Contents

Preface ................................................................. xiii
Acknowledgments .................................................. xix
About the Author .................................................. xxi
Introduction ......................................................... xxiii

Part I: The Roots of Object-Oriented Development ..... 1

Chapter 1: Historical Perspective ............................... 3
   History ............................................................ 3
   Structured Development ....................................... 5
      Functional Decomposition ................................. 7
   Lessons Hard Won ............................................. 12
      Global Data ................................................. 12
      Elephantine Program Units ............................... 12
      Software Structure ........................................ 13
      Lack of Cohesion ......................................... 14
      Coupling ..................................................... 14
   Technical Innovation ......................................... 17
      The Turing Machine ....................................... 17
      Languages and Methodologies ........................... 18
   Sets and Graphs ............................................... 21
   Normal Form (NF) ............................................. 22
   Data Flows ..................................................... 25
   State Machines ............................................... 27

Chapter 2: Object Technology .................................. 29
   Basic Philosophy .............................................. 30
      Maintainability .......................................... 30
   Problem Space Abstraction ................................ 31
   OOA, OOD, and OOP ........................................ 32
<table>
<thead>
<tr>
<th>Subject Matter</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separation of Concerns</td>
<td>34</td>
</tr>
<tr>
<td>Levels of Abstraction</td>
<td>35</td>
</tr>
<tr>
<td>Problem Space Abstraction</td>
<td>37</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>39</td>
</tr>
<tr>
<td>Cohesion</td>
<td>40</td>
</tr>
<tr>
<td>Logical Indivisibility</td>
<td>40</td>
</tr>
<tr>
<td>Communication Models</td>
<td>42</td>
</tr>
<tr>
<td>Breadth-First Processing (aka Peer-to-Peer Collaboration)</td>
<td>44</td>
</tr>
<tr>
<td>Elaboration versus Translation</td>
<td>45</td>
</tr>
<tr>
<td>The Message Paradigm</td>
<td>47</td>
</tr>
<tr>
<td>Object Characteristics</td>
<td>49</td>
</tr>
<tr>
<td>Chapter 3: Generalization, Inheritance, Genericity, and Polymorphism</td>
<td>53</td>
</tr>
<tr>
<td>Generalization</td>
<td>54</td>
</tr>
<tr>
<td>Inheritance</td>
<td>56</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>57</td>
</tr>
<tr>
<td>Inclusion (or Inherent) Polymorphism</td>
<td>60</td>
</tr>
<tr>
<td>Genericity</td>
<td>61</td>
</tr>
<tr>
<td>Chapter 4: MBD Road Map</td>
<td>63</td>
</tr>
<tr>
<td>Problem Space versus Computing Space</td>
<td>63</td>
</tr>
<tr>
<td>Problem Space</td>
<td>67</td>
</tr>
<tr>
<td>Computing Space</td>
<td>68</td>
</tr>
<tr>
<td>Transformation</td>
<td>69</td>
</tr>
<tr>
<td>Maintainability</td>
<td>69</td>
</tr>
<tr>
<td>Domain Analysis</td>
<td>69</td>
</tr>
<tr>
<td>Modeling Invariants</td>
<td>71</td>
</tr>
<tr>
<td>Partitioning the Application</td>
<td>72</td>
</tr>
<tr>
<td>A Static View</td>
<td>72</td>
</tr>
<tr>
<td>A Dynamic View</td>
<td>73</td>
</tr>
<tr>
<td>Chapter 5: Modeling Invariants</td>
<td>77</td>
</tr>
<tr>
<td>So Just What Is Modeling Invariants?</td>
<td>78</td>
</tr>
<tr>
<td>The Invariant Side</td>
<td>78</td>
</tr>
<tr>
<td>The Data Side</td>
<td>80</td>
</tr>
<tr>
<td>The Rewards</td>
<td>81</td>
</tr>
<tr>
<td>Examples</td>
<td>84</td>
</tr>
<tr>
<td>Bank ATM Software</td>
<td>84</td>
</tr>
<tr>
<td>Hardware Interface</td>
<td>89</td>
</tr>
</tbody>
</table>
### Chapter 6: Application Partitioning

**Why Do We Care?**

**Basic Concepts of Application Partitioning**

**Subject Matter**

**Client/Service Relationships**

**Levels of Abstraction**

**Interfaces**

**Identifying Subsystems**

- When abstracting the same entities as other subsystems, does the subsystem in hand have a different view of them?
- Is there a client/service relationship?
- Is the service more detailed than the client?
- Is knowledge shared with other subsystems?
- Is behavior shared with other subsystems?
- Is the subject matter cohesive?
- Is the boundary clearly defined?
- Could it be reused as-is?
- Is it bounded by a network in a distributed environment?
- Is it a different problem space?

**Bridges**

**The Message Paradigm, Yet Again**

**The Bridge Model**

**Describing Subsystems**

**Subsystem Descriptions**

**Relationship Descriptions**

**Requirements Allocation**

**An Example: Pet Care Center**

**Processes**

**Legacy Replacement: A Practical Example**

### Part II: The Static Model

**Chapter 7: Road Map to Part II**

**What Is the Static Model?**
Association Roles ..................................................... 244
Association Paths .................................................... 246
Conditionality .......................................................... 250
Multiplicity ............................................................. 255
Replacing a Problem Space “One-or-Many” with a
Model “One” .......................................................... 256
Supplementing a Problem Space Association with a
Second Association .................................................. 257
Selection versus Participation ...................................... 260
Constraints ............................................................... 261
Association Classes ................................................... 264
Reification ............................................................... 267
Identifying Associations ............................................. 269
Examples ................................................................. 273
The ATM Controller .................................................. 273
Pet Care Center: Disposition ....................................... 277

Chapter 11: Referential and Knowledge Integrity ................. 279
Knowledge Integrity .................................................. 280
Timeliness ............................................................... 280
Consistency ............................................................. 281
Snapshots versus Immediate Access .............................. 282
Dependent Attributes ............................................... 283
Normalization of Attributes ...................................... 285
Referential Integrity .................................................. 289
Identity and Referential Attributes ............................... 290
Association Loops .................................................... 292
Relational versus OO Paradigms: Worlds Apart ................. 295

Chapter 12: Generalization Redux ................................. 299
Subclassing ............................................................. 300
Notation and Rules ................................................... 300
Generalization and Specialization ................................ 304
Categorization ........................................................ 307
Inclusion Polymorphism ............................................ 312
Why {disjoint, complete} Subsets? ............................... 315
Multi-directional Subclassing, Multiple Inheritance, and Composition .................................. 317
Liskov Substitution Principle ..................................... 323
## Contents

Alternatives to Generalization ............................................. 328
Delegation ................................................................. 329
Parametric Polymorphism ..................................................... 331
Basic Abstraction .......................................................... 332

Chapter 13: Identifying Knowledge ........................................... 333
What Is the Nature of OO Knowledge? ................................. 334
Abstracting Aggregates ..................................................... 335
Picking the Right Abstraction .............................................. 341
Abstracting What People Do .............................................. 343
What Does the Underlying Entity Know? ............................... 344
What Subset of Entity Knowledge Should the Object Abstraction Know? ............................................ 345
What Is the Subject Matter? ............................................... 345
What Is the Level of Abstraction? ........................................ 350
Does the Abstraction Need to Coalesce Entity Knowledge? .... 351

Part III: The Dynamic Model .............................................. 355

Chapter 14: Road Map to Part III .......................................... 357
Part III Road Map .......................................................... 357
It’s All about Behavior .................................................... 358
Object Life Cycles .......................................................... 362
Asynchronous Solution ..................................................... 363
Synchronous Services ....................................................... 369
Action Languages .......................................................... 373
Mealy versus Moore versus Harel ....................................... 374
The Learning Curve ........................................................ 376

Chapter 15: The Finite State Machine ................................. 377
Basic Finite State Automata .............................................. 378
Notation ................................................................. 383
The Role of Design by Contract (DbC) ................................ 387
Looking for State Machines .............................................. 390
Knowledge versus Behavior ............................................. 391
Managing Sequences ....................................................... 395
Some Examples ........................................................... 407
Garage Door Opener ....................................................... 407
<table>
<thead>
<tr>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automated Home Kitchen</td>
</tr>
<tr>
<td>Air Traffic Control System</td>
</tr>
<tr>
<td>Rock</td>
</tr>
<tr>
<td>Chapter 16: States, Transitions, Events, and Actions</td>
</tr>
<tr>
<td>States</td>
</tr>
<tr>
<td>Transitions</td>
</tr>
<tr>
<td>Events</td>
</tr>
<tr>
<td>Actions</td>
</tr>
<tr>
<td>The Execution Model</td>
</tr>
<tr>
<td>Naming Conventions</td>
</tr>
<tr>
<td>Chapter 17: Developing State Models</td>
</tr>
<tr>
<td>Designing State Machines</td>
</tr>
<tr>
<td>Examples</td>
</tr>
<tr>
<td>ATM Controller: Character Display</td>
</tr>
<tr>
<td>ATM Controller: Dispatcher</td>
</tr>
<tr>
<td>ATM Controller: Deposit</td>
</tr>
<tr>
<td>Chapter 18: Abstract Action Languages</td>
</tr>
<tr>
<td>AALs and ADFDs</td>
</tr>
<tr>
<td>AAL Syntax</td>
</tr>
<tr>
<td>Examples</td>
</tr>
<tr>
<td>Garage Door Opener</td>
</tr>
<tr>
<td>ATM: Character Display</td>
</tr>
<tr>
<td>Glossary</td>
</tr>
<tr>
<td>Index</td>
</tr>
</tbody>
</table>
Preface

Software development is an enormously complex intellectual activity. It is a relatively young discipline that is still rapidly evolving. It isn’t done very well because everyone is still trying to figure out how to do it.

Nonetheless, there has been substantial improvement over the years in the way software is developed. A number of design methodologies have evolved to facilitate various aspects of software design. One was the Structured Design methodology, which provided a very intuitive approach that matched up well with the hardware computational models of Turing and von Neumann.

The Problem

While Structured Design was clearly superior to the ad hoc approaches that preceded it, it had an Achilles heel: the software tended to be difficult to modify when customer requirements changed over time, especially in large applications. At the same time, applications were rapidly growing in size and complexity. In addition, new languages, technologies, operating systems, data storage paradigms, UI paradigms, hardware, and whatnot were arriving on the computing scene with alarming speed. Yet business conditions were driving the need for faster time-to-market and reduced cost of software products.

The Hope

Consequently, a number of new design methodologies appeared that captured hard-won lessons about good and bad practices. They also presented revolutionary views of computing. One of those was the Object-Oriented (OO) paradigm, with the primary goal of ensuring that large applications are maintainable over time as requirements inevitably change during the software product’s life cycle.

This book is about practicing a particular software design methodology, Model-Based Development (MBD), which is strongly based upon Shlaer-Mellor.\(^1\) Employing

\(^1\) The methodology is named for Sally Shlaer and Steve Mellor. This methodology was originally expounded in the early ’80s. Since then it has gone through several revisions, and there are several variations, of which MBD is one.
the OO paradigm in general and MBD in particular should lead to more robust and maintainable large applications.

This Book

Although the book employs UML as a notation, that is quite peripheral. There are plenty of good books that describe how to express a software design in UML, so UML syntax will not get much mention here. Similarly, this book follows a particular design methodology for MBD, but that is primarily to provide context for the real purpose of this book:

The primary goal of this book is to describe why OO methodologies in general and MBD in particular advocate a particular way of doing things.

There is no single right way to design and develop all software. Too much depends upon the particular development environment, which includes everything from business goals through tools to group culture. In the end, a shop has to decide what set of tools will be most effective in its environment. To do that the decision makers need to understand why the MBD set of methodological tools works in many common situations. More to the point, the practitioners need to understand the fundamentals well enough to adapt them to particular situations.

Practicing OO design requires a unique mindset that is not intuitive in the world of hardware computational models. Rather than focusing on particular notations and methodologies, this book is really about how to think about software design. To that end, this book spends substantial time on the thought processes behind good software designs—even to the point of deliberately providing poor preliminary designs to demonstrate that the approach is self-correcting.

To achieve such understanding it is necessary to describe how traditional (pre-OO) approaches to software development failed in some ways and how the OO paradigm addressed those shortcomings. While Structured Development brought substantial order to the chaos of pre-1970 software development, it was not a panacea, and by the '80s it became clear that software still had serious maintainability problems that the OO paradigm addressed.

Similarly, one cannot describe why a methodology works well without discussing at least some of the underlying theory. Nonetheless, this is a book by a software developer for software developers, so a conscious effort has been made to describe theoretical issues in practical terms without mathematical rigor.
Because this book is primarily about building abstract OOA models, don’t count on seeing a lot of OOPL code. As the methodology name suggests, the emphasis in this methodology lies in abstract modeling rather than writing traditional source language code. In effect, when a translation-quality OOA model is developed the model is the code. To put it another way, the notation of OOA modeling is UML augmented with an MDA-compliant Abstract Action Language (AAL). That notation is a 4GL rather than a 3GL, but the model will be just as executable as any 3GL program. While the model is implementation independent, it is a complete, precise, and unambiguous specification of the solution for functional requirements.

As a final note, we would point out that the practical development experience of the author is measured in decades rather than years. This is definitely not a theoretical book, despite the emphasis on explaining why things are done. It is based upon what works in the real world.

Road Map

The subject matter of this book is limited to application development at the OOA level. This book is organized into three main sections. Part I provides historical perspective and an introduction to basic OO principles. The introduction includes a discussion of the problems with structured development that the OO paradigm sought to correct. Part II is about construction of a static structure for the problem solution. This section represents the largest differences between the OO paradigm and other approaches since it is where the OO paradigm’s unique view of problem space abstraction is primarily manifested. Part III describes the dynamic aspects of the solution, particularly the rather militant use of finite state machines to describe behavior.

---

2. Computer languages are subdivided into generations that roughly correspond to historical innovation and increasing abstraction. The first-generation language (1GL) was machine code, bits set directly in hardware registers. The second generation (2GL) introduced symbolic names for things like instructions. Third-generation languages (3GLs) represented a major advance with notions like reusable procedures, block structure, and stack-based scope. 3GLs represented a huge advance in the size of programs that could practically be written, and they dominated software development for half a century. The fourth-generation languages (4GLs) broke with the previous generations by raising the level of abstraction to the point where the program could be specified in a manner that was essentially independent of specific computing environments.
**Intended Audience**

This book is primarily targeted at people with little OO experience. It is assumed that the reader has some cursory knowledge of UML. The book also assumes that the reader has some software development experience, on the order of a couple of class projects in C. It assumes that the level of experience includes a general knowledge of computers and programming, essentially enough to be familiar with common acronyms like KISS.

A secondary audience is the large number of converts to the OO paradigm from traditional procedural development environments. Many of these developers leapt directly into writing code in an object-oriented programming language (OOPL) since they already had substantial programming experience (i.e., believing that if one has seen one programming language, one has seen them all). Sadly, such converts have written huge volumes of bad OO code because no one told them why Object-Oriented Analysis and Design (OOA/D) is very different from Structured Analysis and Design (SA/D). If you are one of these, you will have to forget everything you ever learned about designing software and start with a clean slate.

**The Role of Translation**

A key characteristic of MBD is that it is one of a family of methodologies based on translation. That is, the methodology is compliant with the approach where a solution is modeled abstractly in a notation like UML and then a full code generator is employed to produce an implementation from that model automatically. Translation has some obvious advantages because it represents a logical extension of automation in the computing space that enhances productivity, enables economies of scale, and improves reliability. The downside is that it is not easy to do; the optimization problems faced for a model compiler are orders of magnitude more complex than those facing a 3GL compiler. Nonetheless, there are several commercial code generators available that provide 100% code generation for translation-based methodologies.

Although most of the translation approaches predate the Object Management Group (OMG) itself, they have been greatly facilitated by the Model-Driven Archi-

---

3. That isn’t critical because the syntax is explained. In addition, books like Kendall Scott’s *UML Explained* (Addison-Wesley, 2001) are cheap and quick reads that provide far more information about UML than you will ever need to record MBD designs.

tecture (MDA) initiative formalized by OMG. MDA has provided a much-needed standardization for plug-and-play tools and a conceptual framework for full code generation. Getting from an abstract, implementation-independent model to 3GL code or Assembly is a nontrivial task, especially in today’s complex IDEs. That task greatly benefits from the formalism and concepts of MDA.

However, MBD is not tied to translation. The models produced in MBD are essentially the same as those that would be produced during OOA in traditional development and then manually elaborated to produce OOD models and 3GL code. The MBD models just happen to be more rigorously constructed than typical OOA models because code generators are quite literal-minded—they do what one says, not what one meant. There is nothing to prevent the developer from manually performing the conversion.
This page intentionally left blank
Acknowledgments

I am deeply indebted to the works of Steve Mellor and Sally Shlaer, whose methodology is the basis of MBD. They pioneered the translation approach to software development and provided the much-needed design rigor for OOA models. In addition, Steve Mellor is the best OO modeler I’ve ever encountered, and his examples have been marvelously instructive.

I am equally indebted to Rebecca Wirfs-Brock for her incredibly detailed and insightful reviews of the manuscripts.

Pathfinder Solutions provided a fertile testing ground for ideas. Greg Eakman, Carolyn Duby, and Peter Fontana were particularly supportive.

I would also like to acknowledge the remarkable patience and faith of Chris Guzikowski and Raina Chrobak at Addison-Wesley. I am also very appreciative of the editing efforts of Elizabeth Ryan, Diane Freed, and Chris Zahn.

Although this book was started after I retired, I am in debt to Teradyne/ATB for providing a world-class software development shop full of superstar developers who formed a crucible for many of the ideas in this book.

Finally, I am indebted to the myriad Internet correspondents over three decades that provided a sounding board for explaining the concepts in this book. Any clarity of expression that readers might find is largely the result of refining explanations in public forums.
About the Author

H.S. Lahman wrote his first software program on a plug board in 1957. That was such a character building experience that he spent the next decade as an exploration geophysicist employing cutting edge techniques in most of the major swamps, deserts, tundras, and jungles of the Free World. He then returned to school to learn about economics, operations research, and computing. For the next three decades he developed software in MIS, scientific, and R-T/E environments. He became an advocate of OO development in 1982. In the '90s he became an advocate of improved software quality and development processes.
Introduction

History is a nightmare from which I am trying to awake.
—James Joyce

Four decades ago a program of a million lines was regarded as huge, reserved for only the most massive mainframe systems in the bowels of the Department of Defense (DoD). It was routinely estimated that to build such programs would require 1,000 engineers working for a decade. Today most of the applications on a PC are well over a million lines, and many are in the 10 million range. Moreover, they are expected to be built in a couple of years or less. So programs are growing ever larger while clients expect them to be developed with less effort.

In today’s high-tech society rapid obsolescence is routine, and the economic model of the firm emphasizes growth more than ever before. These forces create the need for more, different, and better products in shorter time. Since all those new products depend upon ever-increasing amounts of software to run, the pressure on software developers to do their part to decrease time-to-market is relentless.

Developers are also assaulted on another front. Four decades ago the national average was 150 defects per KLOC for released code. That was back in the good old days when no one had a clue how to go about writing “good” software. The GOTO and global data reigned, designs were done on blackboards and cocktail napkins, and the DEL key was the programmer’s favorite. Developing software was the arcane specialty of a few mutated electrical engineers so, frankly, nobody really cared much if it was buggy.

Alas, such a utopia could not last. In the ’70s and ’80s two things happened. First, there was a quality revolution led by the Japanese that was quickly followed by the rest of Asia and, more slowly, by the industrialized West. This led consumers into a new era where they no longer had to plan on their car or TV being in the shop for one week out of six. Consumers really liked that. Second, soon after this the quality revolution software began to weasel its way into virtually everything. Suddenly software stood out as the only thing that was always breaking. Now that consumers were aware of a better way, they didn’t like that at all.

1. This is a book about software, so we have to use acronyms. It is a union rule.
2. If you don’t know what this means, you need to start with a more elementary text than this book. The important factoid about this statistic is that it refers to thousands of raw lines of COBOL code, a rather wordy language.
INTRODUCTION

So while software developers were trying to cope with building ever-larger programs in ever-decreasing time, they suddenly found that they were also expected to get to 5-sigma defect rates. To make matters worse, a popular slogan of the ‘80s became “Quality is Free.” No matter that every software developer was certain that this was hogwash because software quality is a direct trade-off against development time. Sadly, the market forces were then and are now beyond the influence of software developers, so no one paid any attention to the wails of software developers. Thus the ’80s brought new meaning to the term “software crisis,” mostly at the expense of developer mental health.

The second major event was an extraordinary explosion in the technology of computing. Previously the technological advances were mostly characterized by new languages and operating systems. But the huge growth of PC software, the demand for interoperability among programs, and the emergence of the World Wide Web as a major force in computing resulted in a great deal of innovation. Today developers face a mind-boggling set of alternatives for everything from planning tools through architectural strategies to testing processes. Worse, new technologies are being introduced before developers are fluent in the old ones.

Finally, developers are faced with a constant parade of requirements changes. The computing space is not the only domain that is facing increasing innovation and change. For core management information systems the Federal Accounting Standards Bureau (FASB), the IRS, and sundry other agencies in the United States decree changes with annoying regularity. Product competition causes marketeers to relentlessly press for new features as products are developed to keep pace with competitor announcements. Similarly, the standards, techniques, and technologies of the engineering disciplines are also changing rapidly as the academics strive to bring order to this chaos. The days of the “requirements freeze” and the project-level waterfall model are long gone.

To summarize, modern software developers are faced with an ancient problem: putting five pounds of stuff in a three-pound bag as quickly as possible without spilling any. We have better tools, methodologies, processes, and technologies today than

3. Assuming each line of code is an opportunity for one or more defects, 5-sigma represents a frequency of defects that is five standard deviations from a median of one defect per line. The defect rate corresponding to that level of quality is about 230 defects per million LOC—a thousandfold improvement over the norm of the ’60s.

4. Interestingly, much of the core technology associated with the World Wide Web is really a rebirth of old technologies. I actually read an article recently in a fairly prestigious software development magazine where the author claimed the first markup language was invented in 1986. In fact, markup languages were commonly used two decades earlier. Sadly, that myopia meant that the industry learned nothing from the reasons why markup languages had been largely abandoned by the late ’70s. But that is a topic for a different book.
we did a half century ago, but the problems have grown in proportion. The software crisis that Basic Assembly Language (BAL) was supposed to solve is still with us, which segues to this book’s first Ingot of Wisdom:

Software development is an impossible task. The only thing the developer can do is cope and keep smiling.

The State of the Art

A keynote speaker at a conference in 1995 asked everyone who used Microsoft Windows to raise their hand. Several hundred people did. He then asked all those whose system had crashed in the past week to keep their hands up. All but about a half dozen kept their hands up. The keynoter then asked everyone to keep their hands up who thought that Microsoft would go out of business as a result. All but a dozen hands went down. The keynoter used this to validate his assertion that good quality in software was not necessarily a requirement that was important to a software company’s survival.

That conclusion is much less warranted today, as Microsoft discovered in its wars with Linux.5 The keynoter was correct, though, that the key determining factor in a software company’s viability in the marketplace is whether they deliver what their customers want. Broadly speaking, this can be viewed as an analogy of four buckets containing fluids representing reliability, features, ease of use, and availability (in the sense of time-to-market) where the cost of filling each of those buckets is different, as shown in Figure I-1. The end customer clearly wants all of those things but the vendor has a limited budget. So the vendor makes a decision that trades total cost against the mix of levels to which the buckets are filled. The marketplace then determines how well the vendor's decision provides overall satisfaction to the customer.

The decision about how much to fill each bucket is fundamentally a marketing decision. The marketeers earn their big bucks by anticipating what combination of levels will be optimal for a given total price in a given market. The task of minimizing the cost of filling each bucket lies with the software developers. Whenever the developer brings about a cost reduction, that reduction is directly reflected in a competitive advantage for that aspect of the product. The value of that advantage will depend upon the particular market niche. So, to measure the State of the Art of software

5. In fact, Microsoft learned its lesson well. Today’s Microsoft products have pretty much caught up on the reliability front to the point where security is probably more of a competitive issue.
development the question becomes: What tools are available to reduce the costs of filling the various buckets?

We have a variety of software tools to help us, such as version control systems. We also have lots of processes, ranging from XP (eXtreme Programming) to formal methods in clean room shops. And we have as many software design methodologies as there are methodologists. We also have process frameworks like the RUP and the CMM, not to mention languages, integrated development environments (IDEs), metrics systems, estimating models, and all the rest of the potpourri.

We have precious little data to sort out which combinations of the available methodologies, technologies, and practices are best.

The days are long gone when a couple of hackers could sit down in their spare bedrooms and come up with a killer application after a few months of consuming

---

6. An old but still amusing joke: What’s the difference between a terrorist and a software methodologist? You can negotiate with a terrorist.

7. The Rational Unified Process (RUP) is not really a process. It is primarily a process framework that is highly tailor able to local development environments.

8. The Capability Maturity Model (CMM) is a true process framework in that it describes only what capabilities must be in place to produce good software, not how they should be implemented. Thus compliance with the CMM is a necessary but not sufficient condition for good software development.
lots of Jolt Cola and cold pizza. If you tell someone at a job interview that you wrote 100 KLOC last year, you can probably kiss that job off because hiring managers have figured out that the only way you can get that kind of productivity is by writing terminally unreliable and utterly unmaintainable crap. To effectively lower the cost of filling those buckets you need to go at software development in an organized and systematic way.

Because of this there is a lot of lip service paid to software engineering in today’s industry rags. Alas, we are probably a long way from respectability as an engineering discipline. Software development is still an art, although it is more bilious green than black. The art lies in cobbling together a coherent development system that is appropriate for a given environment from a hodgepodge of disparate practices and tools. Instead of providing discipline for putting together such systems, our “software engineering” merely says, “Here’s a framework and lots of alternatives to drape on it. Selecting the right alternatives and gluing them together correctly for the particular development environment is left as an exercise for the student.”

**What Works**

Nonetheless, the reality is that the state of the art of software development has improved significantly over the past half century. The same number of people routinely develop much bigger programs with much better quality in shorter time, so we have to be doing something right. The tricky part is to figure out what things are more right than others.

Initially there was a collection of hard-won lessons based upon decades of rather rocky experience prior to 1970. For example, once upon a time FORTRAN’s assigned-GOTO statement and COBOL’s ALTER statement were regarded as powerful tools for elegantly handling complex programming problems. A decade later, programming managers were making their use grounds for summary dismissal because the programs using those features tended to be unmaintainable when requirements changed.\(^9\)

All these hard-won lessons formed a notion of *developer experience*. The programmers who had been around long enough to make a lot of mistakes and who were bright enough to learn from those mistakes became software wizards. Unfortu-

---

\(^9\) The problem was that such statements changed the flow of control of the program “on the fly” in ways that were not directly visible in the code itself. Understanding other people’s code is tough enough when sequence and branching is explicit. Such programs led to the acronym WORN—Write Once; Read Never.
nately, each developer essentially had to reinvent this wheel because there was no central body of knowledge about software development.

Starting in the late '60s a variety of methodologies for programming in 3GLs and for software design sprang up. The earliest ones simply enumerated the hard-won practices and mistakes of the past in a public fashion for everyone's benefit. Soon, though, various software wizards began to combine these practices into cohesive design methodologies. These methodologies were different in detail but shared a number of characteristics in common, so they were grouped under the umbrella of Structured Development (SD). Because they were rather abstract views—the big picture of software design—the authors found it convenient to employ specialized notations. Since many of the wizards were also academicians, the notations tended to be graphical so that theoretical constraints from set and graph theory could be readily applied.

SD was the single most important advance in software development of the middle twentieth century. It quite literally brought order out of chaos. Among other things, it was the engine of growth of data processing, which later evolved into the less prosaic information technology in the '80s, in the corporate world because acres of entry-level COBOL programmers could now churn out large numbers of reports that provided unprecedented visibility into day-to-day corporate operations. Better yet, those reports were usually pretty accurate.

Alas, SD was only the first step on a long road to more and better software development. As one might expect for something so utterly new, it wasn’t a panacea, and by the late '70s some warts were beginning to show up. Understanding what those warts were is critical to understanding why and how object-oriented development evolved in the '80s.
This page intentionally left blank
Chapter 1

Historical Perspective

Problems are the price of progress.
—Charles F. Kettering

Compared to the physical sciences and the Industrial Revolution, software development is relatively new on the scene of human progress. It took more than a millennia for the physical sciences to play a ubiquitous role in modern life and it took the Industrial Revolution over a century to play a similar role. Yet computers and software have become an invasive and essential part of our lives in three decades. Alas, the road traveled has been a bit rocky.

This chapter provides historical context for why the OO paradigm was developed. To fully understand and appreciate the paradigm we need to understand what problems it sought to solve, so we start with a bit of history. We will then examine some weaknesses of the paradigm that dominated software development immediately before the OO paradigm appeared. Finally, we will provide a technical context for the rest of the book by examining some of the important technical advances made prior to the OO paradigm that were incorporated in it.

History

Essentially there was no systematic development at all through the 1950s. This was the Dark Ages of programming. It is difficult for today’s developers to even imagine the conditions under which software was developed in those days. A mainframe had a few kilobytes of memory and paper tape was a high-tech input system. Western Union had an effective monopoly on teletype input devices that required several footpounds of energy for each key press—it was the machine that crippled programmers with carpal tunnel syndrome before the medical profession had a name for it. There
were no browsers, debuggers, or CRT terminals. Basic Assembly Language (BAL) was the silver bullet to solve the software crisis!

In the late ’50s and early ’60s better tools began to appear in the form of higher-level computer languages that abstracted 1s and 0s into symbolic names, higher-level operations, block structures, and abstract structures such as records and arrays. It was clear that they made life easier and developers were far more productive, but there were no guidelines for how to use them properly. So this renaissance gave birth to the Hacker Era where individual productivity ruled.

The Hacker Era extended from the early ’60s to the mid ’70s. Very bright people churning out enormous volumes of code characterized this period—100 KLOC/yr. of FORTRAN was not unusual. They had to be bright because they spent a lot of time debugging, and they had to be very good at it to get the code out the door that fast. They often developed ingenious solutions to problems. In the ’60s the term hacker was complimentary. It described a person who could generate a lot of code to do wonderful things and who could keep it running.

By the late ’70s, though, the honeymoon was over and hacker became a pejorative. This was because the hackers were moving on to new projects while leaving their code behind for others to maintain. As time passed more special cases were exercised and it was discovered that the code didn’t always work. And the world was changing, so those programs had to be enhanced. All too often it became easier to rewrite the program than to fix it. This was when it became clear that there was something wrong with all that code. The word maintainable became established in the industry literature, and unmaintainable code became hacker code.

The solution in the late ’60s was a more systematic approach that coalesced various hard-won lessons into methodologies to construct software. At the same time,

---

1. My first program was written on a plug board in 1957. A plug board is essentially an extension of the hardware whereby one grounded TTL inputs, which normally float High (logic 1), with a wire to make a Low (logic 0). One wrote every single bit directly into the hardware. My program was a hello-world class exercise, but it took me two weeks to get it to work.

2. In the days of drum memories there was a fellow writing BAL who routinely optimized for the fact that there were two heads on the drum. He wrote his programs by physically separating the logically consecutive statements in his program by a distance that mapped to nearly half the circumference of the drum in address space. The exact placement was computed based on the number of cycles that the previously executed statement would take to execute and the drum rotation speed. This allowed the statements to be executed with something approaching half the latency for the drum rotation that a sequentially ordered program would have. The mind boggles that he could write programs that way, much less debug them.

3. With the advent of the Internet in the ’80s, the term further evolved to refer to people who broke into computer systems for nefarious purposes.
programs were becoming larger and the idea that they had a structure to be designed appeared. Thus software design became a separate activity from software programming. The methodologies that began to appear during the twilight of the Hacker Era had a synergy whereby the various lessons learned played together so that the whole was greater than the sum of the parts. Those methodologies were all under the general umbrella of Structured Development (SD).

The period starting around 1980 was one of mind-boggling advances in almost every corner of the software industry. The OO paradigm—more specifically a disciplined approach to analysis and design—was just one of a blizzard of innovations.

**Structured Development**

This was unquestionably the single most important advance prior to the ’80s. It provided the first truly systematic approach to software development. When combined with the 3GLs of the ’60s it enabled huge improvements in productivity.

SD had an interesting side effect that was not really noticed at the time. Applications were more reliable. It wasn’t noticed because software was being used much more widely, so it had much higher visibility to non-software people. It still had a lot of defects, and those users still regarded software as unreliable. In fact, though, reliability improved from 150 defects/KLOC in the early ’60s to about 15 defects/KLOC by 1980.

SD was actually an umbrella term that covered a wide variety of software construction approaches. Nonetheless, they usually shared certain characteristics:

- **Graphical representation.** Each of these fledgling methodologies had some form of graphical notation. The underlying principle was simply that a picture is worth a thousand words.

---

4. It was more well known as Structured Programming at the time, which was a bit of a misnomer because a common theme of SD was to provide software design using notations that were quite different than the programming languages of that era.

5. The first pure OO programming language, Smalltalk, appeared in 1972, but it was developed with the goal of providing better tools for simulation, and the mapping to general computation was not intuitive. It took about a decade for OOA/D to evolve to the point where general-purpose use was feasible.

6. The empirical data from those times was pretty varied and the quality wasn’t very good, especially for the ’60s. People might argue about the specific numbers, but it is unlikely anyone familiar with the period would argue that defects/KLOC had not improved by several integer factors in that period.
• **Functional isolation.** The basic idea was that programs were composed of large numbers of algorithms of varying complexities that played together to solve a given problem. The notion of interacting algorithms was actually a pretty seminal one that arrived just as programs started to become too large for one person to handle in a reasonable time. Functional isolation formalized this idea in things like reusable function libraries, subsystems, and application layers.

• **Application programming interfaces (API).** When isolating functionality, it still has to be accessed somehow. This led to the notion of an invariant interface to the functionality that enabled all clients to access it in the same way while enabling the implementation of that functionality to be modified without the clients knowing about the changes.

• **Programming by contract.** This was a logical extension of APIs. The API itself became a contract between a service and its clients. The problem with earlier forays into this idea is that the contract is really about the semantics of the service, but the API only defined the syntax for accessing that semantics. The notion only started to become a serious contract when languages began to incorporate things such as assertions about behavior as part of the program unit. Still, it was a reasonable start for a very good idea.

• **Top-down development.** The original idea here was to start with high-level, abstract user requirements and gradually refine them into more specific requirements that became more detailed and more specifically related to the computing environment. Top-down development also happened to map very nicely into functional decomposition, which we’ll get to in a moment.

• **Emergence of analysis and design.** SD identified development activities other than just writing 3GL code. Analysis was a sort of hybrid between requirements elicitation, analysis, and specification in the customer’s domain and high-level software design in the developer’s domain. Design introduced a formal step where the developer provided a graphical description of the detailed software structure before hitting the keyboard to write 3GL code.

SD enabled the construction of programs that were far more maintainable than those done previously. In fact, in very expert and disciplined hands these methods enabled programs to be developed that were just as maintainable as modern OO programs. The problem was that to do that required a lot more discipline and expertise than most software developers had. So another silver bullet missed the scoring rings, but at least it was on the paper. Nonetheless, it is worth noting that every one of these characteristics can be found in modern OO development (though some, like top-down design, have a very limited role).
Functional Decomposition

This was the core design technique that was employed in every SD approach; the *Structured* in SD refers to this. Functional decomposition deals with the solution exclusively as an algorithm. This is a view that is much closer to scientific programming than, say, management information systems programming. (The name of the first 3GL was FORTRAN, an acronym for FORmula TRANslator.) It is also very close to the hardware computational models that we will discuss shortly.

The basic principle of functional decomposition is divide and conquer. Basically, subdivide large, complex functionality into smaller, more manageable component algorithms in a classic top-down manner. This leads to an inverted tree structure where higher-level functions at the top simply invoke a set of lower-level functions containing the subdivided functionality. The leaves at the base of the tree are atomic functions (on the scale of arithmetic operators) that are so fundamental they cannot be further subdivided. An example is illustrated in Figure 1-1.

![Figure 1-1](image)

*Figure 1-1  Example of functional decomposition of a task to compute employee stock benefits into more manageable pieces*

Functional decomposition was both powerful and appealing. It was powerful because it was ideally suited to managing complexity in the Turing world of an algorithmic calculating machine, especially when the 3GLs provided procedures as basic language constructs. In a world full of complex algorithms, functional decomposition was very intuitive, so the scientific community jumped on functional decomposition like tickets for a tractor pull on Hard Liquor and Handgun Night.
It was appealing because it combined notions of functional isolation (e.g., the limbs of the tree and the details of the subdivided functions), programming by contract in that the subdivided functions provided services to higher-level clients, a road map for top-down development once the decomposition tree was defined, APIs in the form of procedure signatures, very basic depth-first navigation for flow of control, and reuse by invoking the same limb from different program contexts. Overall it was a very clever way of dealing with a number of disparate issues.

Alas, by the late ’70s it was becoming clear that SD had ushered in a new set of problems that no one had anticipated. Those problems were related to two orthogonal realities:

- In the scientific arena algorithms didn’t change; typically, they either stood the test of time or were replaced in their entirety by a new, superior algorithms. But in arenas like business programming, the rules were constantly changing and products were constantly evolving. So applications needed to be modified throughout their useful lives, sometimes even during the initial development.

- Hierarchical structures are difficult to modify.7

The problem was that functional decomposition was inherently a depth-first paradigm since functions can’t complete until all subdivided child functions complete. This resulted in a rather rigid up-and-down hierarchical structure for flow of control that was difficult to modify when the requirements changed. Changing the flow of control often meant completely reorganizing groups of limbs.

Another problem was redundancy. The suite of atomic functions was typically quite limited, so different limbs tended to use many of the same atomic operations that other limbs needed. Quite often the same sequence of atomic leaf operations was repeated in the same order in different limbs. It was tedious to construct such redundant limbs, but it wasn’t a serious flaw until maintenance was done. If the same change needed to be made in multiple limbs, one had to duplicate the same change multiple times. Such duplication increased the opportunities for inserting errors. By the late 1970s redundant code was widely recognized as one of the major causes of poor reliability resulting from maintenance.

7. A classical example is the zoological taxonomy that classifies all living things. Academics spent decades developing that taxonomy. Just as they were congratulating themselves on a job well done, someone brought home a duck-billed platypus from Australia. A hairy, egg-laying, web-footed marsupial with a bill simply did not fit into the taxonomy because several of its characteristics were determinants for very different limbs of the taxonomy. Modifying the tree would have been an arduous task; folklore says that zoologists at the time considered financing an expedition to Australia to exterminate the critters. Urban legends aside, to this day the placement of the critter in the tree is a matter of debate.
To cure that problem, higher-level services were defined that captured particular sequences of operations once, and they were reused by invoking them from different limbs of the tree. That cured the redundancy but created an even worse problem. In a pure functional decomposition tree there is exactly one client for every procedure, so the tree is highly directed. The difficulty with reuse across limbs was that the tree became a lattice where services had both multiple descending child functions and multiple parent clients that invoked them, as shown in Figure 1-2.

Figure 1-2  Functional decomposition tree becoming a lattice. Shaded tasks are fundamental elements of different benefits. Dashed lines indicate crossovers from one decomposition limb to another to eliminate redundancy.
Figure 1-1 has been expanded to include the computation of multiple employee benefits that all require basically the same tasks to be done. Some tasks must be implemented in a unique way so the basic tree has the same structure for each benefit. But some tasks are exactly the same. To avoid redundancy, those nodes in the tree are reused by clients in different benefit limbs of the tree. This results in a lattice-like structure where tasks like Get base salary from DB have multiple clients that are in quite different parts of the application.

That lattice structure was a major flaw with respect to maintainability. When the requirements changed for some higher-level function (e.g., Insurance Benefit in Figure 1-2) in a particular limb, the change might need to be implemented in a descendant sub-function much lower in the tree. However, the sub-function might have multiple clients from other limbs due to reuse (e.g., Get base salary from DB). The requirements change might not apply to some of those clients that were in different limbs (e.g., Stock Benefit) because each limb where the sub-function was reused represented a different context. Then the low level change for one client might break other clients in other contexts.

The biggest difficulty for functional decomposition, though, was the implicit knowledge of context that came with depth-first processing. The fundamental paradigm was Do This. Higher-level functions were essentially just a collection of instructions to lower-level functions to do things. To issue an instruction to do something, the higher level function must (a) know who will do it, (b) know that they can do it, and (c) know that it is the next thing to do in the overall solution. All of these break functional isolation to some extent, but the last is most insidious because it requires that the calling function understand the much higher-level context of the whole problem solution. The higher-level function hard-wires that knowledge in its implementation, so when requirements changed for the overall context, that implementation had to be changed. This hierarchical functional dependence, combined with all the other problems just mentioned, resulted in the legendary spaghetti code.

Another way to look at this is through the idea of specification and DbC contracts. When a function is invoked, there is a Do This contract with the client invoking the function. That DbC contract represents an expectation of what service the function will provide. It does not matter whether the function provides the entire service itself (i.e., it is not subdivided) or delegates some or all of its functionality to lower-level functions. From the client’s perspective, the contract with the function in hand is for the entire service. If the function in hand is a higher-level function in the

---
8. An example like this can be picked to death by bringing in other requirements that would ensure that, say, “Check eligibility” would have different requirements for different benefits. This is just an example of the structure issues, so it has been considerably simplified.
tree, the specification of what that function does is the specification of all of the descending limbs. The lower-level functions descending from the higher-level functions are extensions of it, and their individual specifications are subsets of the higher-level function’s specification.

This means that all of the limbs descending from a given higher-level function in the lattice form a complex dependency relationship originating with the higher-level function. That is, to fulfill its responsibilities in the DbC contract with its client, a higher-level function depends on every descendant lower-level function doing the right thing with respect to its client’s expectations. It is that dependency chain that is the real root cause of spaghetti code. A lower-level function’s specification cannot be changed without affecting a potentially long chain of parent (client) specifications.

A similar dependency problem existed for sequences of operations. To modify the sequence in which the leaf functions were invoked, the implementations of the higher-level functions had to be touched. That is, the sequence was determined by moving up and down the limbs in a systematic, depth-first manner. That navigation of the tree was hard-coded in the higher-level functions’ implementations. So to change the sequence, the implementations of multiple higher-level functions had to be changed. In effect, the overall solution flow of control was hard-wired into the tree structure itself, which sometimes required reorganizing the entire tree to accommodate small sequencing changes.

One way this is manifested is in doing unit tests of higher-level functions. Since the specification of the higher-level function includes the specification of every descending lower-level function, it is not possible to unit test a higher-level function from the client’s perspective without working implementations of every descending lower-level function.9

A strong case can be made that the primary goal of the OO paradigm is to completely eliminate the hierarchical dependencies resulting from functional decomposition.

It is not terribly surprising that no one anticipated these problems. They only showed up when SD’s productivity gains were realized. As applications became larger, there were more requirements changes. Meanwhile, the functional decomposition trees were growing in size and complexity. Only when the benefits were realized did the problems show up.

---
9. A short-cut for unit testing was to stub the immediate child functions so that they returned or set variables in a manner appropriate for the test case inputs. Basically, though, this is just self-delusion. All you are doing is testing the test harness.
Lessons Hard Won

By the time the academicians began to ruminate about OT in the late '70s, a lot of crappy software had been written—enough so that the patterns that were common in that crappy software started to be painfully obvious. This led to a lot of papers and letters like Dijkstra’s classic, “Go To Statement Considered Harmful” in 1968. Those patterns represented the rather painfully learned lessons of software development (some of which were just touched on).

Global Data

One basic problem was that persistent state variables were recording the state of the system, and they were getting changed in unexpected ways or at unexpected times when they were globally accessible. This led to errors when code was written assuming a particular state prevailed and that code was later executed when that state did not prevail.

In practice such problems were difficult to diagnose because they tended to be intermittent. There was also a maintenance problem because it was difficult to find all the places where the data was written when, say, some new condition had to be added. So unrelated maintenance tended to introduce new bugs. This problem was so severe that by the late '70s every 3GL IDE had some sort of Find/Browse facility so that you could easily search for every place a variable was accessed. While convenient for the developer, such facilities didn’t really help manage the state data properly. The problem was so pronounced that a functional programming paradigm was developed in parallel to the OO paradigm, largely to eliminate the need for persistent state data.

Elephantine Program Units

Once upon a time a fellow asked a colleague why he was getting a particular compiler error. The colleague hadn’t seen the message before, but it sounded like some compiler’s stack had overflowed, so the colleague took a glance at the code. The procedure where the error occurred was 1800 statements long, and there were several

---

10. E. Dijkstra, “Go To Statement Considered Harmful,” letter to the editor in Communications of the ACM, Vol. 11, No. 3 (March, 1968), pp. 147–148. It seems astonishing now that back then this letter launched a debate about whether GOTOs really were bad, one that lasted nearly a decade. A rather famous guru defended GOTOs as a columnist in Computer Language magazine, writing one of the most convoluted 15-line programs I’ve ever seen outside a coding obscurity contest. It was incorrect, and he got so soundly flamed about it by a gazillion readers that he didn’t put another code example in his articles for years.
procedures of similar size in the file. Printing the compiler listing used nearly half a box of paper! Since the compiler was doing file-level optimizations, it had too much to think about and committed seppuku.

As soon as the colleague saw the length of the routine he suggested that the basic solution was *Don’t Do That*. At that point the code author indignantly lectured him for ten minutes on how screwed up the compiler was because it didn’t allow him to code in the most readable fashion—like nested switch statements that extend for ten pages are readable! The sad thing about this anecdote was that it occurred around 1985 when people were supposed to know better.

Huge program modules led to loss of cohesion. The modules do too many things, and those things tend to have relationships that were all too intimate within the module. That tended to make them more difficult to maintain because changes to one function are not isolated from the other functions in the module. Another common problem was mixed levels of abstraction. Such modules tended to mix both high-level functionality and low-level functionality. This obscured the high-level processing with low-level details, a classic forest-and-trees problem.

Perhaps the worst problem of all, though, was that the larger the module, the more interactions it tended to have with other modules, so the opportunities for side effects were greatly increased. All modern approaches to more maintainable code advocate limiting the scope where side effects can occur. The simplest way to limit scope is to keep modules small.¹¹

**Software Structure**

The word *architecture* started to appear in the software literature in the ’70s because people were coming to understand that a software application has a skeleton just like a building or an aardvark. The painful lesson about structure is that when it has to be changed, it usually requires a great deal of work. That’s because so much other stuff in the application hangs off that structure.

There are four common symptoms that indicate structural changes are being made to an application:

1. Lots of small changes are made.
2. The changes are spread across many of the program units.
3. Changes are difficult to do (e.g., hierarchies have to be redesigned).
4. The changes themselves tend to introduce more errors and rework than normal.

¹¹ The acronym KISS—Keep It Simple, Stupid—was born in the ’70s at least partially from this idea.
Such changes are commonly called Shotgun Refactoring, and in the Hacker Era there was a whole lot of it.

**Lack of Cohesion**

One correlation that became apparent early on was that changes were being made to many modules when the requirements changed. One reason was that fundamental structure was being modified. The other was lack of cohesion. Functionality was spread across multiple modules, so when the requirements for that functionality changed, all the modules had to be touched. Lack of cohesion makes it difficult to figure out how to modify a program because the relevant functionality is not localized. Because different modules implemented different parts of a given functionality, those implementations tended to depend on one another to do certain things in particular orders. Although it was usually easy to recognize poor cohesion after the fact, there were no systematic practices that would guarantee good cohesion when the application was originally developed.

**Coupling**

There have been entire books written about coupling and how to deal with it, so the subject is beyond the practical scope of this book. At the executive summary level, coupling describes the frequency, nature, and direction of intimacy between program elements. The notion of logical coupling grew out of trying to reconcile several different observations:

- Spaghetti code is difficult to maintain.
- Bleeding cohesion across modules resulted in implementation dependencies between them.
- If a client knows a service intimately, it often has to change when the service changes.
- If a service knows a client intimately, it often has to change when the client changes.
- The need to change two elements at once is directly proportional to the nature of access between them.
- The need to change two elements at once is directly proportional to the intimacy of access between them.
- Bidirectional intimacy is usually worse than unidirectional intimacy.
So the notion of coupling arose to describe dependencies between modules resulting from interactions or collaborations between program units. Alas, for a program to work properly, its elements must interact in some fashion. Consequently, developers were faced with a variation on the Three Laws of Thermodynamics: (1) You can’t win; (2) You can’t even break even; and (3) You can’t quit playing.

When thinking about coupling it is important to distinguish between logical and physical coupling. Logical coupling describes how program elements are related to one another based on the roles they play within the problem solution. Physical coupling describes what they need to know about each other to interact from a compiler’s perspective. Logical coupling in some fashion is unavoidable because solving the problem depends upon it. But it can be minimized with good design practice.

Physical coupling, though, is almost entirely due to the practical issues of implementing 3GL-type systems within the constraints of hardware computational models. For the compiler to write correct machine code in one module for processing—say, data passed to it from another module via a procedure call—the compiler must know how that data was implemented by the caller (e.g., integer versus floating point). Otherwise, the compiler cannot use the right ALU instructions to process it. So the compiler needs to know about the implementation of the calling module, which is a physical dependency on that implementation that is dictated by the particular hardware. The most obvious manifestation of this sort of coupling is compile-the-world, where many modules in a large application must be recompiled even though only one had a simple change.

Unfortunately, the literature has been concerned primarily about the frequency of access. Only in the late ’80s did the deep thinkers start worrying about the direction of coupling. They noticed that when the graph of module dependencies was a directed graph without loops, the program tended to be much more maintainable. They also noticed that very large applications were becoming infeasible to maintain because compile times and configuration management were becoming major headaches. The challenge lies in minimizing or controlling the physical dependencies among program elements. The techniques for doing this are generally known as dependency management.

To date the literature has rarely addressed the nature of coupling. Since the nature of coupling plays a major role in justifying some MDB practices, it is worth identifying here a basic way of classifying the nature of coupling (i.e., the degree of intimacy), in order of increasing intimacy:

• **Message identifier alone.** This is a pure message with no data and no behavior. The opportunities for foot shooting are quite limited, but even this pristine form can cause problems if the message goes to the wrong place or is presented at the wrong time.
• **Data by value.** This form is still pretty benign because there is no way to distress the sender of the message and the receiver has full control over what is done with the data. It is somewhat worse than a pure message because the data may no longer be correct in the program context when the message is processed. This is usually only a problem in asynchronous or distributed environments when delays are possible between sending and processing the message. It can also be a problem in threaded applications where parallel processing is possible.

• **Data by reference.** Here we have the added problem of the receiver being able to modify data in the sender’s implementation without the sender’s knowledge. Data integrity now becomes a major issue. In fact, passing data by reference compounds the problems in the parallel processing environments because the receiver (or anyone to whom the sender also passed the data by reference) can change it while the sender is using it.

• **Behavior by value.** This curious situation arises in modern programs that pass applets in messages. This is similar to data-by-reference except that it is the receiver who can be affected in unexpected ways. The reason is that the applet is in no way constrained in the things that it does. When the receiver invokes the behavior it effectively has no control over and cannot predict a potentially unlimited number of side effects. If you don’t think applets are a problem, ask anyone who deals with web site security.

• **Behavior by reference.** Though some languages from previous eras could do this by passing pointers to functions, it was very rare. Alas, in OOPs it can be done trivially by passing object references. Like FORTRAN’s assigned GOTO, it probably seemed like a good idea at the time, but it turned out badly. Today we realize this is absolutely the worst form of coupling, and it opens a huge Pandora’s Box of problems. In addition to invoking behavior with potential for all sorts of side effects, the receiver can also change the object’s knowledge without the sender knowing. Since the class’ entire public interface is exposed, the receiver is free to invoke any aspect of the object, even those aspects that would cause a problem for the sender.

But worst of all, the encapsulation of the sender has been totally trashed. The reason is that the object whose reference is passed is part of the sender’s implementation. By passing it, a bay window has been opened into the implementation of the sender that invites bugs and presents a maintenance nightmare. (In OO terms, it defeats implementation hiding, which is a fundamental OO practice.)

---

12. This only applies to problem space classes that were crafted specifically for the problem in hand. General-purpose computing space classes like String, Array, and Complex_number are really just standardized data structures without significant behavior. In other words, they are simply extensions of fundamental data types.
This means the maintainer must track down every place the reference is used and verify that any change to the way it is used won’t break anything. Even then, we can’t be sure that someone won’t change the receiver during another bout of maintenance to access the instance incorrectly. And even if everything just works, one still has to rebuild the receiver because of the physical coupling.

Because it is such a broad topic, space precludes dealing with coupling in this book. But it is important to note that the notion of coupling formalizes a flock of maintainability issues, and understanding coupling is fundamental to good program development whether you are doing OO development or not.

Technical Innovation

Prior to the OO paradigm all was not chaos, even in the Dark Ages of programming. The academicians were busily adapting mathematics to practical use in the computing environment. During their cogitations the academicians provided a mathematical lingua franca for the underpinnings of software development. This was a collection of theories and models that were particularly well suited to the computing environment. Today this mathematics is the basis of almost everything we do in software development.

This book takes an engineering approach to software development, so it is not going to trot out a lot of theorems. It is, however, worth a couple of subsections to briefly describe at the megathinker level the major relevant mathematical elements and how they influence software development. That’s because those ideas still underlie the OO paradigm, albeit in a very well-disguised manner.

The Turing Machine

In the 1930s Alan Turing developed a detailed mathematical model of a theoretical calculating machine. The model was elegantly simple. It assumed all calculating processes were divided into fundamental, individual steps. Those steps could be expressed as a small set of atomic instructions to his machine. Once so divided any calculating process could be described in the following terms:

Sequence: A sequence of one or more instructions located consecutively in the program

Branch: A shift in the execution of the program from a sequence in one program location to another sequence at a different location in the program based upon the truth of some boolean condition
**Iteration:** A combination of sequence and branch that allowed a sequence to be repeated indefinitely until some boolean condition became true

The mathematics became heavily involved with precise definitions of things like *instruction* and *condition*, and then proving that a Turing machine with a finite instruction set could, indeed, successfully perform any arbitrarily complex calculation. To do so Turing had to examine various strategies for combining and nesting the basic operations.

The Turing machine is the most famous, but there have been a succession of models of computation developed over the years. These models belong to the general class of *hardware computational models*. It is important to realize that such models describe how a *computer* works, not how software is designed. In reality, almost the entire computing space is based upon various hardware computational models. Even exotic mathematical constructs like 3GL type systems have their roots in hardware computational models.

The OO paradigm represents the first major break away from the hardware computational models in creating an approach to software design that is, at least in part, independent of the hardware computational models.\(^\text{13}\) That is not to say, though, that the OO paradigm ignores those models. Ultimately the problem solution has to be executed on Turing hardware, so the OO paradigm necessarily has to provide an unambiguous mapping into the hardware computational models.

**Languages and Methodologies**

As suggested, programs are written in a variety of languages. Initially programs were encoded as binary instructions directly into the hardware on plug boards that directly connected power rails to the hardware to form logic 1 and logic 0, one bit at a time. Instructions to the Arithmetic Logic Unit (ALU) consisted of binary words of a fixed size written directly into hardware *registers*. This was complicated by the fact that the hardware commonly subdivided those words into bit subsets called *fields*.

The first major advance in software development was the introduction of BAL.\(^\text{14}\) As bigger problems were solved, software programs became larger, but people don’t

---

\(^{13}\) Though the OO paradigm is unique in this respect, that uniqueness is not without its price. OO design is not as intuitive as construction approaches that are firmly attached to hardware computational models. While anyone who can spell “C” can learn OO design techniques, there is definitely a longer learning curve.

\(^{14}\) This was enabled by an advance in hardware technology that allowed one to provide binary instructions on media such as paper tape and punched cards that could be loaded into memory registers automatically. The hardware to do such things were the first peripheral devices that supported the ALU to make access to it more convenient.
think well in binary. BAL enabled hardware operations to be described symbolically with mnemonics like ADD, MOV, and JMP. BAL also enabled ALU registers to be represented symbolically as R1, R2, . . ., Rn. Finally, BAL enabled the developer to assign mnemonic names to memory addresses so that data could be named in a meaningful way. All these things were a boon to software development because they allowed the developer to think about the solution in more abstract terms than endless strings of 1s and 0s.

However, for BAL to work, all those mnemonics had to be translated into the binary view that the hardware understood. Thus was introduced the powerful notion of a program, called a compiler, to process other programs by converting them from one representation to another. (Primitive operating systems were already around to manage the housekeeping around running programs through the ALU, but they didn’t significantly modify those programs.)

While all this made life much, much easier for the early software developers, there were still some problems. BAL was necessarily tied to the specific ALU instruction set, and by the 1950s hardware vendors were churning out a wide variety of computers with different instruction sets. What was needed was a more abstract description of the solution that was independent of a particular instruction set. This led to the introduction of FORTRAN (FORmula TRANslator) and COBOL (COmmon Business-Oriented Language) in the late 1950s. Just as BAL raised the level of abstraction of software programs above binary expression, this next generation of languages raised the level of abstraction above the ALU itself.

These 3GLs introduced a number of new abstract concepts for computing. These are the primary ones that characterized that generation of languages.

- **Procedures.** Procedures were blocks of instructions that could be invoked from anywhere in the program. When the block completed, control would return to the point in the program from which the procedure was invoked. This enabled a suite of instructions to be reused in different contexts, which was quite common in large applications.

- **Procedural message passing.** However, the data that was modified for each invocation context of a procedure needed to be different. Procedure parameters enabled a particular context to provide data that was unique to that context. Then such parametric data became a unique message sent to the procedure by an individual invocation context. This went a long way toward decoupling what the procedure did with the data from the source context of the data.

- **Types.** To provide that decoupling an abstract mechanism was needed for describing generally what the data is without the detailed semantics of source context. Types allowed the data to be defined for the procedure very generically
(e.g., as an integer value). Thus a procedure to compute statistics on a bunch of samples could do so without knowing what the samples actually were; it only needed to know they were integers to compute things like means and T-Statistics. Thus, types at the 3GL level initially defined the implementation of data rather than the problem semantics of the data.

- **Stack-based scope.** Since procedures were called from other procedures, there was an inherent hierarchical structure for sequences of operations. Procedures needed to know how to get back to where they came from, and the notion of a call stack was born. The call stack provided a way to keep track of where you came from so that you could unravel nested procedure calls as they completed their tasks.

- **Procedural block structuring.** Procedures provided very useful building blocks for isolating functionality. Such modularity enabled a divide-and-conquer approach to managing complexity. The 3GLs extended this idea to provide modularity within procedures. This supported hard-won lessons like programs being more robust when groups of instructions had single entry and exit points.

Thus the 1960s was a time of profound revolution as software development moved from BAL to 3GLs. New concepts like procedural message passing and stack-based scope required a different view of construction than the only slightly abstracted Turing view of computing in BAL. Into the breach stepped the methodologists. As soon as 3GLs became widely accepted, the need for training developers in using them properly arose.\textsuperscript{15} By the early 1970s it was clear that the larger programs enabled by 3GLs needed to be properly designed. That gave birth to a host of construction approaches under the general umbrella of Structured Development, as discussed previously.

Basically, those approaches recognized that solving large problems requires something more than just Assembly or 3GL coding skills. Somehow all those abstract constructs needed to play together properly to solve the customer’s overall problem. Not the least of the problems was the recognition that programs needed to be maintainable. In addition, large programs needed a structural skeleton and, once in place, that

\textsuperscript{15} Initially this was largely driven by performance issues. The early 3GL compilers weren’t very smart. So in the 1960s experienced programmers wrote “x = a + a + a” in FORTRAN instead of “x = 3 * a” because they knew the compiler was not smart enough to realize that one multiply instruction took an order of magnitude more ALU cycles than an addition and that ALU speed was measured in KHz rather than GHz. So the early methodologists were passing on these sorts of practical tidbits to posterity. But very quickly it became apparent that performance was not the only way to shoot oneself in the foot in a 3GL, and more sophisticated design advice was needed.
skeleton tended to be very difficult to modify because of the sheer number of disparate constructs hanging off it. So the methodologies quickly evolved to a more abstract level than 3GL coding, and the developer started looking at the Big Picture as a separate design activity from 3GL coding.

Programming languages are not the only representations of executables. There have been a large number of pictorial notations used to describe programs. It is no accident that every design notation from early HIPO charts\(^\text{16}\) to today’s Universal Modeling Language (UML) are often referred to as *graphical notations*. Such notations can express the abstract concepts needed for design very efficiently.

Most of the readers of this book will have some software development experience, so the previous discussion of languages and methodologies will probably be quite familiar. However, it was worth the review because there are several important things the rest of the book builds upon.

- A problem solution expressed in any language above binary machine language is a more abstract representation of the machine language that will actually execute on the Turing Machine. Another word for abstract representations of something more concrete is *model*. Thus a program in C is a model of the problem solution that will ultimately execute on the computer in the same sense as a UML OOA model is.

- Software development is continuously evolving towards more abstract ways of expressing problem solutions.

- Because of the ubiquitous presence of mathematics in the computing environment, the computing space is deterministic. For example, compilers, linkers, and loaders automated a lot of drudge work for early BAL and 3GL developers because mapping things like symbolic names to binary addresses is deterministic, even for complex multitasking environments.

**Sets and Graphs**

It is beyond the scope of this book to describe how ubiquitous various mathematics, especially set and graph theory, are in the computing space. However, a few implications of that are very important to the OO paradigm and MBD in particular.

---

\(^{16}\) HIPO charts were one of the earliest attempts at graphical descriptions of programs, so old that I forgot what the acronym actually means. When she reviewed this book, Rebecca Wirfs-Brock reminded me that it means Hierarchy, Input, Process, Output. Rebecca is a dear person and fine methodologist, but I am concerned that cluttering her mind with such arcane trivia might be displacing useful knowledge.
• The computing space is deterministic. That’s because some kind of rigorous mathematics underlies everything in it, however well disguised that might be. This has enormous implications for automation.

• The computing space is disciplined. Everywhere we turn we are constrained by rules like those for language syntax, data structures, identity, GUI design, and so forth. In reality there is very little freedom in implementing problem solutions once we have defined the specific computing environment. Creativity in software development lies in designing solutions, not implementing them.

• The computing space is ultimately based on hardware computational models. However abstract our solution constructs are, the underlying mathematics ensures that they will all map unambiguously into hardware computation.

• The first methodological step in any software construction is expressing the ill-defined, imprecise, ambiguous notions of non-Turing customer spaces in notations that are complete, precise, and unambiguous with respect to the computing space. The mathematics underlying those notations that enable rigorous expression must necessarily be the same mathematics that underlies the computing space.

The methodologies of SD were not simply encyclopedias of a potpourri of hard-won lessons learned in the development trenches prior to 1970. Instead, they became cohesive philosophies of software construction that were based on solid mathematics. Every software construction methodology that has stood the test of time sits on that sort of mathematical platform, however informal its exposition might be and however simple its notation might seem. This was a two-way street; computing has been the driving force in developing entirely new branches of mathematics, from relational theory through type theory to exotic topological theories for non-Turing computation.

**Normal Form (NF)**

One of the most useful tools to evolve concurrently with SD was a particular branch of general set theory that underlies data storage and the static description of programs. That is, it formalizes and constrains the way data is defined, modified, and related to other data. E. F. Codd first defined it in the late ’60s to describe relational databases.17

Codd’s Relational Data Model (RDM) has since been incorporated into both the OO paradigm and extensions of SD. For example, the descriptor attribute associated

---

with knowledge in OO classes is a term Codd introduced in 1970. A suitably arcane branch of mathematics with much broader scope has since been developed, called relational theory. The OO paradigm is based on that broader view.

We can avoid the mathematics of the set theory that forms the basis of NF here and provide the executive summary view because all one needs to know to develop good applications are the implications of the theory. Though NF is more broadly applied, it is most easily understood in terms of tables. The table itself represents an \textit{n-ary relation} of table entities (the rows) that each share the same \textit{n attributes} (the columns). That is, the table describes a set of entities that share characterization in terms of the attributes. Each attribute identifies a particular semantic domain of possible values. Each table entity (a row or \textit{tuple}) has a unique identity and a set of specific values (the table cell values) for the attributes (see Figure 1-3).

\textbf{Figure 1-3} \hspace{1em} \textit{Mapping of problem space entities to an n-ary relation. Each identifiable entity becomes a tuple in the relation. Each relation captures a set of unique values of characteristics that are common to each entity.}

In more practical terms, a column attribute represents a particular data semantic that is shared by every row in the table; it describes \textit{what} the data in the cells of that column is. The cells then represent specific values for that semantic. Then each row represents a set of attribute values that is uniquely associated with the entity. The uniqueness is tied to identity; each row must have a unique identity that distinguishes it from other rows, and the attribute values must depend on that identity. In Figure 1-3 that identity is the name of a person.
In the more restrictive relational data model that applies to databases, some subset of one or more embedded attributes comprises a unique identifier for the table row, and no other row may have exactly the same values for all the identifier attributes. The identifier attributes are commonly known collectively as a \textit{key}. Other attributes that are not part of the key may have duplicated values across the rows, either singly or in combination. But the specific values in a given row depend explicitly on the row key and only on the row key.

Relational theory is about the ways particular entities in different n-ary relations (tables) can be related to each other. For that theory to be practically useful, there need to be certain rules about how attributes and entities can be defined within an n-ary relation. That collection of rules is called \textit{normal form}, and it is critical to almost all modern software development.

The kernel of NF has three rules for compliance:

1. \textit{Attributes must be members of simple domains}. What this essentially means is that attributes cannot be subdivided further (i.e., their values are scalars). That is, if we have a table House with attributes \{Builder, Model\}, we cannot also subdivide Model into \{Style, Price\}. The problem is that the Style/Price combinations would be ambiguous in a single-valued attribute cell for Model in a row of House. So the House table must be composed of \{Builder, Style, Price\} instead.

2. \textit{All non-key attributes, compliant with 1st NF, are fully dependent on the key}. Suppose a House table is defined with attributes \{Subdivision, Lot, Builder, Model\} where Subdivision and Lot are the key that uniquely identifies a particular House. Also suppose only one builder works on a subdivision. Now Builder is really only dependent upon Subdivision and is only indirectly dependent upon Lot through Subdivision. To be compliant we really need two n-ary relations or tables: House \{Subdivision, Lot, Model\} and Contractor \{Subdivision, Builder\}. (The RDM then provides a formalism that links the two tables through the Subdivision attribute that can be implemented to ensure that we can unambiguously determine both the Model and the Builder values for any House. That notion is the subject of a later chapter.)

3. \textit{No non-key attributes, compliant with 2nd NF, are transitively dependent upon another key}. Basically, this means that none of the non-key attribute values depend upon any other key. That is, for House \{Subdivision, Lot, Model\}, if we have a Subdivision = 8, Lot = 4, and Model = flimsy, the Model will remain “flimsy” for Subdivision = 8 and Lot = 4 regardless of what other rows we add or delete from the table.

Generally, most developers depend on the following mnemonic mantra to deal with 3rd NF: \textit{An attribute value depends on the key, the whole key, and nothing but the key.}
Other NF rules have been added over the years to deal with specialized problems, especially in the area of OO databases. We won’t go into them here because they are not terribly relevant to the way we will build application models and produce code. However, the value of the three normal forms above is very general and is particularly of interest when defining abstractions and their relationships.

Data Flows

Data was a second-class citizen until the late 1960s. Most of the early theoretical work in computer science was done in universities where the computer systems were solving the highly algorithmic problems of science and engineering. However, by the late 1960s there was an epiphany when people became aware that most of the software in the world was actually solving business problems, primarily in mundane accounting systems.

In particular, business problems were not characterized by algorithms but by USER/CURD processing of data. For one thing, core business problems were not algorithmic rocket science—doing a long division for an allocation was pretty cutting edge. On the other hand, those very simple calculations were done repeatedly on vast piles of data. What the business community needed was a way to handle data the same way the scientific community handled operations. This led to various flavors of data flow analysis. The basic idea was that processes were small, atomic activities that transformed data, and the processes were connected together by flows of data, as illustrated in Figure 1-4.

As it happens, when we make a data flow graph, it represents a breadth-first flow of control. The processes are self-contained and atomic rather than decomposed. So the problem solution is defined by the direct connections between those processes. The connections are direct and depend only on what needs to be done to the data next. If we want to solve a different problem, we change the connections and move

18. It was Nicolas Wirth who raised consciousness about the importance of data with his work on Pascal. In particular, one of his books was titled, Programs = Algorithms + Data Structures, whose egalitarian view was a quite seminal notion at the time.

19. USER = Update, Sort, Extract, Report and CRUD = Create, Retrieve, Update, Delete. All are simple operations on data.

20. This realization gave birth to data processing, which dominated the ’60s and ’70s. Later that morphed into today’s information technology.

21. Obviously that is no longer true. Many large businesses have entire departments devoted to operations research, often larger than corresponding departments in universities. My earliest practical exposure to operations research was an algorithm to allocate an advertising budget to a variety of national, regional, and local media outlets (TV, magazines, etc.) for a Fortune 500 Company in the ’70s. It was a pretty impressive linear programming application.
the data through the same atomic processes differently. Data flows represented a precursor of the peer-to-peer collaboration that is central to the OO paradigm and will be discussed in detail later.

Alas, this view of the world did not mesh well with the depth-first, algorithm-centric view of functional decomposition. Trying to unify these views in a single software design paradigm led to some rather amusing false starts in the ’70s. Eventually people gave up trying and lived with a fundamental dichotomy in approaches. For data-rich environments with volatile requirements, they focused on the OO paradigm as the dominant paradigm. Meanwhile, functional programming evolved to deal with the pure algorithmic processing where persistent state data was not needed and requirements were stable.
In many respects the purest forms of data flow analysis were the harbingers of the modern OO paradigm. As we shall see later, the idea of small, self-contained, cohesive processes has been largely canonized in the idea of object behavior responsibilities. Ideas like encapsulation and peer-to-peer collaboration have driven a breadth-first flow-of-control paradigm. But in two crucial respects they are quite different. The OO paradigm encapsulates both the behavior and the data in the same object container, and the data flows have been replaced with message flows. The important point to take away from this subtopic is that where SD was based on functional decomposition, the OO paradigm is more closely aligned with data flows.

State Machines

Meanwhile, there was another branch of software development that was in even deeper trouble than the business people. Those doing real-time and/or embedded (R-T/E) hardware control systems were experiencing a different sort of problem than maintainability. They just couldn’t get their stuff to work.

Basically they had three problems. First, such control systems were often used in environments that were extremely resource limited. They didn’t have the luxury of stuffing an IBM 360 into a washing machine or railroad signal. Second, the supporting tools that other developers used weren’t available in their environment. This was especially true for debugging programs when they might not even have an I/O port for print statements, much less a source-level debugger.

The third problem was that they were dealing with a different view of time. Events didn’t happen in synchronization with predefined program flow of control; they occurred randomly in synchronization with the external hardware, and that external hardware never heard of Turing unless it happened to be a computer. Their asynchronous processing completely trashed Turing’s ordered world of preordained sequence, branch, and iteration. The tools have gotten better over the years, but the R-T/E people still have the toughest job in the software industry.

The tool that made life bearable for the R-T/E people was the Finite State Machine (FSM).22 We have no idea who first wrote a program using an FSM, but whoever it was deserves to be in the Software Hall of Fame along with Ada Lovelace and crew. It basically enabled asynchronous processing in a serial Turing world. It also made marvelously compact programs possible because much of the flow of control logic was implicit in the FSM rules for processing events and static sequencing constraints represented by transitions, as shown in Figure 1-5.

22. If you have led a sheltered existence and don’t know exactly what an FSM is or why it is a big deal, have patience. They are the primary topic of several later chapters. For those who have taken courses about finite state automata, it should be noted that the FSM used in software systems has added some capability over the classical theoretical machines.
However, the really interesting thing about FSMs is that they happen to address a number of the same issues that are the focus of OOA/D. In particular, they provide an extraordinary capability for reducing the degree of coupling and truly encapsulate functionality. They also enforce basic good practices of OO design. If we were given the task of coming up with a single construct to capture and enforce core elements of the OO paradigm, the FSM would be that construct. But we will defer the discussion of how that is so until Part III.

Figure 1-5  Example of a finite state machine for a simple traffic light. The traffic light migrates through states in a manner that is constrained by allowed transitions. Transitions from one state to the next are triggered by external events.
Index

Information in figures and tables is denoted by \( f \) and \( t \).

A
AAL. See Abstract Action Language (AAL)
Abstract Action Language (AAL)
abstraction and, 475–476
Action Data Flow Diagram and, 373, 476–478
ATM example for, 484–488
garage door opener example for, 480–484
3GLs vs., 475–476
methods and, 52
syntax, 478, 478–480t, 480
Abstract data type (ADT), 168, 192, 195–197, 333, 335–336
Abstraction
abstract action languages and, 475–476
of aggregates, 335–341
anthropomorphization and, 206–207, 343–344
application partitioning and, 116–117, 117f
of associations, 236
basic, 332
of behavior, 203–204
behavior and, 158, 198
characteristics and, 37–38
choosing correct, 341–351
collaboration and, 396
conditionality and, 252
definition of, 37, 161–162
delegation and, 330–331
domain experts and, 163
emphasis and, 38
encapsulation and, 39
invariants and, 79, 231
levels of, 35, 36f, 116–117, 117f, 350–351, 396–397, 397f
Liskov Substitution Principle and, 326
logical indivisibility and, 396
of methods, 197–198
Model-Based Development and, 35
modification of reality by, 37–38
in object-oriented paradigm, 35, 36f
of object responsibilities, 399–400
of people's actions, 343–344
polymorphism and, 59
as proactive, 162
of problem space entities, 163
representation and, 161–167
of roles, 163
to scalar, 193, 228
sequencing and, 396–397, 397f
in state machine design, 438
subclasses and, 309
subject matter and, 308, 345–349
subsystem implementation and, 110
of subsystems, 128
transitions and, 422
Access constraint, associations and, 238, 272
Accessor(s)
knowledge, 204
as synchronous service, 371
Action(s)
definition of, 379, 427–428
as dynamic, 427
timing of execution, 381–382
Action Data Flow Diagram (ADFD), 373, 476–478
Action languages, 373–374
Addresses, memory, 163
ADFD. See Action Data Flow Diagram (ADFD)
Ad-hoc polymorphism, 59, 60, 80
ADT. See Abstract data type (ADT)
Aggregates, abstraction of, 335–341
Air traffic control example, of finite state machines, 410–411
Allocation, requirements, 130, 147
Alphabet
definition of, 378
in finite state machines, 378–379
knowledge and, 379
ALU. See Arithmetic Logic Unit (ALU)
Analysis
domainmaintainability and, 69–71
object-oriented analysis and, 70
traditional view of, 69–70
object-oriented, 32–33, 36f
domain analysis and, 70
finite state machines and, 377
static model and, 154–155
translation and, 46
problem, 32
of requirements, 70–71
semantic, of subsystems, 146–147
in structured development, 6, 32
AOP. See Aspect-Oriented Programming (AOP)
API. See Application programming interfaces (API)
Application partitioning, 72. See also
Subsystem(s)
basic concepts of, 107–118
bridge definitions and, 227
collaborations and, 270, 272
collaborations and, 270, 272
collaborations and, 270, 272
collaborations and, 270, 272
cohesion and, 108
event of, 130–145, 147–150
feedback to, 231
implementation hiding and, 108
importance of, 105–107
interfaces and, 108, 118
invariants and, 108
legacy replacement event of, 147–150
levels of abstraction and, 116–117, 117f
Liskov Substitution Principle and, 326
maintainability and, 82, 106
pet care center example of, 130–145
problem space abstraction and, 108
static nature of, 158
subject matter and, 109–113
subsystems and, 106, 127–128, 227
synchronous services and, 369–370
Application programming interfaces (API), in structured development, 6
Architecture, 13–14
Arithmetic Logic Unit (ALU)
Basic Assembly Language and, 19
early instructions to, 18
"Artwork," 90
Aspect-Oriented Programming (AOP), 238
Association(s)
abstraction of, 236
access constraint and, 238, 272
ATM as example for identification of, 273–277
as binary, 236
cardinality in notation for, 239, 239f
classes
associations appropriate for, 267
problem space and, 264–265
reification and, 267–268, 268f
value of, 265
classes and, 234
collaborations and, 270, 272
collaborations and, 270, 272
collaborations and, 270, 272
conditionality in
abstraction and, 252
access and, 254–255
in association identification, 271
definition of, 250
elimination of, 252
encapsulation and, 254
hierarchy and, 252–254, 253f
implementation and, 252
notation for, 240, 240f
problem space and, 252
problems with, 250–251
superclasses and, 254
constraints
association identification and, 270–271
attributes and, 262
instances and, 261–262
loops and, 262
notation for, 240, 240f
order and, 262
contracts and, 272
critical, 269
definition of, 234
dependency and, 273
directions of, 244
discriminator
in notation for, 239, 239f
referential attribute and, 291
subclasses and, 300–301
of events with transitions, 424, 449–450
examples in identification of, 273–278
explicit relations and, 272–273
explicit selections and, 273
identification of, 269–273
implementation and, 235
importance of, 233, 270
instantiation and, 234, 235f, 238
locality of scope and, 261
logical connections and, 242–250
loops
constraints and, 262
referential integrity and, 292–295, 293f
multi-class, 236–237
multiple, collaborations and, 242–243, 243f
multiplicity
in association identification, 271
notation for, 239, 239f, 241, 241f
problem space and, 256
navigation and, 235, 236
notation for, 233, 239–242, 239f
obscure, 269
obvious, 269
participation and, 260–261
path
definition of, 237
logical connections and, 242
navigation through, 237, 237f
pet care center as example for
identification of, 277–278
physical connections and, 272
problem space and, 236
questions for identification of, 270
responsibilities and identification of, 270
role in notation for, 240, 240f
roles and, 242, 244–246, 245f
selection and, 260–261
selection consistency and, 261
separation of concerns and, 236
specification and, 273
subject matter and, 269
supplementation of, 257–260
symmetric, 245, 245f
ternary, 236–237
as underutilized, 233
use cases in identification of, 269
Asymmetry, of association roles, 245
Asynchronicity, 42–43
Asynchronous behavior communication, 363–369
Asynchronous implementation, 364
Asynchronous processing, 364
ATM
as Abstract Action Language example, 484–488
as association identification example, 273–277
as class identification example, 173–181
as invariant modeling example, 84–89
as responsibility identification example, 209–222
as state machine design example, 451–473
Attribute(s)
in Abstract Action Languages, 480
abstract data types and, 192, 195–197
abstraction-to-scalar and, 193
accessors and, 205
association constraints and, 262
contracts and, 196
data storage vs., 193
definition of, 168
dependent
knowledge integrity and, 283–285, 283f, 284f
notation for, 283, 283f
as dependent on key, 24
identifier, 24
introduction of term, 23
key and, 24
Attribute(s) (continued)
  knowledge obligations and, 192
  knowledge vs., 191–192
  as members of simple domains, 24
  methods and, 52, 197–207
  non-key, 24
  normalization of, knowledge integrity and, 285–289, 287f, 288f
  notation, 191–193
  object characteristics and, 50–52
  operations and, 197–207
  problem space entities and, 192
  referential
    association discriminator and, 291
    definition of, 290
    identity and, 290–292, 291f, 292f
    responsibility and, 50
    state and, 194–195
    static model and, 192–193
    strong typing and, 197
    value vs., 52
  Automata, finite state, 378–390
  Automated home kitchen, as finite state machine example, 409–410

B
  BAL. See Basic Assembly Language (BAL)
  Balcer, Marc, 154
  Bank ATM software
    as class identification example, 173–181
    as invariant modeling example, 84–89
    as responsibility identification example, 209–222
  Basic Assembly Language (BAL)
    Arithmetic Logic Unit and, 19
    in history of software development, 4
    instruction mnemonics in, 53
    introduction of, 18–19
  Behavior(s). See also Responsibility(ies)
    abstraction and, 158, 198
    abstraction of, 203–204
    anthropomorphization and, 206–207
    asynchronous communication, 363–369
    checking steps, 219
    cohesion in identification of, 202–203
    collaboration between, 361
    complementarity of, 202
    context and, 157
    coupling and, 16
    by data, 16
    finite state machines and, 391–395
    identifying, 201–206
    identifying errors with, 221–222
    importance of, 358–361
    knowledge attributes and, 344–345
    knowledge integrity and, 281
    knowledge vs., 156–158, 391–395
    Liskov Substitution Principle and, 325
    logical cohesion and, 202
    navigation to, 244
    perspective and asynchronous, 366
    polymorphism and, 157
    preconditions and, 388
    by reference, 16
    responsibilities, 198
    self-containment of, 202, 221
    sequencing and, 362
    specificity of, 203
    state machine design and, 438–439
    states and, 428–429
    subclasses and, 314–315
    in synchronous services, 369–370
    testability of, 221
    time slicing and, 368–369
    transitions and constraints on, 360–361
  Behavioral state machine, 382
  Blitz
    object, 169–172
    for responsibility identification, 208–209
    for subsystem identification, 146
  Block structuring, procedural, 20
  Booch notation, 29
  Boundaries, subsystem, 121
  Branch, in Turing Machine, 17
  Breadth-first processing
    in object-oriented paradigm, 44–52
  Bridge(s)
    defining, 147
    descriptions, 129
    model, 125–127, 126f
    subsystems and, 122–127, 227

C
  C++, 29
  Cardinality, in association notation, 239, 239f
Categorization
  subclasses and, 307–312, 310f, 311f

Characteristics
  abstraction and, 37–38
  attributes and, 50–52
  coalescence of, 38
  de-emphasization of, 37
  elimination of, 37
  generalization of, 37–38
  object, 49–52

Class(es)
  association
    associations appropriate for, 267
    problem space and, 264–265
    reification and, 267–268, 268f
    value of, 265
  associations and, 234
  ATM example in identification of, 173–181
  attributes, 168 (See also Attribute(s))
    in class diagram, 155
    definition of, 49
    delegation and, 165–167
    diagram, 155–156, 186–189, 187f, 189f
    embedding of, in other classes, 322
    entities and, 51f
    examples for identification of, 172–186
    identification of, 169–172
    instances and, 51f
    Liskov Substitution Principle and, 323–328, 328f
    logical indivisibility and, 164–165
    method, 168
    notation, 167–168
    objects and, 51f
    operation, 168
    pet care center example in identification of, 181–186
    responsibilities of, 169–172
    subclasses
      abstraction and, 309
      behaviors and, 314–315
      categorization and, 307–312, 310f, 311f
      cohesion and, 307, 328–329
      composition with, 317–323, 318f, 320f, 323f, 324f

  definition of, 300
  disadvantages of, 328–329
  discriminator and, 300–301
  disjoint, 315–317, 316f
  generalization and, 306–307
  inclusion polymorphism and, 312–315, 313f, 314f
  instantiation of, 303
  member characteristics, 303
  multidirectional, 303, 304f, 317–323, 318f, 320f, 323f, 324f
  notation, 300–303
  in problem space, 311–312
  problem space and, 308, 309
  properties and, 300
  reasons for, 310–311
  rules, 300–303
  semantic connection with superclasses, 308–309
  simplicity in, importance of, 307
  specialization and, 304–307
  in superclasses, 303
  subclasses as, 302
  superclasses as, 302
  subclasses in, 303

Client
  definition of, 119
  in object-oriented context, 203
  as state, 204

Client/service relationships
  application partitioning and, 108–109, 114–116, 115f
  subsystems and, 147

COBOL, introduction of, 19

Codd's Relational Data Model, 22–25

Code
  hacker, 4
  size, invariant modeling and, 83–84

Coff, E. F., 22
Cohesion
application partitioning and, 108
in behavior identification, 202–203
bleeding of, across subsystems, 86, 226
composition and, 321–322
design and, 40
lack of, in software development history, 14
logical, behaviors and, 202
in object-oriented paradigm, 40
separation of concerns and, 34
subclasses and, 307, 328–329
subject matter and, 121
of subsystems, 136

Collaboration(s)
abstraction and, 396
associations and, 270, 272
asynchronous model for, 363–369
asynchronous services and, 372
between behaviors, 361
definition of, 237
diagrams, 186–189, 187f, 189f, 208, 213f, 440, 446–447
handshaking and, 441
instantiation vs., 235–236
multiple associations and, 242–243, 243f
referential integrity and, 279
state machine design and, 439
transitions and, 360–361

Commonality of superset, associations and, 261, 272

Communication models
asynchronous, 42–43
in object-oriented paradigm, 42–44

Compiler, introduction of, 19

Complex Number library class, 38

Composition, 317–323, 318f, 320f, 323f, 324f

Computing space
definition of, 68
as deterministic, 22
as disciplined, 22
hardware computational models as basis of, 22
invariants in, 78
as mathematically based, 68–69
problem space vs., 63–69

responsibilities in, vs. problem space, 340–341

Concerns, separation of
cohesion and, 34
decoupling and, 35
isolation and, 34
in object-oriented paradigm, 34–35
subject matter and, 34

Concurrent implementation, 364
Concurrent processing, 281

Conditionality
abstraction and, 252
access and, 254–255
in association identification, 271
in association notation, 240, 240f
definition of, 250
elimination of, 252
capsulation and, 254
hierarchy and, 252–254, 253f
implementation and, 252
problem space and, 252
problems with, 250–251
relationships and, 251
superclasses and, 254

Connections, logical, 242–250

Consistency
concurrent processing and, 281
knowledge integrity and, 281
of selection, associations and, 261

Constraints
association identification and, 270–271
association loops and, 262
in association notation, 240, 240f
attributes and, 262
finite state machines and, 395–396
instances and, 261–262
order and, 262
problem space, 263–264, 263f
sequencing, 397f, 398

Context
behavior and, 157
dynamic, 157
event, 425
functional decomposition and, 10
knowledge integrity and, 280
knowledge value and, 157
Liskov Substitution Principle and, 326
methods and, 198
problem, 448
receiver, in event naming, 434
for requirements, 129
sender, in event naming, 435
state and, 416
static models and, 154
transition, 425
Contract(s)
abstract data types and, 196
associations and, 272
design by, 203–204, 280–281, 387–390
programming by
functional decomposition and, 8
in structured development, 6
Control, flow of, 389, 401, 406, 417, 418
Coupling, 14–17
behavior by reference and, 16
behavior by value and, 16
data by reference and, 16
data by value and, 16
dependency and, 15
interfaces and, 118
logical, 15
message identifier and, 15
physical, 15
"Create" event, 384, 384f
Critical associations, 269
Cromwell, Andrew, 415
Customer
definition of, 67–68
environment, 70
input for subsystem identification, 145–146
invariant modeling and, 79
solution, 70
space
abstraction and, 162
notation for, 22

D
Daisy-chaining, of events, 446–447, 449
Data
coupling and, 16
flows, 25–27, 26f
invariant modeling and, 80–81
knowledge vs., 334
by reference, 16
in relational vs. object-oriented
paradigms, 296–297
storage, attributes vs., 193
by value, 16
Data Flow Diagram (DFD), 476–477. See also Action Data Flow Diagram (ADFD)
DbC. See Design by contract (DbC)
Decomposition, functional
appeal of, 7–8
context and, 10
dependency and, 10–11
functional isolation in, 8
maintainability and, 10
power of, 7
programming by contract in, 8
redundancy and, 8–10, 9f
in structured development, 7–11
unit testing and, 11
Decoupling
message paradigm and, 49
separation of concerns and, 35
of subsystems, 124
Delegation, 165–167
abstraction and, 330–331
as alternative to generalization, 329–331, 331f
definition of, 329
knowledge and, 334–335, 335f
problem space and, 166, 330
Dependency
associations and, 273
coupling and, 15
functional decomposition and, 10–11
management, 15
Dependent attributes
knowledge integrity and, 283–285, 283f, 284f
notation for, 283, 283f
Depreciation, as example of invariant modeling, 93–98
Design
cohesion and, 40
history of, 4–5
instances and, 231
object-oriented
Design (continued)
definition of, 33
elaboration and, 45
levels of abstraction and, 36f
patterns, 78, 231, 312
reuse, 66
role-based, 421
of state machines, 437–450
in structured development, 6
subsystem, 110–111
Design by contract (DbC), 203–204, 280–281, 387–390
Deterministic, computing space as, 22
Development, structured
analysis in, 6, 32
application programming interfaces in, 6
characteristics of, 5–6
design in, 6
functional decomposition in, 7–11
functional isolation in, 6
graphical representation in, 5
history of, 4–11, 5–6
maintainability and, 6
problems in, 8
programming by contract in, 6
reliability and, 5
top-down development in, 6
DFD. See Data Flow Diagram (DFD)
Diagram(s)
action data flow, 373, 476–478
class, 155–156, 186–189, 187f, 189f
collaboration, 186–187, 208, 213f, 440, 446–447
data flow, 476–477
sequence, 214f
state transition, 358, 383, 383f
Dijkstra, E., 12
Disciplined, computing space as, 22
Discriminator
in association notation, 239, 239f
referential attribute and, 291
subclasses and, 300–301
Disjoint generalization, 303, 315–317, 316f
Domain, simple, attributes as members of, 24
Domain analysis
maintainability and, 69–71
object-oriented analysis and, 70
traditional view of, 69–70
Domain expert, definition of, 163
Domain Modeling, 70–71
Drum memory, 4
Duplication, of subsystems, elimination of, 146
Dynamic context, 157
Dynamic models, 73–75
definition of, 355
learning curve with, 376
E
Elaboration
breadth-first processing and, 45–47
stages of, 45
translation vs., 45–47
Embedding, of classes in classes, 322
Encapsulation
abstraction and, 39
conditionality and, 254
in object-oriented paradigm, 39
polymorphism and, 59
synchronous services and, 199
Engine, transformation, 45, 66, 238, 279–280
Entity(ies). See also Object(s)
actions and, 427
classes and, 51f
complexity of, 165
components of, 165–166
defining, example of, 177–179
definition of, 38
gathering of candidate, 169
example, 174
instances and, 51f
intangible, 163
objects and, 51f
objects vs., 164
problem space
abstraction of, 163
attributes and, 192
definition of, 163
problem space abstraction and, 37
roles as, 163
scrubbing, example of, 177–179
Entry action, 374
Environment
    customer, 70
distributed, 121
Event(s)
    assigning, 425–426
    association of, with transitions, 424, 449–450
    context, 425
    "create," 384, 384f
daisy-chaining of, 446–447, 449
    definition of, 380
    in finite state machines, 380, 381
generators, 205–206
    ignoring, 424
    messages, 381
    as messages, 423
    naming conventions for, 433–435
    polymorphic, 426
    queue, 380–381, 431
    receiver context in naming of, 434
    self-directed, 381, 400–403, 400f, 402f
    sender context in naming of, 435
    serialization of, 432–433
    superstates and, 385
    transitions and, 423–424
    transition vs., 381
Executable UML: How to Build Class Models (Starr), 154, 233, 320
Executable UML (Mellor & Balcer), 154
Execution model, 430–433
Expert, domain, definition of, 163
Explicit relations, associations and, 272–273
F
    Falkland, Lord, 77
Finite State Automata (FSA), 378–390
Finite state machines (FSMs), 27–28, 28f
    action execution timing in, 381–382
    action in, 380
    air traffic control example of, 410–411
    alphabet in, 378–379
    basic automata, 378–390
    behavior and, 391–395
    collaborations and, 361
    constraints and, 395–396
    event queue in, 380–381
    events in, 380, 381
flow of control and, 401
    garage door opener example of, 407–408, 409f
    handshaking in, 361, 403–407
    identifying, 390–407
    kitchen example of, 409–410
    knowledge and, 391–395
    Mealy, 382
    Moore, 382
    notation for, 383–386
    object life cycles and, 362–373
    object-oriented analysis and, 377
    object-oriented practice and, 360f
    postconditions in, 387–388
    reasons for use of, 357–358
    rock example of, 410–413
    self-directed events and, 400–403, 400f, 402f
    sequence management and, 395–407
    sequencing and, 357, 361
    state in, 359, 378
    state transition table and, 386, 386f
    time slicing and, 368–369
    transition in, 359, 378
Flawed requirements, 131
    "Floating" transition, 384, 384f
Flow(s)
    of control, 389, 401, 406, 417, 418
    data, 25–27, 26f, 477
FORTRAN, introduction of, 19
FSA. See Finite State Automata (FSA)
FSM. See Finite state machine (FSM)
Functional decomposition
    appeal of, 7–8
    context and, 10
    dependency and, 10–11
    divisions in, 7, 7f
    functional isolation in, 8
    maintainability and, 10
    power of, 7
    programming by contract in, 8
    redundancy and, 8–10, 9f
    in structured development, 7–11
    unit testing and, 11
Functional isolation
    in functional decomposition, 8
    in structured development, 6
INDEX

G
Garage door opener example
  of Abstract Action Languages, 480–484
  of finite state machine, 407–408, 409f
Generalization
  alternatives to, 328–332
  of characteristics, 37–38
  in class diagram, 155
  definition of, 299
  delegation as alternative to, 329–331, 331f
  disjoint, 303, 315–317, 316f
  inclusion polymorphism and, 60
  inheritance and, 56
  joint, 303
  parametric polymorphism as alternative
    to, 331–332
  in problem space, 309
  problem-space abstraction and, 56
  properties and, 304–305
  in set theory, 54–56, 54f, 55f
  specialization and, 304–307, 305f, 306f
  subclasses and, 306–307
Generators, event, 205–206
Generality, 59–60, 61
"Go To Statement Considered Harmful"
  (Dijkstra), 12
Graphical C++, 29
Graphical representation, in structured
development, 5
Graphs, 21–22

H
Hacker, history of term, 4
Hacker code, 4
Hacker Era, 4
Handshaking, 361, 403–407, 441–442
Hardware computational models, 18, 22
Hardware interface, as example of invariant
  modeling, 89–93
Hardware register, knowledge and, 393
Harel model, 375, 385
Harris, Sydney J., 29
Hierarchy, association conditionality and,
  252–254, 253f
HiPO charts, 21
History
  of software development, 3–5
  of structured development, 5–6

I
Identification
  of association roles, 245
  of associations, 269–273
  of behaviors, 201–206
  of candidate objects, 163
  of classes, 169–186
  of finite state machines, 390–407
  of invariants, 70
  of responsibilities, 209–222, 222–231
  of subsystems, 119–122, 145–147
Identifier attributes, 24
Identity, referential attributes and, 290–292,
  291f, 292f
Implementation
  associations and, 235
  asynchronous, 364
  concurrent, 364
  conditionality and, 252
  hiding, application partitioning and, 108
  polymorphism vs., 57
  synchronous, 364
Inclusion polymorphism, 59, 60, 312–315,
  313f, 314f, 323
Indicator, 392–393
Indivisibility, logical
  abstraction and, 396
  classes and, 164–165
  levels of, 40–41
  object and, 41
  in object-oriented paradigm, 40–41
  object responsibility and, 41
  power of, 41
  process and, 41
  subsystem and, 40
Industrial Revolution, 3
Inherent polymorphism, 59, 60
Inheritance
  generalization and, 56
  generalization vs., 57
  multiple, 317–323, 318f, 320f, 323f, 324f
  polymorphism vs., 57
  as rule, 56
Instance(s)
  association constraints and, 261–262
  classes and, 51f
  definition of, 50
  design patterns and, 231
entities and, 51f
objects and, 51f
object vs., 50
Instantiation
associations and, 234, 235–236, 235f, 238
collaboration vs., 235–236
of subclasses, 303
Instantiators
attributes and, 205
as synchronous service, 371
Instruction mnemonics, in BAL, 53
Intangible entities, 163
Integrity
assumptions in, 279
knowledge
attribute normalization and, 285–289, 287f, 288f
behaviors and, 281
consistency and, 281
collaboration vs., 235–236
context and, 280
dependent attributes and, 283–285, 283f, 284f
scope of, 279
snapshots and, 282
timeliness and, 280–281
transformation engine and, 279–280
practices in, 279
referential
association loops and, 292–295, 293f
collaborations and, 279
definition of, 280–281
deception example of, 93–98
collaborations and, 279
identity and, 290–292, 291f, 292f
participants and, 289
referential attributes and, 290–292, 291f, 292f
scope of, 279, 289
selection and, 289
transformation engine and, 279–280
transformation and, 280
Interfaces
application partitioning and, 108, 118
coupling and, 118
large-scale reuse and, 118
Invariant(s)
abstraction and, 79, 231
application partitioning and, 108
in computing space, 78
definition of, 78–79
in design by contract, 387
identification of, 70
Liskov Substitution Principle and, 326
modeling, 71
ATM example of, 84–89
collaboration vs., 235–236
data side in, 80–81
definition of, 78–81
definition of, 78–81
depreciation example of, 93–98
depreciation example of, 93–98
equalities of, 84–103
examples of, 84–103
hardware interface example of, 89–93
importance of, 77
invariant side in, 78–80
maintainability and, 82
problem space in, 79, 80
reduced code size and, 83–84
remote POS entry example of, 99–103
requirements and, 80
rewards of, 81–84
from problem space, extraction of, 342–343
in subsystem descriptions, 128
Isolation
functional
in functional decomposition, 8
in structured development, 6
separation of concerns and, 34
Iteration, in Turing Machine, 18
J
Johnson, Samuel, 333
Joint generalization, 303
K
Kettering, Charles F., 3
Key
definition of, 24
dependence on, 24
Kitchen, automated, as finite state machine
example, 409–410
Knowledge
abstraction, subsystem abstraction and,
accessors, 204
aggregate, abstraction of, 335–341
alphabet and, 379
anthropomorphization and, 333–334
attributes vs., 191–192
### Knowledge (continued)

- behavior and, 344–345
- behavior vs., 156–158, 391–395
- classification of, 333–334
- data vs., 334
- delegation and, 334–335, 335f
- finite state machines and, 391–395
- hardware register and, 393
- integrity
  - attribute normalization and, 285–289, 287f, 288f
  - behaviors and, 281
  - consistency and, 281
  - context and, 280
  - dependent attributes and, 283–285, 283f, 284f
  - scope of, 279
  - snapshots and, 282
  - timeliness and, 280–281
  - transformation engine and, 279–280
- as intrinsic, 334
- level of abstraction and, 350–351
- Liskov Substitution Principle and, 327–328
- nature of, 334–335
- navigation to, 244
- obligations, attributes and, 192
- requirements and, 346
- responsibilities, 167–168
- as static, 157
- subject matter and, 345–349
- synchronous access of, 430
- synchronous services and, 371
- underlying entity and, 344–345

### Logical cohesion

- behaviors and, 202

### Logical connections

- 242–250

### Logical coupling

- 15

### Logical indivisibility

- abstraction and, 396
- classes and, 164–165
- levels of, 40–41
- object and, 41
- in object-oriented paradigm, 40–41
- object responsibility and, 41
- power of, 41
- process and, 41
- subsystem and, 40

### Lombardi, Vince

- 30

### Loops, association

- constraints and, 262
- referential integrity and, 292–295, 293f

### LSP

- See Liskov Substitution Principle (LSP)

### M

- MacPherson, James, 333

### Maintainability

- application partitioning and, 82, 106
- delegation and, 166
- domain analysis and, 69–71
- functional decomposition and, 10
- history of term, 4
- importance of, 30
- invariant modeling and, 82
- object-oriented paradigm and, 29, 30–31
- structured development in, 6
- subclass simplicity and, 307

### Mapping

- 425

### Marvel, Andrew

- 415

### Mealy finite state machine

- 382

### Mealy model

- 374–376

### Mellor, Steve

- 77, 123, 124, 154

### Memory

- addresses, 163
- drum, 4

### Merchant of Venice, The (Shakespeare)

- 53

### Message identifier, coupling and

- 15

### Message paradigm

- 47–49

### Methods

- abstraction of, 197–198
- attributes and, 52, 197–207
- collaborations and, 198
- context and, 198
definition of, 168, 199
preconditions and, 388
Midsummer Night’s Dream, A
(Shakespeare), 105
Mnemonics, in BAL, 53
Model(s)
communication
asynchronous, 42–43
in object-oriented paradigm, 42–44
domain, 70–71
dynamic, 73–75
definition of, 355
learning curve with, 376
execution, 430–433
Harel, 375, 385
invariant, 71
ATM example of, 84–89
code size and, 83–84
customer and, 79
data side in, 80–81
definition of, 78–81
depreciation example of, 93–98
examples of, 84–103
hardware interface example of, 89–93
importance of, 77
invariant side in, 78–80
maintainability and, 82
problem space in, 79, 80
remote POS entry example of, 99–103
requirements and, 80
rewards of, 81–84
Mealy, 374–376
Moore, 374–376, 434
static
abstraction of behavior in, 158
attributes in, 192–193
class diagram and, 155–156
context and, 154
definition of, 154
knowledge vs. behavior in, 156–158
maintainability and, 72–73, 72t
object-oriented analysis and, 154–155
state in, 194
subject matter and, 153
Model-Based Development (MBD)
abstraction and, 35
as object-oriented, 1
road map, 63–75
Moore finite state machine, 382
Moore model, 374–376, 434
Multi-class associations, 236–237
Multidirectional subclassing, 303, 304f,
317–323, 318f, 320f, 323f, 324f
Multiple associations, collaborations and,
242–243, 243f
Multiple inheritance, 317–323, 318f, 320f,
323f, 324f
Multiplicity
in association identification, 271
in association notation, 239, 239f, 241,
241f
participation and, 260–261
problem space and, 256
selection and, 260–261
N
Naming conventions, for states and events,
433–435
Navigation
associations and, 235, 236
to behavior, 244
to knowledge, 244
in relational paradigm, 296
through association paths, 237, 237f
NF. See Normal form (NF)
Normal form (NF), 22–25
Normalization, attribute, knowledge
integrity and, 285–289, 287f, 288f
Notation
associations, 233, 239–242, 239f
attribute, 191–193
Booch, 29
class, 167–168
for customer space, 22
finite state machine, 383–386
responsibility, 199–201
subclass, 300–303
O
Object(s). See also Entity(ies)
abstraction of responsibilities, 399–400
anthropomorphization and, 206–207
blitz, 169–172
characteristics, 49–52
classes and, 51f
definition of, 37, 50
Object(s) (continued)
entities and, 51f
entities vs., 164
identifying candidate, 163
example, 174r–177t
instances and, 51f
instance vs., 50
life cycles, 362–373, 396
logical indivisibility and, 41
methods and, 197–207
operations and, 197–207
in reification, 268, 268f
responsibilities and, 165–166
subsystem implementation with, 110
Object Constraint Language (OCL), 262
Object Design: Roles, Responsibilities, and Collaborations (Wirfs-Brock), 421
Object-oriented analysis (OOA), 32–33, 36f
domain analysis and, 70
finite state machines and, 377
static model and, 154–155
translation and, 46
Object-oriented design (OOD)
definition of, 33
elaboration and, 45
levels of abstraction and, 36f
Object-oriented (OO) paradigm
basic philosophy of, 30–44
breath-first processing in, 44–52
client in, 203
cohesion in, 40
communication models in, 42–44
encapsulation in, 39
hardware computational model vs., 18
levels of abstraction in, 35, 36f
logical indivisibility in, 40–41
maintainability and, 29, 30–31
problem space abstraction and, 31–32, 37–39
relational vs., 295–297
separation of concerns in, 34–35
service in, 203
subject matter in, 33–34
Object-oriented programming (OOP)
definition of, 33
elaboration and, 45
Object responsibility, logical indivisibility and, 41
Obligations, knowledge, attributes and, 192
Obvious associations, 269
OCL. See Object Constraint Language (OCL)
OO. See Object-oriented (OO)
OOA. See Object-oriented analysis (OOA)
OOD. See Object-oriented design (OOD)
Operation(s)
attributes and, 197–207
definition of, 168, 199
implementation of, 200
Overload polymorphism, 59
P
Parametric polymorphism, 59–60, 61, 331–332, 342–343
Partitioning, application, 72
basic concepts of, 107–118
bridge definitions and, 227
client/service relationships and, 108–109, 114–116, 115f
cohesion and, 108
example of, 130–145, 147–150
feedback to, 231
implementation hiding and, 108
importance of, 105–107
interfaces and, 108, 118
invariants and, 108
legacy replacement example of, 147–150
levels of abstract and, 116–117, 117f
Liskov Substitution Principle and, 326
maintainability and, 82, 106
pet care center example of, 130–145
problem space abstraction and, 108
static nature of, 158
subject matter and, 109–113
subsystems and, 106, 127–128, 227
synchronous services and, 369–370
Path, association
definition of, 237
logical connections and, 242
navigation through, 237, 237f
Patterns, design, 78, 231, 312
Peer-to-peer collaboration
in object-oriented paradigm, 44–52
Pellor, Clara, 78
Pet care center
as application partitioning example, 130–145
as association identification example, 277–278
as class identification example, 181–186
as responsibility identification example, 222–231
Petri Nets, 358
Physical connections, associations and, 272
Physical coupling, 15
PLA. See Product Line Architectures (PLA)
Platform, 66
Plug board, 4, 18
Point-of-sale (POS) entry, remote, as example of invariant modeling, 99–103
Polymorphic events, 426
Polymorphism
  abstraction and, 59
  ad-hoc, 59, 60, 80
  behaviors and, 157
  encapsulation and, 59
  genericity, 59–60, 61
  implementation vs., 57
  inclusion, 59, 60, 312–315, 313f, 314f, 323
  inherent, 59
  inheritance vs., 57
  Liskov Substitution Principle and, 326–327
  overload, 59
  overview of, 58
  parametric, 59–60, 61, 331–332, 342–343
  subclasses and, 312–315, 313f, 314f
Postcondition, 387–388
Precondition, 387, 434
Preliminary scrub, for subsystem identification, 146
Problem analysis, 32
Problem context, 448
Problem space
  application partitioning and, 108
  association classes and, 264–265
  associations and, 236
  computing space vs., 63–69
  conditionality and, 252
  constraint, 263–264, 263f
  definition of, 67
  delegation and, 166, 330
  generalization in, 309
  in invariant modeling, 79, 80
  invariants from, extracting, 342–343
  logical connections and, 243
  multiplicity and, 256
  object-oriented analysis and, 32
  reification and, 268
  responsibilities in, vs. computing space, 340–341
  subclasses and, 308, 309
  subclasses in, 311–312
  subject matter and, 33, 111
  subject matter as unique, 156
Problem space entity
  abstraction of, 163
  attributes and, 192
  definition of, 163
Procedural block structuring, 20
Procedural message passing, introduction of concept, 19
Procedures, introduction of concept, 19
Process
  in Action Data Flow Diagram, 477
  logical indivisibility and, 41
Product Line Architectures (PLA), 123
Protocol state machine, 382
Q
Queue, event, 380–381, 431
R
RDM. See Relational Data Model (RDM)
Redundancy, functional decomposition and, 8–10, 9f
Referential attributes
  association discriminator and, 291
  definition of, 290
  identity and, 290–292, 291f, 292f
Referential integrity
  association loops and, 292–295, 293f
  attributes and, 290–292, 291f, 292f
  collaborations and, 279
  identity and, 290–292, 291f, 292f
  participants and, 289
  scope of, 279, 289
  selection and, 289
  transformation engine and, 279–280
Registers, hardware, 18

Reification
  association classes and, 267–268, 268f
  definition of, 267
  problem space and, 268

Relational Data Model (RDM), 22–25

Relational paradigm
  navigation in, 296
  object-oriented vs., 295–297

Relationship(s)
  association identification and, 272–273
  conditionality and, 250–251
  definition of, 234
  descriptions, 129

Reliability, structured development and, 5

Remote POS entry, as example of invariant modeling, 99–103

Representation, abstract, 161–167

Requirements
  allocation, 130, 147
  analysis of, 70–71
  context for, 129
  delegation and, 166–167
  elicitation of, 70–71
  flawed, 131
  invariant modeling and, 80
  knowledge and, 346
  object-oriented analysis and, 32
  problem analysis and, 32
  roles and, 249–250
  specification of, 70–71
  subject matter and, 158
  subject matter example, 173–174
  subsystem descriptions and, 129
  volatility of, 31

Responsibility(ies).  See also Attribute(s)
  abstract data type and, 168
  abstraction of object, 399–400
  anthropomorphization and, 207
  association identification and, 270
  asynchronicity and, 42–43
  ATM example for identification of, 209–222
  attributes and, 50
  behavior, 198
  in class diagram, 155
  of classes, 169–172

  in class notation, 167
  combined in action, 440
  definition of, 50
  delegation and, 166
  fleshing out of, 132
  as intrinsic, 334
  knowledge, 167–168
  level of abstraction and, 350–351
  logical indivisibility and, 41
  modified object blitz for identification of, 208–209
  notation, 199–201
  object life cycle and, 362
  pet care center example for identification of, 222–231
  preconditions and, 388
  in problem space vs. computing space, 340–341
  splitting, 165–166
  state machine design and, 439
  states and, 428–429
  substituting knowledge for behavior, 342–343
  subsystem, 128
  testability of, 202

Reuse
  design, 66
  large-scale, 82, 105, 118, 124

Rock example, of finite state machine, 410–413

Role(s)
  in association notation, 240, 240f
  associations and, 242, 244–246, 245f
  asymmetry in, 245
  as entities, 163
  as explanatory, 246
  information added by, 245
  naming of, 246
  precision vs. brevity in, 246
  requirements and, 249–250
  in state naming, 434
  states and, 421
  subject matter and, 245

Role-based design, 421

S
  Sampling, 282
  Scalar, abstraction to, 193, 228
Scope
   locality of, associations and, 261
   stack-based, 20
Scrub, preliminary, for subsystem identification, 146
SD. See Structured development (SD)
Self-containment
   of actions, 429
   of behaviors, 202, 221
Self-directed events, 381, 400–403, 400/1,
   402f
Semantic analysis, of subsystems, 146–147
Separation of concerns
   associations and, 236
   cohesion and, 34
   decoupling and, 35
   isolation and, 34
   in object-oriented paradigm, 34–35
   subject matter and, 34
Sequence(s)
   behaviors and, 362
   constraints, 397f, 398
   finite state machines and, 357, 361
   level of abstraction and, 396–397, 397f
   managing, 395–407
   rules, 362–363
   in Turing Machine, 17
Sequence diagrams, 186–189, 187f, 189f,
   214f
Serialization, of events, 432–433
Service(s)
   definition of, 119
   event generators, 205–206
   instantiators, 205
   knowledge accessors, 204
   in object-oriented context, 203
   in subsystem descriptions, 128
   synchronous, 199, 204–206, 369–373
   tests, 204
   transforms, 205
Service oriented architecture (SOA), 116
Set theory
   generalization in, 54–56, 55f, 55f
   overview of, 21–28
Shakespeare, William, 53, 105
Shlaer-Mellor methodology, 77, 123, 154
Situation, condition vs., 422–423
Size, code, invariant modeling and, 83–84
Snapshots, knowledge integrity and, 282
SOA. See Service oriented architecture (SOA)
Software design, history of, 4–5
Solution, customer, 70
Space
   computing
      definition of, 68
      as deterministic, 22
      as disciplined, 22
      hardware computational models as
      basis of, 22
      invariants in, 78
      as mathematically based, 68–69
      problem space vs., 63–69
      responsibilities in, vs. problem space,
      340–341
   customer
      abstraction and, 162
      notation for, 22
   problem
      application partitioning and, 108
      association classes and, 264–265
      associations and, 236
      computing vs., 63–69
      conditionality and, 252
      constraint, 263–264, 263f
      definition of, 67
      delegation and, 166, 330
      entity, 163
      generalization in, 309
      in invariant modeling, 79, 80
      invariants from, extracting, 342–343
      logical connections and, 243
      multiplicity and, 256
      object-oriented analysis and, 32
      reification and, 268
      responsibilities in, vs. computing
      space, 340–341
      subclasses and, 308, 309
      subclasses in, 311–312
      subject matter and, 33, 111
      subject matter as unique, 156
Specialization
   definition of, 299
   generalization and, 304–307, 305f, 306f
   generalization vs., 56
   subclasses and, 304–307
Specification
  associations and, 273
  of requirements, 70–71
Stack-based scope, 20
Starr, Leon, 154, 233, 320
State(s)
  adding, 441
  attributes and, 194–195
  behaviors and, 428–429
  of being, 420–421
  client as, 204
  as condition of being, 415
  context and, 416
  definition of, 359
  dynamic view of, 194
  in finite state machines, 378
  mapping of, 417f
  naming conventions for, 433–435
  preconditions and naming of, 434
  responsibilities and, 428–429
  roles and, 421
  roles in naming of, 434
  static view of, 194
  superstates, 384–385
  variables, 194
Statechart, 383
State machines. See also Finite state machines (FSMs)
  abstraction in design of, 438
  adding transitions to, 447–449
  behavioral, 382
  behaviors and design of, 438–439
  collaborations and design of, 439
  designing, 437–450
  examples for design of, 450–473
  handshaking and design of, 441–442
  protocol, 382
  responsibilities and design of, 439
State Transition Diagrams (STD), 358, 383, 383f
State Transition Tables (STT), 358, 386, 386f
Static model
  abstraction of behavior in, 158
  attributes in, 192–193
  class diagram and, 155–156
  context and, 154
  definition of, 154
  knowledge vs. behavior in, 156–158
  maintainability and, 72–73, 72f
  object-oriented analysis and, 154–155
  state in, 194
  subject matter and, 153
STD. See State Transition Diagrams (STD)
Storage, data, attributes vs., 193
Strong typing, attributes and, 197
Structure, 13–14
Structured development (SD)
  analysis in, 6, 32
  application programming interfaces in, 6
  characteristics of, 5–6
  design in, 6
  functional decomposition in, 7–11
  functional isolation in, 6
  graphical representation in, 6
  history of, 5–6
  maintainability and, 6
  problems in, 8
  programming by contract in, 6
  reliability and, 5
  top-down development in, 6
STT. See State Transition Tables (STT)
Subclass(es)
  abstraction and, 309
  behaviors and, 314–315
  categorization and, 307–312, 310f, 311f
  cohesion and, 307, 328–329
  composition with, 317–323, 318f, 320f, 323f, 324f
  definition of, 300
  disadvantages of, 328–329
  discriminator and, 300–301
  disjoint, 315–317, 316f
  generalization and, 306–307
  inclusion polymorphism and, 312–315, 313f, 314f
  instantiation of, 303
  Liskov Substitution Principle and, 323–328, 328f
  member characteristics, 303
  multidirectional, 303, 304f, 317–323, 318f, 320f, 323f, 324f
  notation, 300–303
  in problem space, 311–312
Subclass(es) (continued)
properties and, 300
reasons for, 310–311
rules, 300–303
semantic connection with superclasses, 308–309
simplicity in, importance of, 307
specialization and, 304–307
as superclass, 302
in superclasses, 303

Subject matter(s)
abstraction and, 308, 345–349
application partitioning and, 109–113
association identification and, 269
cohesion and, 121
definition example, 173
definition of, 33
knowledge and, 345–349
level of abstraction and, 350–351
Liskov Substitution Principle and, 326
in object-oriented paradigm, 33–34
problem space and, 33, 111
requirements and, 158
requirements example, 173–174
roles and, 245
separation of concerns and, 34
static model and, 153
subsystems and, 121
trivial, 132
as unique problem space, 156

Substitution principle, Liskov, 323–328, 328/

Subsystem(s). See also Application partitioning
abstraction and, 110
application partitioning and, 106, 127–128, 227
bleeding of cohesion across, 86, 226
blitz technique for identification of, 146
boundaries of, 121
bridge model for, 125–127, 126/
bridges between, 122–127, 227
characteristics of, 109–110
client/service relationships and, 147
cohesion, 136
common, 131
customer input for, 145–146
decoupling of, 124
definition of, 40, 109
describing, 127–130
design of, 110–111
distributed environment for, 121
eliminating duplication of, 146
event queues and, 431
example of, 130–145
identification of, 119–122, 145–147
input interface for, 125–126, 126/
interfaces in, 118
invariants in description of, 128
"is" vs. "how" descriptions of, 128
large-scale reuse and, 124
level of abstraction in description of, 128
logical indivisibility and, 40
message problem with, 123–125
mission in descriptions of, 128
as objects, 123
objects in implementation of, 110
output interface for, 125–126, 126/
pet care center example for, 130–145
preliminary scrub technique for, 146
relationship descriptions, 129
responsibilities, 128
reuse of, 121
semantic analysis of, 146–147
services in description of, 128
as skeleton, 153
subject matter and, 121
text tags for, 127–128

Superclasses
conditionality and, 254
definition of, 300
Harel view of, 385
Liskov Substitution Principle and, 323–328, 328/
polymorphic events addressed to, 426
problem space and, 308
semantic connection with subclasses, 308–309
subclasses as, 302
subclasses in, 303
Superset commonality, associations and, 261, 272
Superstates, 384–385
Symmetric roles, 245, 245/
Synchronous implementation, 364
Synchronous knowledge access, 430
Synchronous services, 199, 204–206, 369–373
Synchronous wormhole, 124–125

**T**
Tags, text, for subsystems, 127–128
Taxonomy, zoological, 8
Technical innovation, in history of object-oriented programming, 17–28
Ternary associations, 236–237
Tester, as synchronous service, 371
Testing, unit, functional decomposition and, 11
Tests, as synchronous service, 204
Text tags, for subsystems, 127–128
Timeliness, knowledge integrity and, 280–281
Time slicing, 368–369
Top-down development, structured development, 6
Transformation
definition of, 69
engine, 45, 66, 238, 279–280
integrity and, 280
reification and, 267–268, 268f
Transformer, as synchronous service, 371
Transforms, 205
Transition(s)
abstraction and, 422
adding, in state machine design, 447–449
association of, with events, 424, 449–450
behavior constraints and, 360–361
collaborations and, 360–361
context, 425
definition of, 422
events and, 423–424
event vs., 381
in finite state automata, 380
in finite state machines, 378
"floating," 384, 384f
as sequencing constraint, 421–422
situations vs. conditions in, 422–423
in state machines, 359
in State Transition Diagram, 383
in state transition table, 386, 386t
Translation
advantages of, 46
in breadth-first processing, 45–47
definition of, 69
disadvantages of, 47
elaboration vs., 45–47
Turing, Alan, 17
Turing Machine, 17–18
Types, introduction of concept, 19–20

**U**
UML. See Universal Modeling Language (UML)
Unit testing, functional decomposition and, 11
Universal Modeling Language (UML), 21
Use cases
association identification and, 269
Model-Based Development and, 71
USER/CRUD processing, 25

**V**
Value(s)
attribute vs., 52
transforms and, 205
Volatility, of requirements, 31

**W**
Wirfs-Brock, Rebecca, 21, 163, 421
Wirth, Nicolas, 25
Wormhole, synchronous, 124–125

**Z**
Zoological taxonomy, 8