# The Object Primer

## Third Edition

## Agile Model–Driven Development With UML 2.0

**Scott W. Ambler**
Ronin International, Inc.

Cambridge
University Press

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

© Cambridge University Press 2004

This book is in copyright. Subject to statutory exception
and to the provisions of relevant collective licensing agreements,
no reproduction of any part may take place without
the written permission of Cambridge University Press.

First published 2004

Printed in the United States of America

*Typefaces* ITC Berkeley Oldstyle 11/13.5 pt. and ITC Franklin Gothic *System* LaTeX $2_\varepsilon$ [TB]

*A catalog record for this book is available from the British Library.*

*Library of Congress Cataloging in Publication data available*

ISBN 0 521 54018 6 paperback

# Contents


**Acknowledgments** *page* xvii

**Foreword** xix

**Preface** xxi

**About the Author** xxv

**Chapter 1**
**Leading-Edge Software Development** **1**

1.1 Modern Development Technologies 2
  1.1.1 Object Technology 2
  1.1.2 Extensible Markup Language (XML) 5
  1.1.3 Relational Database (RDB) Technology 6
  1.1.4 Web Services 7
1.2 Modern Development Techniques 8
  1.2.1 Agile Software Development 9
  1.2.2 Unified Modeling Language (UML) 11
  1.2.3 The Unified Process (UP) 13
  1.2.4 Model-Driven Architecture (MDA) 14
  1.2.5 Using Them Together 15
1.3 The Organization of This Book 16
1.4 The Case Studies 19
  1.4.1 The Bank Case Study 20
1.5 What You Have Learned 22

## Chapter 2
## Understanding the Basics–Object-Oriented Concepts 23

2.1 A Brief Overview of OO Concepts 24
2.2 OO Concepts from a Structured Point of View 27
2.3 The Diagrams of UML 2 28
2.4 Objects and Classes 28
2.5 Attributes and Operations/Methods 32
2.6 Abstraction, Encapsulation, and Information Hiding 34
2.6.1 Abstraction 34
2.6.2 Encapsulation 34
2.6.3 Information Hiding 35
2.6.4 An Example 35
2.7 Inheritance 37
2.7.1 Modeling Inheritance 38
2.7.2 Inheritance Tips and Techniques 39
2.7.3 Single and Multiple Inheritance 40
2.7.4 Abstract and Concrete Classes 43
2.8 Persistence 43
2.9 Relationships 44
2.9.1 Associations 45
2.9.2 Modeling the Unknown 48
2.9.3 How Associations Are Implemented 49
2.9.4 Properties 50
2.9.5 Aggregation and Composition 50
2.9.6 Dependencies 52
2.10 Collaboration 54
2.11 Coupling 57
2.12 Cohesion 58
2.13 Polymorphism 59
2.13.1 An Example: The Poker Game 59
2.13.2 Polymorphism at the University 60
2.14 Interfaces 62
2.15 Components 63
2.16 Patterns 65
2.17 What You Have Learned 66
2.18 Review Questions 67

**Chapter 3**
**Full Lifecycle Object-Oriented Testing (FLOOT) 68**

3.1 The Cost of Change 69
3.2 Testing Philosophies 74
3.3 Full Lifecycle Object-Oriented Testing (FLOOT) 75
3.4 Regression Testing 78
3.5 Quality Assurance 79
3.6 Testing Your Models 80
3.6.1 Proving It with Code 80
3.6.2 Usage Scenario Testing 81
3.6.3 Prototype Reviews/Walkthroughs 84
3.6.4 User-Interface Testing 85
3.6.5 Model Reviews 85
3.6.6 When to Use Each Technique 87
3.7 Testing Your Code 88
3.7.1 Testing Terminology 88
3.7.2 Testing Tools 89
3.7.3 Traditional Code Testing Concepts 89
3.7.4 Object-Oriented Testing Techniques 92
3.7.5 Code Inspections 94
3.8 Testing Your System in Its Entirety 95
3.9 Testing by Users 96
3.10 Test-Driven Development (TDD) 97
3.11 What You Have Learned 99
3.12 Review Questions 99

**Chapter 4**
**Agile Model–Driven Development (AMDD) 101**

4.1 Modeling Philosophies 102
4.2 Project Stakeholders 106
4.3 What Is Agile Modeling (AM)? 107
4.4 The Values of AM 108
4.5 The Principles of AM 109
4.6 The Practices of AM 109
4.7 Easing into Agile Modeling 109
4.8 Agile Model–Driven Development (AMDD) 118
4.8.1 How is AMDD Different? 120

4.9 Fundamental Information Gathering Skills 121
4.9.1 Interviewing 121
4.9.2 Observation 123
4.9.3 Brainstorming 123
4.10 Agile Documentation 124
4.11 Making Whiteboards Work for Software Development 125
4.12 AMDD and Other Agile Methodologies 129
4.13 Agile Modeling and Test-Driven Development (TDD) 129
4.14 What You Have Learned 132
4.15 Review Questions 132

**Chapter 5**
**Usage Modeling 134**

5.1 Use Case Modeling 134
5.1.1 Starting Agile 136
5.1.2 Essential Use Case Diagrams 139
5.1.3 Identifying Actors 146
5.1.4 Writing an Essential Use Case 149
5.1.5 Identifying Use Cases 151
5.1.6 System Use Case Diagrams 153
5.1.7 System Use Cases 153
5.1.8 Writing Alternate Courses of Action 160
5.1.9 Other Use Case Styles 163
5.1.10 Comparing Essential and System Use Cases 164
5.1.11 Reuse in Use Case Models: <<extend>>, <<include>>, and Inheritance 167
5.1.12 Packages 172
5.1.13 Use Case Modeling Tips 173
5.1.14 Remaining Agile 176
5.2 User Stories 177
5.2.1 What About System User Stories? 180
5.3 Features 180
5.4 What You Have Learned 182
5.5 Review Questions 183

**Chapter 6**
**User-Interface Development** **184**

6.1 Essential User-Interface Prototyping 185
6.2 Traditional User-Interface Prototyping 191
6.3 User-Interface Flow Diagramming 197
6.4 Usability 199
6.5 User-Interface Design Strategies 200
6.6 Agile Stakeholder Documentation 205
6.7 What You Have Learned 207
6.8 Review Questions 207

**Chapter 7**
**Supplementary Requirements** **209**

7.1 Business Rules 210
7.2 Technical Requirements 214
7.3 Constraints 215
7.4 Object Constraint Language (OCL) 216
7.5 Glossaries 217
7.6 Supplementary Specifications 218
7.7 What You Have Learned 218
7.8 Review Questions 219

**Chapter 8**
**Conceptual Domain Modeling** **220**

8.1 Robustness Diagrams 221
8.2 Object Role Model (ORM) Diagrams 227
8.3 Class Responsibility Collaborator (CRC) Cards 231
8.4 Analysis Class Diagrams 237
8.4.1 Modeling Classes and Responsibilities 241
8.4.2 Modeling Associations 245
8.4.3 Introducing Reuse between Classes via Inheritance 247
8.4.4 Modeling Composition and Associations 249
8.4.5 Modeling Vocabularies 250
8.5 Logical Data Models (LDMs) 251
8.6 Applying Analysis Patterns Effectively 254

8.7 UML Object Diagram 256
8.8 Keeping Conceptual Domain Modeling Agile 258
8.9 What You Have Learned 259
8.10 Review Questions 260

**Chapter 9**
**Business Process Modeling 262**

9.1 Data Flow Diagrams (DFDs) 263
9.2 Flowcharts 268
9.3 UML Activity Diagrams 270
9.4 What You Have Learned 277
9.5 Review Questions 277

**Chapter 10**
**Agile Architecture 278**

10.1 Architecture Techniques and Concepts 280
10.1.1 Put Architectural Decisions Off as Long as Possible 280
10.1.2 Accept That Some Architectural Decisions Are Already Made 281
10.1.3 Prove It with Code 281
10.1.4 Set an Architectural Change Strategy 282
10.1.5 Consider Reuse 283
10.1.6 Roll Up Your Sleeves 284
10.1.7 Be Prepared to Make Trade-offs 285
10.1.8 Consider Adopting the Zachman Framework 285
10.1.9 Apply Architectural Patterns Gently 289
10.2 Looking to the Future with Change Cases 289
10.3 UML Package Diagrams 291
10.3.1 Class Package Diagrams 292
10.3.2 Data Package Diagrams 294
10.3.3 Use Case Package Diagrams 294
10.4 UML Component Diagrams 296
10.4.1 Interfaces and Ports 298
10.4.2 Designing Components 300
10.4.3 Creating Component Models 302
10.4.4 Remaining Agile 306

10.5 Free-Form Diagrams 306
10.6 UML Deployment Diagrams 308
10.7 Network Diagrams 313
10.8 Layering Your Architecture 314
10.9 What You Have Learned 317
10.10 Review Questions 317

**Chapter 11**
**Dynamic Object Modeling 319**

11.1 UML Sequence Diagrams 321
11.1.1 Visual Coding Via Sequence Diagrams 332
11.1.2 How to Draw Sequence Diagrams 333
11.1.3 Keeping It Agile 334
11.2 UML Communication Diagrams 334
11.3 UML State Machine Diagrams 337
11.4 UML Timing Diagrams 344
11.5 UML Interaction Overview Diagrams 346
11.6 UML Composite Structure Diagrams 348
11.7 What You Have Learned 350
11.8 Review Questions 350

**Chapter 12**
**Structural Design Modeling 351**

12.1 UML Class Diagrams 351
12.1.1 Modeling Methods during Design 352
12.1.2 Modeling Attributes during Design 360
12.1.3 Inheritance Techniques 366
12.1.4 Association and Dependency Techniques 368
12.1.5 Composition Techniques 372
12.1.6 Introducing Interfaces into Your Model 373
12.1.7 Class Modeling Design Tips 376
12.2 Applying Design Patterns Effectively 380
12.2.1 The *Singleton* Design Pattern 380
12.2.2 The *Façade* Design Pattern 381
12.2.3 Tips for Applying Patterns Effectively 382
12.3 Physical Data Modeling with the UML 383
12.4 What You Have Learned 390

12.5 Review Questions 390
12.5.1 The Bank Case Study Six Months Later 391

**Chapter 13**
**Object-Oriented Programming 393**

13.1 Philosophies for Effective Programming 394
13.2 Programming Tips and Techniques for Writing High-Quality Code 397
13.3 Test-Driven Development (TDD) 400
13.3.1 TDD and AMDD 402
13.3.2 Why TDD? 403
13.4 From Object Design to Java Code 404
13.4.1 From UML Sequence Diagrams to Code 404
13.4.2 From UML Class Diagrams to Code 406
13.4.3 Implementing a Class in Java 408
13.4.4 Declaring Instance Attributes in Java 409
13.4.5 Implementing Instance Methods in Java 410
13.4.6 Implementing Static Methods and Attributes in Java 412
13.4.7 Documenting Methods 417
13.4.8 Implementing Constructors 419
13.4.9 Encapsulating Attributes with Accessors 420
13.4.10 Implementing Inheritance in Java 426
13.4.11 Implementing Interfaces in Java 426
13.4.12 Implementing Relationships in Java 429
13.4.13 Implementing Dependencies 438
13.4.14 Implementing Collaborations in Java 439
13.4.15 Implementing Business Rules 439
13.4.16 Iterate, Iterate, Iterate 440
13.5 What You Have Learned 440
13.6 Review Questions 440

**Chapter 14**
**Agile Database Development 442**

14.1 Philosophies for Effective Data Development 444
14.2 Mapping Objects to Relational Databases 445

14.2.1 Shadow Information 446
14.2.2 Mapping Inheritance Structures 447
14.2.3 Mapping Relationships 451
14.3 Strategies for Implementing Persistence Code 453
14.4 From Design to Database Code 455
14.4.1 Defining and Modifying Your Persistence Schema 455
14.4.2 Creating, Retrieving, Updating, and Deleting Data 456
14.4.3 Interacting with a Database from Java 458
14.4.4 Implementing Your Mappings 460
14.5 Data-Oriented Implementation Strategies 460
14.5.1 Concurrency Control 462
14.5.2 Transaction Control 464
14.5.3 Shared Logic and Referential Integrity 466
14.5.4 Security Access Control 471
14.5.5 Searching for Objects 473
14.5.6 Reports 476
14.6 Database Refactoring 477
14.7 Legacy Analysis 481
14.7.1 Formalizing Contract Models 482
14.7.2 Common Legacy Challenges 483
14.7.3 Creating Contract Models 484
14.8 What You Have Learned 485
14.9 Review Questions 486

**Chapter 15**
**Where to Go from Here 487**

15.1 Become a Generalizing Specialist 487
15.2 Continuing Your Learning Process 490
15.3 Parting Words 492

**Glossary 493**

**References and Recommended Reading 525**

**Index 533**

# Chapter 1

# Leading-Edge Software Development

*Modern software development requires modern ways of working.*

The only constant in the information technology (IT) industry is change. To remain employable, let alone effective, software developers must continually take the time to identify and then understand the latest development approaches. The goal of this chapter is to introduce you to leading-edge technologies and techniques that enable you to succeed at developing modern business systems. I will try to steer you through the marketing hype surrounding these approaches, and in one case try to dissuade you from adopting it—just because something is new and well hyped does not mean that it has much of a future. In short, this chapter provides you with a foundation for reading the rest of this book.

This chapter discusses

- Modern development technologies;
- Modern development techniques;
- How this book is organized; and
- The case studies.

## 1.1 Modern Development Technologies

Effective developers understand the fundamentals of the technologies that they have available to them. The good news is that we have many technologies available to us; the bad news is that we have many technologies available to us.

Figure 1.1, which depicts a high-level architecture detailing how these technologies are used together, shows how some applications may be *n*-tiered—an approach where application logic is implemented on several (*n*) categories of computing devices (tiers)—whereas others fall into the "fat client" approach where most business logic is implemented on the client. Object technology is used to implement all types of logic, including both business and system logic. XML is used to share data between tiers, and Web services are used to access logic that resides on different tiers. Most business data are stored in relational databases, which are accessed either via structured query language (SQL) or persistence frameworks (see Chapter 14). Data are returned from the database as a collection of zero or more records and then marshaled either into objects or into XML documents. In the case of a browser-based application the XML structures are in turn converted into HTML documents, often through XSL-T (extensible stylesheet language transformations).

Although the focus of this book is the development of business systems, much of the advice is also applicable to the development of other types of software. My specialty is business software so that is what I will stick to in this book. My experience is that when it comes to building modern business systems, you are very likely to use a combination of the following:

- Object technology;
- Extensible markup language (XML);
- Relational database (RDB); and
- Web services.

### 1.1.1 Object Technology

The object-oriented (OO) paradigm (pronounced "para-dime") is a development strategy based on the concept that systems should be built from a collection of reusable parts called objects. Examples of OO languages and

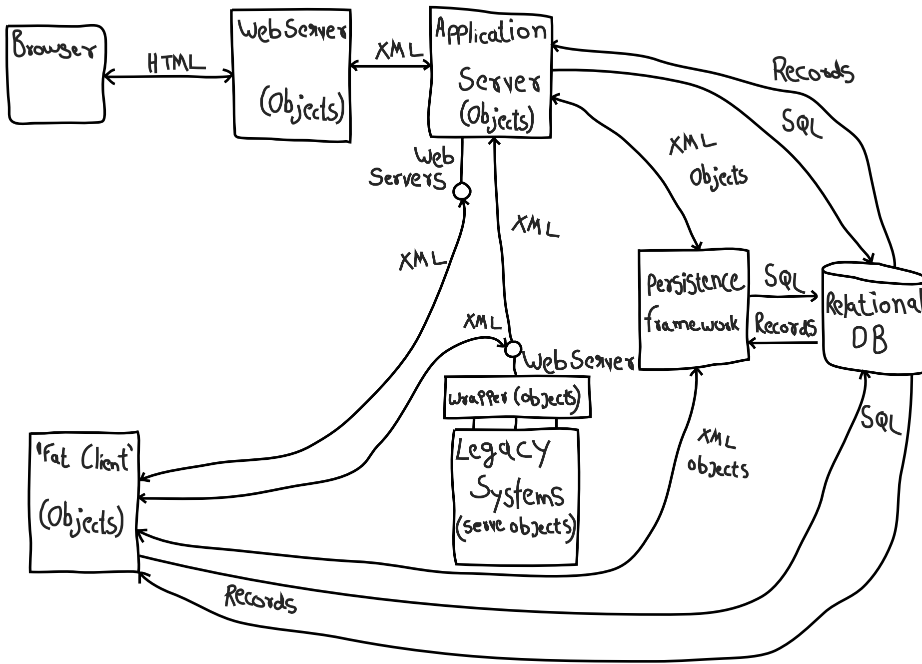


**FIGURE 1.1. High-level application architecture.**

technologies include the Java, C#, and C++ programming languages and the Enterprise JavaBeans (EJB) framework. The original motivation of the object paradigm was that objects were meant to be abstractions of real-world concepts, such as students in a university, seminars that students attend, and transcripts that they receive. This was absolutely true of business objects, but as you will see throughout this book, business objects are only one part of the picture—you also need user interface objects to enable your users to work with your system, process objects that implement logic that works with several business concepts, system objects that provide technical features such as security and messaging, and potentially some form of data objects that persists your business objects.

The use of object technology can be in fact quite robust. In fat client applications object technology is typically used on client machines, such as personal computers or personal digital assistants (PDAs), to implement both user interface code and complex business logic. In thin-client or *n*-tier applications object technology is often used to implement business logic on application

**Table 1.1. Evaluating Object Technology**

| Advantages | Disadvantages |
|---|---|
| • Enables development of complex software<br>• Wide industry acceptance<br>• Mature, proven technology<br>• Wide range of development languages and tools to choose from<br>• Very easy to find people with object experience | • Significant skillset is required<br>• No single language dominates the landscape (although Java, C#, C++, and arguably Visual Basic are clearly popular and here to stay)<br>• Not all IT professionals, in particular some within the data community, accept it<br>• Technical "impedance mismatch" with structured technologies and RDBs |

servers and sometimes even on other nodes such as database servers, security servers, or business rule servers. Table 1.1 summarizes the strengths and weaknesses of object technology for business system development.

It is useful to contrast these concepts with the structured paradigm and structured technology. The structured paradigm is a development strategy based on the concept that a system should be separated into two parts: data (modeled using a data model) and functionality (modeled using a process model). Following the structured approach, you develop applications in which data are separate from behavior both in the design model and in the system implementation (that is, the program). Examples of structured technologies include the COBOL and FORTRAN programming languages. The main concept behind the object-oriented paradigm is that instead of defining systems as two separate parts (data and functionality), you now define systems as a collection of interacting objects. Objects do things (that is, they have functionality) and they know things (they have data). While this sounds similar to the structured paradigm, in practice it actually is quite different.

However, it is equally important to recognize that structured techniques and technologies still have their place. As you will see later in Section 1.1.4, it is quite common to transform legacy systems, typically implemented with structured technologies, and then wrap them with Web services to reuse their functionality. Furthermore, in coming chapters you will discover that structured

```
<office>
  <name>Ronin International, Inc. HQ</office:name>
  <state>
    <name>Colorado</state:name>
    <area>North West</state:area>
  </state>
  <country>United States of America</country>
</office>
```

**FIGURE 1.2. An example of an XML document.**

modeling techniques such as data flow diagrams (DFDs) and data models are still critical to your success.

## 1.1.2 Extensible Markup Language (XML)

XML is a subset of standard generalized markup language (SGML), the same parent of hypertext markup language (HTML). The critical standards are described in detail at the World Wide Web Consortium Web site (http://www.w3c.org). XML is simply a standardized approach to representing text-based data in a hierarchical manner and for defining metadata about the data. From a programmer's point of view XML is a data representation, backed by metadata, plus a collection of standardized technologies for parsing that data. The data are stored in structures called XML documents and the metadata are contained in document-type definitions (DTDs) and XML schema definitions. Figure 1.2 provides an example of a simple XML document and Table 1.2 overviews the advantages and disadvantages of XML.

XML is often used to transfer data within an application when that application has been deployed across several physical servers. It is also used in enterprise application integration (EAI) as a primary means of sharing data between applications. You will also see XML used for permanent storage in data files; it is quite common to use XML for configuration files in both J2EE and .NET applications, and sometimes even in databases. Chapter 14 discusses database issues in more detail, and you will see at that point that it is often better to "shred" an XML document into individual columns instead of saving it as a single column when storing data in a relational database.

**Table 1.2. Evaluating XML Technology**

| Advantages | Disadvantages |
|---|---|
| • XML is widely accepted<br>• XML is cross platform<br>• (Small) XML documents are potentially human readable<br>• XML is standards based; the World Wide Web Consortium defines and promotes technical standards for XML and XML.org (http://www.xml.org) promotes vertical XML standards within specific industries<br>• XML separates content from presentation | • XML documents are very bulky, causing performance problems<br>• XML requires marshaling (conversion of XML to objects and vice versa), causing performance problems<br>• XML standards are still evolving<br>• XML is overhyped, resulting in unrealistic expectations<br>• XML business standards will prove elusive because most businesses compete and do not collaborate with their industry peers |

### 1.1.3 Relational Database (RDB) Technology

A relational database is a persistent storage mechanism that stores data as rows in tables. Most relational databases enable you to implement functionality in them as stored procedures, triggers, and even full-fledged Java objects. Although other alternatives to RDBs exist—object-oriented database management systems (OODBMSs), XML databases (XDBs), and object-relational databases (ORDBs)—the fact is that RDBs are the database technology of choice for the vast majority of organizations. Table 1.3 summarizes the pros and cons of using RDBs for modern business applications.

It is important to understand that there is a technical impedance mismatch between RDBs and other common implementation technologies. When it comes to RDBs and objects, RDBs are based on mathematical principles, whereas objects are based on software engineering principles (Ambler 2003a). The end result is that you need to learn how to map your objects into RDBs as well as how to use the two technologies together, the topic of Chapter 14. Similarly, there is a difference between XML and RDBs—XML structures are hierarchical trees, whereas RDB table structures are “bushier” in nature.

**Table 1.3. Evaluating RDB Technology**

| Advantages | Disadvantages |
| --- | --- |
| • Wide industry acceptance<br>• Very easy to find RDB expertise<br>• Mature industry dominated by several strong vendors (Oracle, IBM, Sybase, Microsoft)<br>• Open source databases, for example, MySQL, are available<br>• Wide range of development tools<br>• Sophisticated and flexible data processing are supported | • Impedance mismatch with other common technologies, in particular objects and XML |

This requires you either to write marshaling code that maps individual XML elements to table columns, degrading performance, or to simply store XML documents in a single column, negating many of the benefits of RDBs.

### 1.1.4 Web Services

According to the World Wide Web Consortium, a Web service is "a software application identified by a Uniform Resource Identifier (URI), whose interface and bindings are capable of being identified, described, and discovered by XML artifacts and supports direct interactions with other software applications using XML-based messages via Internet-based protocols." Whew! An easier definition is that a Web service is a function that is accessible using standard Web technologies in accordance to standards (McGovern et al. 2003). The Web Services Interoperability Organization (WS-I, http://www.ws-i.org) is a consortium of mostly vendor companies that focus on Web services standards.

Web services are being used to implement functionality that is accessible via Internet technologies, often following an approach referred to as utility computing where the use of a computing service is charged for by the vendor on a usage basis much as electricity or water is charged for. It is far more common to use Web services "behind the firewall" to implement reusable functionality or to wrap legacy systems, including both programs and databases, so that they may be reused by other applications. Internal Web services such as this

**Table 1.4. Evaluating Web Service Technology**

| Advantages | Disadvantages |
|---|---|
| • Promotes reusability through a standardized approach<br>• Supports location transparency through a UDDI server<br>• Contains scaleable architecture<br>• Enables you to reduce dependency on vendors | • Searching for services via UDDI is time consuming<br>• Overhead of XML detracts from performance<br>• Does not yet support transaction control (this is coming)<br>• Does not yet support security (this is coming) |

are often managed within an internal UDDI (universal description, discovery, and integration) registry or better yet a reuse repository such as Flashline (http://www.flashline.com). A system built from a collection of cohesive services has a service-oriented architecture (SOA). Table 1.4 summarizes the advantages and disadvantages of Web services.

## 1.2 Modern Development Techniques

Now that we understand the fundamentals of modern technologies, we should now consider modern development techniques. The IT industry is currently undergoing what I consider to be a significant shift—a move from prescriptive development techniques to agile techniques. Until just recently management often bemoaned the fact that developers did not want to follow a process, not understanding what was wrong with the 3,000 pages of procedures they expected everyone to follow. Along came agile software processes such as extreme programming (XP) (Beck 2000), feature-driven development (FDD) (Palmer and Felsing 2002), and agile modeling (Ambler 2002) and developers embraced them. Unfortunately many managers are still leery of agile techniques and fight adoption of them. This is a truly ironic situation—developers are now demanding to follow proven software processes yet are not being allowed to do so. Sigh.

In this section I briefly explore four important development techniques that all developers should be familiar with:

- Agile software development;
- Unified modeling language (UML);
- The unified process; and
- Model-driven architecture (MDA).

### 1.2.1 Agile Software Development

Over the years several challenges have been discovered with prescriptive software development processes, such as the waterfall lifecycle characterized by the ISO 12207 standard (http://www.ieee.org), the Object-Oriented Software Process (OOSP) (Ambler 1998b, 1999), and the Rational Unified Process (RUP) (Kruchten 2000). First, the Chaos report published by the Standish Group (http://www.standishgroup.com) still shows a significant failure rate within the industry, indicating that prescriptive processes simply are not fulfilling their promise. Second, most developers do not want to adopt prescriptive processes and will find ways to undermine any efforts to adopt them, either consciously or subconsciously. Third, the "big design up front" (BDUF) approaches to software development, particularly those followed by ISO 12207, are incredibly risky due to the fact that they do not easily support change or feedback. This risk is often ignored, if it is recognized at all, by the people promoting these approaches. Fourth, most prescriptive processes promote activities only slightly related to the actual development of software. In short, the bureaucrats have taken over.

To address these challenges a group of 17 methodologists formed the Agile Software Development Alliance (http://www.agilealliance.org), often referred to simply as the Agile Alliance, in February 2001. An interesting thing about this group is that they all came from different backgrounds, and yet they were able to come to an agreement on issues that methodologists typically do not agree upon. They concluded that to succeed at software development you need to focus on people-oriented issues and follow development techniques that readily support change. In fact, they wrote a manifesto (Agile Alliance 2001a) defining four values for encouraging better ways of developing software:

1. **Individuals and interactions *over* processes and tools.** The most important factors that you need to consider are the people and how they work together because if you do not get that right the best tools and processes will not be of any use.