. Integration is effectively dynamic in nature. This *Dynamic Integration* can be envisioned as a business process traversing a specified path through a set of nodes. The nodes represent the business objects. Hence, for a given set of nodes, the execution of different business processes result in different outputs.

The abstract data view (ADV) concept is central to the notion of adaptive business process being presented. The ADV design concept promotes reuse of interface specification through the principle of composition since it allows complex interfaces to be built from simpler interface components. In addition, the specification constructors have the ability to extend the capability of the module components being combined. The business models represented as automated workflows are, in fact, the extension capability provided by the ADV or business process components. In so doing, the ADV mechanism provides the capability for specification of the collaboration logic between the enterprise business objects as well as the enterprise workflows that define the business models of the respective organizations. In principle, the ADV approach for business processes is an extension of the basic delegation mechanism. Figure 6.8 provides a schematic representation of the concept.

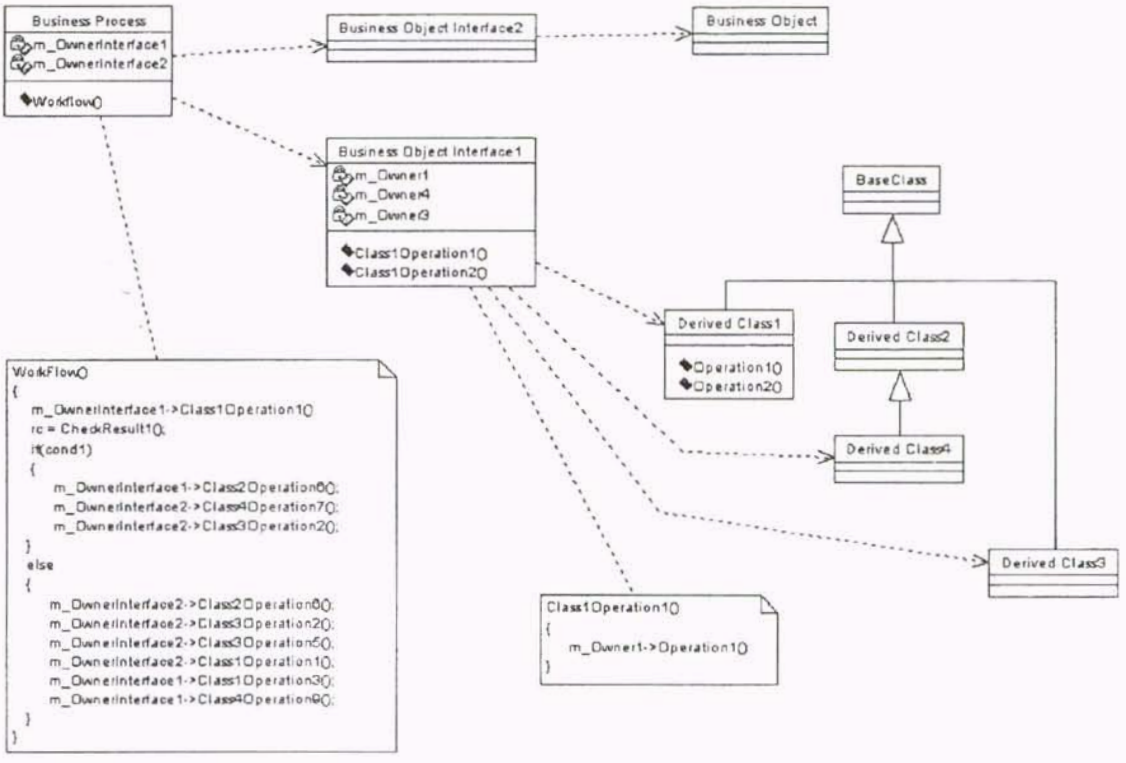


Figure 6.8: ADV Representation of Business Processes

6.7 Putting It Together

The Adaptive EAI Architecture Framework presented in Figure 5.1 lends itself to significant automation. In Chapter 8, we present a UML based component development approach using model-based development to automate much of the coding. There is significant scope for reuse in this approach. The domain application, domain adapters, and mediation core facilitate wholesale reuse. As long as the domain application has the functionality, no coding is needed for reuse. It is the business process objects that will have to undergo the most modification to implement the enterprise business models. This is understandable since this is what affords an enterprise the ability to differentiate itself.

In Chapter 8, we also propose a translation process that can take a UML model of the business process and generate the deployable runtime component.

Chapter 7

OSS Integration in the Telecommunications Industry

Operations support systems (OSS) are the mission-critical enterprise hardware and software systems that telecommunications service providers use to implement, manage, and support the complex transmission and delivery systems of the communications environment. OSS software systems can be segmented into three very broad categories: (1) Customer care, customer support, and billing, (2) provisioning and order management, and (3) network management. The deregulation of the telecommunications industry has resulted in significant structural changes in the telecommunications market, causing major modifications of the business models for telecom providers. The changes in the business models have direct impact on telecom companies' OSSs and interoperability of the OSSs.

The deregulation of the telecommunications industry has resulted in the proliferation of new service providers and technology offerings, such as wireless, long distance, Internet service providers (ISPs), and cable. The new players have fueled a robust market for telecom OSSs, as they position themselves for battle with the incumbent service providers. As these providers jostle for market position, convergent service offerings and other valued-added services are used to differentiate and attempt to gain customer lock-in.

The solutions offered by OSS vendors are highly fragmented. Most vendors offer solutions that can be categorized in one of the three segments mentioned above. These solutions have little or no integration or interoperability between the various categories. Thus, integration of OSS systems, as a general rule, is non-existent, resulting in severe inefficiencies in the back office. This situation is exacerbated by the fact that each of the domain-specific OSS software application is developed using diverse software technologies and architectural frameworks, resulting in very disparate solutions that are very difficult to integrate and interoperate. Swivel–chair integration is the order of the day.

The business processes are normally integrated into the OSS applications' APIs. This results in service providers not being able to be responsive to market pressure arising from competition, consolidations, and technological innovations. In addition, because of the complexities of these systems, traditional peer-to-peer integration approaches are not a viable solution. More innovative solutions must be adopted.

We need an architecture that takes the business processes out of the APIs and elevates them to a higher level of abstraction – such as a business process management layer. This layer should be integrated to the service management layer using configurable soft-edge domain adapter frameworks. The impact of such an orthogonal architecture is that the service provides can adapt rapidly to changes in their business models and these changes do not percolate to the lower-level core OSS applications. Hence, the adaptive EAI architecture model and framework.

7.1 Key Industry Standards

Experience with telecommunications in the deregulated market has proven the utility of the ITU Telecommunications Management Network (TMN) information model as the organizing principle for the information/data architecture (ITU-T M.3100, *Generic Network Model*). The information model of TMN is distinct from its corresponding implementation model. The TMN information model defines layers of abstraction that are appropriate to different aspects of the overall telecommunication enterprise. TMN also specifies an agent-based implementation model for the network management software. The information model, as specified by ITU, is illustrated in Figure 7.1. A brief summary of the main components and associated features of the TMN model is presented below.

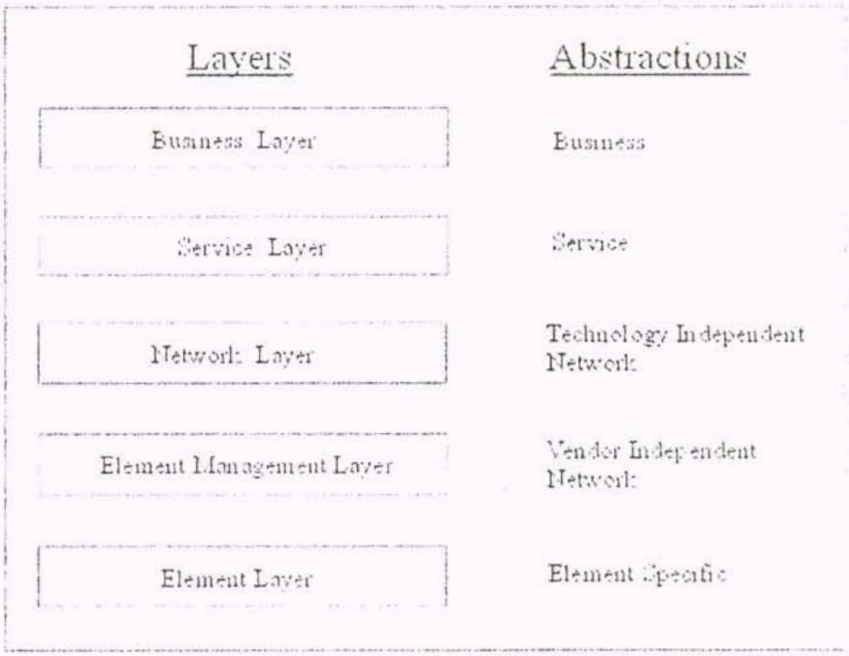


Figure 7.1: ITU Standard TMN Information Model

The TMN model has five layers of abstraction, namely:

1. Element Layer

   This is the least abstract view of the total system, consisting of the software interface to the hardware components that comprise the network.

2. Element Management Layer

   This layer abstracts the differences among similar components, hiding the differences between different products of equivalent type. This abstraction allows management of technology types by the upper layers.

3. Network Management Layer

   At this layer, we can abstract the difference between technologies to transform the network configuration management problem into a graph problem.

4. Service Layer

   At this layer, applications are constructed to provide services to the customer of the communications firm. This layer provides service software developer with a service-based, rather than a hardware-based, view of the network.

5. Business Layer

   This layer consists of business applications such as billing, rate control, and customer care. This layer provides the business software developer with business-model based, rather than hardware-based, view of the network.

7.2 Solution to the Telecom OSS Integration Problem – A Business Process Centric Approach

Traditional application development approaches often embed the business process logic within the application APIs. Any attempt to modify the business model result in

programming changes to the applications that have corresponding down time to the application availability when installing a new version that embeds updated business processes. The corresponding business implications are many folds: application down time resulting in lost revenues, dissatisfied customers, and difficulty in the specification and implementation of business process. Most OSS applications were developed with little or no concern for interoperability and as such, resulted in enterprise stovepipes. The competitive market pressure in the telecommunications industry has propelled service providers to demand fully integrated operational functionality from all the OSS applications that they have to operate in the commission of their business. A number of approaches to integration have emerged to address the OSS interoperability problem. The most notables are the peer-to-peer and hub and spoke (broker) models.

The intensely competitive nature of the telecommunications business market place mandates that a telecom service provider must have the ability to perform very flexible business process reengineering. Telecom service providers are continually redefining their business models in an effort to differentiate themselves from their competitors and to ward off competition.

In this environment, high availability and customer care management is essential to establishing market acceptability and high customer retention. These are essential ingredients to successfully operate a business in the Internet driven economy. These requirements coupled with the necessity of flexible business process reengineering have driven us to reevaluate the approaches that have been taken to address the enterprise application integration (EIA) problem. The above requirements mandate a business process driven integration framework that allows individual business processes to be

represented, monitored, and integrated with existing systems and users across the enterprise. With the ability to dynamically reconfigure active business processes (software fault tolerant and hot swappable capabilities), allowing users to continuously adapt to rapidly changing business conditions. This process-driven infrastructure provides businesses the ability to adjust and alter their operational systems to changing market conditions without any down time.

Application portfolio integration is mandatory in order to support many carriers' organizational goals. These goals include operational efficiency via process flow through and customer intimacy to enhance customer satisfaction. Examples of this include knowing what a customer has ordered across multiple products, what problems he or she has experienced, and his or her billing and payment history. Addressing this integration challenge requires a comprehensive application portfolio assembly approach that can exchange information among multiple application architectures each with different data and process models and with different data exchange models.

The challenge is to create a means of integration at the business process level. An information broker can create generalized event and object models to normalize the flow of information between OSSs. We can create adapters to the various OSS applications. This serves the purpose of migrating the architecture from a multi-point, spaghetti architecture into a much more manageable hub-and-spoke arrangement. However, the adapter approach is just the starting point because it does not allow for a flexible business architecture. To create an architecture that enables best-of-breed OSS selection, while not sacrificing integration and data sharing, requires another layer of "business aware" software that runs above the information broker. With such a layer in place the business

process can change without affecting the underlying applications. And conversely, IT should be able to change applications without affecting the business processes.

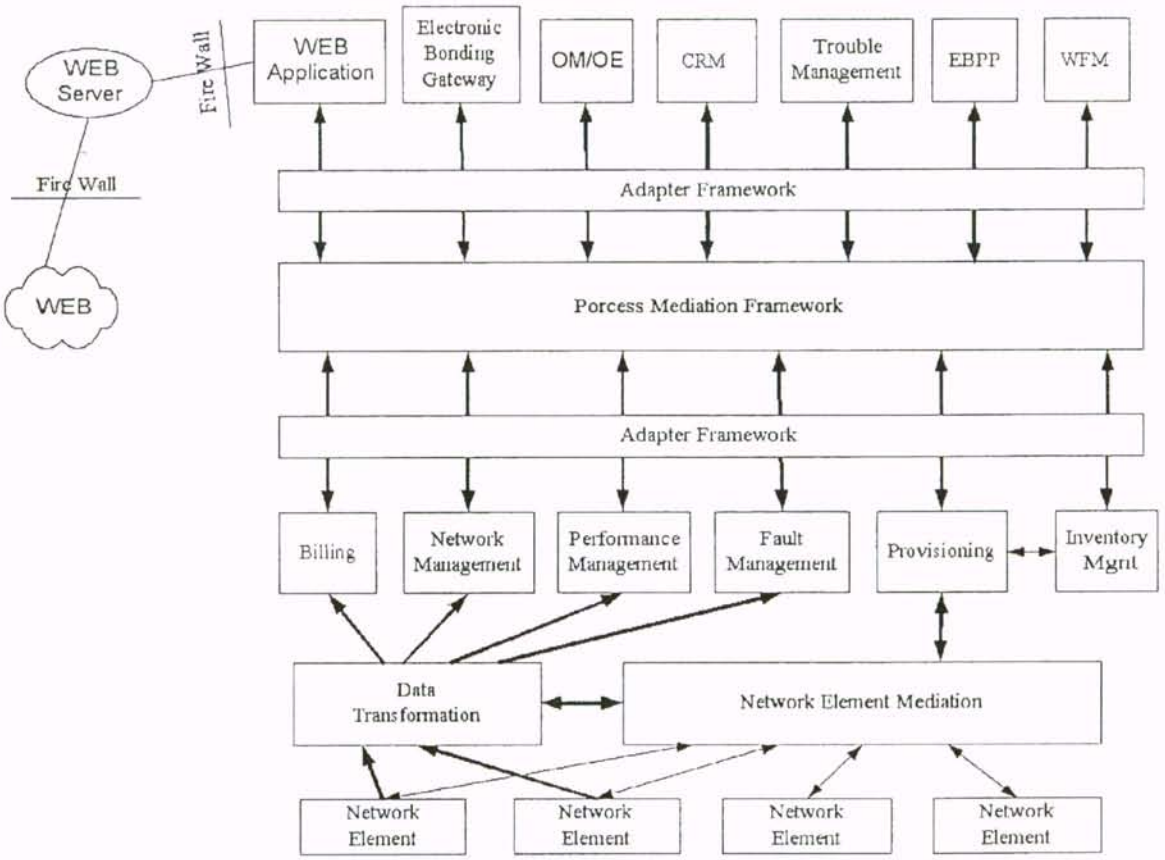


Figure 7.2: Generic OSS Integration Architecture

The architectural approach outlined in Chapters 4-6 are used to build the schematics of integrating the typical OSS domain applications that the Telecom Service Providers will use in the commission of telecom services to their customer. Figure 7.2 shows a logical OSS Integration architecture. This diagram has most of the applications a telecom service provider will need.

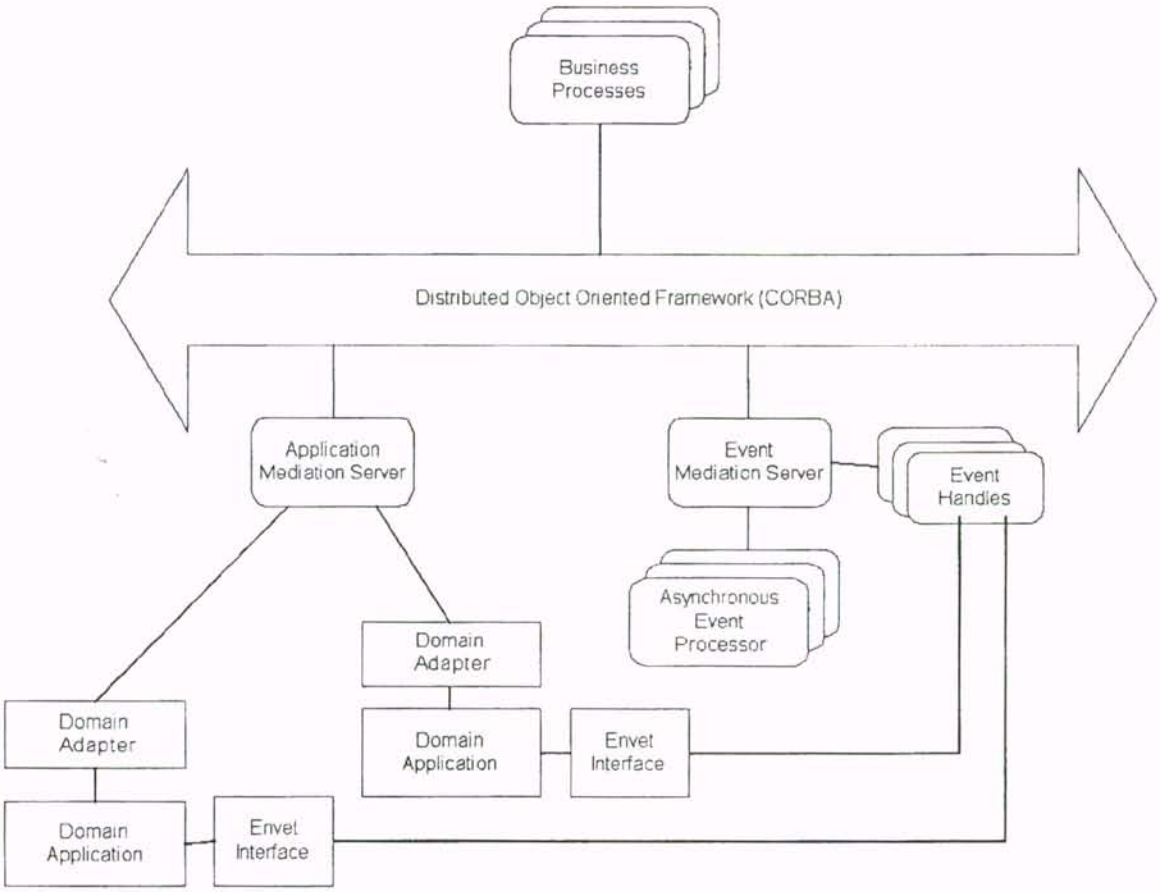


Figure 7.3: EAI Context Diagram

Figure 7.3 presents a simplified context diagram for the EAI problem. Domain silo applications are plugged into an application mediation server via their respective domain adapters. These applications can be retrofitted with asynchronous event delivery mechanisms that can be interfaced to an event mediation server via event handlers. Event processors (event handlers) can be registered to the event mediation server. The event processors respond to event notifications and can trigger operations in the business processes. This provides an automated mechanism for legacy applications to trigger business processes.

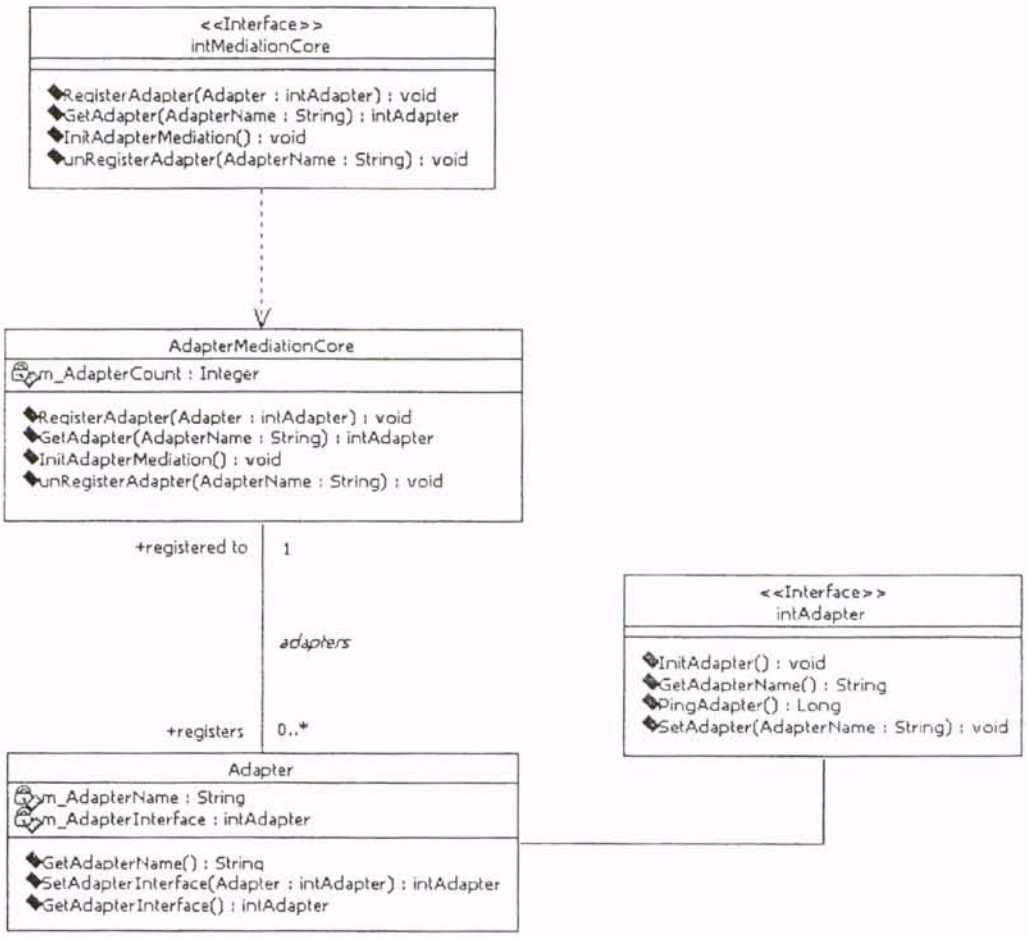


Figure 7.4: Application Mediation Server

Figure 7.4 presents the main components of the application mediation server. Its primary function is to provide for registration and use of the application domain adapters. Adapters as fitted with interfaces corresponding to abstract data view classes.

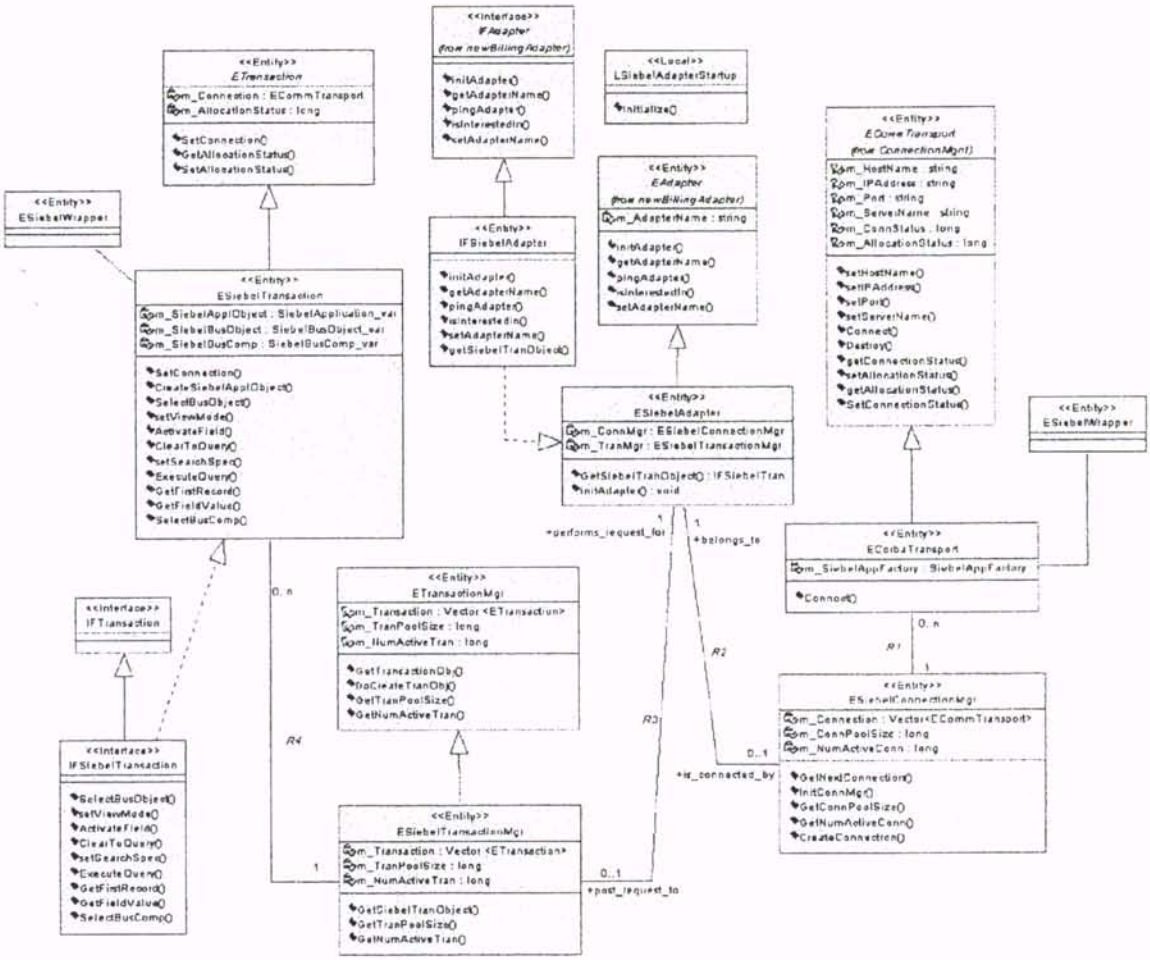


Figure 7.5: Billing Application Adapter

Figure 7.5 provides the major classes in the adapter for the billing application. This adapter was developed around the Portal Billing system.

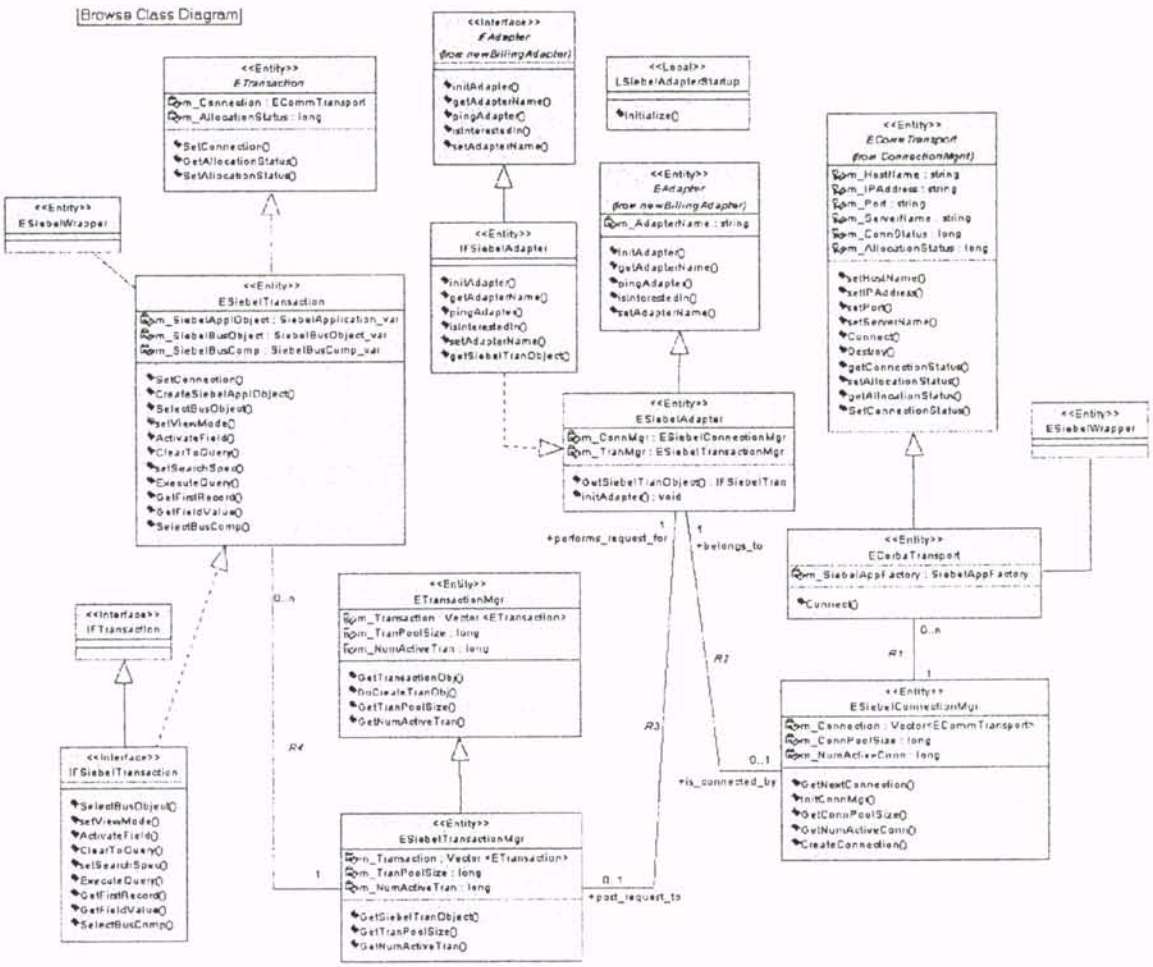


Figure 7.6: CRM Application Adapter

Figure 7.6 provides the major classes in the adapter for the customer relationship management (CRM) application. This adapter was developed around the Siebel CRM system.

7.3 Information Architecture: Static Domain Model

This section presents the information architecture corresponding to the enterprise mediation layers (domain objects, business objects, and business processes) for the adaptive orthogonal EAI architecture model presented in Chapters 4 and 5. The material provides a gradient walk through the logical architecture from a domain analysis viewpoint.

7.3.1 Customers and Orders

Figure 7.7 shows the simple and obvious relationship between customers and their orders. An entity (company or individual) who has ordered service(s) from a telecommunication service provider is called a customer.

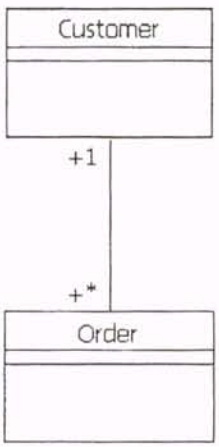


Figure 7.7: Customers and Orders

Customers can place any number of orders for service. There is a one-to-many relationship between a customer and its orders. The system should remember all orders a customer has placed, even after the entity ceases to be a customer. If the entity becomes a

customer again, all past order history is restored. The navigation between customers and orders should be two-way. Users should be able to access all past and present order information from the customer information and reference customer information, such as billing address, from the order information.

7.3.2 Service Enrollment Simplified

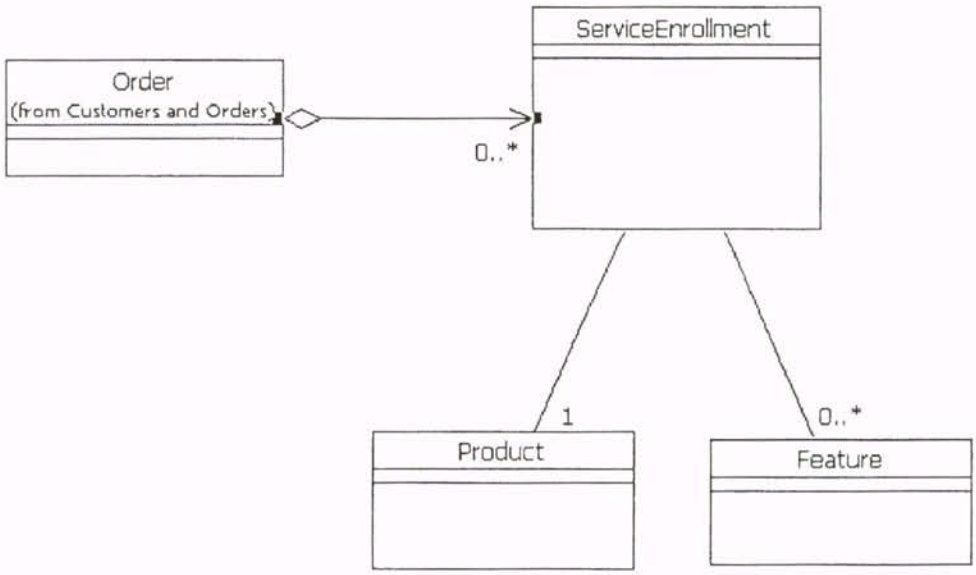


Figure 7.8: Service Enrollment Simplified

Figure 7.8 presents a simplified diagram of the details of a service enrollment. We will elaborate in Section 7.3.4 to make it represent a more realistic real world model, but this view will suffice for now. A service enrollment is a specific instance of some telecom service ordered by a customer. The term *enrollment* is used to represent the customer's use of, or enrollment onto, a service that can potentially be disconnected later.

An order can have several service enrollments, and each enrollment is associated with a particular product and an optional list of features. Examples of products are a business line or a combo trunk. Examples of features are call waiting or call forwarding. Features are dependent on products and are not inventoried. Products are independent and are inventoried or allocated, and thus there are individual instances of products. Features have instances so as to be associated with products in orders.

A specific product can be allocated to a customer via an order; later it can be de-allocated (i.e., returned to the catalog of available products) and allocated to a different customer. Line can be moved and telephone numbers changed. Thus, enrollment signifies the changeable relationship between products and customers (via their orders). The customer enrolls in the service. The service is added to the customer's service portfolio (list of services the customer has).

7.3.3 Order Operations

A more complete picture of the relationship between orders and their service enrollments, Figure 7.9, introduces the concept of *order operations*. In addition to requests to acquire and turn on service, orders are actually the primary medium of exchange between a customer and a telecom service provider, and embody all requests for modifying the customer's status. Orders include requests to change some aspect of service (Change orders) and requests to discontinue part or all service (Disconnect orders) as well as requests for new service and several miscellaneous kinds of orders. The subtypes of order operation shown in Figure 7.9 are not complete; however the list is easy to expand. Other

operations include From and To operations, which are two sides of a customer move, and Records orders, which make a change to customer or service data in some small way.

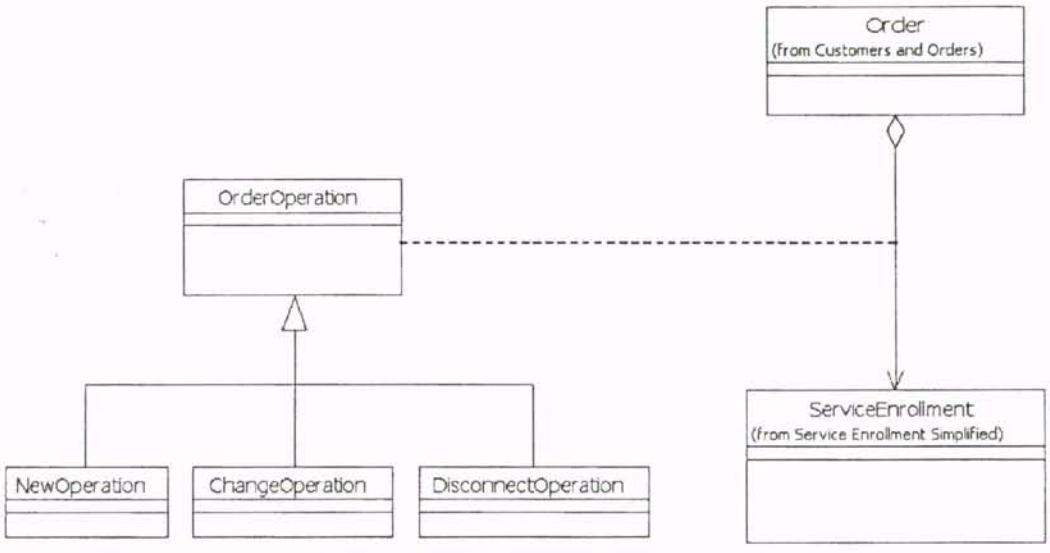


Figure 7.9: Order Operations

The order operation is conceived to be an association between an order and its service enrollments (SEs), as the operation describes what will be done to the enrollments. For example, for a disconnect order, the list of SEs represents the lines that will be disconnected.

Order operations are commonly thought of as being associated with orders rather than with individual items. This means that normally an operation applies to all SEs on the order. However, the structure presented in Figure 7.9 is more flexible and makes way for an ordering system where different items can be handled differently on the same order. For example, a single order could specify a *disconnect* of three lines while adding four lines of some other kind. If a single *order type* for each order was required, then

users or the system would have to create an order type at the order level and have rules to force all order operations to the same kind.

7.3.4 Offerings and Offering Instances

The diagram in Figure 7.10 completes and corrects Figure 7.8 and introduces the concept of offerings. An offering is a template for describing a product or feature. Each different kind of product (i.e., a product type) has a unique set of attributes that are relevant to it. For example, a business line has a telephone number, whereas a trunk does not. A trunk has a circuit ID, which may be considered irrelevant for a business line. Some types of features, which are also offerings, have attributes; some do not.

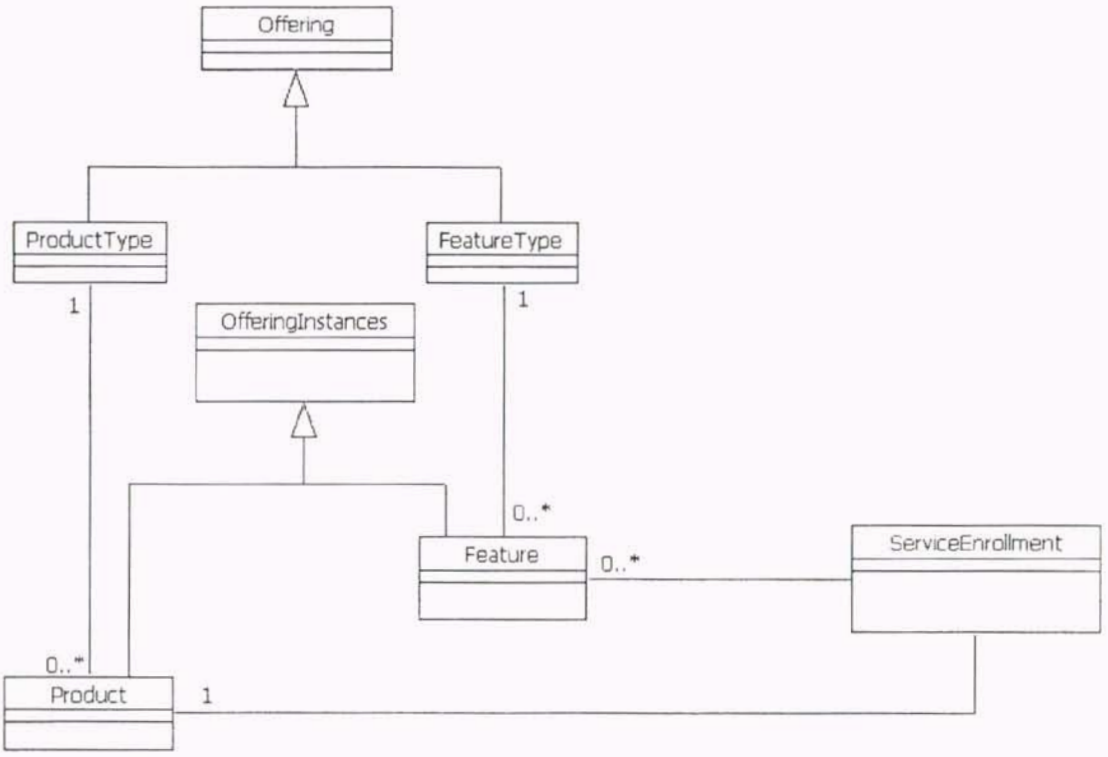


Figure 7.10: Offering and Offering Instances

An offering instance is a specific item being sold and provisioned for the customer. The two main kinds of offerings are product types and feature types. Each has an associated instance class, product and feature, respectively. Products can be allocated and inventoried. If you sell a customer a telephone number, nobody else can have it. Features simply serve to further enhance and modify products. A service enrollment can have only one product, but many features can modify the product.

7.3.5 Offerings

The diagram in Figure 7.11 expands the offering world. From the top, the OfferingCatalogHolder and its subclasses indicate that different kinds of enterprises can hold catalogs of offering, among which are service providers, offering vendors, and market organizations. This list is by no means complete and could vary in many ways. Offering catalogs are simple collections of offerings.

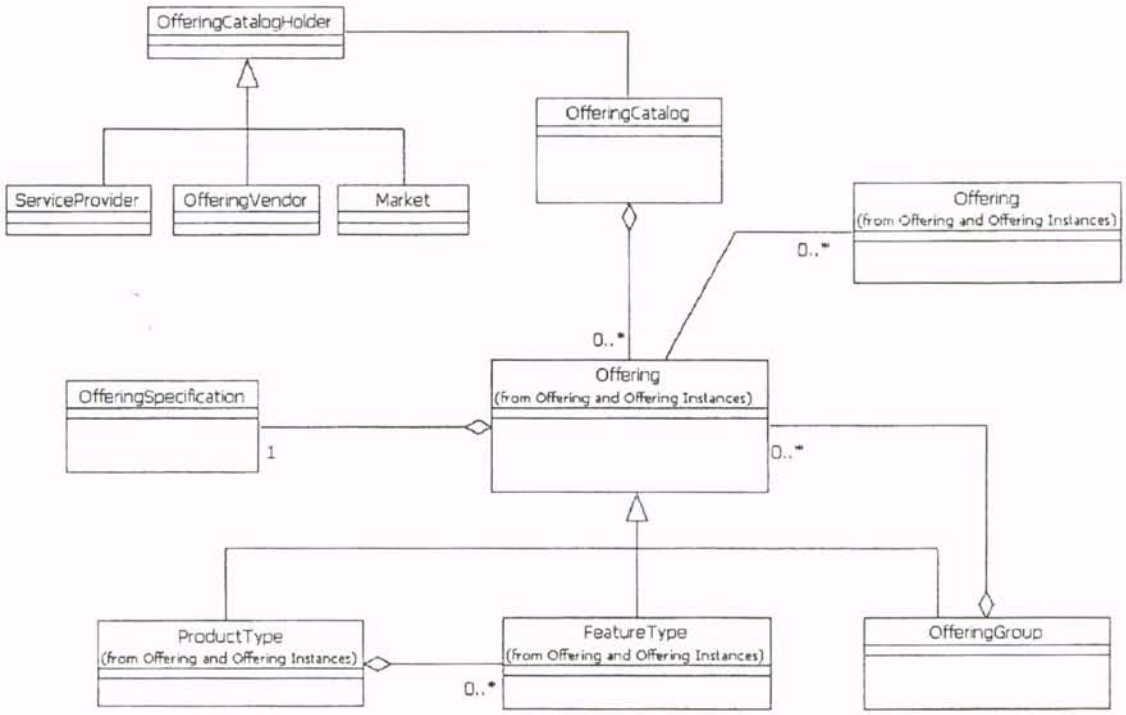


Figure 7.11: Offerings

The recursive or nesting structure of offering makes it interesting. An offering can contain other offering in offering groups. Product types and feature types retain their identity as leaf nodes of an offering tree.

This structure is the "implementation" view of products and features. That is, it addresses the problem of specifying complex configuration of products and feature definitions and their dependencies when defining the offering in the catalog, not when putting an order together. This structure can be used to manage complex rules for inclusion of products and features.

The offering class lists and describes the unique attributes of a certain offering. Example implementation of this could be a collection of (name, type, length) tuples for a

DDL definition. Each offering has an optional specification, which manages the rules of combination and exclusion between offering and their component offering (groups, products, and features). It represents a placeholder for constraints such as allowed features and nested products.

The *contains* relationship between product type and feature type is not structural in nature; it merely represents the rule that features are subordinate to products.

Implicit, but not shown, in the diagram in Figure 7.11 is the relationship between an offering and its offering instances, which relates this diagram closely with Figure 7.10. In Figure 7.10, offerings are seen as related to one or more offering instances.

7.3.6 Customer and Service Locations

Figure 7.12 provides the relationship between a customer and its service locations. Customers can have many service locations, which are specific places that they receive service and where circuits terminate. However, locations exist independently of customers – customers can move in and out of locations, but many of the facts about a location remain constant (or change independently of who is occupying the location).

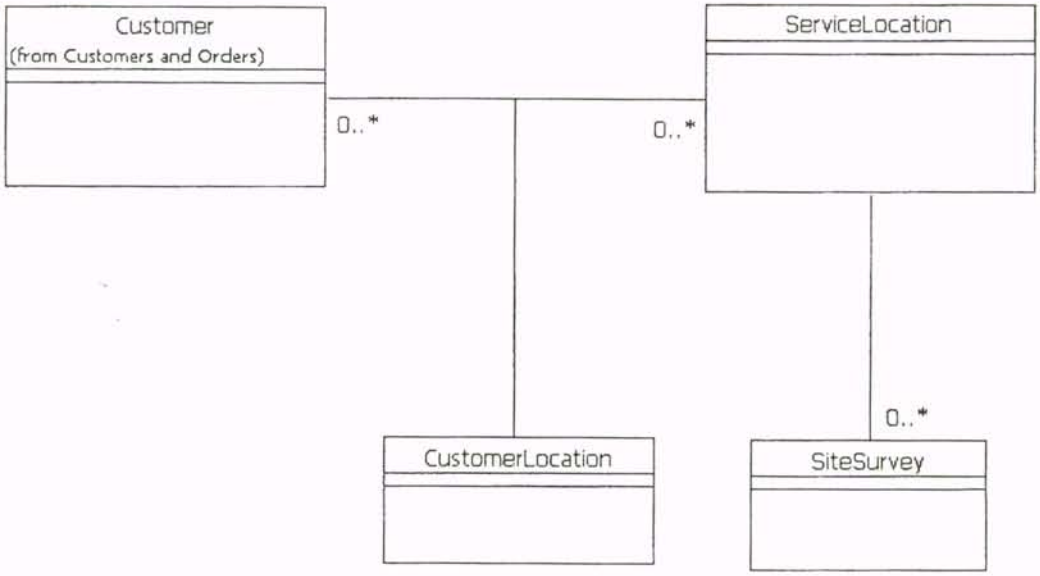


Figure 7.12: Customer and Service Locations

A many-to-many relationship exists between customers and locations. A customer can have many offices, and a location can serve several customers. The complexity of this relationship and the fact that the connection between a customer and its location is transient call for an object to manage and track the relationship. This is the customer location object, which signifies a customer at a location.

A site survey is an event in which an engineer from a service provider visits the service location and inspects the facilities. A report or form describing what was found is called a *site survey*, and may be printed at the time of the visit. Thus, conceptually, a site survey is a complete collection of information about a site as prepared at a certain time. A service location can be surveyed any number of times. Site surveys apply to locations, not customers, even though some of the data on the form pertains to the customer, as theoretically a site survey could be taken on a location unoccupied by any customer.

7.3.7 Customers and Service Enrollments

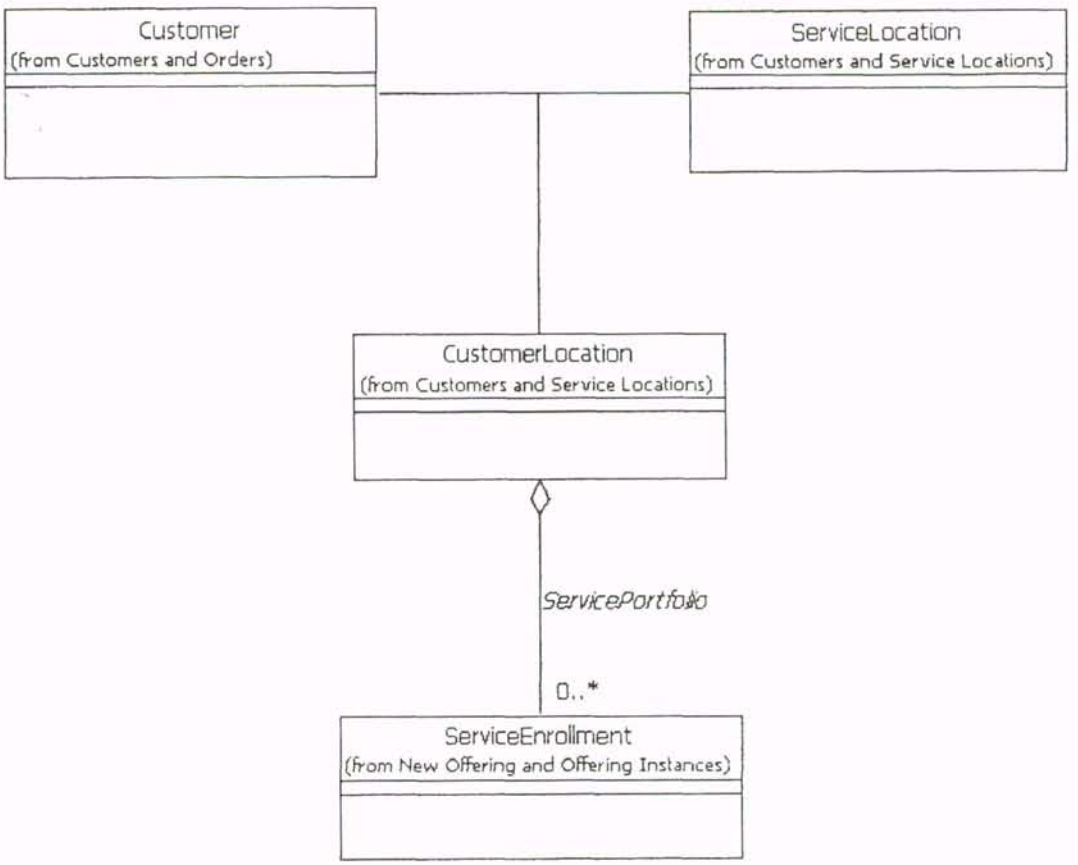


Figure 7.13: Customer and Service Enrollments

The diagram in Figure 7.13 introduces the concept of service portfolio. A customer's portfolio is the collection of all telecommunication products to which the customer is currently subscribed. Service portfolios are grouped by location, as the facilities at different locations determine many constraints about the possible service. Also, customers usually keep their locations separate from an accounting viewpoint. For example, bills are prepared for different locations.

Note that the service enrollment is the same object used to specify an individual product in an order. Thus, an order can be considered to be a delta (i.e., an incremental change) to the customer's service portfolio. A new customer has an empty service portfolio. The enrollments of the completed order are copied to the service portfolio, resulting in a portfolio that looks just like the first order's contents. A subsequent order to disconnect a line would result in a removal of that line from the portfolio, a New or Change order would affect an addition to the service portfolio, and so on.

An interesting way to view orders and service portfolios is from a configuration management viewpoint. The customer's portfolio is the aggregation of all the orders and could be reconstructed by starting from an empty portfolio and simulating the installation of each of the customer's past orders in sequence.

7.3.8 The Order World

The diagram in Figure 7.14 completes the picture of the concepts surrounding the order. In addition to the key relationships previously discussed, some more details are mentioned in this section.

Orders are related to other orders. There are various reasons for this. For example, an order can be related to an earlier incompletely fulfilled order or to an order for facilities (such as a T-1 line) on which it is dependent.

An order can have any number of remarks entered for it during the process of installing the service. An order is assigned to a sales representative for commission purposes. The sales representative can be a person, a team, or even an external organization.

An order is also assigned to a customer care person who acts as the single point of contact where applicable and also acts as the assurance person, making sure all tasks are completed to turn up service.

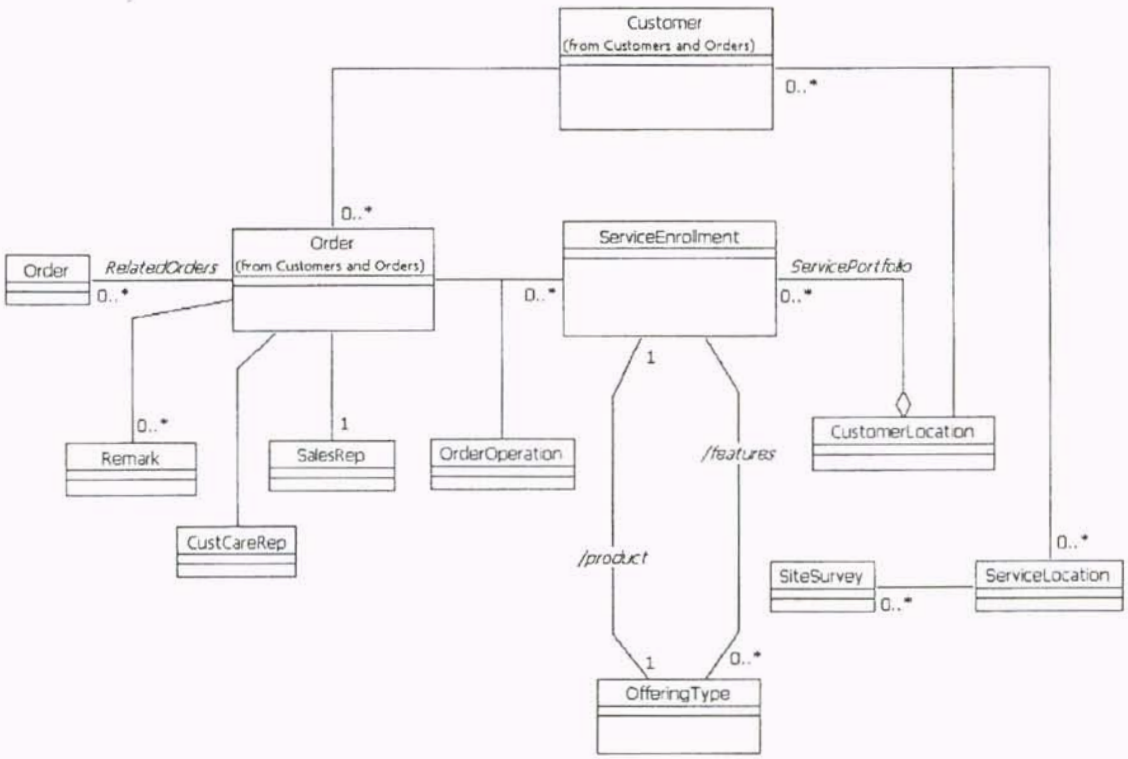


Figure 7.14: The Order World

Note that offerings are illustrated differently than in the previous diagrams. This diagram has hidden the inheritance relationship between products and features and the offering instance. Both products and features are shown here as offering instances (which they are) and the fact that they are products and features is derived (the slash before the label in the line).

7.3.9 The Customer World

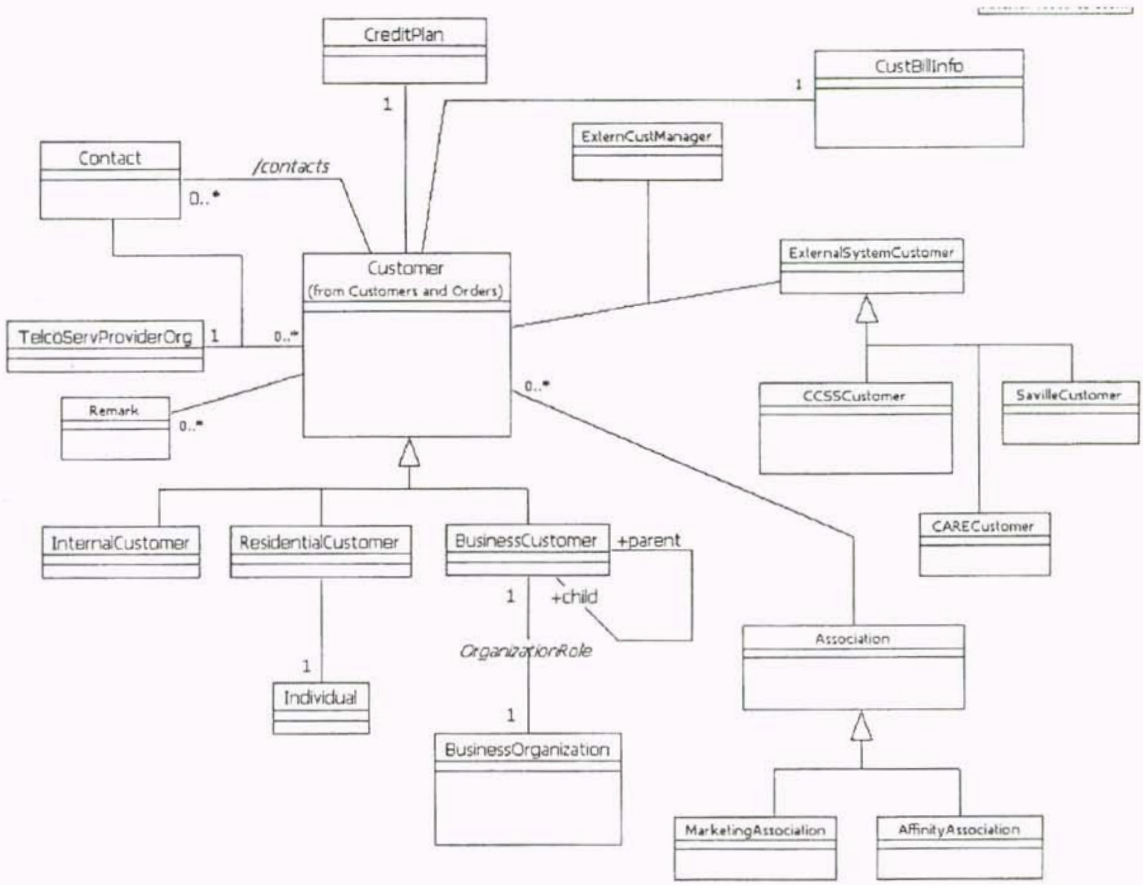


Figure 7.15: The Customer World

The diagram presented in Figure 7.15 completes the picture surrounding the customer, beyond its association with orders. This diagram shows details not covered in the earlier diagrams which discussed only the essential concepts.

A customer does business with a single service provider organization, usually an affiliate or a market office. A single point of contact, usually the Customer Care Rep, manages the relationship between the customer and the service provider.

The individual customer class is an association between customer and individual that can assume many other roles than simply the customer role. The same is true for the business customer, which is an association between business organization and customer. Note that business customer can nest in an organizational pattern.

7.3.10 Simplified Telco Organization Structure

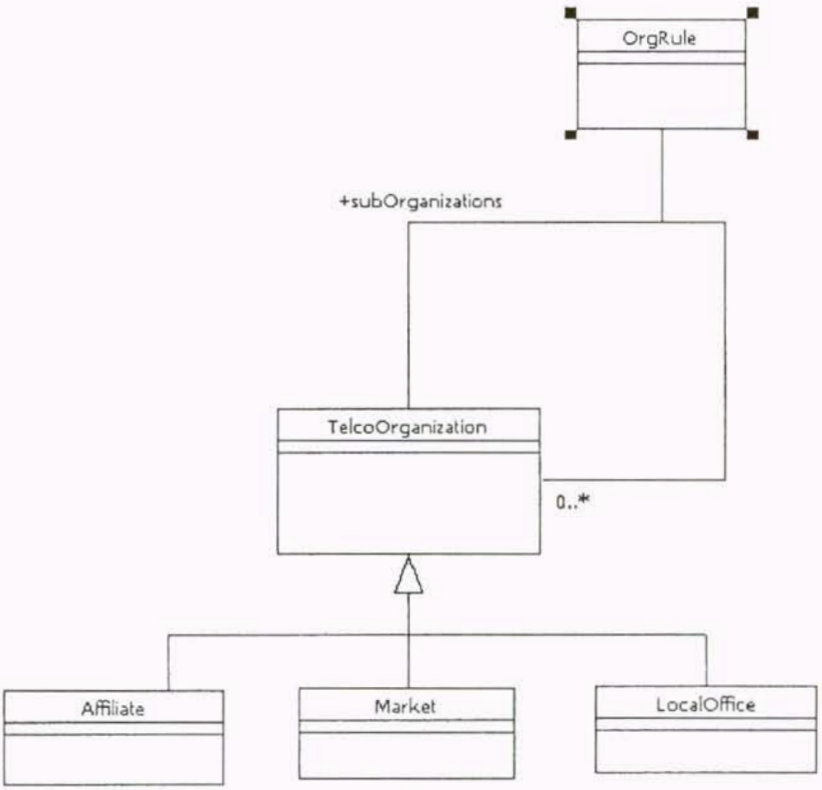


Figure 7.16: Simplified Telco Organization Structure

Figure 7.16 provides a simplified organizational structure for a typical telecom service provider. A service provider is composed of affiliates, which are semi-autonomous

operating organizations with their own presidents, staff, local policies, and business activity flow.

Affiliates often are divided into markets, which are also called branch offices. Markets also have their own staff and management but report their business statistics (number of orders and lines connected) to the affiliate.

The structure presented in Figure 7.16 is very flexible and represents the fact that new companies are constantly entering the telecom market space. Young companies do not have solidified organizational lines and need to be modeled with a structure that can facilitate change.

In order to control this structure and not allow, for example, markets to certain affiliates, an organization rule is needed. This class is needed to enforce the company's organization rules.

7.3.11 Telco Organization in Detail

Figure 7.17 illustrates the details of a telecom service provider organization and the things that are closely related to them. A structure like the one shown in Figure 7.17 will help manage the future as the organization grows and discovers new organizational relationships. Task management capabilities are assigned to the top-level organization.

In this structure, the Telco organizations can contain other Telco organizations, without any predefined labeling hierarchy for the subdivision or containment. The affiliates and their branches actually provide service to customers. Consequently, affiliates do business with customers and LECS and have local business practices, often enforced by the processes of the local BELL operating companies that they must interact

with. The column of classes on the right side on the model shows the different unique collections that the affiliates maintain (this list is not complete).

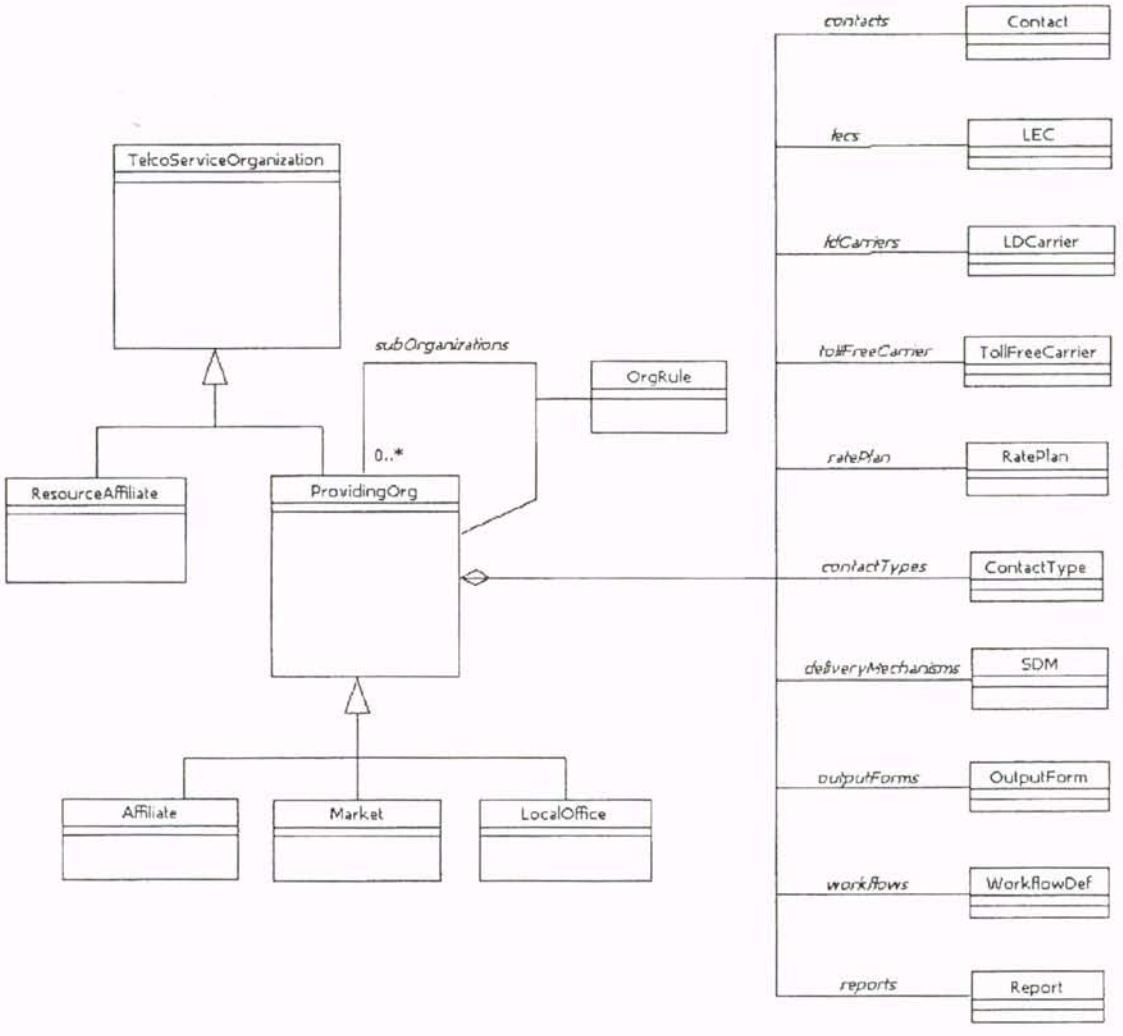


Figure 7.17: Detail Telco Organization Structure

7.3.12 Instances of Business Activity Flows

Figure 7.18 illustrates the basic process and activity flows as it applies to a typical Telco service provider. Each affiliate organization has a collection of business process definitions (ProcessDef). The individual steps of a process definition are called activities

and are defined by activity definitions (ActivityDef). A process definition may contain many activity definitions along with the relationships between them, which can be complex. There are different process definitions for different order operations. For example, starting a new service for a customer is different from disconnecting service and so has a different process definition.

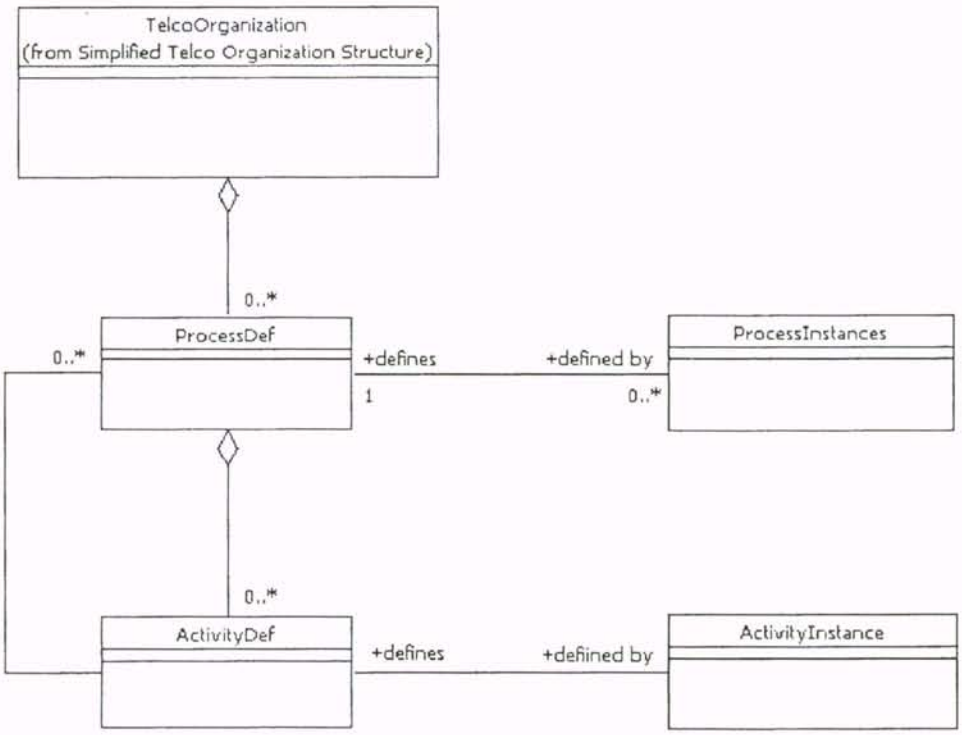


Figure 7.18: Instances of Business Activity Flows

Process instances are the individual, currently running processes defined by process definitions. Likewise, activity definitions serve as templates for individual, currently executing activity instances. An alternative term for activity instance is *task*.

7.3.13 Orders in the Flow of Business Activity

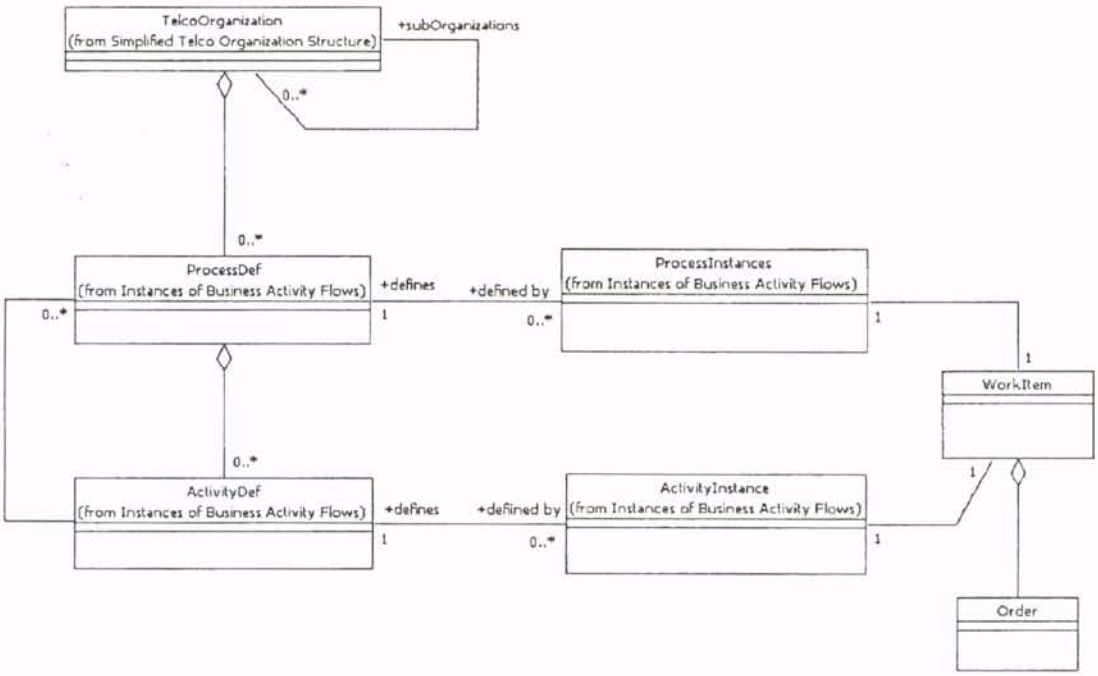


Figure 7.19: Orders in the Flow of Business Activities

Figure 7.19 presents the relationship of a typical business activity flow and an order. Service activation is accomplished through a sequence of business activities. The individual business activities are represented in the diagram as a work item. The work item is the basic thing that is passed from step to step as work proceeds. The work item can accumulate many attachments – related documents and artifacts – as work proceeds. The premiere work item attachment is the order.

Every instance of a process is related to one and only one order, via the work item. (Orders can relate to other orders, which can be somewhere in the execution of a business process themselves.) Likewise, every activity instance pertains to one and only

one order. The line between an activity instance and an order is probably a derived association, from the fact that an activity is related to a process.

7.3.14 Worklists

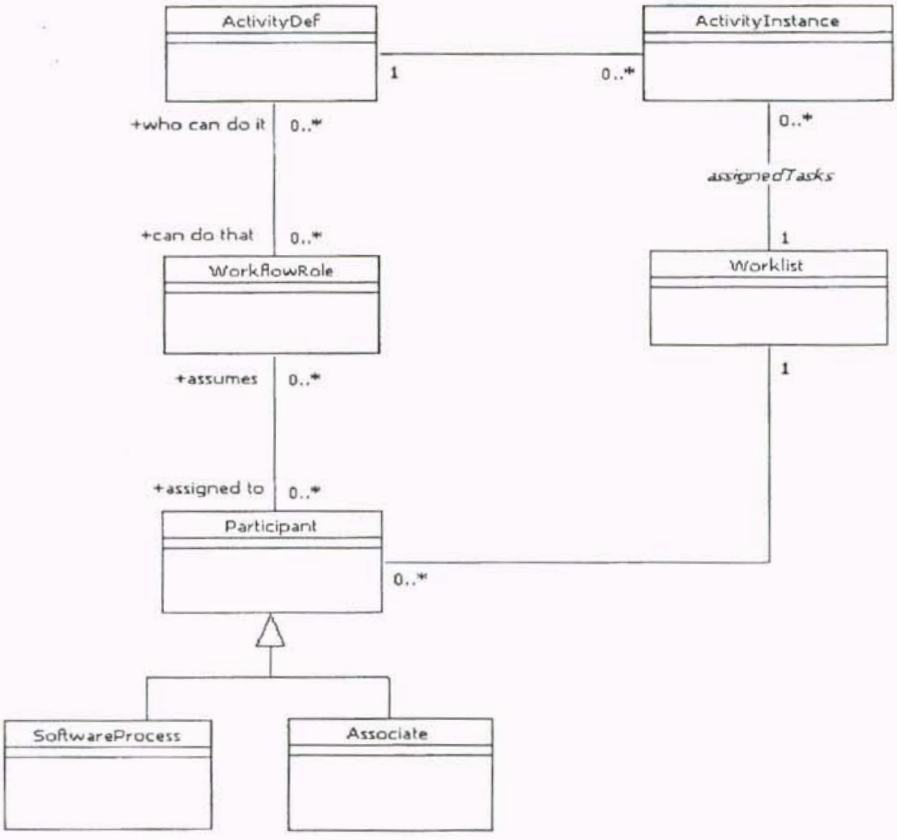


Figure 7.20: Worklists

The diagram in Figure 7.20 introduces the concepts of roles and Worklists. A business activity role is a kind of participant who is capable or authorized to perform a certain kind of activity. A participant can be many different kinds of entities, including but not limited to software processes and human employees (associates). The most

common participant is an associate. A participant is defined as some person or thing that is assigned to a business activity role.

Whereas the business activity role describes who is authorized to do what kind of activity, the worklist defines who is actually assigned to do a specific task in real time. All assigned tasks for all of associates' roles appear on their worklists. An assigned task is one which has been assigned to a participant but has not been completed (it may not have been started). If one associate acts as both a salesperson and as a customer care rep, all assigned salesperson and customer care rep tasks will appear in that person's worklist.

7.3.15 Combined Business Activity Flow

Figure 7.21 presents a comprehensive view on typical business activity flow. There is a slight modeling twist. Here, business activity role and worklist are seen as association classes between the participants and their activity definitions and activity instances. The sense of the previous diagram is retained.

As an association class, the business activity role manages the assignment of participants to kinds of activities. It keeps the list of who can do what activity. Likewise, the worklist maintains the list of who is currently assigned to what task in real time.

One point that is not shown in Figure 7.26 is that the business process can branch conditionally. In most cases, the data to be tested in the branch decision is in the order.

The business activity flow is usually modeled using the UML activity or state diagrams.

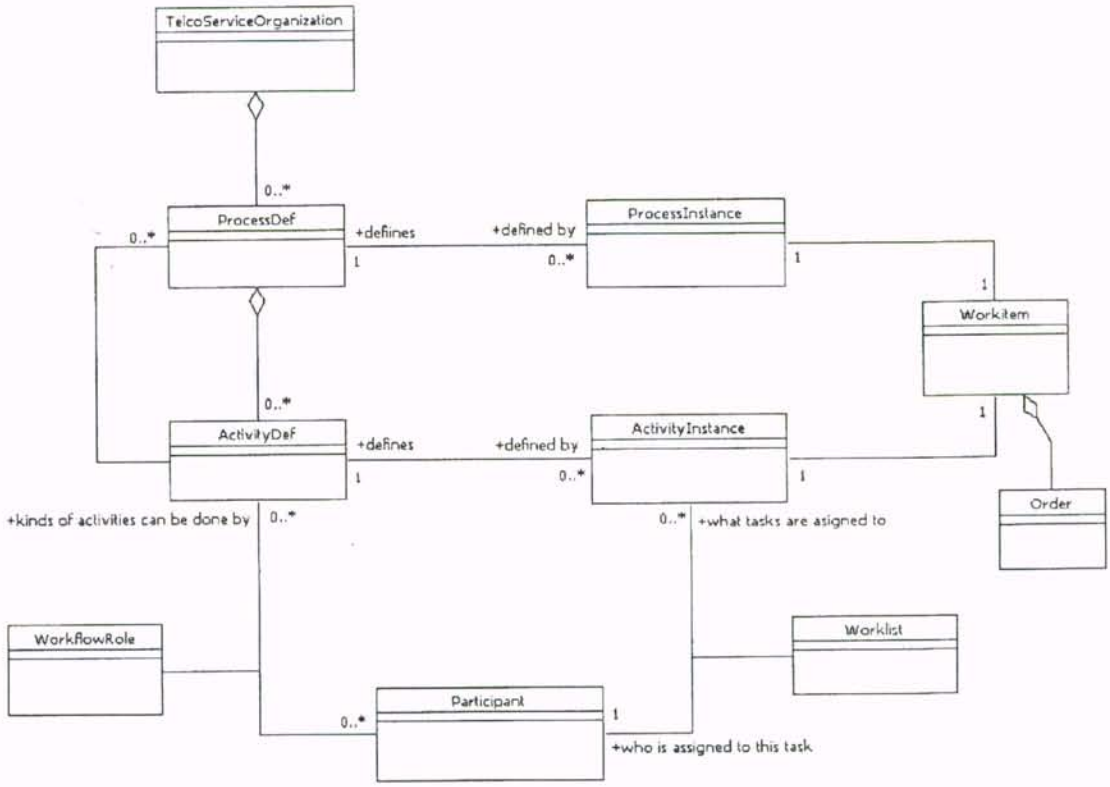


Figure 7.21: Combined Business Activity Flow

7.4 Example: Get Customer Data for Viewing

This section provides details pertaining to the retrieval and presentation of the customer data for viewing. The customer data resides in the customer relationship manager (CRM) application and the billing application. Figure 7.22 provides some details of the customer business object.

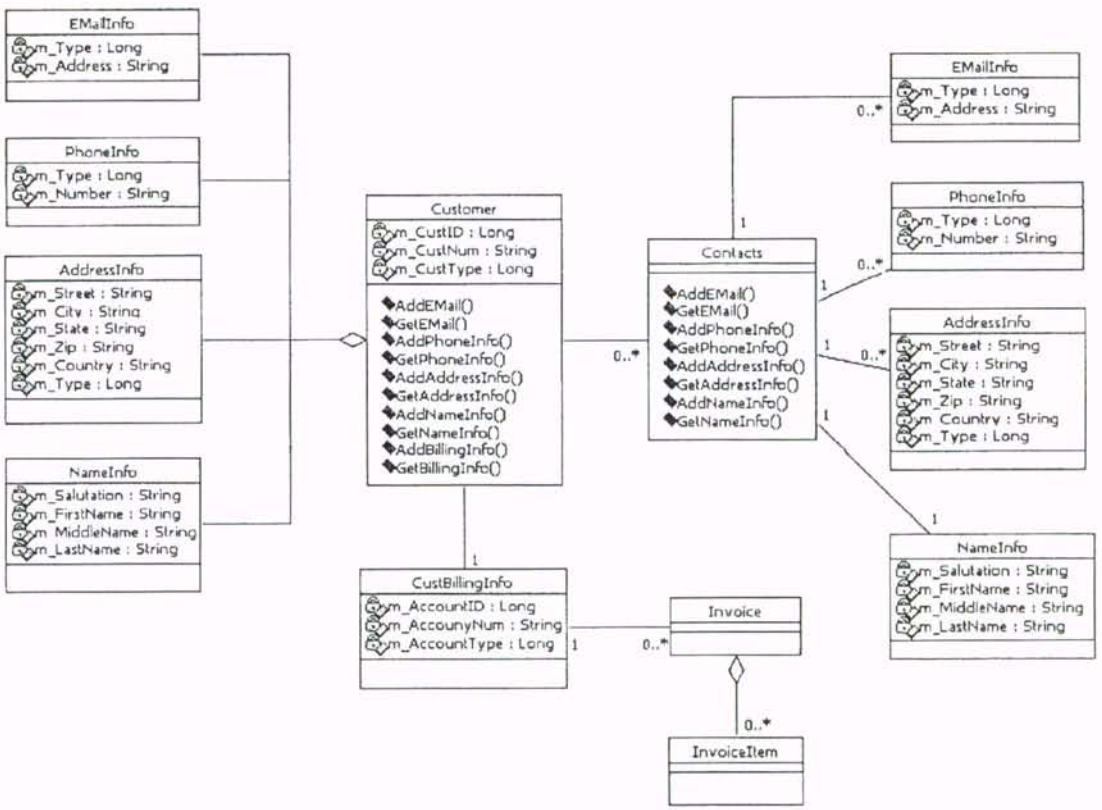


Figure 7.22: Customer Business Object

The contact information resides in the Siebel CRM application. Figures 7.23 and 7.24 show the entities and associated state diagram for retrieving the customer contact information from the CRM application.

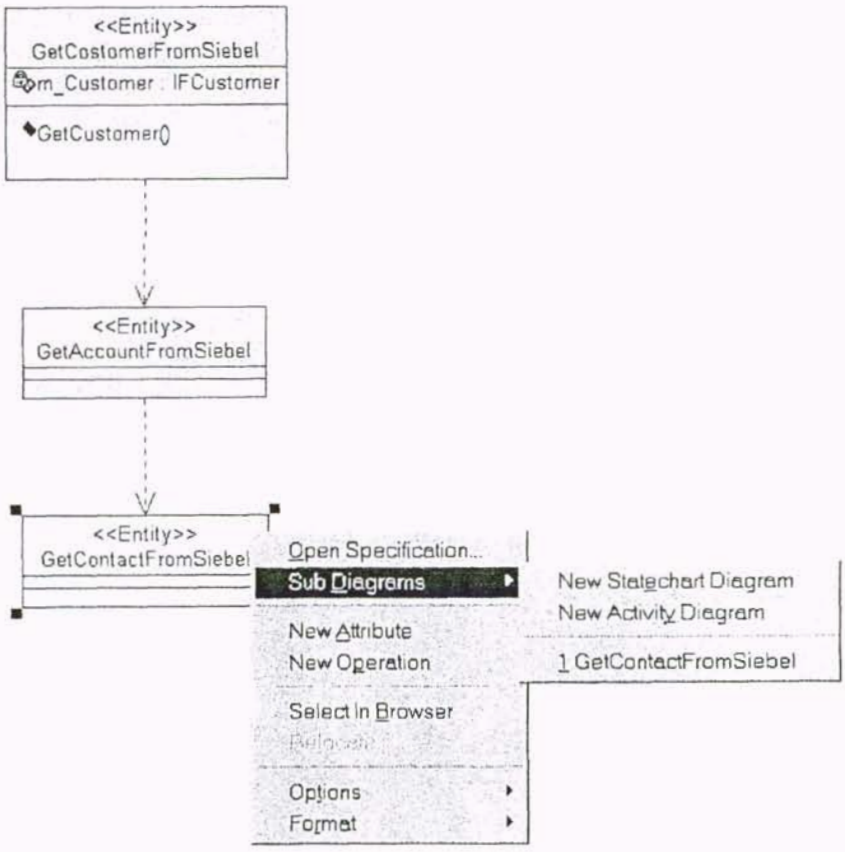


Figure 7.23: Get Customer from CRM Operation

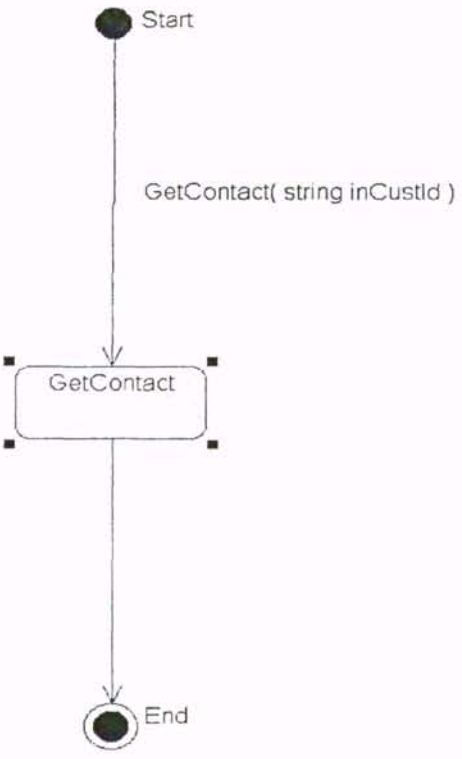


Figure 7.24: Get Contact Information from CRM State Diagram

Figure 7.25 shows the state specification for the GetContact state operation. The specification uses a high-level action semantic like language for defining the operation. This layer also inherently specifies the mapping and translation between the application domain model exposed via the domain adapter and the enterprise business objects.

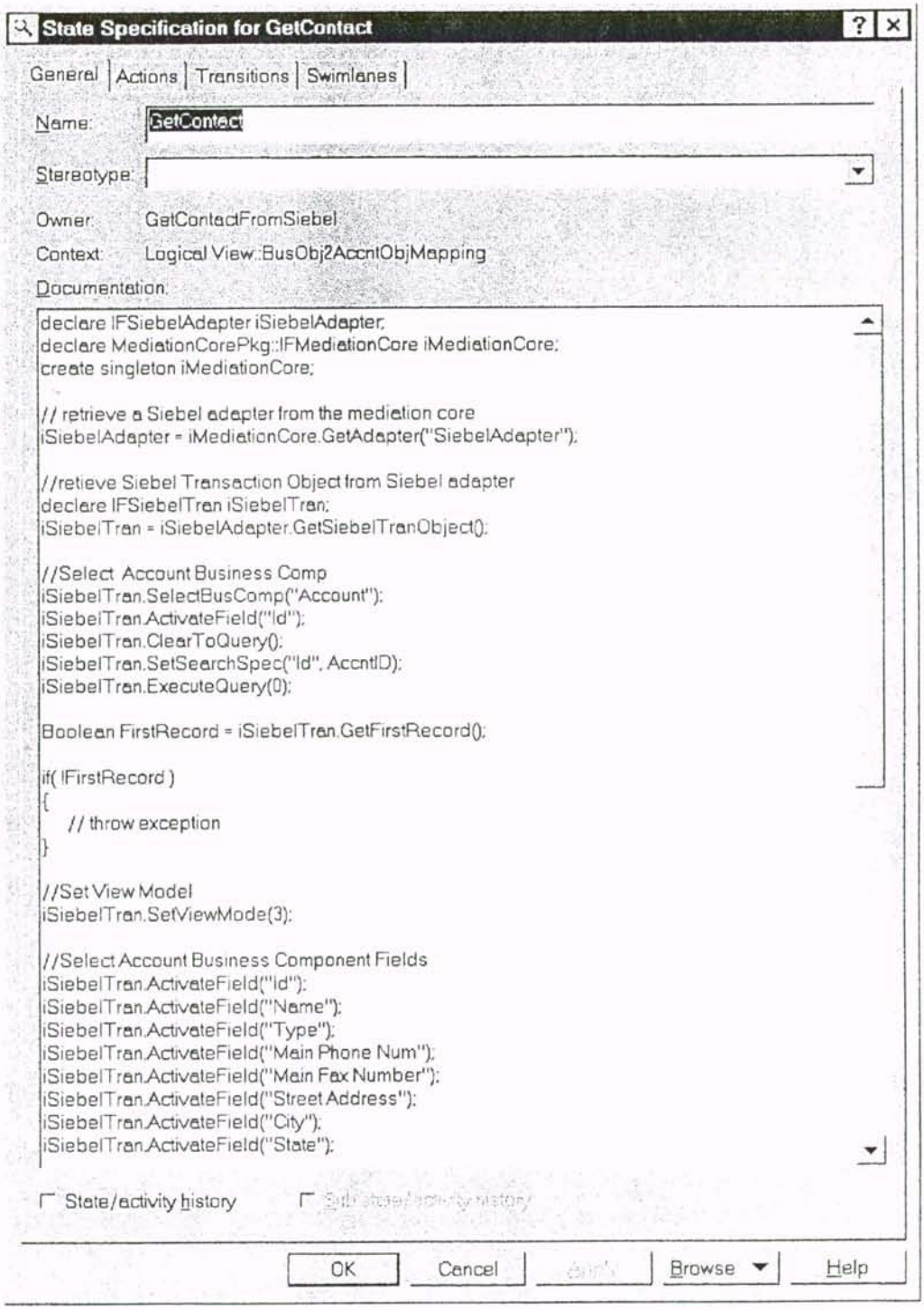


Figure 7.25: Get Contact Action Semantic Language

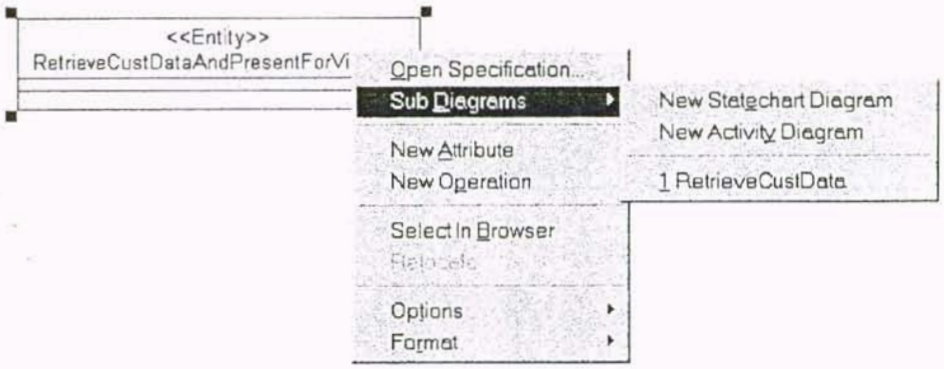


Figure 7.26: Retrieve Customer Data for Viewing Operation

Figures 7.26 and 7.27 capture the overall business process for retrieval and presentation of the customer data to the viewing application. Figure 7.27 is a typical activity diagram with swimlanes. Each swimlane specifies operations for a domain silo operation. The horizontal lines indicate synchronization points.

7.5 Summary

The proposed adaptive architecture approach and techniques have been used to develop a natural solution for the telecom OSS integration problems. Telecom OSS integration is at the extreme end of the spectrum of application integration problems. Using our approach to address this problem is a testament to its capability. The generic nature of the proposed adaptive architecture approach makes it applicable to any customer centric business software application that is transactional by nature. Thus, the approach can be used to

address integration concerns in any enterprise that has a need for application portfolio interoperability.

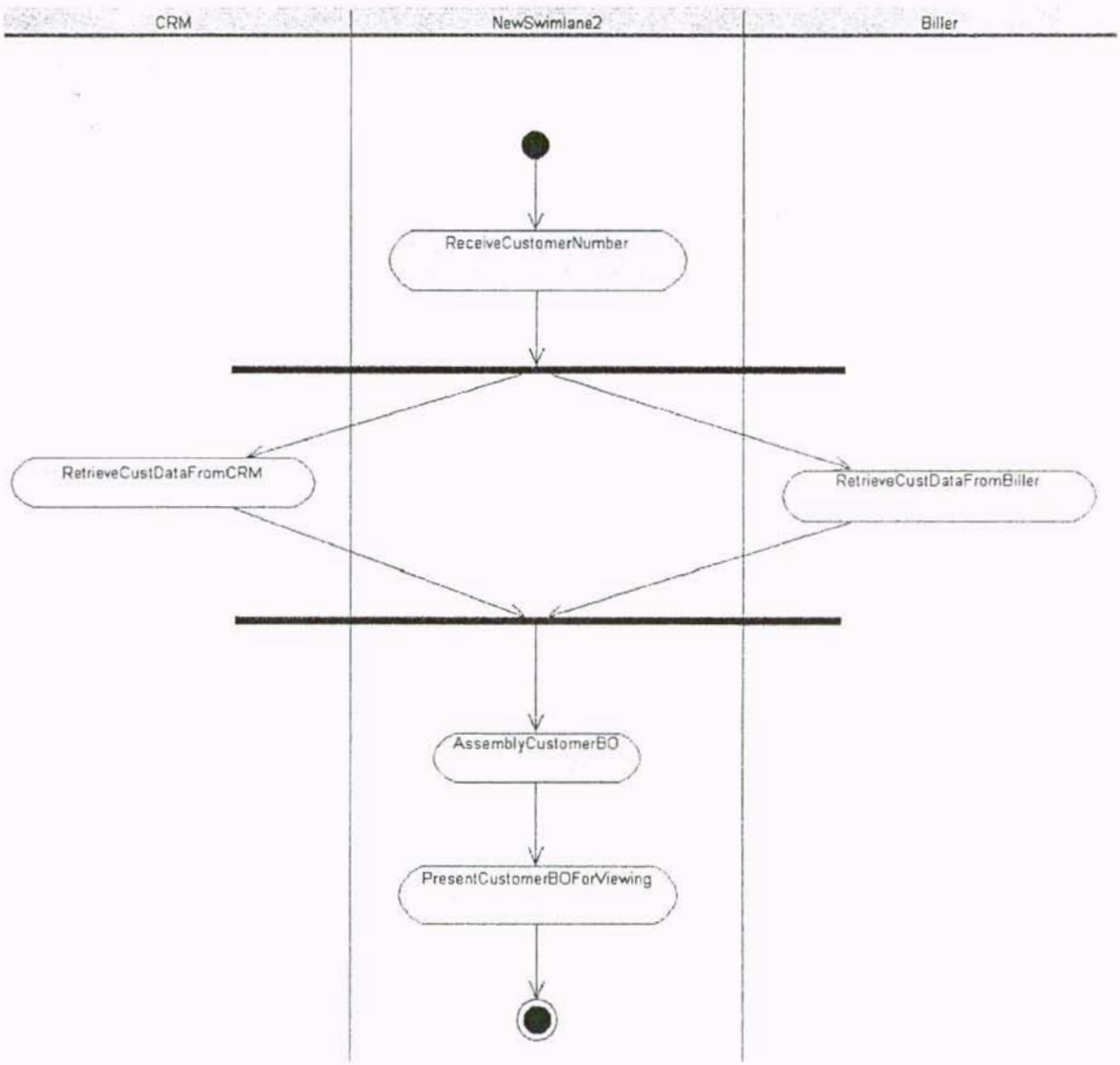


Figure 7.27: Retrieve Customer Data Activity Diagram

Chapter 8

UML Model-Based Component Development Framework

Dramatic and rapid changes in the computer industry make it impossible for application developers to stay current with technological advances. Developers are expected not only to create the applications solutions, but also to design the recovery, scaling, distribution, and other infrastructure services needed to support the mission critical business solutions of today's enterprises. This is an unrealistic expectation and results in the software application landscape being littered with failed projects.

To address this mismatch in expectations between what is currently achievable and what the business enterprises desire, we put forward the concept of model based software construction. This is explained in the following sections.

8.1 Model-Based Software Construction

The objective of the Model-Based Component Development Framework is to isolate core application logic specifications from infrastructure services that the software components will use. This will enable developers to create complex, robust, operation-critical software solutions without embedding infrastructure services into the core application logic. However, this is only the first step. What we really want is to have programming language, infrastructure services, and execution environment neutrally in the

specification of software components. That is, what we would like is to be able to specify the component object model, complete with behavioral specification, all in a meta-formalism such as some extended version of UML (EUML) [OMG 2000b] or some other Universal Design Language (UDL) [OMG 2000]. Figure 8.1 gives a schematic representation of a model-based component development framework.

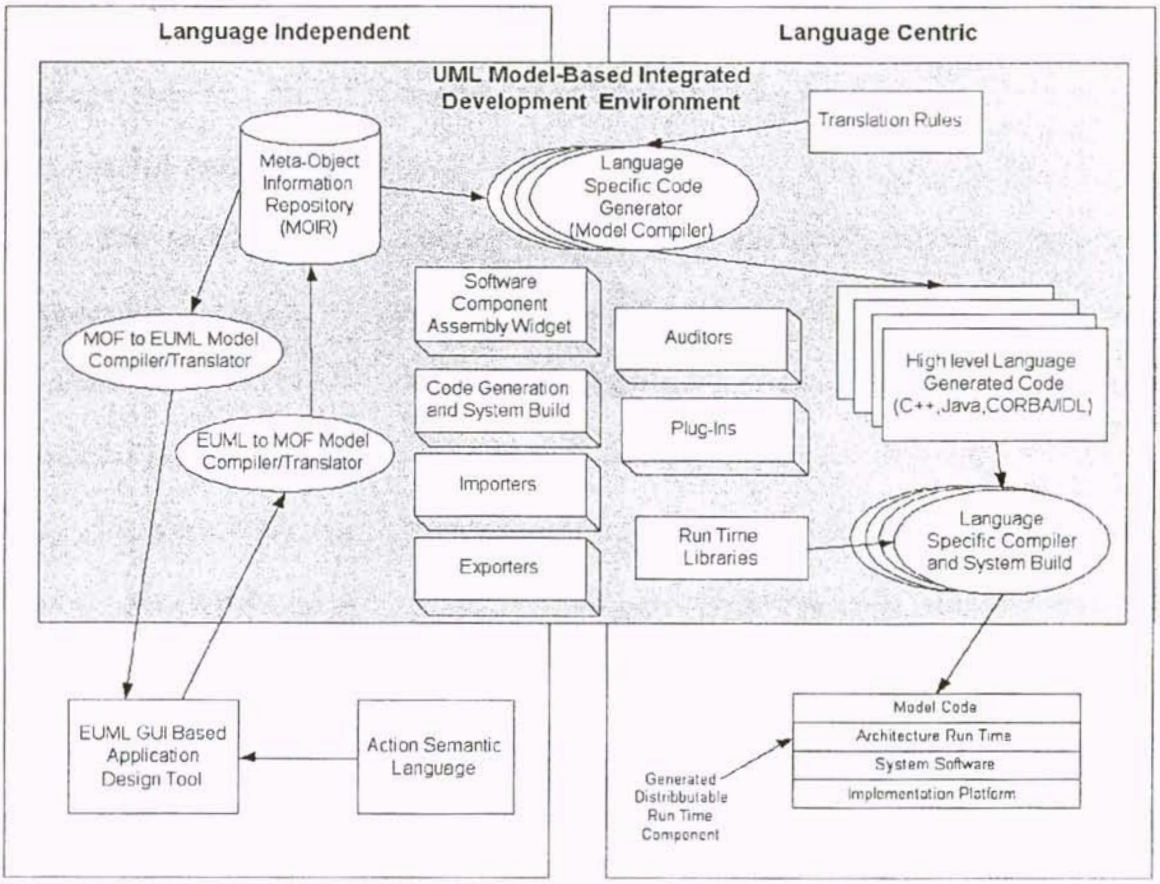


Figure 8.1: Model-Based Component Development Framework

CORBA architecture framework goes a long way to accomplish some of these goals [OMG 1997; OMG 2001]. With such an approach we could provide the complete

specification of a software component (object model) using a meta specification formalism that is independent of any imperative high level programming language specificity, independent of any execution environment, and independent of the runtime infrastructure services that it will be using. Such components would lend themselves to significant reusability since they could be looked at as higher-level abstract design artifacts. An Integrated Development Environment (IDE) can be used during program creation time to translate the meta model into a specified high level programming language equivalent specification and link in all the runtime infrastructure services that the generated component will use.

The application developer would thus work exclusively within a high-level programming language independent meta-specification to determine the component functionality. Infrastructure services and runtime binding can be specified in the IDE and automatically generated into the resulting executable component.

Implementation induces programming language, infrastructure services, and runtime environment specificity. If these issues are taken into consideration and addressed during the application design then they will ultimately impose limits on the reusability of the resulting software components. The only reasonable way to address these issues is to have complete separation of the application business functionality from imperative programming language, infrastructure services, and runtime environment specificity. This requirement mandates that the application must be specified using a meta-object formalism from which the resulting deployable software component can be generated.

This approach provides an effective mechanism for raising the level of abstraction at which an application developer works. The application developer effectively works within a graphical environment using some extended form of the UML meta-metamodel formalism derived from the Meta Object Facility (MOF) [OMG 2000].

A great deal of the efforts, maybe up to 80%, expended during the traditional software development process, goes into the development of the application infrastructure. This may be even more for highly distributed applications. Only about 20% of the effort goes into the design of the application logic. If we can change this process to be one in which the application developer specifies, using a metamodel, the infrastructure service he/she wishes to use and the manner of use, and then allow the integrated development environment to generate the specified software component and associate it with all the infrastructure services it requires, we could have the developer working at a higher level of abstraction. This approach will fundamentally change the software development process to be that of "model and generate" as opposed to "model and code". Model based software construction will be the new paradigm in which we develop software systems.

The Integrated Development Environment should let the developer perform the following functions:

- Load a component metamodel specification into the Meta-Object Information Repository.
- Specify target implementations (colorings), such as database type, caching, CORBA Services, etc., without corrupting the business processes defined in the model.
- Audit models to verify correctness.

- Generate server components from the models.

Raising the abstraction level of the software developer should result in a number of tangible advantages that we should be able to associate with metrics. These include the following:

- Faster time to market for new products and services being offered by the enterprise.
- Since applications would be define using a high-level meta-metamodel, the enterprise needs to recruit technocrats who are skilled using this technology to define the enterprise business model in the relevant domains. People who are skilled in middleware technologies such as CORBA and high-level programming languages such as C++ and Java would not be required to develop high performance business applications. The enterprise would put greater emphasis on employing business and domain analysts.
- The enterprise could have business analysts performing the majority of the application enhancement and refinement during the maintenance cycle. Since this is the phase during which most of the cost for an application is incurred, the enterprise should be able to significantly reduce its application maintenance expenditure.
- Since infrastructure services such as externalization, transactionality, concurrency, persistence, and synchronization can be specified in a high level meta formalism and language and environment specificity associated with these services generated during the build process, there is a distinct possibility that we could reverse the effort role. The developer or modeler could now spent 80% or more of his or her time designing the application functionality as opposed to programming infrastructure services in some high level language.

- Applications developed along this model will be high-level programming language and runtime environment independent allowing for easy migration and interoperability across different computing platforms.

Some of the major components of such an integrated development environment include the following:

- A UML based modeling tool
- UML Extended with Action Semantic Language
- Meta-Object Information Repository
- UML based Model Compiler/Translator
- Language specific code generators
- Importers
- Exporters
- Auditors
- Plug-Ins

8.2 Meta-Object Information Repository

In their efforts to advance the development of distributed software systems, the Object Management Group (OMG) proposed two standard specifications for modeling distributed software architecture and systems [OMG 2000; OMG 2000b] that are consistent with the CORBA Object Management Architecture (OMA). The two complementary specifications are as follows:

- Unified Modeling Language Specification
- Meta-Object Facility Specification

The Unified Modeling Language (UML) Specification defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems. The specification includes the formal definition of a common Object Analysis and Design (OA&D) metamodel, a graphical notation, and a CORBA IDL facility that supports model interchange between OA&D tools and metadata repositories. The UML provides the foundation for specifying and sharing CORBA-based distributed object models.

The Meta-Object Facility (MOF) Specification defines a set of CORBA IDL interfaces that can be used to define and manipulate a set of interoperable metamodels and their corresponding models. These interoperable metamodels include the UML metamodel, the MOF meta-metamodel, as well as future OMG adopted technologies that will be specified using metamodels. The MOF provides the infrastructure for implementing CORBA-based design and reuse repositories. The MOF specifies precise mapping rules that enable the CORBA interfaces for metamodels to be automatically generated, thus encouraging consistency in manipulating metadata in all phases of the distributed application development cycle.

The MOF and OA&D metamodels are architecturally aligned to use the MOF IDL mapping for generating CORBA IDL for both the MOF and UML models. This was accomplished by defining the MOF and UML models using the MOF and by generating the IDL interfaces based on the MOF specification. Alignment of UML, MOF, and CORBA paves the way for future extensibility of CORBA in key areas such as richer semantics, relationships, and constraints. Likewise the longer-term benefits to UML and

MOF include better recognition and addressing of distributed computing issues in developing CORBA-compliant systems.

The extension of UML with Action Semantic Language (ASL) behavioral specification can be accomplished by extending the UML meta-metamodel with the new ASL constructs using the MOF. The Meta-Object Information Repository (MIR) service would be an implementation of the extended UML MOF metamodel interfaces. This service would be accompanied by tools (e.g., compilers or graphical editors) that allow the designer to input information models using a human readable notation for the MOF model.

Chapter 9

A Mathematical Formalism for Specifying Design Patterns

Within the context of software engineering the nomenclature pertaining to object-oriented methodologies depicts an *object* as an instance of a *class*. An object is a self-contained entity that is complete with its sets of data and associated operations. A class is a specification of an abstract data type. Many-sorted algebra is the mathematical formalism used for the specification of abstract data types and represents a straightforward generalization of classical (i.e., single-sorted) algebras [Loeckx 1996].

In this chapter we present a mathematical formalism for the specification of design patterns. This specification constitutes an extension of the basic concepts from many-sorted algebra. In particular, the notion of signature is extended to that of a vector, consisting of a set of linearly independent signatures. The linearly independence property is necessary to satisfy non-interference that is essential for compositional based construction [Cowan 1993a; Loeckx 1996; Enderton 1972]. This is of fundamental concern in the building of large-scale software systems where we have the composition of smaller components to form larger components. In what follows, the major concepts for the specification of design patterns are developed using successive extensions of the relevant many-sorted algebraic concepts.

9.1 Definitions and Concepts

This section outlines the definitions and concepts relevant to the formal specification of the design patterns.

Signatures

A many-sorted algebra consists of sets and functions. A signature may be viewed as the syntax of an algebra for fixing the names of the sorts and functions.

Definition 1: Signature

A *signature* Σ is a pair $\Sigma = (S, F)$ of sets, the elements of which are called *sorts* and *operations* respectively. Each operation consists of a (k+2)-tuple

$$n = s_1 \times s_2 \times \ldots \times s_k \rightarrow s$$

with $s_1, s_2, \ldots, s_k, s \in S$ and $k \geq 0$; n is called the *operation name* of the operation and $s_1, s_2, \ldots, s_k \rightarrow s$ its arity; the sorts $s_1, s_2, \ldots, s_k$ are called argument sorts of the operation and the sort s its target sort. In the case $k = 0$ the operation $n: \rightarrow s$ is called a constant of sort s.

Informally, a sort denotes (i.e., is a name of) a type and an operation denotes a function. Note that different operations may have the same operation name. In particular, the equality of two operations implies the equality of their names and the equality of their arities. One may write n instead of $(n : s_1 \times \ldots \times s_k \rightarrow s)$ if no ambiguities arise.

The operations and relations defined on sets are lifted to signatures by applying them componentwise. For example, if $\Sigma = (S, F)$ and $\Sigma' = (S', F')$ are signatures, $\Sigma \subseteq \Sigma'$ stands for $S \subseteq S'$ and $F \subseteq F'$; similarly, $\Sigma \cup \Sigma'$ stands for $(S \cup S', F \cup F')$. □

Definition 2: Vector Signature, extension of signature

A *vector signature* Σ_v is an n-tuple of linearly independent (Σ-) signatures represented as $\Sigma_v = (\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$, such that $\Sigma_i = (S_i, F_i)$ and

S_i represents the set of sorts for Σ_i

F_i represents the set of operations for Σ_i

The linearly independent property states that

$$S_i \cap S_j = \varnothing \text{ for } i \neq j \text{ and } F_i \cap F_j = \varnothing.$$

An operation of Σ_v is a vector. For example the vector w is defined as $w = (w_1, w_2, \ldots, w_n)$ such that $w_i \in F_i$ with

$$w_i = S_1^j \times S_2^j \times \ldots \times S_k^j \rightarrow S^j$$

with

$$S_1^j, S_2^j, \ldots, S_k^j, S^j \in S_j \text{ and } k \geq 0 \text{ and } 1 \leq i, j \leq n. \qquad \square$$

Algebras

A many-sorted algebra assigns a meaning to a signature by associating a set of data to each sort and a function to each operation.

Definition 3: Total Algebra (Algebra)

Let $\Sigma = (S, F)$ be a signature. A total algebra for Σ assigns the following:

1. A set $A(s)$ to each sort $s \in S$, called a carrier set of the sort s; the elements of a carrier set are called carriers;
2. A total function

$$A(n : s_1 \times \ldots \times s_k \rightarrow s) : A(s_1) \times \ldots \times A(s_k) \rightarrow A(s)$$

to each operation $(n : s_1 \times \ldots \times s_k \rightarrow s) \in F, k \geq 0$; when $k = 0$, $A(n:\rightarrow s)$ denotes an element of the carrier set $A(s)$. □

Definition 4: Vector Algebra, extension of algebra

Let $\Sigma_v = (\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$ be a vector signature. A total vector algebra for Σ_v (Σ_v–algebra) is a vector of Σ-algebras. A Σ_v-algebra A is depicted as the vector $A = (A_1, A_2, \ldots, A_n)$ where $A_{i=1..n}$ are Σ_v-algebras corresponding to $\Sigma_i = (S_i, F_i)$ and $1 \leq i \leq n$. A total vector algebra (Σ_v-algebra) assigns the following:

1. A set $A(\Sigma_i)$ to each algebra A_i; $Alg(\Sigma_i)$ is the class of Σ-algebra corresponding to Σ_i.

$$A(\Sigma_i) = \bigcup A(s_j) \text{ with } s_j \in S_i.$$

2. A total function A_v that is the union of the total functions of the Σ-algebra corresponding to the individual Σ_i in the vector signature $\Sigma_v = (\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$. A_v is represented as

$$A_v(w) = \bigcup A(s_j)$$

$$\equiv \bigcup A_i(n_i : S_1^i \times S_2^i \times \ldots \times S_k^i \rightarrow S^i) : A_i(S_1^i) \times A_i(S_2^i) \times \ldots \times A_i(S_k^i) \rightarrow A_i(S^i)$$

such that each operation $n_i \in F_i$ is covered. □

Homomorphism

Homomorhpisms constitute mappings between the carrier sets of algebras that *respect* the functions.

Definition 5: Homomorphism

Let A, B be two Σ-algebras, $\Sigma=(S, F)$. A Σ-homomorphism h: $A \rightarrow B$ from A to B is a family $h = (h_s)_{s \in S}$ of functions

$$h_s : A(s) \rightarrow B(s) \equiv h_s : A_s \rightarrow B_s$$

such that for any operation $w \in F$, say $w = (n : s_1 \times s_2 \times \ldots \times s_k \rightarrow s)$, $k \geq 0$, the following condition holds:

$$h_s(A(w)(a_1, a_2, \ldots, a_k)) = B(w)(h_{s_1}(a_1), \ldots, h_{s_k}(a_k)) \qquad (1)$$

for all $(a_1, \ldots, a_k) \in A(s_1) \times \ldots \times A(s_k)$. The above equation, equation (1), is called the homomorphism condition of the homomorphism h for the operation w. Note that in the case $k = 0$ the condition simply states:

$$h_s(A(w)) = B(w)$$ □

Figure 9.1 shows the commuting diagram that graphically illustrates the homomorphism condition of the homomorphism $h : A \rightarrow B$ for the operation $w = (n : s_1 \times \ldots \times s_k \rightarrow s),\ k \geq 0.$

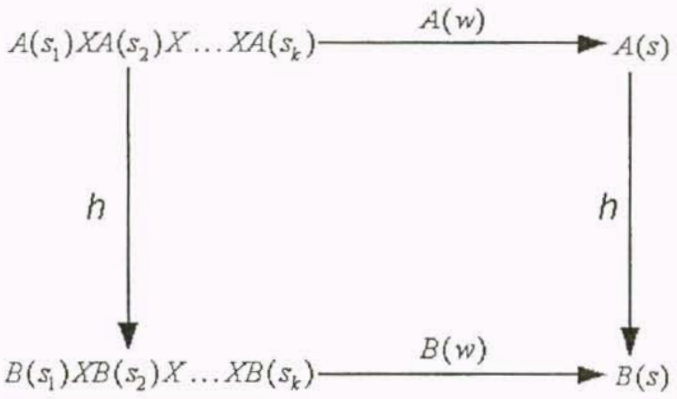


Figure 9.1: Commuting Diagram illustrating the homomorphism condition of the homomorphism $h: A \rightarrow B$ for the operation $w = (n : s_1 \times \ldots \times s_k \rightarrow s),\ k \geq 0.$

Vector homomorhpisms constitute mappings between the carrier sets of vector algebras. The mappings *respect* the functions of the corresponding vector signature.

Definition 6: Vector Homomorphism, extension of homomorphism

Let $A = (A_1, A_2, \ldots, A_n)$, $B = (B_1, B_2, \ldots, B_n)$ be two Σ_v–algebras, $\Sigma_v = (\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$.

A Σ_v–homomorhpism H: $A \rightarrow B$ is a vector of Σ–homomorhpisms represented as

$$H = (h_{s^i})_{i=1..n} = (h_{s^1}, h_{s^2}, \ldots, h_{s^n})$$

where h_i is a homomorphism from A_i to B_i over Σ_i given by the mapping h_i: $A_i \rightarrow B_i$.

For any operation $w \in \Sigma_v = (\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$, say $w=(w_1, w_2, \ldots, w_n)$, such that $w_i \in F_i$ with

$w_i = S_1^i \times S_2^i \times \ldots \times S_k^i \rightarrow S^i$ and $S_1^i, S_2^i, \ldots, S_k^i, S^i \in S_i$ and $\Sigma_i = (S_i, F_i)$

and $k \geq 0$ and $1 \leq i \leq n$ the following conditions hold:

$$H(A)(w)(\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_n) = B(w)(H(\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_n))$$

$$= B(w)(H(\overline{a}_1), H(\overline{a}_2), \ldots, H(\overline{a}_n)) \qquad (2)$$

where $\overline{a}_i = (a_1, a_2, \ldots, a_k) \in S_1^i \times S_2^i \times \ldots \times S_k^i$ and $S_1^i, S_2^i, \ldots, S_k^i \in S_i$.

The linearly independent property mandates that the application of w over $\overline{a}_i$ must be done on a pair-wise basis. That is,

$$w(\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_k) = w_1(\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_k), w_2(\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_k), \ldots, w_n(\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_k)$$

$$= w_1(\overline{a}_1), w_2(\overline{a}_2), \ldots, w_n(\overline{a}_n)$$

where $w_i \times \overline{a}_j = \phi$ for $i \neq j$

Thus, we have,

$$H(A)(w)(\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_n) = B(w)(H(\overline{a}_1), H(\overline{a}_2), \ldots, H(\overline{a}_n))$$

$$\equiv H(A_1)(w_1)(\overline{a}_1), H(A_2)(w_2)(\overline{a}_2), \ldots, H(A_n)(w_n)(\overline{a}_n)$$

$$= B_1(w_1)(H(\overline{a}_1)), B_2(w_2)(H(\overline{a}_2)), \ldots, B_n(w_n)(H(\overline{a}_n)) \qquad (3)$$

Again, the linearly independent property mandates that the homomorphisms must be applied on a pair-wise component basis. Hence,

$$h_{s^1}(A_1)(w_1)(\overline{a}_1), h_{s^2}(A_2)(w_2)(\overline{a}_2), \ldots, h_{s^n}(A_n)(w_n)(\overline{a}_n)$$

$$= B_1(w_1)(h_{s^1}(\overline{a}_1)), B_2(w_2)(h_{s^2}(\overline{a}_2)), \ldots, B_n(w_n)(h_{s^n}(\overline{a}_n))$$

$$= B_i(w_i)(h_{s_1^1}(a_1), \ldots, h_{s_k^1}(a_k)), B_2(w_2)(h_{s_1^2}(a_1), \ldots, h_{s_k^2}(a_k)), \ldots, B_n(w_n)(h_{s_1^n}(a_1), \ldots, h_{s_k^n}(a_k))$$

$$\forall(a_1, a_2, \ldots, a_k) \in A(s_1^i) \times A(s_2^i) \times \ldots \times A(s_k^i)$$

The above equation, equation (3), is called the homomorphism condition for the vector homomorphism H for the operation w. In the case when $k = 0$, the condition simply states:

$$H(A_1)(w_1), H(A_2)(w_2), \ldots, H(A_n)(w_n) = B_1(w_1), B_2(w_2), \ldots, B_n(w_n)$$

$$\equiv h_{s^1}(A_1(w_i)), h_{s^2}(A_2(w_2)), \ldots, h_{s^n}(A_n(w_n)) = B_i(w_i), B_2(w_2), \ldots, B_n(w_n) \quad \square$$

Figure 9.2 shows the commuting diagram for the homomorphism condition for the vector homomorphism H for the operation w.

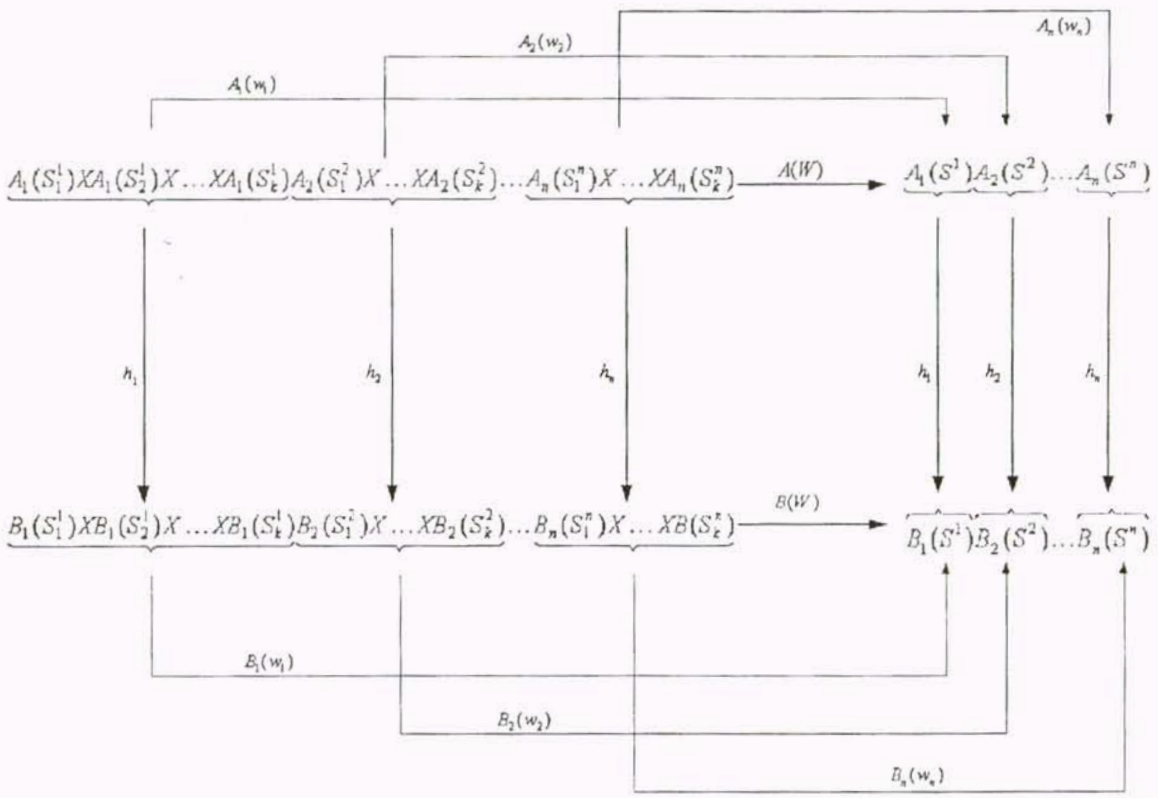


Figure 9.2: Commuting diagram illustrating the vector homomorphism condition

In our discussion of software components we have seen that the application of the principle of abstraction partitions a software component into a specification part and a realization part. The specification part corresponds to the interface of the software component. The Abstract Data View (ADV) concept further extends this notion. ADV corresponds to interfaces that are extensible. Hence, we can think of extending the functionality of a software module by extending the ADV interface. This approach

preserves encapsulation and enhances reusability by effectively applying the principle of composition to the existing module functionality.

Design Patterns

Informally, a design pattern or micro-architecture software artifact is an aggregate of abstract data types (ADTs). The class of objects corresponding to each of the ADTs is represented by a Σ–algebra. Assuming that the components are linearly independent and thus satisfy the non-interference proof obligation, then we can represent a design pattern as a vector of Σ–algebras. One of the major attributes of design patterns is that it captures knowledge from past experience. Thus, relationship between the component ADTs must be made explicit in any reasonable representation of design pattern. Hence, the vector of Σ–algebras is not sufficient to represent a design pattern.

One reasonable representation is to extend the n-tuple of Σ–algebras by including a *relation* that is capable of encoding the requisite knowledge. That is, the relation must be able to encode relationship, associations, roles, and multiplicity between entities in a design pattern. The relation depicted below is capable of encoding the requisite knowledge in a design pattern

$$R \subseteq L(Alpha) \times L(Alpha) \times A\lg(\Sigma_{v}) \times A\lg(\Sigma_{v}) \times L(Alpha) \times L(Alpha) \times Nat \times Nat$$

where

$L(Alpha)$: depicts a set of alphabetic strings representing the name of relationship between entities

$Alg(\Sigma_v)$: is the class of Σ–algebra corresponding to the vector signature Σ_v

Nat: is the set of natural number.

The components for the relation R are defined as follows: the first component of R depicts the name of the relationship or association between two entities. The second component depicts the type of relationship. The third and forth components depict the entities that the relation is defined between. Components five and six define the roles of the relationship. Components seven and eight define the multiplicity of the relationship.

More formally, a design pattern can be defined as an $(n+1)$-tuple containing n Σ–algebras and R. The n Σ–algebras corresponds to the various object entities in the design pattern while R is used to encode the relationships between the entities. Thus, the design pattern DP can be represented as $DP = (A_1, A_2, \ldots, A_n, R)$.

Example

The design pattern fragment given in Figure 9.3 shows a number of object entities and their associations. $DP1$ is an $(n+1)$-vector representation of the design pattern fragment.

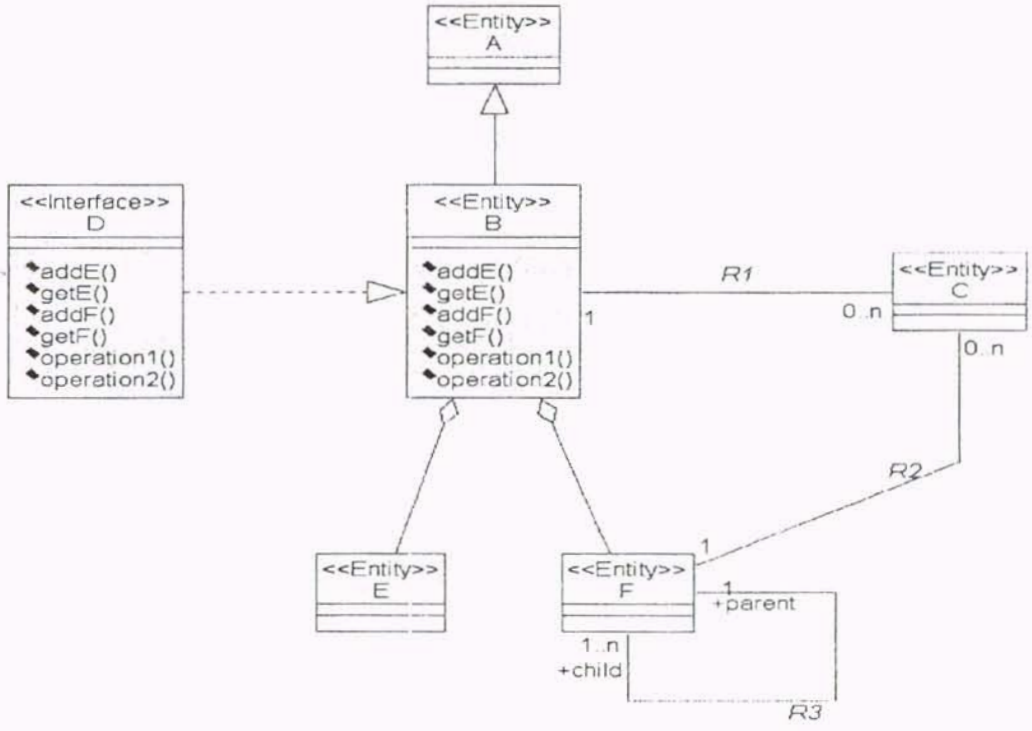


Figure 9.3: A Design Pattern Fragment

The relation R is enumerated by the following set of relationships:

R = { (nil, Aggregation, E, B, nil, nil, nil, nil),

(nil, Aggregation, F, B, nil, nil, nil, nil),

(R1, Association, B, C, nil, nil, 1, n),

(R2, Association, F, C, nil, nil, 1, n),

(R3, Association, F, F, parent, child, 1, 1..n),

(nil, Inheritance, A, B, nil, nil, nil, nil,),

(nil, Interface, D, B, nil, nil, nil, nil) }

Thus, the vector $DP1 = ((A, B, C, D, E, F), R)$ can be used to represent the design pattern fragment shown above.

Definition 7: Module Signature

A *module signature* is a pair (Σ_i, Σ_e) of signatures; Σ_i and Σ_e are called *import signature* and *export signature* respectively. A sort or operation from the signature Σ_i, Σ_e or $\Sigma_i \cap \Sigma_e$ is respectively called *imported*, *exported* or *inherited*. □

Figure 9.4 gives a graphical representation of the module signature (Σ_i, Σ_e) with Σ_i=({s, r}, {wi, w2}), Σ_e=({s},{w1, w3}). In this representation the inherited sorts and operations are shown by broken lines. Informally, the module signature fixes the signatures of the argument and of the value of a modularized abstract data type.

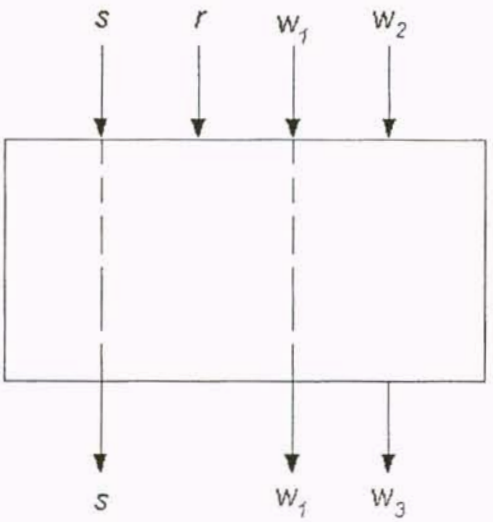


Figure 9.4: Graphical Representation of a Module Signature

It is now possible to introduce a formal notion of the *syntactic specification* for design pattern. Three dimensions of design pattern are characterized: the major classes forming the body or realization of the pattern, the interface or specification of the pattern, and the relationship between the classes in the body and interface of the pattern.

Definition 8: Module Vector Signature, extension of Module Signature

A *module vector signature* is a 3-tuple consisting of a pair of signatures and the knowledge relation discussed above. This is represented as follows:

$$(\Sigma_i^v, \Sigma_e^v, R^+)$$

where Σ_i^v represents the vector signature corresponding to abstract data types depicting the main classes in the body or realization of the design pattern. Σ_e^v represents the vector signature corresponding to the abstract data views (ADVs) depicting the main classes in the interface or specification of the design pattern. R is the relation that captures the inter-relationship between the classes in Σ_i^v and Σ_e^v. □

Definition 8a: Expanded Version of Module Vector Signature

A module vector signature is a 3-tuple consisting of three pairs. The first component of each pair is a vector signature representing the classes of the corresponding abstract data types and the second component represents the knowledge relation that define the actual binary relationship between the instances of the abstract data types and between abstract data types and binary relationships. This is represented as follows:

$$(\Sigma_i^v, \Sigma_e^v, R) \equiv ((\Sigma_i^v, R_1), (\Sigma_e^v, R_2), ((\Sigma_i^v \cup \Sigma_e^v), R_3)$$

where $R_2 \subseteq R_1^+$ and $R = \bigcup_{1 \le i \le 3} R_i$

and

$$R_1 \cong R^{\sim} = (Alg(\Sigma^{\nu}) \cup R_i^{\sim}) X Alg(\Sigma^{\nu}). \qquad \square$$

$R^{\sim}$ is defined to be a set of binary relationships between the entities in a design pattern. The binary relationships can be defined between abstract data types (or classes) in the design pattern or between an abstract data type and a binary relation that has been defined in the design pattern. As a result there are at least two kinds of binary relationships to be considered in the specification of design pattern: (1) The primitive binary relationship between abstract data types and (2) A higher order binary relationships between abstract data types and primitive binary relationships. Conceivably, this process of defining higher order relationship can be continued, defining tertiary and quaternary relationship similar to the concepts in the entity relationship (ER) model.

The schematic design pattern shown in Figure 9.5 shows a relationship in the specification that does not explicitly exist in the realization. The binary relationship between A and C, $(A,C) \in R$, in the specification is preserved via transitivity of A to B,$(A,B) \in R$, and B to C, $(B,C) \in R$, in the realization. The relationship between A and B is that of inheritance. Hence, B is a *type* of A. Therefore the set defining relationships between B and C can be extended to include relationships between C and A. This accounts for the inclusion of the transitive closure condition in the above definition.

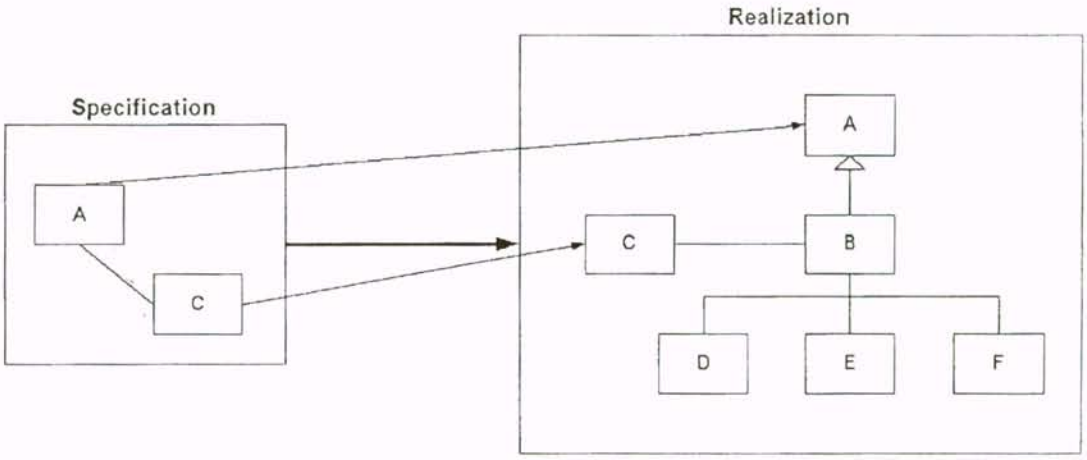


Figure 9.5: Schematic Representation of Design Patterns

Claim 1: Definition 8 is consistent

An important observation of definition 7 is that implicit in the definition is the fact that there is a structural relationship existing between the two signatures. This relationship is effectively the inheritance relationship. Therefore the basic module signature definition can be extended to explicitly include the inheritance relationship. This form is shown below:

$$(\Sigma_i, \Sigma_e) \equiv (\Sigma_i, \Sigma_e, I)$$

If the signatures in the vector export and import signatures are all empty except for one in each vector signature then the vector module signature reduces to the module signature. Consider

$$\begin{aligned}
&(\Sigma_i^v, \Sigma_e^v, R^+) = ((\Sigma_i)_{1\le i\le n}, (\Sigma_e)_{1\le e\le m}, R^+) = (\Sigma_{i=k}, \Sigma_{e=l}, R^+ = I) \\
&\equiv ((\Sigma_1 = \phi, \phi, \ldots, \Sigma_k, \ldots, \Sigma_n = \phi), (\Sigma_1 = \phi, \phi, \ldots, \Sigma_l, \ldots, \phi), R^+ = I) \\
&\cong (\Sigma_{i=k}, \Sigma_{e=l}, R^+ = I) \\
&\equiv (\Sigma_i, \Sigma_e, I)
\end{aligned}$$

with $1 \le i \le n$, $\Sigma_i = \phi$, for $i \neq k$ and $1 \le e \le m$, $\Sigma_e = \phi$, for $e \neq l$.

and $\Sigma_i = (\phi, \phi) \cong \phi$ and $\Sigma_i^v = (\Sigma_1 = \phi, \ldots, \phi, \Sigma_k \neq \phi, \phi, \ldots, \phi, \Sigma_m = \phi) \cong \Sigma_k$.

Thus, the more general vector module signature reduces to the simpler module signature.

□

Definition 9: Modularized Abstract Data Type

i. A *modularized abstract data type* for the module signature (Σ_i, Σ_e) or, briefly, a (Σ_i, Σ_e)-*module* is a total function

$$M : Alg(\Sigma_i) \rightarrow \wp(Alg(\Sigma_e))$$

such that for each algebra $A \in Alg(\Sigma_i)$ the class $M(A) \subseteq Alg(\Sigma_e)$ is an abstract data type.

ii. A (Σ_i, Σ_e)-module M is called *persistent for an algebra* $A \in Alg(\Sigma_i)$, if:

for each $B \in M(A)$:

$$(A \mid \Sigma_i \cap \Sigma_e) \cong (B \mid \Sigma_i \cap \Sigma_e).$$

It is called *persistent* if it is persistent for all $A \in Alg(\Sigma_i)$.

iii. A (Σ_i, Σ_e)-module M is called *consistent for an algebra* $A \in Alg(\Sigma_i)$, if $M(A) \neq \phi$.

It is called *consistent* if it is consistent for all $A \in Alg(\Sigma_i)$.

iv. A (Σ_i, Σ_e)-module M is called *monomorphic for an algebra* $A \in Alg(\Sigma_i)$, if $M(A)$ is monomorphic. It is called *monomorphic* if it is monomorphic for all $A \in Alg(\Sigma_i)$.

□

Informally, persistency means that the inherited sorts and operations have the same meaning in A and $M(A)$ up to isomorphism. Consistency expresses the fact that the mapping M is "effective".

Clearly, an abstract data type may be viewed as a "constant" module, i.e., a module with an empty import signature.

Definition 10: Modularized Vector Abstract Data Type, extension of modularized ADT

A *modularized vector abstract data type* for the module signature $(\Sigma_i^v, \Sigma_e^v, R^+)$ is a family of total functions that define the relationships between the various classes in a design pattern. $\Sigma_i^v = (\Sigma_1, \Sigma_2, \ldots, \Sigma_m)$ and $\Sigma_e^v = (\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$ are vector signatures representing abstract data types corresponding to classes in the main body (realization part) and interface (specification part) of a design pattern respectively.

i. The *realization part* of the *modularized vector abstract data type* for the module signature $(\Sigma_i^v, \Sigma_e^v, R^+)$ is defined by the following function:

$$M_1 : Alg(\Sigma_k) \rightarrow \wp(Alg(\Sigma_l))$$

where the following conditions hold: $1 \le k, l \le m$ and $k \ne l$ and $\Sigma_k, \Sigma_l \in \Sigma_i^v$ and for each algebra $A \in Alg(\Sigma_k)$ the class $M_1(A) \subseteq Alg(\Sigma_l)$ is an abstract data type.

ii. The *specification part* of the *modularized vector abstract data type* for the module signature $(\Sigma_i^v, \Sigma_e^v, R)$ is defined by the following function:

$$M_2 : \bigcup_{1 \le k \le m} Alg(\Sigma_k) \rightarrow \bigcup_{1 \le l \le n} \wp(Alg(\Sigma_l))$$

where the following conditions hold: $1 \le k \le m$, $\Sigma_k \in \Sigma_i^v$ and $1 \le l \le n$, $\Sigma_l \in \Sigma_e^v$ and for $A \in \wp(Alg(\Sigma_k))$ the class $M_2(A) \subseteq \wp(Alg(\Sigma_l)$ is an abstract data view. □

The function M_1 effectively defines the use of object-oriented design principles such as inheritance, composition, and aggregation in the progressive build up of the realization part of design patterns.

The mapping allows subsets of the component ADTs of the design pattern to present their interfaces through a combined abstract data view. The ADV can be used to specialized or extend the functionality provided by the component ADTs comprising the realization part of the design pattern.

Definition 11: Loose Module Specification

Let L be a logic.

i. *Abstract Syntax*: A loose module specification in L is a pair $msp = ((\Sigma_i, \Sigma_e), \Phi)$ where (Σ_i, Σ_e) is a module signature with $\Sigma_i \subseteq \Sigma_e$ and $\Phi \subseteq L(\Sigma_e)$ is a set of formulas.

ii. *Semantics*: The meaning $M(msp)$ of the loose module specification $msp = ((\Sigma_i, \Sigma_e), \Phi)$ is the (Σ_i, Σ_e) – module defined by:

$$M(msp)(A) = \{B \in Alg(\Sigma_e) \mid B \models \Phi \text{ and } (B \mid \Sigma_i) \cong A\}$$

for each $A \in Alg(\Sigma_i)$. □

Clearly, a loose module specification defines a persistent but not necessarily consistent module.

Definition 12: Design Pattern Specification, Extension of Loose Module Specification

Let L be a logic.

i. *Abstract Syntax*

A *design pattern specification* in L is a pair $dpsp = ((\Sigma_i^v, \Sigma_e^v, R^+), \Phi)$ where $(\Sigma_i^v, \Sigma_e^v, R^+)$ is a vector module signature with $R = \bigcup_{1 \le i \le 3} R_i^+$ and $(R_1 modifies \Sigma_i^v)$, $(R_2 modifies \Sigma_e^v)$, $(R_3 modifies (\Sigma_i^v \cup \Sigma_e^v))$ and $\Phi = \Phi_1 \cup \Phi_2$ with $\Phi_1 \subseteq L(\Sigma_i^v)$ and $\Phi_2 \subseteq L(\Sigma_e^v)$. Φ is a set of formulas that defines the derivation sequence to establish a relationship between two instances (entities) of the abstract data types corresponding to the vector signatures. R is the resultant static relationship that is determined by Φ.

ii. Semantics

The meaning $M(dpsp)$ of the design pattern specification $dpsp = ((\Sigma_i^v, \Sigma_e^v, R^+), \Phi)$ is the $(\Sigma_i^v, \Sigma_e^v, R^+)$-module defined by the following set of mappings:

1. The meaning of the relationships in the realization part of the design pattern specification is given by:

$$M(dpsp)(A) = \{B \in \mathrm{Alg}(\Sigma_l) \mid A \models_{\Phi_1} B, iff \text{ for each } op_i \in \Phi_1,$$

$$A_i' \models_{op_i} A_{i+1}', \text{ implies } (A_i', A_{i+1}') \in R_1^+ \text{ and } op_i(A_i') = A_{i+1}' \}$$

for each algebra $A \in Alg(\Sigma_k)$ and $1 \le k, l \le m$ and $k \ne l$ and $\Sigma_k, \Sigma_l \in \Sigma_i^v$ and $A_i', A_{i+1}' \in Alg(\Sigma_i^v)$.

Note: $A \models_{\Phi_1} B \equiv B \models \Phi_1$.

2. The meaning of the relationships in the interface part of the design pattern specification is given by:

$$M(dpsp)(A) = \{B \in \wp(Alg(\Sigma_l)) \mid A \models_{\Phi_2} B, iff \text{ for each } op_i \in \Phi_2,$$

$$A_i' \models_{op_2} A_{i+1}', \text{ implies } (A_i', A_{i+1}') \in R_2^+ \text{ and } op_i(A_i') = A_{i+1}'\}$$

for each $A \in \wp(Alg(\Sigma_k))$ and $1 \le k \le m, 1 \le l \le n, \Sigma_k \in \Sigma_i^v$ and $\Sigma_l \in \Sigma_e^v$. □

The set of formulas represented by Φ_1 characterizes the nature of the relationship between abstract data types constituting the realization part of the design pattern that is consistent with the mapping defined by M_1 of definition 10. Φ_1 is depicted as follows:

$$\Phi_1 \subseteq L(\Sigma_i^v) \equiv (\bigcup_{i \le l \le m} L(\Sigma_l))$$

The set of formulas represented by Φ_2 characterizes the nature of the relationship between the specification part and realization part of the design pattern that is consistent with the mapping defined by M_2 of definition 10. Φ_2 is depicted as follows:

$$\Phi_2 \subseteq L(\Sigma_e^v) \equiv (\bigcup_{i \le l \le n} L(\Sigma_l))$$

We can think of $A \models_\Phi B$ as having the meaning of abstract data type B derived from abstract data type A through a sequence of *formulas* or operations belonging to Φ. In addition, an operation is only permissible if the resulting relationships between the

abstract data types for each of the derivation steps are contained in the transitive closure of R. That is, if the following relationship holds:

$$A_i' \models_{op_i} A_{i+1}', \text{iff}(A_i', A_{i+1}') \in R^+$$

and $op_i \in \Phi$ and $A_i', A_{i+1}' \in \wp(A\lg(\Sigma_i^v) \cup \wp(A\lg(\Sigma_e^v)$

This process of deriving abstract data type B from abstract data type A can be interpreted using a derivation tree for the operations in Φ. The process of building the derivation tree is constrained by the relationship set depicted by R^+. Figure 9.6 gives the schematics of the derivation tree for deriving B from A.

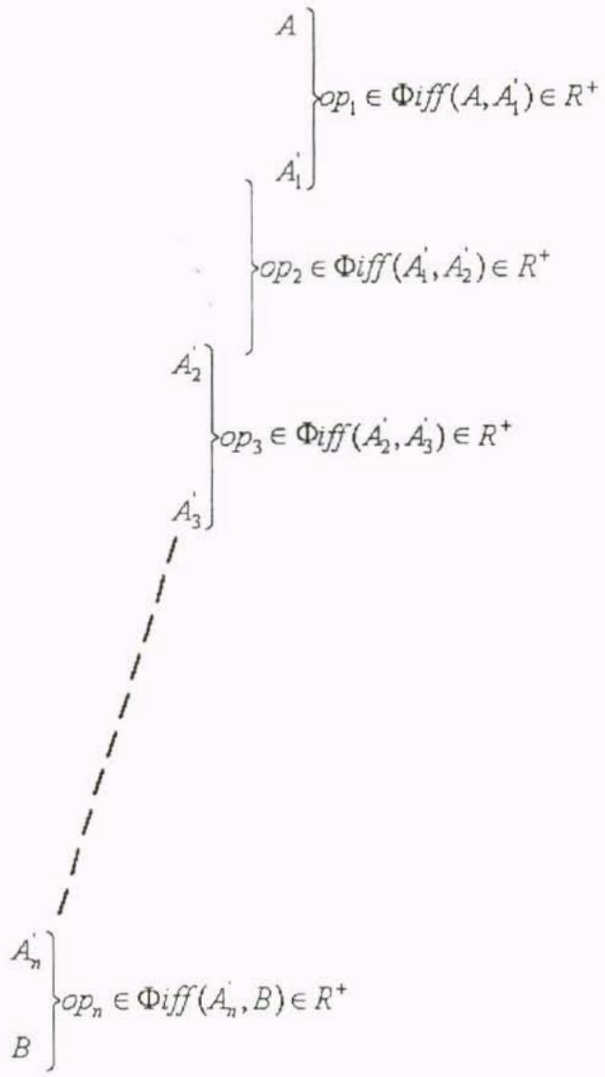


Figure 9.6: Schematic Derivation Tree for Vector Algebra *B* being derived from Vector Algebra *A*

It is conceivable to have more than one derivation tree for a design pattern. Each derivation tree will result in a difference structural version of the pattern. This variation could account for differences in implementation approaches. For example, one implementation may favor delegation over an inheritance-based strategy.

The concept of derivation tree presented above can be used to give some insight into the effort required to reuse a component. Algebra A represents the component to be reused and algebra B represents the desired component. The derivation tree gives the sequence of transformations that can be used to go from A to B. The sequence of transformations is a quantitative measure of the effort to reuse a particular component.

9.2 Semantics of Design Patterns and their Specification Constructors

A reasonable representation for the Semantics of Design Patterns and their Specification Constructors is presented based on the concept of Abstract Data Views (ADV) that was proposed by Donald Cowan and his associates [Cowan 92, Cowan 93, Cowan 93a]. The ADV specification is consistent with the theoretical model put forward in this work and is based on the general principle of term writing system.

The semantics of the specification constructor for composition is given in terms of the representation of both ADVs and ADTs. The interpretation of the relationship between ADTs and ADVs is done using the variable *owner* (see Chapter 3). A general schema for an ADV is shown in Figure 9.7.

ADV_Type = ADV [is ADV] [for ADT]

declaration	$x_i : T_i, (i = 1,...,I)$
invariant	Inv
component	*ADV_Type* = *ADV* [for ADV'] [for ADT']
	.....
component	set of ADV_Type1 = ADV [for ADV'] [for ADT']
	.....
component	seq of ADV_Type2 = ADV [for ADV'] [for ADT']
	.....
function	$f_i(a_{k,j} : U_{k,j}; k = 1,...,K_j) a_{k+1,j} : U_{k+1,j} (j = 1,...,J)$
	declaration $y_{l,j} : V_{l,j} (l = 1,...,L)$
	external $status_{m,j} : z_{m,j}, (m = 1,...,M)$

where $status_{m,j} \in \{`wr', `rd'\}$

pre_condition $pre\_ADV\_f_i$

post_condition $post\_ADV\_f_i$

event $e_n(b_{o,n} : R_{o,n}; o = 1,...,O_n)(n = 1,...,N)$

declaration $v_{p,n} : S_{p,n}, (p = 1,...,P)$

external $status_{q,n} : w_{q,n}, (q = 1,...,Q),$

where $status_{q,n} \in \{`wr', `rd'\}$

pre_condition $pre\_ADV\_e_n$

post_condition $post\_ADV\_e_n$

end ADV

Figure 9.7: A General Schema for an ADV

In the schema of Figure 9.7, the declaration "*ADV_Type* = ADV[is ADV][for ADT]" expresses the definition of a type ADV. The symbol "[..]" encloses optional syntactic items, i.e., the declaration "is ADV" and "for ADT" are optional. The declaration "is ADV" represents the inheritance relationship, i.e., the *ADV_Type* is being defined as a specialization or extension of another *ADV_Type*. The declaration "for ADT" represents the association of an *ADV_Type* with an *ADT_Type*.

ADT_Type = ADT

declaration $\overline{x_i} : \overline{T_i}, (i = 1,...,\overline{I})$

invariant $\overline{Inv}$

component $ADT\_Type = \overline{ADT}$

.....

function $\overline{f_i}(\overline{a_{k,j}} : \overline{U}_{k,j}; k = 1,...,\overline{K}_j)\overline{a}_{k+1,j} : \overline{U}_{k+1,j}(j = 1,...,\overline{J})$

declaration $\overline{y}_{l,j} : \overline{V}_{l,j}(l = 1,...,\overline{L})$

external $status_{m,j} : \overline{z}_{m,j}, (m = 1,...,\overline{M})$

where $status_{m,j} \in \{'wr','rd'\}$

pre_condition $\overline{pre\_ADT\_f_j}$

post_condition $\overline{post\_ADV\_f_j}$

end ADT

Figure 9.8: A General Schema for an ADT

An *ADV_Type* is composed of a declaration part, invariant, components, functions, and events. The declaration part represents the private variables of the *ADV_Type*. The invariant part describes the constraints on variables that compose the *ADV_Type*. The components represent the structural composition of the *ADV_Type*. Finally, the functions and events describe the behavior of the *ADV_Type*.

A general schema for an ADT is presented in Figure 9.8. The symbols and constructs used in this schema are the same ones used in the ADV representation. The variable *owner* represents the association of an ADV with an ADT. This association is illustrated by the representation in Figure 9.9.

ADV_Type = ADV for ADT

declaration $x_i : T_i, (i = 1,...,I)$

owner : ADT_Type

invariant Inv

component .....

function $f_i(a_{k,j} : U_{k,j}; k = 1,...,K_j)a_{k+1,j} : U_{k+1,j}(j = 1,...,J)$

declaration $y_{l,j} : V_{l,j}(l = 1,...,L)$

external $status_{m,j} : z_{m,j}, (m = 1,...,M)$

where $status_{m,j} \in \{`wr', `rd'\}$

pre_condition $pre\_ADV\_f_i$

post_condition $post\_ADV\_f_i$

event $e_n(b_{o,n} : R_{o,n}; o = 1,...,O_n)(n = 1,...,N)$

declaration $v_{p,n} : S_{p,n}, (p = 1,...,P)$

external $status_{q,n} : w_{q,n}, (q = 1,...,Q),$

where $status_{q,n} \in \{`wr', `rd'\}$

pre_condition $pre\_ADV\_e_n$

post_condition $post\_ADV\_e_n$

end ADV

Figure 9.9: A General Schema Showing Inclusion of an ADT in an ADV

9.3 Closure of Design Pattern under Composition

Let $\Sigma_v = (\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$ be a vector signature. The composition of two Σ_v-homomorphisms, say $H : A \to B$ and $G : B \to C$, yields a Σ_v-homomorphism $G \circ H : A \to C$ that is a family of functions of the form $G \circ H = (g_{s^i} \circ h_{s^i})_{i=1..n}$.

Theorem

For any vector signature Σ_v the composition of two Σ_v-homomorphisms yields a Σ_v-homomorphism.

Proof:

Given a vector signature $\Sigma_v = (\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$ and the two Σ_v-homomorphisms $H : A \to B$ and $G : B \to C$, we want to show that $G \circ H : A \to C$ satisfies the vector homomorphism condition, thus,

$$G \circ H(A(w)(\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_n) = C(w)(G \circ H(\overline{a}_1), G \circ H(\overline{a}_2), \ldots, G \circ H(\overline{a}_n)).$$

That is,

$$g \circ h_{s^1}(A_1)(w_1)(\overline{a}_1), g \circ h_{s^2}(A_2)(w_2)(\overline{a}_2), \ldots, g \circ h_{s^n}(A_n)(w_n)(\overline{a}_n)$$

$$= C_1(w_1)(g \circ h_{s^1}(\overline{a}_1)), C_2(w_2)(g \circ h_{s^2}(\overline{a}_2)), \ldots, C_n(w_n)(g \circ h_{s^n}(\overline{a}_n))$$

Therefore, we have

$$G(H(A(w)(\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_n) = G(B(w)(H(\overline{a}_1), H(\overline{a}_2), \ldots, H(\overline{a}_n))$$

$$= G(B_1(w_1)(h_{s^1}(\overline{a}_1)), B_2(w_2)(h_{s^2}(\overline{a}_2)), \ldots, B_n(w_n)(h_{s^n}(\overline{a}_n)))$$

- Applying vector homomorphism condition

$$= g_{s^1}(B_1(w_1)(h_{s^1}(\overline{a}_1))), g_{s^2}(B_2(w_2)(h_{s^2}(\overline{a}_2))), \ldots, g_{s^n}(B_n(w_n)(h_{s^n}(\overline{a}_n)))$$

- Applying G component wise

$$= C_1(w_1)(g_{s^1} \circ h_{s^1}(\overline{a}_1)), C_2(w_2)(g_{s^2} \circ h_{s^2}(\overline{a}_2)), \ldots, C_n(w_n)(g_{s^n} \circ h_{s^n}(\overline{a}_n))$$

$$= C_1(w_1)(G \circ H(\overline{a}_1)), C_2(w_2)(G \circ H(\overline{a}_2)), \ldots, C_n(w_n)(G \circ H(\overline{a}_n))$$

$$= C(w)(G \circ H(\overline{a}_1), G \circ H(\overline{a}_2), \ldots, G \circ H(\overline{a}_n))$$

9.4 Examples Illustrating the Use of the Formalism Presented Above

The Factory Method design pattern primary intent is to define an interface for creating an object, but let subclasses decide which class to instantiate [Gamma1995]. Thus, the Factory Method lets a class defer instantiation to subclasses. Figure 9.10 provides a generic structure of the Factory Method design pattern.

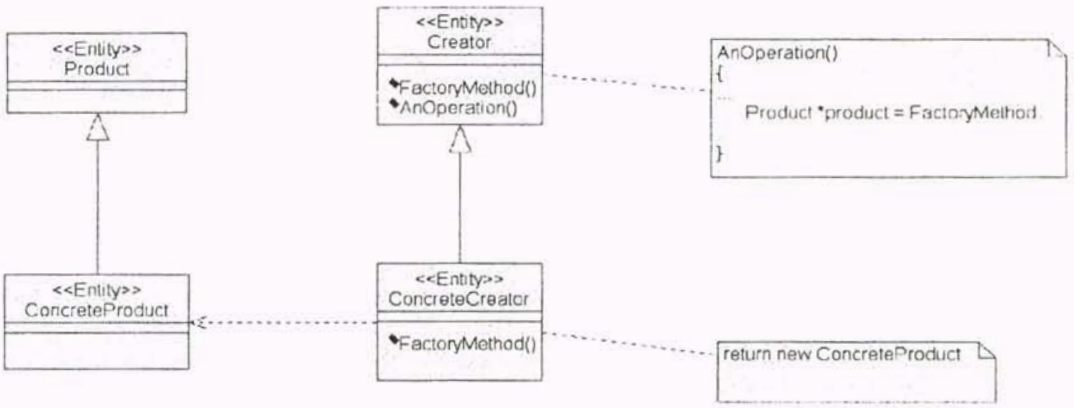


Figure 9.10: Generic Structure of the Factory Method Design Pattern

Use of the Factory Method design pattern is applicable when either of the following conditions apply: (1) a class can't anticipate the class of objects it must create, (2) a class wants its subclasses to specify the objects it creates, or (3) classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate [Gamma 1995]. The Factory Method is ideal for use in frameworks that use abstract classes to define and maintain relationships between objects. Frameworks are often responsible for creating the objects as well.

Figure 9.11 gives an instance of the Factory Method design pattern that can be used in a multiple document framework. The key abstractions are *Document* and *Application*. The Factory Method pattern encapsulates the knowledge of which Document subclass to create and moves this knowledge out of the framework. Application subclasses redefine an abstract *CreateDocument* operation on Application to return the appropriate Document subclass. Once an Application subclass is instantiated, the application subclass can then instantiate application-specific Documents without knowing their classes.

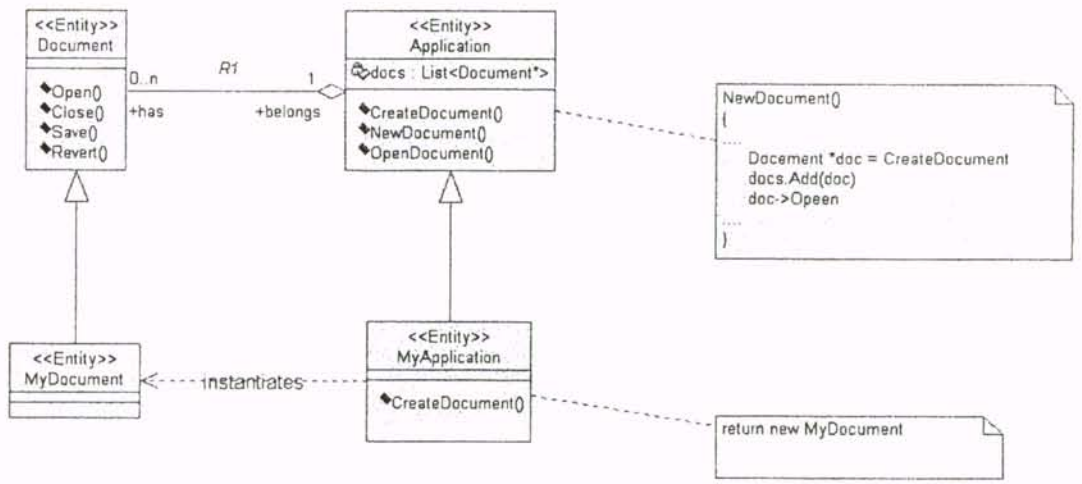


Figure 9.11: Instance of Factory Method Design Pattern

The Factory Method design pattern for the document framework is represented by FM_{DP} using the formalism presented above. Thus,

$$FM_{DP} = ((Alg(\Sigma_i^v), R_1), (Alg(\Sigma_e^v), R_2), ((Alg(\Sigma_i^v) \cup Alg(\Sigma_e^v)), R_3))$$

The realization part of the factory method design pattern is represented by the tuple $(Alg(\Sigma_i^v), R_1)$. These components are explained below.

$$\Sigma_i^v = ((\Sigma_{Document} = (\{void,\ldots\},\{Open : void \rightarrow void, Close : void \rightarrow void,$$
$$Save : void \rightarrow void , Revert : void \rightarrow void,\ldots\})),$$
$$(\Sigma_{MyDocument} = (\{Document, void,\ldots\},\{Open : void \rightarrow void, Close : void \rightarrow void,$$
$$Save : void \rightarrow void , Revert : void \rightarrow void,\ldots\})),$$
$$(\Sigma_{Application} = (\{Document, void, int,\ldots\},$$
$$\{CreateDocument : void \rightarrow Document,$$
$$NewDocument : void \rightarrow int,$$
$$OpenDocument : string \rightarrow int,\ldots\})),$$
$$(\Sigma_{MyApplication} = (\{Application, Document, void, int,\ldots\},$$
$$\{CreateDocument : void \rightarrow Document,$$
$$NewDocument : void \rightarrow int,$$
$$OpenDocument : string \rightarrow int,\ldots\}))$$

The *class* of Σ_v-algebras corresponding to Σ_i^v is represented by $Alg(\Sigma_i^v)$,

$$Alg(\Sigma_i^v) = (Document, MyDocument, Application, MyApplication).$$

The relationships between the algebraic entities is represented by R_1,

$$R_1 = \{\ (R1,\ Aggregation,\ Document,\ Application,\ belongs,\ has,\ 1,\ 0..n),$$
$$(nil,\ Inheritance,\ MyDocument,\ Document,\ nil,\ nil,\ nil,\ nil),$$
$$(nil,\ Inheritance,\ MyApplication,\ Application,\ nil,\ nil,\ nil,\ nil),$$

$(nil, Instantiates, MyApplication, MyDocumnet, nil, nil, nil, nil)$ }

The Specification part or interface of the Factory Method design pattern is represented by the tuple $(Alg(\Sigma_e^v), R_2)$. There is no explicit specification part or interface corresponding to this design pattern. However, the *Application* and the set of *ConcreteApplication* classes can be combined to form the interface specification for this pattern. In addition, it is quite easy to extend this pattern with an interface. Figure 9.12 shows an expanded example that has an interface.

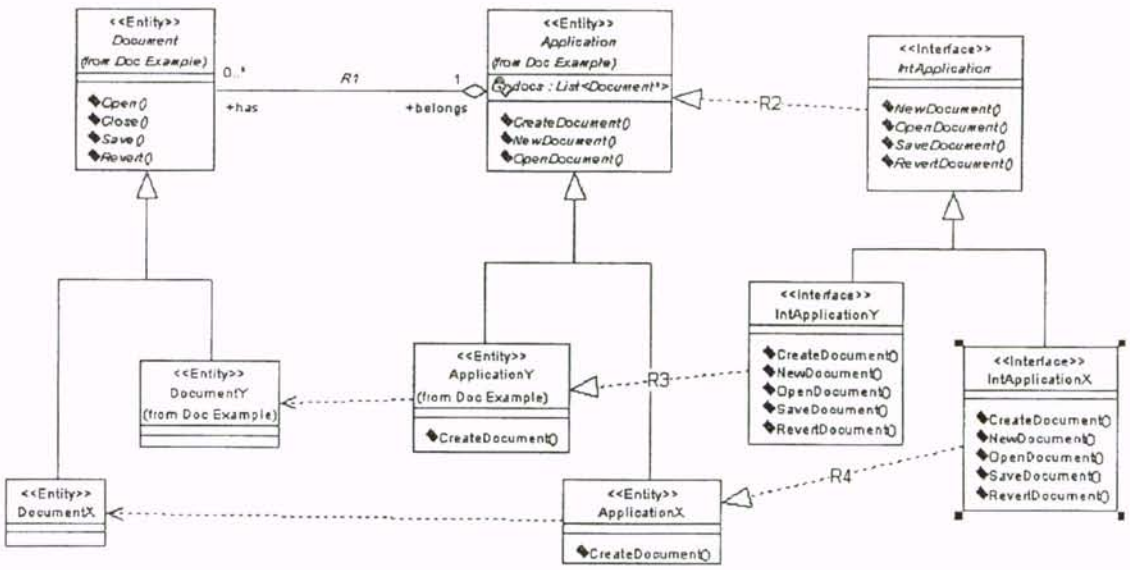


Figure 9.12: Factory Method Design Pattern with Interface

The interface corresponding to Figure 9.11 is given by $(Alg(\Sigma_e^v), R_2)$ where we have the following:

$$\Sigma_e^v = ((\Sigma_{Application} = (\{Document, void, int, \ldots\},$$

$$\{CreateDocument : void \rightarrow Document,$$

$$NewDocument : void \rightarrow int,$$

$$OpenDocument : string \rightarrow int,\ldots\})),$$

$$(\Sigma_{MyApplication} = (\{Application, Document, void, int,\ldots\},$$

$$\{CreateDocument : void \rightarrow Document,$$

$$NewDocument : void \rightarrow int,$$

$$OpenDocument : string \rightarrow int,\ldots\}))$$

The *class* of Σ_v-algebras corresponding to Σ_e^v is represented by $Alg(\Sigma_e^v)$,

$$Alg(\Sigma_e^v) = (Application, MyApplicationX)$$

The relationships between the algebraic entities in the interface is represented by R_2,

$$R_2 = \{\ (nil, Inheritance, MyApplication, Application, nil, nil, nil, nil)\ \}$$

The relationship between interface and the realization is represented by R_3,

$$R_3 = \{\ (nil, Instantiates, MyApplication, MyDocumnet, nil, nil, nil, nil)\ \}$$

The interface corresponding to Figure 9.12 is given by $(Alg(\Sigma_e^v), R_2)$ where we have the following:

$$\Sigma_e^v = ((\Sigma_{intApplication} = (\{Application, Document, void, int,\ldots\},$$

$$\{CreateDocument : void \rightarrow Document,$$

$$NewDocument : void \rightarrow int,$$

$$OpenDocument : string \rightarrow int,\ldots\})),$$

$$(\Sigma_{intApplicationX} = (\{intApplication, Document, DocumentX, void, int,\ldots\},$$

$$\{CreateDocument : void \rightarrow Document,$$

$$NewDocument : void \rightarrow int,$$

$$OpenDocument : string \rightarrow int,\ldots\})),$$

$$(\Sigma_{intApplicationY} = (\{Application, Document, DocumentY, void, int,\ldots\},$$

$$\{CreateDocument : void \rightarrow DocumentX,$$

$$NewDocument : void \rightarrow int,$$

$$OpenDocument : string \rightarrow int,\ldots\}))$$

The *class* of Σ_v-algebras corresponding to Σ_e^v is represented by $Alg(\Sigma_e^v)$,

$$Alg(\Sigma_e^v) = (intApplication, intApplicationX, intApplicationY)$$

The relationships between the algebraic entities in the interface is represented by R_2,

R_2 = { (*nil*, *Inheritance*, *intApplicationX*, int*Application*, *nil*, *nil*, *nil*, *nil*),

(*nil*, *Inheritance*, *intApplicationY*, int*Application*, *nil*, *nil*, *nil*, *nil*) }

The relationship between interface and the realization is represented by R_3,

R_3 = { (*R2*, *Realizes*, *intApplication*, *Application*, *nil*, *nil*, *nil*, *nil*),

(*R4*, *Realizes*, *intApplicationX*, *ApplicationX*, *nil*, *nil*, *nil*, *nil*),

(*R3*, *Realizes*, *intApplicationY*, *ApplicationY*, *nil*, *nil*, *nil*, *nil*) }

Realize is a special form of inheritance. It allows you to inherit a subset of a signature.

9.4.1 The Transformation Process of Building the Document Framework Pattern

The document framework version of the Factory Method design pattern shown in Figure 9.12 can be built using a sequence of elemental transformations that are consistent with the permissible relationships.

The set of transformation operations include the following:

- Inheritance of operation
- Inheritance of sorts or types
- Addition of sort to the signature
- Addition of operation to the signature
- Rename of operation
- Rename of sort
- Addition of variable
- Rename of variable
- Aggregate a class to another
- Instantiate
- Realize a class by another
- Duplication of a design pattern

The sequence of transformation is as follows:

1. Duplicate the pattern in Figure 9.10
2. Rename Product to Document
3. Rename Creator to Application

4. Rename ConcreteProduct algebra to DocumentX
5. Rename ConcreteCreator algebra to ApplicationX
6. Aggregate (Document, Application)
7. Add operation Open to Document
8. Add operation Close to Document
9. Add operation Save to Document
10. Add operation Revert to Document
11. Inherit operation Open (DocumentX, Document)
12. Inherit operation Close (DocumentX, Document)
13. Inherit operation Save (DocumentX, Document)
14. Inherit operation Revert (DocumentX, Document)
15. Rename operation (FactoryMethod, CreateDocument) Application
16. Update operation signature CreateDocument (void→Document) Application
17. Rename operation (AnOperation, NewDocument) Application
18. Add operation OpenDocument to Application

The resulting vector module signature after the above sequence of operations is given below:

$$\Sigma_t^v = ((\Sigma_{Document} = (\{void,\ldots\},\{Open : void \rightarrow void, Close : void \rightarrow void,$$
$$Save : void \rightarrow void\ , Revert : void \rightarrow void,\ldots\})),$$
$$(\Sigma_{DocumentX} = (\{Document, void,\ldots\},\{Open : void \rightarrow void, Close : void \rightarrow void,$$
$$Save : void \rightarrow void\ , Revert : void \rightarrow void,\ldots\})),$$
$$(\Sigma_{Application} = (\{Document, void, int,\ldots\},$$
$$\{CreateDocument : void \rightarrow Document,$$

$$NewDocument : void \rightarrow int,$$

$$OpenDocument : string \rightarrow int,\ldots\})),$$

$$(\Sigma_{ApplicationX} = (\{Application, Document, void, int,\ldots\},$$

$$\{CreateDocument : void \rightarrow Document,$$

$$NewDocument : void \rightarrow int,$$

$$OpenDocument : string \rightarrow int,\ldots\}))$$

The relationships between the algebraic entities is represented by R_1,

$$R_1 = \{\ (R1, Aggregation, Document, Application, belongs, has, 1, 0..n),$$

$$(nil, Inheritance, MyDocument, Document, nil, nil, nil, nil),$$

$$(nil, Inheritance, MyApplication, Application, nil, nil, nil, nil),$$

$$(nil, Instantiates, MyApplication, MyDocumnet, nil, nil, nil, nil)\ \}$$

9.5 Applicability to Reuse

With the exception of the *duplicate pattern* operation, the operations presented above are all elemental. Since the set of elemental operations needed to build a component is finite then we can use the aggregate of the sequence of operations as a quantitative measure of the effort to reuse a component. The operations can be fitted with a differential-weighting scheme based on relative weights of the operations.

The prevailing conclusion of the collective wisdom of building complex distributed software over the past decade or so is that the software construction process must be iterative and incremental. The software practitioner must have a very good

understanding of what he wants to build and must be able to give a reasonable specification of it, albeit incomplete. A tool environment that takes advantage of the above mentioned formal principles will give the practitioner the ability to play scenario games with very complex modules specification and therefore help to guide the development process.

The formal principle explored above can be incorporated into the very large-scale software construction process to facilitate automatic program verification. Given the beginning and ending specifications, a tool could use the principles above to verify that the resulting component is consistent. In fact, it would be able to identify the offending code giving developers the ability to zero in on supposedly suspected code.

The applicability of the formal principles explored in this thesis to automatic program verification can be put into the format of a theorem prover based on the principle of *interpretation between theories* [Enderton 1972]. The vector signature concept can be incorporated into a logic based on predicate calculus. This can then be used to represent a design pattern as a *theory*, to which the principle of interpretation between theories can be applied. Thus, we can develop a formal mathematical basis for the theorem prover.

Chapter 10

Conclusions and Future Work

Software reuse is the reapplication of a variety of existing knowledge during the construction of a software system in order to reduce the effort of development and maintenance of the new system. This reused knowledge includes artifacts such as domain knowledge, development experience, design decisions, architectural structures, module-level implementation structures, specification, design, code, etc. Different reuse techniques may emphasize or de-emphasize certain of these artifacts.

Creating a complex software system with a smaller amount of effort and less cognitive burden on the part of the software developer implies a higher level of abstraction. For a developer to effectively select, specialize, and integrate reusable artifacts, the reuse technology must provide natural, succinct, high-level abstractions in which the abstraction specifications describe the artifacts in terms of what they do. The ability of a developer to practice software reuse is primarily limited by the abstraction mechanism employed by the reuse technology. That is, there must be a small cognitive distance between informal reasoning and the abstract concepts defined by the reuse technology.

Why is software reuse so difficult? The answer is simply that raising the level of abstraction of an artifact is extremely difficult. For example, early reuse required the

development of the entire body of knowledge of Formal Language Theory before unlocking the secrets of compiler construction.

The object-oriented approach to software development has emerged as one of the primary vehicles for the realization of software reuse. The features of inheritance, dynamic binding, and polymorphism offered by this paradigm provide an extremely powerful and elegant approach to software reuse, which differs fundamentally from other mechanisms.

Design patterns express the static and dynamic structures and collaborations of components in software architectures. Patterns aid the development of extensible distributed system components and frameworks by expressing the structure and collaboration of participants in software architecture at a level higher than (1) source code and (2) object-oriented design models that focus on individual objects and classes.

Design patterns are an effective mechanism for capturing successful designs and micro-architectures. Expressing proven techniques as design patterns makes them more accessible to new systems and thus facilitates greater reuse. The ability to reuse a successful pattern without any modification is highly desirable. However, this is hardly a realistic expectation because the interface exported may not be an exact match. Hence, the next best thing is to be able to superimpose on the design pattern the requisite interface. The abstract data view (ADV) concept performs this task perfectly.

10.1 Summary

We have shown how to use the concepts of ADV, design pattern, and software architecture to create a very powerful software architecture framework for developing

new applications and integrating existing applications into a unified adaptive business centric solution. To illustrate the approach, we have applied this framework to solving the OSS integration problem in the telecommunications industry.

We have presented a model-based software development approach. This is an approach to raise the abstraction level at which application developers work and to automate the process of translation from an application model to its corresponding distributable runtime component. The basic thesis here is that we can effectively reverse the effort role in the software development process in which about 80% of the development effort goes into the development of infrastructure services and 20% into the development of application logic.

We have presented a mathematical formalism for the specification of design patterns. This specification constitutes an extension of the basic concepts from many-sorted algebra. In particular, the notion of signature is extended to that of a vector, consisting of a set of linearly independent signatures. The linearly independence property is necessary to satisfy non-interference that is essential for compositional based construction. This is of fundamental concern in the building of large-scale software systems where we have the composition of smaller components to form larger components. The approach can be used to determine efforts for component reuse and facilitate program verification. The approach has the potential to be able to aid complex software development by providing the developer with different design alternatives.

10.2 Future Work

The material from Chapters 8 and 9 present opportunities for the construction of various tools to explore the concepts proposed by this research effort. The algebraic specification formalism presented illustrates an approach to determine the effort to reuse a software artifact. This concept can also be applied to automatic program verification and other related concepts.

The applicability of the formal principles explored in this thesis to automatic program verification can be put into the format of a theorem prover based on the principle of *interpretation between theories* [Enderton 1972]. The vector signature concept can be incorporated into a logic based on predicate calculus. This can then be used to represent a design pattern as a *theory*, to which the principle of interpretation between theories can be applied. Thus, we can develop a formal mathematical basis for the theorem prover.

The proposed adaptive integration architecture can be combined with the model-based software development approach presented in Chapter 8 to provide a very powerful IDE. The architecture framework could be transparently provided by the IDE along with all the relevant infrastructure services. Such an environment would definitely facilitate raising the abstraction level at which application developers work by allowing them to focus on *pure application* model specification using a meta-metamodel formalism. The application model is independent of infrastructure and imperative programming language specificities.

The adaptive architecture technique explored in this research undertaking is of a generic nature and applicable to any customer centric business software application that

is transactional by nature. Some of the relevant application domains include financial services, telecommunication OSSs, item tracking, energy and water utilities, back-office item processing, and office automation. Applying the proposed approach to such systems will provide further experiments in supporting the findings of this research effort.

REFERENCES

[Alencar 1994] The semantics of abstracts data views: A design concept to support reuse-in-the-large, A. Alencar, L. Carneiro, D. D. Cowan, and C. Lucena, Proc. Colloquium Object-Orientation in Database and Software Engineering. Kluwer Press, May 1994.

[Alencar 1994a] Towards a formal theory of abstract data views, A. Alencar, L. Carneiro, D. D. Cowan, and C. Lucena, Technical Report 94-18, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, April 1994.

[Arango 1988] Domain Engineering for Software Reuse, Ph.D. Thesis, G. Arango, Computer Science Department, University of California, Irvine, CA, 1988.

[Arnold 1988] The REUSE System: Cataloging and Retrieval of Reusable Software, S. P. Arnold and S. L. Stepoway, Editor: W. Tracz, Software Reuse: Emerging Technologies, IEEE Computer Society, 1988, pp. 138-141.

[Atkinson 1991] Object-Oriented Reuse, Concurrency and Distribution: An Ada-based approach, Colin Atkinson, ACM Press, New York, NY, 1991.

[Balzer 1989] A 15 year perspective on automatic programming, Frontier Series: Software Reusability: Application and Experience, Volume II, Ted J. Biggerstaff and Alan J. Perlis (editors), ACM Press, New York, NY, 1989, pp. 289-311, Chapter 14.

[Bass 1991] Developing Software for the User Interface, L. Bass and J. Coutaz, Reading, MA: Addison-Wesley, 1991.

[Betts 1990] Math packages multiply, Kellyn S. Betts, Mechanical Engineering, Volume 112, Number 8, August 1990, pp. 32-38.

[Biggerstaff 1989] Reusability Framework, Assessment, and Directions, Frontier Series: Software Reusability: Concepts and Models, Volume I, Ted J. Biggerstaff and Alan J. Perlis (editors), ACM Press, New York, NY, 1989, pp. 1-17.

[Biggerstaff 1989b] Reusability Framework, Assessment, and Directions, Frontier Series: Software Reusability: Application Experience, Volume II, Ted J. Biggerstaff and Alan J. Perlis (editors), ACM Press, New York, NY, 1989.

[Bigus 1998] Constructing intelligent agents with Java: A programmer's guide to smarter applications, Joseph P. Bigus and Jennifer Bigus, John Wiley & Sons, Inc., New York, NY, 1998.

[Blair 1989] Genericity vs. Inheritance vs. Delegation vs. Conformance vs. ..., S. G. Blair, J.J. Gallagher, and J. Malik, Journal of Object-Oriented Programming, Volume 2, Number 3, September/October 1989, pp. 11-17.

[Booch 1987] Software Components with Ada: Structure, Tools, and Subsystems, G. Booch, Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

[Booch 1994] Object-Oriented Analysis and Design with Applications, Second Edition, G. Booch, Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[Booch 1999] The Unified Modeling Language User Guide, Grady Booch, James Rumbaugh, and Ivar Jacobson, Addison Wesley Longman, Inc., Reading Massachusetts, 1999.

[Breu 1991] Algebraic Specification Techniques in Object Oriented Programming Environments, R. Breu, Springer-Verlag, Berlin, Heidelberg, 1991.

[Breymann 1998] Designing Components with the C++ STL: A new approach to programming, Ulrich Breymann, Addison Wesley Longman Limited, Edinburgh Gate, Harlow, England, 1998.

[Buchanan 1979] Theory of Library Classification, B. Buchanan, Clive Bingley, London, UK, 1979.

[Buschmann 1998] Pattern-Oriented Software Architecture: A System of Patterns: Volume 1, Frank Buschmann, Regine Meunier, Michael Stal, Hans Rohnert, and Peter Sommerlad, John Wiley & Sons, Inc., New York, NY, July 1998.

[Carneiro 1993] User interface higher-order architectural models, L. M. F. Carneiro, M. H. Coffin, D. D. Cowan, C. J. P. Lucena, Technical Report 93-14, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, 1993.

[Carneiro 1994] ADVcharts: A visual formalism for interactive systems, L. M. F. Carneiro, D. D. Cowan, and C. J. P. Lucena, SIGCHI Bulletin, 1993, pp. 74-77.

[Carneiro 1995] ADVcharts: A Visual Formalism for Highly Interactive Systems, L. M. F. Carneiro-Caffin, D. D. Cowan, and C. J. P. Lucena, Software Engineering in Human-Computer Interaction, eds. M. D. Harrison and C. W. Johnson, Cambridge University Press, Cambridge, 1995.

[Carothers 1997] Design and Implementation of HLA time management in the RTI version F.0, Christopher D. Carothers, Winter Simulation Conference Proceeding, 1997. IEEE, Piscataway, NJ, USA 97CB36141, pp. 373-380.

[Cleaveland 1988] Building application generators, J. C. Cleaveland, IEEE Software, Volume 5, Number 4, July 1988, pp. 25-33.

[Cleeland 1996] External Polymorphism: An Object Structural Pattern for Transparently Extending C++ Concrete Data Types, Chris Cleeland, Douglas C. Schmidt, and Timothy H. Harrison, Proceedings of the 3rd Pattern Languages of Programming Conference, Allerton Park, Illinois, September 4-6, 1996.

[Coplien 1995] Pattern languages of program design, edited by James O. Coplien and Douglas C. Schmidt, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1995.

[Coutaz 1991] Applications: A dimension space for user interface management systems, J. Coutaz and S. Balbo, Reaching Through Technology, CHI 1991 Conference Proceedings, Editors: S. P. Robertson, G. M. Olson, and J. S. Olson, New Orleans, LA, April 27-May 2, 1991, pp. 27-32.

[Cowan 1992] Program design using abstract data views – An illustrative example, D. D. Cowan *et al*, Technical Report 92-54, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, December 1992.

[Cowan 1993] Abstract Data Views, D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien, Structured Programming, Volume 14, January 1993, pp. 1-13.

[Cowan 1993a] Application Integration: Constructing composite applications from interactive components, D. D. Cowan, Software Practice and Experience, Volume 23, March 1993, pp. 255-276.

[Cowan 1993b] Abstract Data Views: A module interconnection concept to enhance design for reusability, D. D. Cowan and C. J. P. Lucena, Technical Report 93-52, Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, November 1993.

[Cowan 1995] Abstract Data Views: An interface specification concept to enhance design for reuse, Donald D. Cowan and Carlos J. P. Lucena, IEEE Transactions on Software Engineering, Volume 21, Number 3, March 1995, pp. 229-242.

[Dahmann 1997] Department of Defense High Level Architecture, Judith S. Dahmann, Winter Simulation Conference Proceeding, 1997. IEEE, Piscataway, NJ, USA 97CB36141, pp. 142-149.

[DEC 1991] The common object request broker: Architecture and specification, Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design Inc., and SunSoft Inc., OMG 91.12.1, December 1991.

[Dewey 1979] Decimal Classification and Relative Index, M. Dewey, 19th ed., Forest Press Inc., Albany, N.Y., 1979.

[Dennis 1973] Modularity, Advanced Course on Software Engineering, J. Dennis, Springer-Verlag, New York, 1973.

[DeRemer 1972] Programming-in-the-large versus programming-in-the-small, F. DeRemer and H. Kron, IEEE Transactions on Software Engineering, Volume 2, 1976.

[Derr 1995] Applying OMT: a practical step-by-step guide to using the object modeling technique, Kurt W. Derr, SIGS Books, New York, NY, 1995.

[Dick 2000] XML: a manager's guide, Kevin Dick, Addison Wesley Longman, Inc., Reading, Massachusetts, 2000.

[Dodd 1999] The essential guide to telecommunications, Annabel Z. Dodd, Prentice Hall, Inc., Upper Saddle River, NJ, 1999.

[Doh 1994] The facets of action semantics: Some principles and applications, Kyung-Goo Doh and David A. Schmidt, Proceedings 1st International Workshop on Action Semantics, Edinburg, 1994, pp. 1-15.

[Eeles 1998] Building business objects, Peter Eeles and Oliver Sims, John Wiley & Sons Inc., New York, NY, 1998.

[Even 1990] Category sorted algebra-based action semantics, Susan Even and David A. Schmidt, Theoretical Computer Science, Volume 77, 1990, pp. 73-96.

[Fiadeiro 1993] Verifying for Reuse: Foundations of Onbject-Oriented System Verification, J. Fiadeiro and T. Maibaum, Technical Report, Imperial College of Science and Technology, University of London, London, 1993.

[Enderton 1972] A mathematical Introduction to Logic, H. B. Enderton, Academic Press, New York, NY, 1972.

[Fowler 1999] Analysis patterns: reusable object models, Martin Fowler, Addison Wesley Longman, Inc., Reading, Massachusetts, October 1999.

[Freeman 1983] Reusable software engineering: Concepts and research directions, P. Freeman, Workshop on Reusability in Programming (Newport, R.I., Sept. 1983), ITT Programming, Stratford, Conn., pp. 2-16.

[Freeman 1987] A conceptual analysis of the Draco approach to constructing software systems, P. Freeman, IEEE Transactions on Software Engineering, SE-13, 7, July 1987, pp. 830-844.

[Frichman 1992] The Assimilation of Software Process Innovations: An Organizational Learning Perspective, R. Frichman and C. Kemerer, MIT Center for Information Systems Research WP 281, Management Science, 1992.

[Gamma 1996] Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley Publishing Company, Inc., Reading, MA, December 1996.

[Gaudel 1986] Towards Structured Algebraic Specifications, M. C. Gaudel, ESPRIT 85 Status Report, North-Holland, Brussels, 1986, pp. 493-510.

[Green 1983] Design notations and user interface management systems, M. Green, User Interface Management System, Proceedings on Workshop User Interface Management System, Seeheim, FRG, November 1-3, 1983.

[Hartson 1989] User-interface management control and communication, R. Hartson, IEEE Software, volume 26, January 1989, pp. 62-70.

[Harmon 1997] Understanding UML: the developer's guide: with a web-based application in Java, Paul Harmon and Mark Watson, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997.

[Helm 1990] Contracts: Specifying behavioral composition in object-oriented systems, R. Helm, I. M. Holland, and D. Gangopadhyay, OOPSLA, 1990, pp. 169-180.

[Henning 1999] Advanced CORBA Programming with C++, Michi Henning and Steve Vinoski, Addison Wesley Longman, Inc., Reading, Massachusetts, April 1999.

[Henry 1995] Large-scale industrial reuse to reduce cost and cycle time, Emmanuel Henry and Benoit Faller, IEEE Software, September 1995.

[Hill 1986] Supporting concurrency, communication, and synchronization in human-computer interaction – The Sassafras UIMS, R. D. Hill, ACM Transactions on Graphics, Volume 5, July 1986, pp. 179-210.

[Hill 1992] The abstraction-link view paradigm: using constraints to connect user interface to applications, R. D. Hill, CHI 1992, ACM, May 1992, pp. 335-342.

[Hoare 1969] An axiomatic approach to computer programming, C. A. R. Hoare, Communications of the ACM, Volume 12, 1969, pp. 576-580, 583.

[Iona 1999] Orbix Programmers' Manual, Iona Technologies, Inc, January 1999.

[Jacobson 1992] Object-Oriented Software Engineering – A Use Case Driven Approach, Ivar Jacobson, Magnus Christerson, Patrik Johnson, and Gunnar Overgaard, Addison-Wesley, Wokingham, England, 1992.

[Johnson 1991] Delegation in C++, Ralph E. Johnson and Jonathan Zweig, Journal of Object-Oriented Programming, Volume 4, Number 11, pp. 22-35, November 1991.

[Jones 1990] Systematic Software Development Using VDM, C. B. Jones, Prentice-Hall, New York, NY, 1990.

[Krasner 1988] A cookbook for using the model-view-controller user interface paradigm in smalltalk-80, G. E. Krasner, JOOP, August-September 1988, pp. 26-49.

[Krueger 1992] Software Reuse, Charles W. Krueger, ACM Computing Surveys, Volume 24, Number 2, June 1992.

[Levy 1986] A metaprogramming method and its economic justification, L. S. Levy, IEEE Transaction on Software Engineering, SE-12, Number 2, February 1986, pp.272-277.

[Lieberman 1986] Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems, H. Lieberman, OOPSLA'86 Conference Proceeding, SIGPLAN Notices, Volume 21, Number 11, pp. 214-223, 1986.

[Linthicum 1999] Enterprise application integration, David S. Linthicum, Addison Wesley Longman, Inc., Reading, Massachusetts, November 1999.

[Loeckx 1996] Specification of Abstract Data Types, Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf, John Wiley & Sons, Inc., New York, NY, 1996.

[Lucena 1992] A programming Model for User Interface Compositions, C. J. P. Lucena, D. D. Cowan, and A. B. Potengy, Technical Report 92-61, Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, March 1992.

[Lucena 1993] A programming approach for parallel rendering applications, C. J. P. Lucena, D. D. Cowan, and A. B. Potengy, Technical Report 93-62, Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, March 1993.

[Maarek 1991] An information retrieval approach for automatically constructing software libraries, Y. S. Maarek, D. M. Berry, and G. E. Kaiser, IEEE Transactions on Software Engineering, Volume 17, No. 8, August 1991, pp. 800-813.

[Marciniak 1994] Encyclopedia of Software Engineering, J.J. Marciniak, John Wiley & Sons, Inc., New York, NY, 1994.

[Martin 1997] Object-oriented methods: a foundation, James Martin and James Odell, Prentice-Hall, Inc., Upper Saddle River, NJ, 1997.

[Maruyama 2000] XML and Java: developing web applications, Hiroshi Maruyama, Kent Tamura, and Naohiko Uramoto, Addison Wesley Longman, Inc., Reading, Massachusetts, January 2000.

[Matsumoto 1987] A Software Factory: An overall approach to software production, IEEE Tutorial on Software Reusability, P. Freeman (editor), IEEE Computer Society Press, Los Alamitos, CA, 1987, pp. 155-178.

[McDysan 1999] ATM theory and applications, David E. McDysan and Darren L. Spohn, McGraw-Hill, New York, NY, 1999.

[McCormack 1988] An overview of the X toolkit, J. McCormack and P. Asente, Proceedings of the ACM SIGGRAPH Symposium on User Interface Software, October 1988, pp. 46-55.

[McIlory 1968] Mass Produced Software Components, M.D. McIlory, Software Engineering Concepts and Techniques, Brussels 39, Belgium: Pertrocelli/Charter, 1968, pp. 88-98. Paper presented at the 1968 NATO Conference on Software Engineering.

[Meyer 1988] Object-oriented software construction, B. Meyer, Prentice-Hall, 1988.

[Mosses 1992] Action Semantics, Peter D. Mosses, Cambridge Tracts in Theoretical Computer Science, Number 26, Cambridge University Press, 1992.

[Mosses 1996] Theory and practice of action semantics, Peter D. Mosses, MFCS: Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science: Volume 1113, Springer-Verlag, Cracow, Poland, September 1996, pp. 37-61.

[Mowbray 1997] Corba design patterns, Thomas J. Mowbray and Raphael Malveau, John Wiley & Sons, Inc., New York, NY, 1997.

[Mowbray 1998] Inside CORBA: Distributed object standards and applications, Thomas J. Mowbray and Willam A. Ruh, Addison Wesley Longman, Inc., Reading, MA, February 1998.

[MSC 1991] Microsoft Visual Basic Programmer's Guide, Microsoft Corporation, 1991.

[Myers 1991] Separating application code from toolkits: Eliminating the spaghetti of call-backs, B. A. Myers, UIST-4th Annual Symposium on User Interface Software Technology, 1991, pp. 211-220.

[Neighbors 1983] The Draco approach to constructing software from reusable components, J. M. Neighbors, Workshop on reusability in programming (Newport, R.I., September 1983), ITT Programming, Conn., pp. 167-178.

[Neighbors 1989] Draco: A method for engineering reusable software systems, Frontier Series: Software Reusability: Concepts and Models, Volume 1, Ted J. Biggerstaff and Alan J. Perlis (editors), ACM Press, New York, NY, 1989, pp. 295-319, Chapter 12.

[Nierstrasz 1992] Component-oriented software development, O Nierstrasz, S. Gibbs, and D. Tsichritzis, Communications of the ACM, Volume 35, September 1992, pp. 160-165.

[OG 1994] Distributed Computing Services (XDCS) Framework: X/Open Framework and Models, The Open Group, October 1994.

[Orfali 1998] Client/server programming with Java and CORBA, Robert Orfali and Dan Harkey, John Wiley & Sons, Inc., New York, NY, 1998.

[Olsen 1983] Presentational, syntactic, and semantic components of interactive dialogue specifications, D. R. Jr. Olsen, User Interface Management System, Proceedings on Workshop User Interface Management System, Seeheim, FRG, November 1-3, 1983.

[OMG 1997] A Discussion of the Object Management Architecture, OMG, January 1997.

[OMG 1998] CORBA Finance: Financial Domain Specifications: Version 1.0, OMG, December 1998.

[OMG 1998a] CORBA Telecoms: Telecommunications Domain Specifications: Version 1.0, OMG, June 1998.

[OMG 2000] Meta Object Facility (MOF) Specification: Version 1.3, OMG, March 2000.

[OMG 2000a] OMG XML Metadata Interchange (XMI) Specification: Version 1.1, OMG, November 2000.

[OMG 2000b] OMG Unified Modeling Language (UML) Specification: Version 1.3, OMG, March 2000.

[OMG 2001] The Common Object Request Broker Architecture: Architecture and Specification: Version 2.4.2, OMG, February 2001.

[OSF 1990] Application Environment Specification (AES) User Environment Volume, Open Software Foundation, 1990.

[Ousterhoust 1994] Tcl and Tk Toolkit, J. K. Ousterhoust, Reading, MA: Addison-Wesley, 1994.

[Parnas 1989] Enhancing Reusability with Information Hiding, Frontier Series: Software Reusability: Concepts and Models, Volume 1, D.L. Parnas, P.C. Clements, and D. M. Weiss, Ted J. Biggerstaff and Alan J. Perlis (editors), ACM Press, New York, NY, 1989, pp. 141-157.

[Potengy 1993] A Programming Approach for Parallel Rendering Applications, A. B. Potengy, C. J. P. Lucena, and D. D. Cowan, Technical Report 93-62, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, March 1993.

[Poulin 1995] Populating software repositories: incentives and domain-specific software, J. S. Poulin, Journal of Systems and Software, Volume 30, Number 3, September 1995, pp. 187-199.

[Prieto-Diaz 1985] A software classification scheme: Ph.D. thesis, R. Prieto-Diaz, Department of Information and Computer Science, University of California, Irvine, 1985.

[Prieto-Diaz 1986] Module Interconnection Languages, R. Prieto-Diaz and J.M. neighbors, Journal of System Software, Volume 6, Number 4, November 1986, pp. 307-334.

[Prieto-Diaz 1987] Classifying Software for Reusability, R. Prieto-Diaz and P. Freeman, IEEE Software, Volume 4, Number 1, January 1987, pp. 6-16.

[Prieto-Diaz 1989] Classification of Reusable Modules, Software Reusability: Concepts and Models, Volume 1, Ted J. Biggerstaff and Alan J. Perlis (editors), ACM Press, New York, NY, 1989, pp. 99-123.

[Prieto-Diaz 1991] Domain Analysis and Software System Modeling, R. Prieto-Diaz and G. Arango, IEEE Computer Society Press, Los Alamitos, 1991.

[Prieto-Diaz 1991a] Implementing Faceted Classification for Software Reuse, R. Prieto-Diaz, Communications of the ACM, Volume 34, Number 5, May 1991, pp. 89-97.

[Rising 1998] The Patterns Handbook: Techniques, Strategies, and Applications, collected by Linda Rising, Cambridge University Press, Cambridge, UK, 1998.

[Rumbaugh 1991] Object-Oriented Modeling and Design, J. Rumbaugh et al., Prentice Hall, Englewood Cliffs, NJ, 1991, pp. 156-161.

[Schmidt 1995a] An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Systems, Douglas C. Schmidt and Tatsuya Suda, BCS/IEE Distibuted Systems Engineering Journal, 1995.

[Schmidt 1995b] Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching, Douglas C. Schmidt, Pattern Languages of Program design, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1995.

[Schmidt 1995c] Object-Oriented Components for High Speed Network Programming, D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, Proceedings of the Conference on Object-Oriented Technologies, (Monterey, CA), USENIX, June 1995.

[Schmidt 1996] A Family of Design Patterns for Flexibly Configuring Network Services in Distributed Systems, Douglas C. Schmidt, International Conference on Configurable Distributed Systems, Annapolis, Maryland, May 6-8, 1996.

[Schmidt 1996a] A Family of Design Patterns for Application-Level Gateways, Douglas C. Schmidt, Theory and Practice of Object Systems, Wiley & Sons, Volume 2, Number 1, December 1996.

[Schmidt 2000] Pattern-Oriented Software Architecture: Patterns for Concurrent and Network Objects: Volume 2, Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, John Wiley & Sons, Inc., New York, NY, 2000.

[Shaw 1984] Abstraction techniques in modern programming languages, M. Shaw, IEEE Software, Volume 1, Number 4, October 1984, pp. 10-26.

[Shaw 1989] Larger scale systems require higher-level abstractions, M. Shaw, In Proceedings of the 5th International Workshop on Software Specification and Design, IEEE Computer Society Press, Los Alamitos, Calif., May 1989, pp.143-146.

[Shaw 1991] Heterogeneous design idioms for software architecture, M. Shaw, In Proceedings of the 6th International Workshop on Software Specification and Design (Como, Italy), IEEE Computer Society Press, Los Alamitos, Calif., October 1991, pp.143-146.

[Shepard 2000] Telecommunications Convergence, Steven Shepard, McGraw-Hill, New York, NY, 2000.

[Shlaer 1988] Object-oriented systems analysis: modeling the world in data, Sally Shlaer and Stephen J. Mellor, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992

[Shlaer 1992] Object lifecycles: modeling the world in states, Sally Shlaer and Stephen J. Mellor, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992

[Sikkel 1992] Abstract data types as reusable software components: the case for twin ADTs, K. Sikkel and J. C. van Vliet, Software Engineering Journal, May 1992, pp. 177-183

[Stroustrup 1986] The C++ Programming Language, B. Stroustrup, Addison-Wesley Publishing Company, 1986.

[Szyperski 1998] Component Software: Beyond Object-Oriented Programming, Clemens Szyperski, ACM Press, New York, NY, 1998.

[Summerville 1989] Software Design with Reuse, I. Summerville, J. Mariani, N. Haddley, and R. Thomson, Internal Report, Department of Computing, Lancaster University, Bailrigg, lancaster, 1989.

[Sun 1996] Term rewriting and Hoare logic – coded rewriting, Yong Sun, Information Processing Letters, Volume 60, Number 5, December 9, 1996, pp. 237-242.

[Texel 1997] Use cases combined with BOOCH/OMT/UML: process and products, Tutnam Texel and Charles Williams, Prentice-Hall, Inc., Upper Saddle River, NJ, 1997.

[TMF 1999] Telecom Operations Map: Evaluation Version 1.1, TeleManagement Forum, Morristown, NJ, April 1999.

[TMF 1999a] Network Management Detailed Operations Map: Evaluation Version 1.1, TeleManagement Forum, Morristown, NJ, March 1999.

[Tracz 1987] Software Reuse: motivators and inhibitors, W. J. Tracz, Proceedings of COMPCON, San Francisco, CA, February 1987, pp. 358-363.

[Turski 1987] The Specification of Computer Programs, W. M. Turski and T. S. E. Maibaum, Addison-Wesley, New York, 1987.

[Udupa 1999] TMN: Telecommunications Management Network, Divakara K. Udupa, McGraw-Hill, New York, NY, 1999.

[Vogel 1998] Java programming with CORBA, Andreas Vogel and Keith Duddy, John Wiley & Sons, Inc., New York, NY, 1998.

[Wang 1999] Telecommunications Network Management, Haojin Wang, McGraw-Hill, New York, NY, 1999.

[Wegner 1983] Varieties of Reusability, P. Wegner, Workshop on reusability in programming, ITT programming, Stratford, Conn., 1983, pp. 30-44.

[Wegner 1987] Dimensions of Object-Based Language Design, P. Wegner, OOPSLA'87 Conference Proceedings, SIGPLAN Notices (Special Issue), Volume 22, Number 12, 1987, pp. 168-182.

[WIC 1993] Watcom VX-REXX for OS/2 Programmer's Guide and Reference, Watcom Int. Corporation, Waterloo, Ontario, Canada, 1993.

[Zahavi 1999] Enterprise Application Integration with CORBA, Ron Zahavi, John Wiley & Sons, Inc., New York, NY, 1999.

[Zave 1984] The operational versus the conventional approach to software development, P. Zave, Communications of the ACM, Volume 27, Number 2, February 1984, pp. 104-118.

[Zilles 1974] Algebraic Specification of Data Types, S. Zilles, Project MAC Progress Report 11, MIT, 1974.