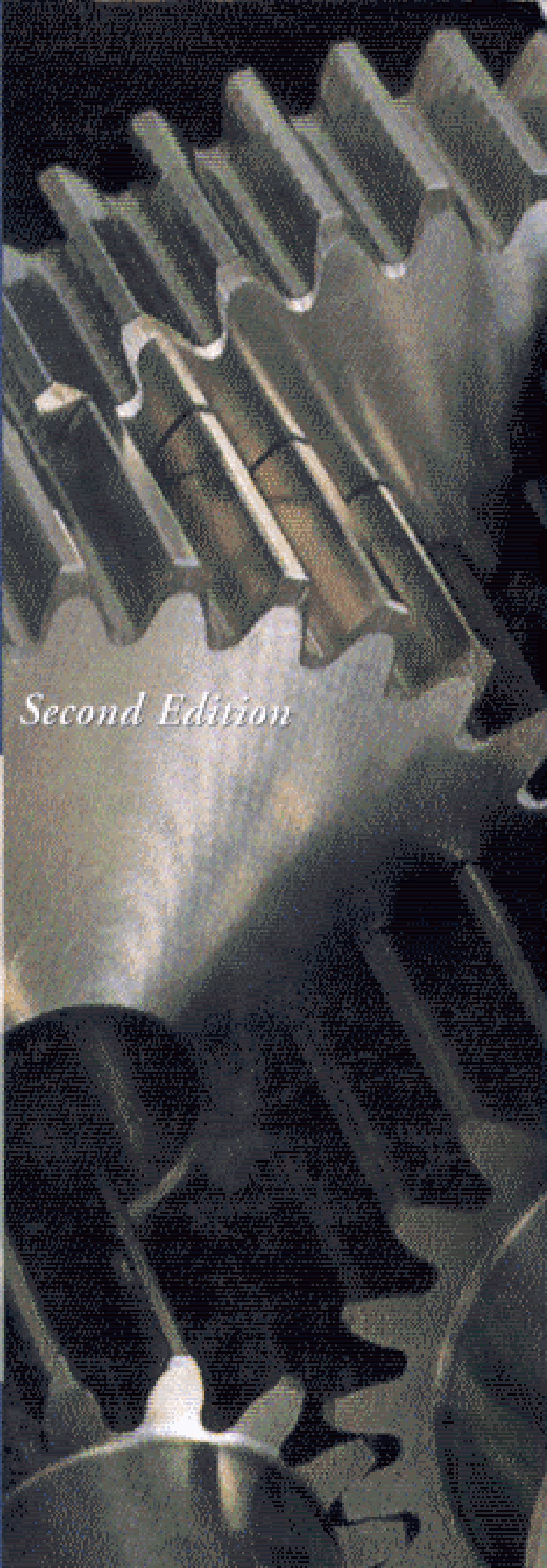
C++

Effective Object-Oriented Software Construction

Concepts, Principles, Industrial Strategies and Practices

Second Edition



- What you should know to succeed in any object-oriented environment
- Structuring your C++ development project for maximum efficiency
- Includes an under-the-hood look at language design
- Class diagrams in UML

KAYSHAV DATTATRI

Foreword by Eric Gamma

Contents

Foreword **xv**

Preface **xvii**

Acknowledgments **xxi**

Part I: Concepts, Practices, and Applications

1 What Is Object-Oriented Programming? 1

Background 1

Procedure-Oriented Programming Example 2

Representation of a Bank Account 3

Security for a Bank Account 4

Solving it with Object-Oriented Programming 5

Understanding the Object Model 7

Terminology 8

Understanding Messages, Methods, and Instance Variables 8

What is Contained in an Object? 9

Instantiating (or Creating) Objects 11

What can be Categorized as a Class? 11

What is NOT a class? 12

Purpose of a Class 13
More about Objects 15
State of an Object 15
Why is the State of an Object Important? 15
Who Controls the State of an Object? 17
Behavior of an Object 18
Phases in an Object-Oriented Software Development 18
Object-Oriented Analysis 18
Object-Oriented Design 20
Object-Oriented Programming 21
Key Elements of the Object Model 21
OOP Paradigms and Languages 24
Requirements of an Object-Oriented Programming Language 24
Benefits of the Object Model 25
Summary 26

2 What Is Data Abstraction? 27
Analysis of the LD Player Design 30
Separation of Interface and Implementation 30
Meaning of Interface 30
Why the Interface to the Object Is So Important 31
What is a Sufficient Interface? 31
Meaning of Implementation 32
Protecting the Implementation 32
What is the Benefit of Data Encapsulation? 34
Relation Between Interface, Implementation, and Data Encapsulation 35
Precautions with Data Encapsulation 36
What (and When) to Encapsulate? 36
Abstract Data Types 37
Implementation of the Abstract Data Type—Stack 38
Data Abstraction with C++ 40
Access Regions in a Class 41
Some Terminology Used with Classes 47
Who is the Implementor of a Class? 48
Implementing the Member Functions 49
Identifying the Target Object in Member Functions 49
A Sample Program 52
The Focus is on the Object 53
A Second Look at Interface 53
What is a Multi-Thread Safe Class? 55
Ensuring Reliability of Abstractions—Class Invariants and Assertions 57
Class Invariants 57
Preconditions and Postconditions 57
Using Assertions for Implementing Invariants and Conditions 59
Using Assertions Effectively 60
Notations for Representing OO Designs 60
The Booch Notation 61

Class Relationships 61
Association 62
Aggregation (has-a) 62
The Using Relationship 65
The Inheritance Relationship (is-a) 66
Class Categories 66
The Unified Modeling Language (UML) 67
Class Relationships 68
Association 69
Association as an Aggregation 71
The OR Association 72
Composition 72
The Generalization Relationship (is-a) 74
The Importance of the has-a Relationship 75
Summary 76

3 Data Abstraction with C++ 77
Fundamentals of the Class Concept 77
Details of Class Elements 78
Access Regions 78
Copy Constructor 81
Accessing Data Members of an Object—the C++ Model 85
The Assignment Operation 89
More on the this Pointer and Name Mangling 95
The Concept of a const Member Function 98
How a Compiler Implements a const Member Function 99
Differences Between Class and Struct in C++ 100
What Can a Class Contain? 100
Focus Is on the Interface of a Class During Design 101
Class Names, Method Function Names, Argument Types, and Documentation 102
Argument Passing Modes—the Client's View 103
The Adopt Semantics 106
Selecting the Correct Mode for Arguments 108
Function Return Values 109
Returning References from Functions 111
Writing Memory Safe Classes 112
Improving Performance 112
Client's Responsibilities with Classes and Functions 113
Summary 114

4 Initialization and Garbage Collection within OOP 115
What is Initialization? 115
Initialization by Constructors 117
Rules To Be Followed with Embedded Objects 124
Garbage Collection Issues 125
Garbage 125

Dangling Reference 125
Remedies for Garbage Collection and Dangling Reference 126
Garbage Collection and Language Design 127
When is Garbage Created in C++? 129
When Does an Object Acquire Resources? 130
Garbage Collection In C++ 130
Identity of Objects 132
Semantics of Object Copying 136
Semantics of a Simple Copy Operation 137
Semantics of Object Assignment 142
Assignment as an l-value Operation 145
Semantics of Object Equality 145
Object Equality and Object Equivalence 147
Why Copy Control Is Needed? 149
A Semaphore Example 150
A License Server Example 152
A String Class Example 154
Analysis 160
The Copy-On-Write Concept 161
When to Use Reference Counting? 167
Summary of Copy-On-Write 168
Classes and Types 169
Summary 170

5 The Inheritance Concept 171
Basics of Inheritance 172
Meaning of the is-a Relationship 186
Effect of Inheritance Relationship 187
Direct and Indirect Base Classes 187
Polymorphic Substitution Principle 187
Initializing the Base Class Object 191
Extending Class Hierarchies with Inheritance 195
Some Basic Advantages of Inheritance 197
Dynamic Binding, Virtual Functions, and Polymorphism 198
Meaning of Dynamic Binding 201
Support for Dynamic Binding—Virtual Functions 202
Effect of Inheritance on Data Encapsulation 204
The Meaning of Polymorphism 206
Using Virtual Functions (Dynamic Binding) Effectively 207
The Concept of Overriding 208
Need for a Virtual Destructor 210
Constructors and Virtual Functions 214
The Generalization-Specialization Concept 215
The Abstract (Deferred) Class Concept 215
Purpose of an Abstract Class 219
A More Complete Example of Abstract Classes—The Game of Chess 224
The Power of Inheritance 232

Effective Code Reuse 233
Clients of an Abstract Base Class 236
Summary of Benefits of Inheritance 237
Perils of Inheritance and Dynamic Binding 238
How Dynamic Binding (Virtual Function) Is Implemented In C++ 240
Cost of Virtual Functions 240
Dynamic Binding and Type Checking 241
Unnecessary Inheritance and Dynamic Binding 242
The Library Checkout System 242
Different Modes for Using Virtual Functions 254
Summary 255

6 The Concept of Multiple Inheritance 257
Simple Definition of Multiple Inheritance 258
The University Example 258
Code Reuse with Refinement 262
The Meaning of Multiple Inheritance Relationships 264
The MI Scenario 265
Resolving Name Conflicts in C++ 266
Problem of Ambiguous Base Classes 270
Preliminary Benefits of Multiple Inheritance 271
Alternatives to Multiple Inheritance 271
First Alternative 272
Second Scenario 274
Repeated Inheritance 276
Solution to Repeated Inheritance 279
Sharing Objects with Virtual Base Classes in C++ 279
Benefits of Virtual Base Classes 282
New Problems Due to Virtual Base Classes 282
Comparing MI in Eiffel and C++ 287
General Problems with Inheritance 290
Using Mixin Classes for Adding Static Capabilities 291
Definition of a Mixin Class 291
When to Use Mixin Classes? 296
Designing for Dynamically Changing Situations 296
Design Flexibility of Role Playing Classes 302
How to Use Role Playing Classes? 302
Another Alternative for Managing Roles 311
Polymorphic Usage of TUniversityMember Objects 312
Changes Required to the Existing Classes 313
Mixin Classes vs Role Objects—Applicability 314
Private Derivation in C++ 316
When to Use Private Derivation 319
Re-exporting Members of the Private Base Class 321
Alternative to Private Derivation—Containment 322
The Need for Private Derivation 324
A Very Useful Example of Mixin Classes and Private Derivation 327

Inheritance vs Containment 333
Summary 334

7 **Selective Exports from Classes (Friend Functions)** **336**
What We Need 337
The C++ Scenario 337
Implications of the Friend Relationship 340
Applications of Friend Functions 343
Case I: Minimizing Heavy Interactions Between Classes 343
Case II: Overcoming Syntax Problems 349
Case III: Functions that Need to Communicate with More than One Class 361
The Advantage of Non-member Functions 362
Choosing Friend Functions vs Member Functions 365
Summary 366

8 **The Concept of Operator Overloading** **367**
Difference Between Language Types and Programmer Defined Types 367
What Is an Overloaded Operator? 370
Why (Not) Operator Overloading—Pros and Cons 371
More Elegant Abstract Data Types 372
Convoluted Operator Overloading 372
Failure to Understand Precedence and Associativity Rules 373
Overloaded Operators in C++ 376
Another Application for the ++ and – – operators 380
The Subscript Operator:operator [ ] 382
A More Sophisticated Operator—The Member Access Operator:Operator -> 387
Operators as Friend Functions 395
Operators that Are Member Functions 396
Operators that Are Implemented as Non-member Functions 398
Why Do We Need Conversions? 402
Conversion Functions 402
Interactions Between Converting Constructors and Conversion Functions 405
Eliminating the Need for Temporary Objects 409
Returning Results from Operator Functions 411
The Assignment Operator 416
Summary 416

9 **Generic Types** **417**
The Repeated Coding Problem 417
The Smart Solution—Generic Programming 424
Fundamentals of a Generic Type (Class) 427
What Happens When a New Template Class is Instantiated in C++? 429
Generic Types and Code Duplication 433
Contract Between a Generic Class Implementor and Clients 434
Can This Be Considered Good Design? 439
Operators vs Member Functions in Generic Class Implementations 441
The Alternative Solution—Specialization of Generic Classes 443

Template Specializations 444
Specialization of a Template Member Function 444
Another Alternative: Separating Comparison of Objects 446
What if One Can't Specialize a Template Member Function? 448
Template Class Specialization 449
The Concept of Generic Functions 451
Instantiation of Template Classes and Member Functions in C++ 455
Generic Types and Type Checking 462
Constrained and Unconstrained Genericity 463
Constraints on Template Arguments in C++ 467
Specific Types as Template Arguments in C++ 468
Default Values for Template Arguments 469
Enforcing Constraints on Template Arguments in C++ 470
Generic Classes and Selective Exports 473
Inheritance and Generic Classes 476
Polymorphism and Generic Classes 481
Useful Applications of Inheritance with Generic Classes 483
The Singleton Approach 484
A General Technique for Controlling Object Creation 485
Implementing Counted Pointers 487
Minimizing Code Duplication with Template Objects 496
Memory Footprint of a Program 498
Schemes for Reducing Template Code Size 498
Template Classes and Source Code Protection 510
Template Classes in Shared (Dynamic) Libraries 510
Template Classes in Shared Libraries—Multiple Instantiations Problems 513
Eliminating Multiple Instantiations in Shared Libraries 515
Linking with Existing Shared Libraries 516
Container Classes 517
Comparing Generic Types and Inheritance 518
Summary 519

10 Expecting the Unexpected 520
Why Should We Handle Error Conditions? 520
What's Wrong With Error Codes? 521
What is the Alternative? 522
The C++ Exception Handling Model 523
How the C++ Exception Mechanism Works? 524
Significance of the Try Block 526
Significance of the Throw Expression 526
Understanding the Dynamic Call Chain 528
Handling Multiple Exceptions 530
Responsibilities of a Catch Block 531
The Exceptions Model in Eiffel 532
Merits and Drawbacks of the Eiffel and C++ Exceptions Models 536
Using C++ Exceptions Effectively 538
Creating Exception Hierarchies 538

Ordering of Catch Handlers 541
Writing Exception Safe Functions 543
Architecting Exception Handling into Projects 545
When to Throw an Exception 545
Strategies for Successful Error Management in Projects 547
A Function Is Not a Firewall 549
Designing Exception Hierarchies 549
Resource Management in an Exceptions Environment 552
Automating Resource Management 553
Generalizing the Resource Management Solution 556
Exceptions and Constructors 558
Returning Safe Resources from Functions 558
A Helper Class for Managing Arrays of Objects 562
Cost of Automatic Garbage Collection 567
Partial Completion of Constructors 568
Creating a Safe Array Using Exceptions 568
Summary 574

Part II: Using Object-Oriented Programming Effectively

11 Mastering Data Abstraction 575
Hiding Implementation Details of an Abstraction 575
Advantages of Using Handles 579
Disadvantages of Using Handles 579
Using Pointers as Data Members (Lazy Evaluation) 584
Controlling Object Creation 586
Allowing Object Creation Using the new () Operator Only 586
Preventing Object Creation Using the new () Operator 589
Using Pointers and References Instead of Embedded Objects 589
Avoid Using Large Arrays as Automatic Variables (or Data Members) 590
Using Arrays of Objects and Arrays of Pointers to Objects 591
Prefer Objects Instead of Primitive Type Pointers for Data Members and Return Values of Member Functions 593
Compatibility with C 596
Pay Attention to Implementation Alternatives: Object Size vs Code Efficiency 598
Avoid Temporary Objects 601
Use the Copy to Initialize Objects 602
Using Proxy (or Surrogate) Objects Effectively 603
Proxy Objects to Facilitate Safe Sharing of Objects 604
Proxy Objects for Ease of Use 605
Proxy Objects that are Stand-ins for Remote Objects 606
Smart Proxy Objects that Provide Additional Functionality 607
Proxies for Solving Syntactic/Semantic Problems 608
A General Subscript Proxy Technique 611
Use Simple Abstractions to Build More Complex Abstractions 613
Abstractions Must Allow Clients to Use a Class in Many Different Ways 614

Summary 616

12 Using Inheritance Effectively 617
Using Inheritance to Implement Elegant Menus and Commands 617
Supporting Different Types of Menus 623
Encapsulating Object Creation Details 624
The Concept of Virtual Constructors 626
Combining Virtual and Non-virtual Functions for Protocol Control 629
The Double Dispatching Concept 638
Designing and Implementing Container Classes 645
Designing to Deal with Different Containers 647
Implementing Homogeneous Container Classes Using Generic Programming 659
Design Goals 660
Advantages of Template-Based Homogeneous Containers 665
Disadvantages of Template-Based Containers 666
Implementing Heterogeneous Containers Using Pointer-Based Homogenous Containers 667
Navigating Containers 669
Passive Iterators 670
Active Iterators 673
Iterators as Objects 676
Managing Collections and Iterators—Client's View 682
Style 1. Create and Return an Iterator from the Container for the User to Use 683
Style 2. Return a Container by Value that the User Can Manipulate by Using Iterators 683
The C++ Standard Template Library (STL) 685
STL Containers 686
Iterators 687
Algorithms in STL 687
Summary 690
Implementation Code for TArray Container 691

13 Understanding the C++ Object Model 701
Efficient Implementation 701
How Objects are Represented in C++ 701
Classes without Any Virtual Functions 702
Member Functions 702
Static Data Members 703
Constructor Functions 704
Classes with Virtual Functions 705
Placement of the vtbl Pointer 706
Sharing Virtual Function Tables Across Shared Libraries 708
Virtual Functions and Multiple Inheritance (No Virtual Base Classes) 709
Virtual Base Classes 715
Member Access with Virtual Base Classes 715
Virtual Base Classes with Virtual Functions 717

Implementation Support for RTTI (Run-Time Type Identification) 719
Object-Based and Object-Oriented Programming 720
References, Pointers, and Values 721
  Assignment of References and Pointers 721
  Copy Constructor 722
  Responsibility of a Constructor 723
Responsibility of a Copy Constructor 726
Optimizations for Pass by Value and Return by Value of Objects 727
  Pass by Value 727
  Return by Value 729
Run-Time Initialization 732
Summary 732

Appendix **733**

Bibliographical References and Recommended Reading **737**

Index **741**