Contents

Preface 2

Preface 2

1 Introduction:
Software Husbandry and Software Development 1
   1.1 The Software Development Cycle  ........................................ 2
   1.2 Maintaining Software Quality ........................................... 5
   1.3 Framework  ........................................................................ 6
   1.4 A Software Development Methodology  .......................... 10
       1.4.1 Software Development as a Group Activity ............. 10
       1.4.2 What Methodology to Use?  ...................................... 12
   1.5 SOS Methodology ............................................................ 13
   1.6 Notation: Source Code Representation ......................... 14
       1.6.1 Advantages of Source Code Representation ............ 14
       1.6.2 The Fallacy of the Specification Language .......... 17
       1.6.3 Socializing The Design ........................................... 19
   1.7 The Process: The Outside-In Development (OID) ........... 21
       1.7.1 Advantages of OID .................................................. 31
   1.8 Timing: Short Cycle Development (SCD) ....................... 32
       1.8.1 Advantages of SCD .................................................. 33
   1.9 The Challenges of SOS .................................................... 35
       1.9.1 The Challenge of Source Code Representation ....... 35
       1.9.2 The Challenge of Outside-In Development ............. 36

1
2 Attribute-based Interface Design

2.1 Interface Design Challenges ................................................................. 52

2.1.1 Completeness and Consistency ......................................................... 53

2.1.2 Semantics and Naming ................................................................. 53

2.1.3 Function Signature: Parameters and Return Values .................... 54

2.2 Attributes .......................................................................................... 54

2.2.1 Attribute Data Types ................................................................. 56

2.2.2 Attribute Naming and Semantics .................................................. 57

2.2.3 Attribute Operations ................................................................. 60

2.3 Attribute Semantics ........................................................................... 63

2.3.1 Read Accessor Semantics .......................................................... 63

2.3.2 Write Accessor Semantics .......................................................... 64

2.3.3 Specific Semantics ...................................................................... 66

2.4 Collection Attributes ........................................................................ 66

2.4.1 Specialized Collections .............................................................. 71

2.4.2 Membership Test ...................................................................... 71

2.4.3 Collection versus Singleton Attribute ...................................... 73

2.5 Constant Attributes ......................................................................... 74

2.6 Composite Attributes ...................................................................... 77

2.7 Compound Attributes ....................................................................... 79

2.8 Benefits of Attribute-based Class Interface .................................... 82

2.8.1 Simple Clear Semantics .............................................................. 82

2.8.2 Behavioral Encapsulation ............................................................ 82

2.8.3 Completeness ............................................................................. 83

2.8.4 Consistency ............................................................................... 83
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8.5</td>
<td>Less Documentation</td>
<td>83</td>
</tr>
<tr>
<td>2.9</td>
<td>The Attribute-Only Discipline</td>
<td>84</td>
</tr>
<tr>
<td>2.9.1</td>
<td>Example: Sorted Directory</td>
<td>84</td>
</tr>
<tr>
<td>2.9.2</td>
<td>Other Examples</td>
<td>86</td>
</tr>
<tr>
<td>2.9.3</td>
<td>ADT Classes</td>
<td>87</td>
</tr>
<tr>
<td>2.9.4</td>
<td>Benefits of the Attribute-Only Approach</td>
<td>87</td>
</tr>
<tr>
<td>2.9.5</td>
<td>Uniform Reference Principle</td>
<td>87</td>
</tr>
<tr>
<td>2.10</td>
<td>Experience with Attribute-only Interface</td>
<td>88</td>
</tr>
<tr>
<td>2.10.1</td>
<td>Retrofitting an Object-oriented Operating System Interface</td>
<td>88</td>
</tr>
<tr>
<td>2.10.2</td>
<td>Converting the Microsoft Foundation Class</td>
<td>89</td>
</tr>
<tr>
<td>2.10.3</td>
<td>Not Hungarian</td>
<td>90</td>
</tr>
<tr>
<td>2.10.4</td>
<td>Adapting to Existing Software</td>
<td>90</td>
</tr>
<tr>
<td>2.10.5</td>
<td>C# Properties</td>
<td>91</td>
</tr>
<tr>
<td>2.10.6</td>
<td>The Difficulty of Attribute-Only Interface Design</td>
<td>92</td>
</tr>
<tr>
<td>2.11</td>
<td>Attribute-structured Object Memory</td>
<td>93</td>
</tr>
<tr>
<td>2.12</td>
<td>Inlining Read Accessors</td>
<td>95</td>
</tr>
<tr>
<td>2.12.1</td>
<td>Protected Cached Attribute Access</td>
<td>97</td>
</tr>
<tr>
<td>2.12.2</td>
<td>Extreme Space/Time Trade-offs</td>
<td>100</td>
</tr>
<tr>
<td>2.13</td>
<td>The Modeling and Simulation Perspective</td>
<td>100</td>
</tr>
<tr>
<td>2.13.1</td>
<td>The Modeling Perspective</td>
<td>101</td>
</tr>
<tr>
<td>2.13.2</td>
<td>The Simulation Perspective</td>
<td>101</td>
</tr>
<tr>
<td>2.13.3</td>
<td>Benefits of Modeling and Simulation</td>
<td>102</td>
</tr>
<tr>
<td>2.13.4</td>
<td>The Challenges with the Modeling and Simulation Perspective</td>
<td>103</td>
</tr>
<tr>
<td>2.14</td>
<td>The Base Framework</td>
<td>105</td>
</tr>
<tr>
<td>2.14.1</td>
<td>Conceptual Framework</td>
<td>108</td>
</tr>
<tr>
<td>2.14.2</td>
<td>Type/Interface Framework</td>
<td>108</td>
</tr>
<tr>
<td>2.14.3</td>
<td>Style Framework</td>
<td>109</td>
</tr>
<tr>
<td>2.14.4</td>
<td>Framework Benefits</td>
<td>109</td>
</tr>
<tr>
<td>2.14.5</td>
<td>The Distributed Systems Imperative</td>
<td>110</td>
</tr>
<tr>
<td>2.14.6</td>
<td>Space/Time Trade-Offs and Costs</td>
<td>110</td>
</tr>
<tr>
<td>2.15</td>
<td>Synopsis</td>
<td>111</td>
</tr>
<tr>
<td>2.16</td>
<td>Concluding Remarks</td>
<td>113</td>
</tr>
</tbody>
</table>
3 Objects: Entities, Values and Descriptions

3.1 What is an Object? ........................................... 117
3.2 What is Object-oriented Programming? .................. 120
   3.2.1 Object Model .................................. 121
   3.2.2 Uniform Interface Reference ..................... 121
3.3 Entities: Real-world Objects ............................. 124
   3.3.1 Entity Creation as A Domain Event ............... 124
   3.3.2 Basic Entity Interface .......................... 126
   3.3.3 Restrictions on Entity Programming .............. 126
   3.3.4 Components and Aggregates ..................... 128
3.4 Values .................................................. 128
   3.4.1 Value Types Define Equality .................... 130
   3.4.2 Value Types as Abstract Collections .......... 131
   3.4.3 Descriptions as Value Types .................... 137
3.5 Named Descriptions ................................. 137
   3.5.1 Versus Common Names ............................ 137
   3.5.2 Versus Entity and Value Types ................ 138
   3.5.3 Versus Types .................................... 139
3.6 Objectifying Abstract Concepts and Actions .......... 140
3.7 Other Programming Paradigms .......................... 141
   3.7.1 Procedure-oriented Programming ............... 141
   3.7.2 Value-Oriented Programming .................... 142
   3.7.3 Abstract Data Type Programming ............... 142
   3.7.4 Functional Programming ....................... 143
   3.7.5 Modular Programming ........................... 143
   3.7.6 Prototype-based Programming ................... 144
3.8 Synopsis ............................................. 144
3.9 Concluding Remarks .................................. 145

4 The Expressability of Object-oriented Programming ... 149

4.1 User-defined Types .................................... 151
   4.1.1 Expressing the Model ............................ 151
4.1.2 Expressing Interface ............................................ 152
4.1.3 Expressing Type Constraints ................................. 155
4.1.4 Optimizations .................................................. 158
4.1.5 Benefits ....................................................... 159
4.2 Type Hierarchy and Inheritance ............................... 160
  4.2.1 Interface - Implementation Relationship ................. 162
  4.2.2 Base Implementation - Specialized Implementation .... 166
  4.2.3 Basic - Extended Interface Relationship ................ 170
  4.2.4 English versus C++ Inheritance ......................... 176
4.3 Genercity ..................................................... 177
  4.3.1 Generic Accessory Types ................................ 180
  4.3.2 Inheritance-based Genercity ............................... 181
  4.3.3 Genercity: Templates versus Inheritance ............... 185
4.4 Implementation ............................................... 187
  4.4.1 Virtual Function Implementation ......................... 187
  4.4.2 Time Efficiency and Virtual Functions ................. 190
  4.4.3 Space Efficiency and Virtual Functions ................ 191
4.5 Synopsis .................................................... 192
4.6 Chapter Summary .............................................. 193

5 Exceptions and Exception Handling .......................... 196
  5.1 Logging for Error Reporting ................................. 198
    5.1.1 Log Interface ........................................... 199
    5.1.2 Log Implementation .................................... 200
    5.1.3 Logging Advantages ..................................... 200
    5.1.4 Auditing .................................................. 201
  5.2 Exceptions in Value-oriented Programming ................ 202
  5.3 The Alternate Return Model ................................ 205
  5.4 Basic Use of Exceptions .................................... 207
    5.4.1 Simple Example Function ............................... 208
    5.4.2 Explicit Error Checks ................................... 209
    5.4.3 Only Catching at the Program Periphery ............... 210
5.4.4 Functions That Should Not Throw Exceptions ............. 211
5.4.5 Read Accessors .................................................. 215
5.4.6 Sufficiency for Transactional Semantics ........................ 218
5.5 Exception Types .......................................................... 219
  5.5.1 Range Exception ................................................. 220
  5.5.2 Resource Exception ............................................. 220
  5.5.3 Resource Exception Types ....................................... 224
  5.5.4 C++ Standard Exceptions ....................................... 227
5.6 Programming With Exceptions ....................................... 228
  5.6.1 Structuring Functions for Transactional Semantics ......... 228
  5.6.2 Try Block Exception Handlers as Optimizations ............. 228
  5.6.3 Top-level Exception Handlers .................................. 230
  5.6.4 Programming Return Exceptions .............................. 231
  5.6.5 Exception Variable Type ...................................... 233
  5.6.6 Exceptions in Exception Handlers ............................. 233
  5.6.7 Call-specific Exception Handling ............................ 234
  5.6.8 Exception Specifications ..................................... 235
5.7 Advantages and Alternatives ...................................... 238
  5.7.1 Language-supported Exceptions ............................. 239
  5.7.2 The Alternate Return Model .................................. 239
  5.7.3 Pass-through Exceptions ..................................... 242
  5.7.4 Framework-wide Exception Types ............................ 242
  5.7.5 Probability-based Exceptions ................................. 243
  5.7.6 Versus Asserts .................................................. 244
  5.7.7 Limits on Error Handling ..................................... 245
5.8 Exceptions and Signals/Interrupts† ............................... 246
5.9 C++ Exception Handling Implementation† ........................ 247
  5.9.1 Performance ..................................................... 249
5.10 Evolving to the Use of Exceptions ................................. 251
5.11 Synopsis ............................................................. 252
5.12 Concluding Remarks ................................................ 253
6 Events, Notifications and Callbacks

6.1 Notification Interface ................................................. 260
  6.1.1 Notification ADT and Attribute Enum ...................... 260
  6.1.2 SingleNotifiee Class Template ............................... 262
  6.1.3 Selective Notification ........................................ 266
  6.1.4 Notifiee Behavior ............................................. 269

6.2 Implementation ...................................................... 270
  6.2.1 Optimizations ................................................ 270
  6.2.2 Distributed Implementation .................................. 273

6.3 Advantages ............................................................ 274

6.4 Activities ............................................................. 277
  6.4.1 Activity Use .................................................. 278
  6.4.2 Scheduling .................................................... 279
  6.4.3 Implementation ................................................. 279
  6.4.4 Advantages .................................................... 280

6.5 Simulation ............................................................. 282

6.6 Designing with Notifications ...................................... 283
  6.6.1 Call-Up/Notify-Down ........................................ 284
  6.6.2 Immediate Mode .............................................. 287
  6.6.3 Separating Configuration and Reporting Interfaces ....... 287

6.7 Functors Considered Harmful ...................................... 293

6.8 Java Event Handling ................................................ 294
  6.8.1 Inner Classes ................................................. 294

6.9 C# Event Handling and Delegates ................................ 296

6.10 Synopsis .............................................................. 297

6.11 Concluding Remarks ................................................ 297

7 Reference and Resource Management with Smart Pointers .... 300

7.1 Reference Management Interface .................................. 302
  7.1.1 PtrInterface: Common Interface ............................. 302
  7.1.2 Programming with the PtrInterface ......................... 306

7.2 The PIn Class Template .............................................. 306
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.1</td>
<td>The <code>Ptr</code> Interface</td>
<td>306</td>
</tr>
<tr>
<td>7.2.2</td>
<td>The <code>Ptr</code> Implementation</td>
<td>309</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Smart Pointers as Accessory Types</td>
<td>310</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Valid Pointer Type</td>
<td>311</td>
</tr>
<tr>
<td>7.2.5</td>
<td>Default Pointer Value</td>
<td>313</td>
</tr>
<tr>
<td>7.2.6</td>
<td>Modeling Support and Pointer Classes</td>
<td>318</td>
</tr>
<tr>
<td>7.3</td>
<td>Automatic Reference Management with <code>Ptr</code></td>
<td>319</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Rules for Using <code>Ptr</code></td>
<td>319</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Avoiding Cyclic References</td>
<td>323</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Using the Default Destructor</td>
<td>324</td>
</tr>
<tr>
<td>7.4</td>
<td>Handling Other Resources</td>
<td>325</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Proxy Objects</td>
<td>326</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Files and Other Resources</td>
<td>327</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Wrappers</td>
<td>329</td>
</tr>
<tr>
<td>7.4.4</td>
<td>Example: Supporting Deletable Objects</td>
<td>329</td>
</tr>
<tr>
<td>7.4.5</td>
<td>Handling Objects with No Reference Counts</td>
<td>331</td>
</tr>
<tr>
<td>7.4.6</td>
<td>Handling Micro Objects</td>
<td>331</td>
</tr>
<tr>
<td>7.5</td>
<td>Iterators as “Interval” Pointers</td>
<td>333</td>
</tr>
<tr>
<td>7.6</td>
<td>The C++ Standard <code>auto_ptr</code></td>
<td>334</td>
</tr>
<tr>
<td>7.6.1</td>
<td><code>auto_ptr</code> Usage</td>
<td>335</td>
</tr>
<tr>
<td>7.7</td>
<td>Benefits of One Smart Pointer Class</td>
<td>337</td>
</tr>
<tr>
<td>7.8</td>
<td>Other Smart Pointer Classes</td>
<td>339</td>
</tr>
<tr>
<td>7.8.1</td>
<td>Non-Invasive Pointers</td>
<td>339</td>
</tr>
<tr>
<td>7.8.2</td>
<td>Ownership versus Anchor Model</td>
<td>341</td>
</tr>
<tr>
<td>7.8.3</td>
<td>Copying Pointers</td>
<td>342</td>
</tr>
<tr>
<td>7.8.4</td>
<td>The Handle Class</td>
<td>343</td>
</tr>
<tr>
<td>7.8.5</td>
<td>Extended Pointers</td>
<td>343</td>
</tr>
<tr>
<td>7.8.6</td>
<td>“Managed/Smart Pointers”</td>
<td>344</td>
</tr>
<tr>
<td>7.8.7</td>
<td>Guarded Pointers</td>
<td>344</td>
</tr>
<tr>
<td>7.9</td>
<td>What about C++ References?</td>
<td>345</td>
</tr>
<tr>
<td>7.9.1</td>
<td>Efficient Pass-by-Value</td>
<td>345</td>
</tr>
<tr>
<td>7.9.2</td>
<td>Efficient Return Values</td>
<td>346</td>
</tr>
</tbody>
</table>
7.9.3 C++ Reference Risks ................................. 346
7.10 Java Reference Management ........................ 348
7.11 Synopsis ............................................. 349
7.12 Concluding Remarks ................................. 350

8 Garbage Collection 

8.1 Limits of Garbage Collectors ....................... 356
8.2 Usage Scenarios .................................... 359
  8.2.1 Rapid Prototyping with Paging Virtual Memory Systems ..... 359
  8.2.2 Memory Management in Paging Virtual Memory Systems ..... 360
  8.2.3 Non-VM Systems .................................. 360
  8.2.4 Memory Management in Servers ..................... 361
  8.2.5 Long-running Memory-intensive Applications ............. 362
8.3 Why is Garbage Collection Popular? ............... 363
8.4 Java .................................................. 365
8.5 The Garbage Collector as an Auditor ............... 365
8.6 Synopsis ............................................. 366
8.7 Concluding Remarks ................................. 366

9 Naming, Managers and Modules 

9.1 Naming ............................................... 369
  9.1.1 The NamedInterface Class ...................... 370
  9.1.2 The Directory Class ............................ 371
  9.1.3 Per-Module Directory ........................... 376
  9.1.4 Manager-based vs. Component-based Naming .......... 376
  9.1.5 Mounting ...................................... 377
9.2 Manager Interface .................................. 378
  9.2.1 Module-specific Manager Interface ............... 380
9.3 Advantages and Limitations ........................ 381
  9.3.1 Inapplicability to ADT Classes ................... 384
  9.3.2 Related Concepts ................................ 385
9.4 Design Patterns .................................... 386
9.5 Module Interface and Structure .................................. 388
9.6 Module Implementation ......................................... 388
   9.6.1 Implementation Namespace ................................. 389
   9.6.2 Adding New Implementations .............................. 389
   9.6.3 Optimizing Manager Classes ............................... 391
   9.6.4 Memory Management ..................................... 393
   9.6.5 Morphing .................................................. 393
   9.6.6 Module Dependencies ..................................... 395
9.7 Synopsis .......................................................... 395
9.8 Concluding Remarks ............................................. 396

10 Inter-Object Relationships: Components to Collisions ........ 399
10.1 Relationships and Categories of Attributes .................. 399
10.2 Components and Component-oriented Design .................. 400
   10.2.1 Component-oriented Design .............................. 400
   10.2.2 Evolving from Simple to Complex ....................... 402
   10.2.3 Optimizing Components ................................ 403
   10.2.4 The Component-Super Interface .......................... 404
   10.2.5 Component Attachment ................................... 405
   10.2.6 Additional Component Attachment Information ........ 408
   10.2.7 Component Visibility and Reference Management .... 411
10.3 The Collision Model of Object Interaction ................... 411
   10.3.1 Collision Service ....................................... 412
   10.3.2 Examples ................................................ 412
   10.3.3 Collision Model Advantages .............................. 415
   10.3.4 Collision Implementation ................................. 416
   10.3.5 Collision Optimizations ................................ 420
   10.3.6 Composition as a Collision Optimization .............. 421
10.4 Transitive Relationships ....................................... 422
10.5 Other Models of Relationships .................................. 423
10.6 Synopsis .......................................................... 423
10.7 Concluding Remarks ............................................. 424
11 Templates and Generic Programming 427

11.1 Example: Automated Dimensional Analysis ..................... 429
11.2 Function Templates ............................................. 432
  11.2.1 Control of Types Instantiating Function Templates ......... 433
  11.2.2 Templates as a Macro Facility ............................... 434
  11.2.3 Exceptions in Templates ..................................... 434
  11.2.4 Iterator-based Generic Programming ......................... 436
  11.2.5 Type Tagging ............................................... 436
  11.2.6 More Dangers with Function Templates ....................... 438
11.3 Member Templates ............................................... 440
11.4 Class Templates ................................................. 441
  11.4.1 Template Type Parameter Binding ............................ 442
  11.4.2 Template Constant Parameters ............................... 442
  11.4.3 Template Template Parameters ............................... 444
  11.4.4 Template Specialization ..................................... 445
  11.4.5 Member Class Templates ..................................... 447
  11.4.6 Preinstantiation ............................................. 447
  11.4.7 Disadvantages of Class Templates ............................ 448
11.5 Concepts and the Use of Class Templates ......................... 450
  11.5.1 Collections ............................................... 451
  11.5.2 Template as a Wrapper ....................................... 454
  11.5.3 Class Template as a Common Interface Specification ...... 454
  11.5.4 Compile Preconditions: Compile-Time Assertions .......... 458
  11.5.5 Traits ..................................................... 459
  11.5.6 Type-based or Template Programming ......................... 461
11.6 Policies ......................................................... 463
11.7 Generic Programming ............................................ 467
11.8 Synopsis ......................................................... 469
11.9 Concluding Remarks ............................................ 470
13.8 Audit Support in Memory Pools .............................................. 515
  13.8.1 Self-Audit .......................................................... 515
  13.8.2 External Audit Support ............................................. 515
13.9 Other Topics ............................................................ 517
13.10 How “Engineer” Memory Management ................................. 518
13.11 Java ................................................................. 519
13.12 Synopsis ............................................................... 520
13.13 Concluding Remarks ................................................... 521

14 Collections ................................................................. 524
  14.1 Invasive Collections of Entities ...................................... 526
    14.1.1 Example: Invasive Queue ..................................... 526
    14.1.2 Implementation ............................................... 528
    14.1.3 Advantages .................................................... 529
    14.1.4 Apparent Limitations ........................................ 532
  14.2 Iteration ............................................................. 535
    14.2.1 Base Iterator Support ......................................... 539
    14.2.2 Additional Collection Operations ............................ 540
    14.2.3 Collection Copy Construction ............................... 541
    14.2.4 Advantages .................................................... 542
    14.2.5 Limitations .................................................... 543
  14.3 Array-based Collections .............................................. 545
  14.4 Comparison to STL .................................................. 547
  14.5 Iterator Alternatives ................................................ 551
    14.5.1 The Separate Iterator Class Template ...................... 551
    14.5.2 The Iterate Function Approach ............................. 555
    14.5.3 The Non-iterator Approach .................................. 555
  14.6 Generic Attribute-based Collection Interface ...................... 556
    14.6.1 Ordering and Iteration ...................................... 559
    14.6.2 Other Query and Update Support ............................. 559
    14.6.3 Conventional Collections .................................... 560
  14.7 Other Collection Issues ............................................. 561
14.7.1 Collection versus Containers .......................... 561
14.7.2 Collections versus Types and Extents .................. 562
14.7.3 Type Relationships between Collections ................. 562
14.7.4 Collections and Relations .............................. 563
14.7.5 Collection versus Group Object ......................... 563
14.8 Synopsis .................................................. 565
14.9 Concluding Remarks ....................................... 566
14.10 Exercises ................................................ 566

15 Auditing .................................................. 568
15.1 Types of Bugs ............................................. 569
15.2 Integrated Software Audit .................................. 570
15.3 Use Of Auditing .......................................... 572
15.4 The Audit Framework ..................................... 574
  15.4.1 The AuditErrors Attribute .............................. 575
  15.4.2 Scope ................................................ 575
15.5 Class Invariants ........................................... 576
  15.5.1 Implementation Class Invariants ......................... 577
  15.5.2 Interface Class Invariant Specification .................. 578
15.6 Implementation ............................................ 579
  15.6.1 Structuring the Audit Routine .......................... 580
  15.6.2 Impact on Software Structure .......................... 581
  15.6.3 Extending to Behavior ................................ 582
  15.6.4 Audit-based Unit Testing .............................. 582
  15.6.5 Audit in the Software Development Process .......... 585
15.7 Advantages of Integrated Audit ............................ 586
  15.7.1 Non-inline Versus Inline .............................. 586
  15.7.2 Same Class versus Different Class ...................... 588
  15.7.3 Manual versus Automatic Generation ................. 588
  15.7.4 No Repair versus Repair ................................ 589
  15.7.5 Snapshot vs. Running Audit ........................... 590
15.8 Experience ................................................ 590
16 Value-Oriented Programming and Abstract Data Types 597

16.1 Application-specific ADTs 598
16.2 Type-Isolation 599
16.3 Operator-Restricted ADTs 603
   16.3.1 Nominal ADTs 604
   16.3.2 Ordinal ADTs 609
   16.3.3 Interval ADTs 610
16.4 Measure Value Types 613
   16.4.1 Range Restrictions 615
   16.4.2 Alternatives 616
16.5 Other ADTs 617
   16.5.1 Specialized Base Types 618
   16.5.2 ADTs and Friend Functions and Operators 619
16.6 ADT Programming Caveats 620
   16.6.1 Caution in Defining ADTs 621
   16.6.2 Completeness and Consistency 622
   16.6.3 Inheritance and ADTs 622
   16.6.4 Avoid Logical Operators 622
   16.6.5 Minimize the Use of Conversion Operators 623
   16.6.6 Other Non-arithmetic Operators 625
   16.6.7 Cute/Unconventional Use of Operators 626
   16.6.8 Binary Operators and Ambiguity 628
   16.6.9 Automatic Generation of Operators 630
16.7 Attributes and Operators 630
   16.7.1 Defining an Attribute ADT Interface 630
   16.7.2 Operator Overloading and Convenient Syntax 632
16.8 Algebraic Semantics of ADTs 633
16.9 Other Useful ADT Classes ........................................ 634
  16.9.1 Extended Number Types ..................................... 634
  16.9.2 Numeric Scientific Computing ............................. 634
16.10 Synopsis .......................................................... 636
16.11 Chapter Summary .................................................. 637

17 Type Structure of Programs ......................................... 641
17.1 Static Type Hierarchy .............................................. 641
  17.1.1 Recursive Structuring ..................................... 643
  17.1.2 Extending with Subtyping ................................. 644
  17.1.3 Other Systems ............................................. 645
17.2 Templates and the Type Structure ................................ 645
17.3 RTTI .............................................................. 646
  17.3.1 Dynamic Cast Form of RTTI .............................. 647
  17.3.2 Use of RTTI ............................................... 648
  17.3.3 RTTI with General Framework Mechanisms ............. 648
  17.3.4 Intra-Module Uses of RTTI ............................... 649
17.4 GUIs, Scripting Languages and RTTI ............................ 651
17.5 Dispatching On Multiple Parameters ........................... 654
17.6 Synopsis .......................................................... 655
17.7 Concluding Remarks .............................................. 656

18 Descriptions .......................................................... 658
18.1 Managing Descriptions ........................................... 660
  18.1.1 Other Examples ........................................... 661
  18.1.2 Subdescriptions ........................................... 662
18.2 Modifying Descriptions .......................................... 662
18.3 Value Types with Hidden Descriptions ......................... 665
18.4 Modular Description Support .................................... 666
18.5 Multi-layer Descriptions ........................................ 667
18.6 Layers ............................................................ 669
  18.6.1 Separate Layer or Component ............................ 672
18.7 Communication between Layers .......................... 672
   18.7.1 Supporting Components ............................ 673
   18.7.2 Consistency ........................................ 673
18.8 Structuring Description Layers .......................... 674
18.9 Component Descriptions ................................ 676
18.10 Synopsis .................................................. 676
18.11 Concluding Remarks .................................... 677

19 Inter-Class Relationships and Dependencies ............... 680
   19.1 Problems with Dependencies .......................... 681
   19.2 Categorizing Inter-class Dependencies ............... 682
      19.2.1 Representation Level of Inter-class Dependencies . 682
      19.2.2 Strengths of Inter-class Dependencies .......... 682
      19.2.3 Scopes of Inter-class Dependencies ............. 683
   19.3 Interface Techniques To Minimize Dependencies ....... 685
   19.4 Interface Issues ....................................... 686
      19.4.1 Use Virtual Functions ............................ 686
      19.4.2 Use Interface Classes ............................ 687
      19.4.3 Use Multiple Interfaces Per Object ............. 687
      19.4.4 Use Notifications ................................. 688
      19.4.5 Avoiding Artificial Dependencies ............... 688
   19.5 Implementation Techniques ............................ 691
      19.5.1 Limiting the Visibility of Classes ............... 691
      19.5.2 Stable Data Members ............................. 692
   19.6 Synopsis ................................................ 693
   19.7 Concluding Remarks .................................... 693
## List of Figures

1.1 The Software Development Cycle: A new requirement extends the previous requirements which expands the design after a design phase, which leads to an extended implementation which, after testing, leads to a new product, which is then further analyzed by customers and the developers to identify new requirements, completing the cycle. .......................... 3

1.2 Software Framework Pyramid .................................................. 7

1.3 The Outside-in Development Process ......................................... 21

1.4 Scenario development and class interface design as vertical and horizontal views of the product ........................................... 24

1.5 Interface and Implementation Type Visibility ............................... 26

1.6 The Hour Glass Structure to Software Development ..................... 28

1.7 Short Cycle Development versus Linear Development .................... 32

2.1 Attributes as the boundary between values and objects ................. 94

2.2 Building a House: A Frame(work) versus Useful Components ........... 106

2.3 Conventional vs. OOP Program Structure ................................ 107

3.1 An object encapsulates state and behavior behind an interface identified by an identifier. In this figure, client0 accesses the object through id0 which designates an interface to the object and client1 accesses the object through a second interface designated by id1. ............................ 118

3.2 Objects (modules) of an object-oriented flight simulation program .... 120

3.3 A Real airplane and an Airplane object. The Airplane may have separate interfaces for a variety of purposes including pilot control, geometric collision detection, graphical display, radar sensing. .................... 124

3.4 Airplane: Aggregation and Components ................................. 129
3.5 Views of Value Types: a value type defines equality, a value type can be viewed as an abstract collection, a value type can be viewed as a description .......................................................... 129
3.6 Abstract Collection Class as ADT ......................................................... 133
3.7 Object Classes as ADT programming plus inheritance .................................. 134
3.8 The orthogonality of object-oriented versus procedure-oriented structuring of a program .......................................................... 141

4.1 Interface Type with Multiple Derived Implementation Types .................. 163
4.2 Virtual Function Implementation .......................................................... 188

5.1 Error Communication: Call and Return vs. Logging .................................. 198
5.2 The Alternate Return Model of Exceptions ............................................. 205
6.1 Call-down, Notify-up Versus Call-up, Notify-down ................................... 284

7.1 Proxy Object as Interface to Object Implementation .................................. 326
7.2 Reference Counting Proxy Object ......................................................... 331
7.3 Using a Separate Reference Count Object .............................................. 340

9.1 Interface and Manager Classes ............................................................. 382

11.1 Collection Class Templates versus Model Types ..................................... 452
11.2 Type Hierarchy using the Instantiable class template .............................. 457

13.1 MemPool as a Common Interface ......................................................... 496
13.2 Memory Regions, Segments and Files ................................................... 502

15.1 Behavior versus Audit Orthogonality ..................................................... 571

18.1 Object Modularity along 3 Dimensions. The X dimension represents inter-object modularity. The Y dimension represents the “partOf” or component decomposition, which is discussed in Chapter 10. The Z dimension is the division of an object into separate representations .................................................. 668

18.2 Physical/Geometric Simulation Layering .............................................. 670
18.3 Layer Structuring with Database of Model ............................................ 675

19.1 Interclass Dependencies ................................................................. 684
19.2 Indirection through an Interface Class ............................ 687
19.3 Two Uniform Print Structures ............................... 689
List of Tables

1.1 Phase to Documentation Correspondence .......................... 22

2.1 Attribute Dictionary .................................................. 59

4.1 Function Call Costs(ussec) - 33MHz MIPS R3000 ............... 191

5.1 Exception Costs(ussec) on RS/6000 xlC Compiler ............... 249
5.2 Exception Costs(ussec) on Various Machines .................... 250
Object-Oriented Programming from a Modeling and Simulation Perspective

David R. Cheriton
Computer Science Department
Stanford University
Stanford, CA 94305-9040

September 27, 2004

Copyright 1992,3,4,6,7,8,9,2001-3 David R. Cheriton, all rights reserved.
Preface

I’ve been programming for more than 25 years, and I love it! I love to confront new challenges of a new application or system, work out a software design that meets the requirements, carefully encode that design for clarity and efficiency, and see a computer or network of computers executing my “grand” design at the impressive speed of the computer, which these days continues to become more and more amazing.

However, throughout these years, there has always been a strong frustration with programming. The beautiful structures I developed in my head became a jumble of weird constructs in whatever programming language I was using. I always felt the language encoded my design, not expressed it.

This frustration was particularly pronounced because of my job. As a researcher in operating systems and distributed systems, my job is to come up better understanding of how to structure systems. I thought about the programming-in-the-large structures for these systems, yet could only represent the designs with encodings “in-the-small”.

One emphasis in my work has been the object-oriented structuring of interfaces and state in operating systems, communication protocols and distributed and parallel applications. Unfortunately, for years, the only languages that supported objects were languages such as Smalltalk and Simula that were just not suitable for systems programming and large-scale performance-critical application development. I was trapped in C with no good way to express the true structure of my designs.

When I started using C++ in 1988, I was delighted to find a language that supported the object-oriented aspects of my design thinking and yet provided the efficiency of C. It also had the added bonus of better compile-time type checking than C or Smalltalk. However, with the help of various C++ textbooks on the market, I managed to write some truly bad code — it didn’t work, it was convoluted, it was laden with C++ features or it was too slow. There had to be a better way, and I set out to find it. I was convinced that C++ was a tool to produce far better programs, but there were some really major pitfalls and feature interactions in the language to avoid — I have the wounds to prove it!

I was also convinced that most C++ books took far too much delight in describ-
ing the language and these features, and put far too little into describing how to express your designs in the language and avoid the pitfalls. These books were written with examples that illustrated how features could be used, but without adequate consideration of whether they ever would be or should be used in this form in real programming. In fact, many of the example programs and structures are just not something I would ever use in my programming, and probably you won’t, and shouldn’t, either.

I was also egged on in my grand crusade by the poor software I had seen over the years, from students and from companies I had worked with in my consulting. (There were lots of exceptions as well of course.) Reading the news and talking to others in the industry, enough software horror stories come up that it is clear there is the need for major improvements in programming quality.

Once empowered with the idea that I really had something to say about programming based on the thousands of lines of code I had written or supervised the writing of, I undertook to develop a discipline of object-oriented programming that exploited the best of C++, avoided the worst, and provided the efficiency and maintainability of real-world programs. This book is the result of these efforts.

As a university professor, I also had the great luxury of being paid to learn things, and the wonderful opportunity to learn by teaching, or trying to teach, the subject. Thus, in the winter of 1993, I launched a new course at Stanford University on object-oriented programming and waltzed into class with a collection of my own notes and various selected research papers. The course, which has now been taught annually since then, was a tremendous vehicle and forcing function for developing this material. The students brought their programming experience, questions and insights to bear, and sent me scrambling to fill the holes in my thinking, even revise my whole approach in some areas. I also had a number of guest lecturers from industry that brought important insights and experience to the class, and contributed to the development of this material.

Another aspect that contributed strongly to the development of this material was my interest in simulation. I became fascinated with the notion that all of computing could be looked at as a simulation of some referent system, real or imagined. In the earlier days of computing, we were too limited by computer resources to structure programs this way, instead forced to focus on minimizing memory usage, I/O and processor time. However, the great advances in hardware price/performance had changed all that. For instance, it is now feasible to program an operating system as a simulation of a computer, one with a variable number of processors (i.e. processes), a variable-sized disks (i.e. files) and so on. As another example, it is also feasible to simulate a network by having (virtual) packets flowing around between (virtual) switches, rather than using some closed form representation of a queuing approximation to the network dynamics. The amount of memory consumed by these packet objects could be significant, but memory is now available. It is also feasible
to structure an animated 3-D graphics program as objects being projected into a 3-D space that is in turn projected onto a display.

In 1989, I bought myself a multiprocessor Silicon Graphics computer and started programming in C++ to develop an object-oriented framework that supported the broad simulation framework I envisioned. The resulting software was a testing ground for my ideas and led to the modeling and simulation perspective that permeates the book (and my thinking). Let me say more about this.

The Modeling and Simulation Perspective

This may sound corny, but I’ll say it because I believe it. I think the modeling and simulation perspective represents a new generation of programming. In this new generation, applications will be structured as simulations of referent systems and programming will be largely concerned with the development of models. The critical frameworks will provide the simulation infrastructure, the simulation operating systems for this new generation. The indicators and enabling technologies are all around.

Previously, we were limited by scarce and expensive hardware resources. I remember paying 9 thousand dollars for a 32 kilobyte memory card, running my whole research group on two 25 megabyte disk drives and a machine with 144 kilobytes of memory in total, and being excited with a 9600 BAUD terminal¹. I recall half the grade in my first programming course (in assembler) being based on minimizing time and space requirements, and pulling some devious tricks to get good grades. Now, however, the computer prices have dropped amazingly and the performance improved similarly, and it seems like the trend will continue for some time. Nevertheless, programming the hardware to be useful is still a major challenge.

Early programs benefited to some degree in a salvation through suffering sense that extraneous features and performance problems were removed to save on hardware resources. Since the improvement in hardware in the 1980’s, programs have gotten bigger and slower in many cases because the discipline imposed by scarce hardware resources has been removed.

Nonetheless, we still need better programming, to allow our bigger applications and systems to evolve and adapt, and even to get them right in the first place. That is, they have to be refined and improved for robustness and extensibility, not so much to fit within hardware limitations. The modeling and simulation perspective is a basis for this new discipline. And besides, it matches what applications want to do!

The simulation perspective also comes up with the user interfaces of applications: The application is the GUI! In the 1970’s and 80’s, the computing community de-

¹Of course, some older-timers can cite far more resource-starved situations!
veloped the desktop metaphor of GUI. However, a new generation of users were brought up on computer games, animated and 2.5 or 3 dimensional. The static 2-D desktop looks pretty dead and boring! The next generation of GUI interfaces will be 3-D visual simulations. We have the hardware resources to do it so let’s get on with it. Of course, the computer game industry is a significant part of the software industry as well, particularly including its fast growing respectable “edutainment” off-shoot.

As a final motivation for this modeling and simulation perspective, programs and systems have suffered from the introverted perspective that the concern with hardware resources produced. The programming too often translated real world behavior into highly restricted computer approximations. A lot was lost in the translation, especially when you tried to extend the original program. We all have our favorite horror stories of running into the limitations of computer systems — “You can’t do that because the computer won’t allow it”. As programmers, we also run into the situation that you cannot extend the program to support some new requirement without rewriting it or dreadfully hacking it. For example, one of the earliest programs I wrote for someone else was supposed to simulate the growth of a population of snails in a swamp. The snail population was represented as a vector, each entry corresponding to the number of snails of a given size. The change in the population was represented by a matrix that gave the probability of a snail growing to the next size category and reproducing in that category to produce more snails in the smallest category. Unfortunately, in running this program, we discovered that the snail population tended to die out rather quickly or explode to takeover the world, not the behavior you see in the real world. The matrix model seemed just too simple to accurately model the real world and it did not admit any significant extensions. I was also severely constrained by the amount of memory available on the computer I was using. Now, however, it is feasible to represent individual snails and let them interact with each other and their environment to provide a much more realistic simulation, SimAnt\(^2\) being a commercial example of this for ants.

In general, matrix-based approximations of systems are computer-oriented models that are driven more by hardware constraints than the reference system, and fail to provide accurate modeling. They are also encouraged by languages like FORTRAN that are optimized around statically and uniformly structured array-like computations. FORTRAN makes more dynamic simulation structures difficult to program in return for optimization focused on the restrictive matrix computations.

The rigid matrix models also fail to recognize that most of the interesting action takes place in a few places. It’s too detailed for much of the referent system and too coarse for the interesting parts. For example, modeling a wing surface, there are vast areas of simple air flow and small areas of high and complex turbulence. A

\(^2\)SimAnt is a trademark of Maxis, Inc.
simple matrix-based model forces a compromise between the accuracy you want in the turbulent areas and the excessive cost it would incur if applied to the areas of simple air flow. (Complex multi-grid techniques do try to deal with these problems to some degree.)

With the modeling and simulation perspective, the program is structured to match the referent system and thus retains a higher fidelity. Dynamic object-oriented models allow you to focus on these small but critical areas without incurring the same computational overhead for the large uninteresting majority. Moreover, using object-oriented techniques such as inheritance, the program can be successively extended in detail to provide more accurate simulations.

These application demands and hardware enablers will make this new generation happen. However, simulation is notoriously difficult programming. It is highly concurrent, highly dynamic and the state is the product of millions of object interactions. A new generation of programming is required, and probably a new generation of programmers. This book is targeted as helping you be part of this new generation, and not being left behind in the old.

**Your Background**

This book should not be your first exposure to computer programming (unless you are a genius). Ideally, you know C++ already, or at least C, and have written some programs longer than 1000 lines. You should also have been exposed to the basic concepts in operating systems, at least at the level of having used operating system services in some non-trivial programs. However, the key background required to understand this book is an appreciation of the problems with writing “real” software, the type of software that people buy or pay to have written. Let me draw some analogy to the way people learn writing of English or any natural language.

In your early education, you were taught how to spell individual words and express simple sentences and their associated grammar. Later in your education, the grammar was extended to more complex constructs and issues of style and organization were introduced for units of writing, such as paragraphs, book reports and on up to essays. It was only having written these smaller unit of composition that you were really ready to write long reports, short stories and even novels. However, if you did proceed to the stage of writing these large units, I expect you found that the problems of large-scale composition reflected back and influenced and refined how you organized individual paragraphs and even sentences. It certainly has for me.

Your programming education was probably similar. You first learned basic syntax and how to write small programs. Subsequent courses lead you into writing larger programs, possibly as programming exercises or as part of studying another topic.
area, such as compiler construction. Along the way you gained experience and a better understanding with the problems of writing large programs.

This book appeals to that programming experience and explores the techniques for dealing with the programming-in-the-large problems. For example, one topic I discuss is naming economy, how to avoid a proliferation of names for operations and objects in a large program. This problem does not exist in small programs and is hard to even appreciate unless you have spent hours puzzling over a huge number of names in a large program that has not dealt with this problem well. None of these issues are really deep or difficult to understand, but you do not appreciate how importance of handling them well unless you have experienced the pain of handling them badly.

The syntax and semantics of C++ are reasonable to learn with the type of background I need from you, and most people that don’t already know C++ at least know C. Therefore, I have included an appendix that is a rapid-fire introduction to C++ that explains the language, roughly presented in the sequence that the language features are used in the text. If you are a C programmer, you should be able to learn C++ from this appendix in conjunction with learning how to do object-oriented programming well using the main chapters of the text.

Why C++?

I choose to base this text on C++ for several reasons. First, this is the language in which I program, it is enjoying wide-spread use, and there is a growing collection of programming environments and aids available for it. Moreover, regarding programming as an engineering activity, I believe that C++ is the best programming language at the present time, and for the foreseeable future, for expressing most programs because it provides not only the important abstraction capabilities for programming-in-the-large, but also and (and critically important), the ability to control the hardware down to the bit, address and cycle levels. Java is an obvious alternative, but it does not provide the control that C++ does, by design. It is sufficiently close to C++ that you should be able to apply the relevant ideas to Java as well. However, I do not believe that it is adequate for a professional programmer to be only competent in Java.

C++ has been used as the language of examples and illustration. As such, this text can be regarded as an advanced C++ programming text. However, it is more generally focused on object-oriented programming in languages that exhibit the combined static and dynamic binding of the Simula class of languages. Recall that my objective is to explore how to express good object-oriented design using C++ syntax and facilities, but not teaching C++ per se.

There are certainly those that have strong opinions against C++ and favor other
object-oriented languages. I admit to a strong preference in favor of C++, particularly the static type checking capabilities, its efficiency, and its simplicity of run-time (such as no garbage collection). I've tried to avoid an excessively strong prejudice for C++, but if you want to learn about object-oriented programming but don't like C++, I suspect you will find this book rather strong on C++.

**Topic Selection**

My approach to topic selection was decidedly non-academic. I simply wrote chapters on what had struck me as the biggest issues in object-oriented programming in the real world. This selection was obviously strongly influenced by my own personal experience and those I talked with, but then the content on each topic is also strongly influenced by the same factors.

As an example of a subject that I did not quite fit in, *reification* and *reflection* are considered very important by some, i.e. the ability to query and modify the language implementation. However, C++ provides only limited support for reification and reflection, and much of the facilities that other languages provide here are present in the program environments for C++, which are consciously separated from the language and run-time as part of the C++ philosophy. I have also not found it to be a key issue in the programming that I have been involved with.

The basic strategy of selecting key topics to include was modified because I simply ran out of space and time to include some of the topics for which I had planned chapters. In particular, I believe that persistence, concurrency and distribution are important aspects of object-oriented programming. These areas are also rapidly evolving and relatively few programmers are directly involved with these issues, although multi-threaded or multi-thread safe code is becoming more and more of an issue. Moreover, the absence of support for these facilities in C++ means that these facilities are provided in vendor- or library-specific forms. The current ANSI and ISO standard for C++ is also not addressing these issues. I have made comments throughout the text on some aspects of persistence, concurrency and distribution as they arise. I hope to provide a follow-on book which includes chapters on these subjects once I recover from this one.

My strong goals-oriented approach to this book also affected the selection of topics to some degree. I only wanted to cover material that a real programmer could and would take and use in active programming practice. In the context of teaching, I've viewed it as important to figure out what you want someone to know as a goal at the end of the course, then structure a series of assignments that exercise skills that lead to that goal, and then provide the material that supports doing these assignments. This focus on goals and objectives also affected the book organization and chapter structure.
Book Organization and Chapter Structure

The key topics were relatively easy to identify as chapters. For instance, I knew there was the importance and the material to have a chapter on inheritance, memory management, exceptions and so on.

As I developed these chapters, I also found myself moving topics forward, from a specific chapter to an earlier chapter, like the introduction of the interface class before the chapter on interface classes. Some of the chapters and treatment of materials also evolved from a broad discussion of the possibilities to a more focused treatment of what I regard as the best approach, and comparison with the alternatives. For instance, my original treatment of inheritance explored all the in’s and out’s of inheritance, whereas the current chapter is more focused on how to avoid multiple inheritance, and why. The how-to focus has reduced the amount of text you have to wade through to figure out the best way to address some issue. This organization may leave you feeling rather directed, compared to being offered a range of alternatives with less winner-picking. However, as a programmer, I was looking for the best approach, not just a compendium of all the possibilities.

There is no perfect order for the chapters. It would be nice, for instance, to cover error handling early because the handling of errors is part of any non-trivial program. Yet, some aspects of error handling are difficult to motivate until other topics have been covered. They also depend on techniques introduced in still other chapters. In some ways, it would be interesting to have a sequence of chapters that tackled progressively more complex and sophisticated programming across all the topics in each subsequent chapter. Like, Programming 1, Programming 2, and so on. However, that would mean scattering the coverage of a given topic across several chapters and make it hard to review a topic comprehensively without significant repetition. I also note that, with the background I assume, you have some idea of how to do memory management, error handling and so on. The focus of the book is to improve how you are tackling these issues, so it is not essential to know everything to program anything.

Recognizing the impossibility of a best ordering, I have fantasized about a hypertext version of this book. However, in the meantime, I have tried to structure the chapters so there is basic material at the beginning and advanced topics that can be omitted or at least skipped on first reading at the end or at least marked. I have also tried to minimize the dependence of the later chapters on each other, making other orderings feasible. This structure and the organization of the chapters is also designed to facilitate teaching from this book, accommodating the distinct possibility that you may not want to, or have to time to, cover all the chapters.
So, you are thinking of teaching a course using this book!

Let me first of all encourage you to teach a course of material on the topics in this book. In my experience at Stanford, this material is really valuable to students. Computer science students are generally provided with a few introductory courses in programming and then expected to program for specific topic courses such as compilers, operating systems, databases and so on. Thus, in these courses, the issues of how to program well are secondary to exercising knowledge about the specific topic of the course. After going through a bunch of these core topic courses, many students know how to program but have also confronted the challenges with programming well, and really welcome a course that helps them polish their programming skills without the burden of learning a new application area of programming.

The book has been used for a course for advanced undergraduates and masters students at Stanford. I might have said that it was designed for that level. However, by selective omission of material, I think it could be taught at a lower division level. I also think we will see this course moving down the curriculum as the sophistication of computing education at the high school level moves forward. After all, drawing the parallel to writing, the first and second year English courses in most colleges teach style, discipline and organization, not grammar, and assume that students have already done a fair amount of writing when they enter college — actually they ensure this in the admission process if the college has any standards. At some point in the future, high school students can be expected to have a similar background in programming.

As another comment on the direction of programming courses, we are seeing beginning programming courses starting to use sophisticated libraries and frameworks so that students can generate interesting applications while still writing only a small amount of code. This contrasts with more conventional early programming in which the students are given just the raw programming language capabilities to build on, and no library support. Building on this trend, I see this book as serving a subsequent programming course on how to program frameworks, a natural follow-on to the “how to use frameworks” introductory courses, especially if the introductory courses use C++ and C++ libraries.

Finally, I view this book and its approach as an interesting alternative to a course focused just on object-oriented design, as one might teach from several books out on the market. This book provides higher-level abstractions and a discussion of analysis and design techniques, but does so in an way that is integrated with C++, helping students understand the overall process. The techniques in this book are more practical and more disciplined than what you would get from a typical object-oriented design text. Moreover, I think students enjoy producing programs that work, not just large, complex object diagrams. (And, I cannot imagine anyone enjoying grading these diagrams — they are very difficult to check for correctness
compared to running a submitted program.)

There are several routes through the material in the book. Chapters 1, 3 and 4 are basic, and should be covered in all courses, although selected sections could be omitted.

The next 3 chapters focus on memory management as the foundation of programming. You can regard programming as creating abstractions on top of raw memory, so having a solid basis of memory management is critical to establish. Chapter 7 describes the use of smart pointers for reference management — this also provides an example of using class templates. Chapter 8 and 13 cover the key issues of reference management, automatic garbage collection and memory management. The memory management techniques of Chapter 13 also serve as examples of interface classes and manager classes, explored further in subsequent chapters. You might view these three chapters as less relevant to Java because Java has automatic garbage collection, but you need to design the memory management in Java too, at least for non-trivial programs. Some few class-specific techniques can be used in Java as well.

These 3 chapters on memory management also serve to explore software design decisions in almost nauseating detail, as an illustration of the careful reasoning required to get software really well designed. Many programmers seem to think there are a variety of “right” approaches in various situations, and the choice is a matter of personal style or preference. I think there is generally a right approach, which only becomes apparent when you look hard enough at the real problem and real trade-offs.

The next three chapters are focused on function and class interface design. Chapter 2 is important because the style it advocates for class interface design does permeate the whole book. Besides promoting a particular approach, the chapter does makes students strongly aware of the design issues that arise in interface design. It’s a “soft” issue, but an important one. Chapter 5 discusses errors and exceptions as part of the class interface to use the C++ exception mechanism. Exception support is relatively new to C++ but will be become increasingly available and increasingly important. I think at least it is good for the students to understand the benefits of exceptions relative to conventional error return approach and the strong impact that exceptions have on design. It also establishes the basics for error handling as an extension of the interface design of the previous chapter. Chapter 11 discusses the use of templates with focus on their use for common interfaces.

Chapters 6, 9 and Chapter 18 develop the notion of events, managers and descriptors as separate interfaces to objects. Chapter 22 describes the general aspects of supporting multiple interfaces to an object. Chapter 10 explores components and object interactions through a general collision model, viewing the component relationship as an optimization.
Chapter 17 describes run-time type identification. Chapter 15 covers auditing interceding with a discussion of an approach that logically follows from the manager structure.

Chapter 19 is important but has more of software engineering favor than much of the rest of the book. It can be skipped when the focus is on programming as opposed to software engineering.

Chapter ?? tries to completely tackle the issues of multiple inheritance. Multiple inheritance is a sufficiently challenging and prevalent complication that it seems unfortunate to omit this chapter altogether but a number of the sections could be skipped.

Chapters 16, ?? and 14 deal with the value-oriented aspects of object-oriented programming that support the modeling and simulation levels. A wide range of this functionality is available in standard class libraries which are increasingly being used instead of “rolling your own”. Thus, you should consider skipping these chapters if time is short and expect the students to use these standard libraries in their programming.

Chapters ?? to ?? discuss the SOS software development methodology introduced in Chapter 1 in some detail. These are the closest parts of the book to what has been taught as object-oriented design and analysis. In my mind, these chapters draw together the issues and techniques explored in the earlier chapters into an integrated analysis and design process for developing object-oriented software. I have found it effective to cover at this stage because, with suitable programming assignments to this point, the students appreciate the discipline that has been covered to this point, but also recognize the difficulty in developing programs according to this discipline. These chapters help to reduce the difficulty and server to map out an approach for even larger projects. I favor integrating analysis and design into a programming course, that is, one in which students write programs and are judged in part on programming style and discipline, because it is part of the process of programming. This part of the text can be viewed as an introduction to analysis and design if there is a separate design course or software engineering course.

The appendices are primarily for students to refer to. The introduction to C++ is obviously just background, but you might check through the key things to remember. Some of the points are not that well known. The style appendix could be skipped, assigned as reading as an example of a style guide, or required to be followed in the programming assignments. I’ve done the latter with quite good results. Even if the students didn’t appreciate every aspect of the style guideline, many did appreciate it overall, and most benefited from the experience of having to follow a specific style guideline. They will probably encounter such a requirement when they enter industry, and also not agree with everything in the style guideline they are expected to adopt then either.
I’ve always liked to include some history in whatever field I teach, and that motivates the last appendix. I think we owe it to the people that have made significant contributions to this technology to recognize where it came from. Moreover, students should appreciate that what we teach is a snapshot of an evolving technology that they will have to deal with evolving further. History provides a sense of this evolution, and helps indicate directions for its future evolution. However, time constraints have made this an optional topic in my teaching of the course at Stanford, and it did not have a natural place among the other chapters, so it ended up as an appendix.

One piece of further guidance: An introductory or short course, particularly one oriented for practitioners, can focus on the “how-to” and techniques aspect of the material, and downplay the “why” and design space exploration aspects of the course. A more advanced version could focus on these more investigative issues. Certainly, I have found that discussions on these topics have been stimulating for my graduate students, and me! Although none of my students have pursued them from a research standpoint, it seems possible.

Finally, a programming course needs to flow based as much on the assignments (if not more so) as on the material. I like the notion of assignment-driven teaching, where assignments are tied to the goals of the course. Students don’t really learn a technique unless they have done it, so why teach it unless it is covered in an assignment?

I have provided an extensive collection of exercises and programming projects that can be used in assignments. You might identify the goals you have for your course, select assignments that exercise students on the techniques that achieve these goals, and then select the material you need to cover to allow them to tackle these assignments. My approach has been to assign the chapters as reading, but only lecture on those aspects that are central to the assignments, again depending on the careful selection of the assignments to structure the course.

So, you are a Professional Programmer Reading this Book

If you are a professional programmer, I’m sure you have encountered lots of challenging situations, and may be inclined to just go directly to your current concern, such as exceptions or memory management. I would hope that you can take the time to first read or skim the first 4 chapters so you at least know where I am “coming from”, as they say. Beyond that, I have tried to make it easy with the table of contents and the index to jump around and leap into specific topics. However, there is a logical development to the text, and it would be even better if you can make a
pass through the book at least once, following the chapter plans and omissions as I have suggested in the previous section for a course.

Acknowledgements

So many people have influenced my thinking, aided in this book development and helped me improve the presentation, it’s hard to know where to start.

The members of my research group have spent countless hours arguing with me about my ideas, trying them out as part of our projects, and bringing back insights, comments and criticism. Ken Duda, Hugh Holbrook and John Chapin have made particularly valuable comments. Let me particularly recognize Matt Zelesko, Sandeep Singhal and Fusun Ertemalp who served as teaching assistants in my course. Thanks also to the several hundred students that have been through the Stanford course that led to this book, and put up with me as I struggled to develop and refine these ideas.

I have had on-going interactions with several key people in the C++ community as I worked through my ideas. Mark Linton taught a course based on an earlier draft of this book at Stanford plus provided many insightful comments as a reviewer and user of the material. John Vlissides and I had many useful conversations. Tom Cargill also reviewed earlier version of the book and brought his uncannily scrutiny to bear on my ideas, and the coding of the examples. Bob Hagmann of SUN and Edward Jung of Microsoft also gave guest lectures in my course.

Finally, I want to thank Bjarne Stroustrup and the C++ community overall for the tremendous effort they have made to produce a great language and a wonderful improvement in programming in general. Programming has got to be one of the most wonderfully demanding and fulfilling activities, and C++ certainly has made it better for me and for thousands of others. Hopefully, it will do the same for you!
Chapter 1

Introduction:
Software Husbandry and Software Development

Software development is more like raising horses than building a bridge. Rather than designing and building a static structure once, a software development effort iterates and improves a collection of software components over many generations, trying to incrementally improve how the product runs. Each generation of the software — a release — builds on the previous generation(s), incorporating new features, improving on old, and hopefully also eliminating some of the previous deficiencies. I use the term *software husbandry* to capture this notion\(^1\).

**Software as an Asset**  For many companies, not just software companies, their software base is a key corporate asset, if not the major asset. Maintaining the quality and currency of this asset must therefore be a priority for the company, just as maintaining the quality of the herd is a priority to a horse breeder. Producing and selling a particular software product is important but analogous to selling individual horses — it should not compromise the longer term health of the herd.

You might quibble that, unlike husbandry, the software product is an exact clone of one core gene combination. However, in actual instantiation, most software runs on a variety of different platforms with different configurations and even different operating systems. Moreover, most software products can be customized by the user through a variety of options or preferences. These products are also exposed to a

\(^1\)I discovered in a recent Google search that Nathan Myrvold used this term in his talk to ACM'97 on the future of software. However, as I gather, his use was for future software that is more cominged with humans, rather than for the software development process itself.
variety of different environments. So, in terms of the product as a running execution on a customer’s machine, the software is a quasi-clone from a common gene pool similar to a horse from a herd. Moreover, some companies produce various products based on different configurations of the same software. For example, a router company may configure a variety of switch and router products from a common software “herd” of components or subsystems, configuring the protocol software, network management and drivers required for a particular configuration.

Following this view, the key long-term measure of software productivity for programmers is how they are improving this software asset, not the number of new lines of code, etc.

Let’s take the analogy to husbandry further. The “gene pool” of the software in terms of its overall structure, style, consistency, robustness, etc. dictates in part what is feasible to evolve the software to. For instance, a person raising draft horses can breed them for increased speed but can hardly hope to compete with someone doing the same, starting with race horses.

The process of evolving software and horses is an iterative process, a cycle of design, production and refinement. It is not a linear terminating sequence. Let’s consider the software development cycle in this context.

### 1.1 The Software Development Cycle

As with real animal husbandry, software is developed in a cycle, as illustrated in Fig. 1.1.

Prior to beginning a cycle, a new requirement arises for a software product (bottom left of the figure) as an extension or modification to the software base. The design phase then refines this new requirement into a (revised) design (in the top box in the figure). Working from this new design, the implementation phase produces an extended implementation. After that, the testing phase in the bottom right of the figure checks that the new software satisfies this requirement, including testing that it actually works as required. Then, in the best case, your happy customers say, “This product is great, but it would be nice if it also ...”. You then feed this input into the next iteration of the software development cycle. (These “requirements” are really a customer “wish list” that should be judiciously prioritized and pruned.) Of course, the other situation is that the software does not satisfy the requirements, and you are back into the software development cycle again to fix the problems (or out of a job!)

A software bug fits into this picture by viewing that the new requirement identified is to fix a discovered bug in the software.
1.1. THE SOFTWARE DEVELOPMENT CYCLE

Figure 1.1: The Software Development Cycle: A new requirement extends the previous requirements which expands the design after a design phase, which leads to an extended implementation which, after testing, leads to a new product, which is then further analyzed by customers and the developers to identify new requirements, completing the cycle.
If there is no iteration ... Conversely, if the development of software does not continue in a cycle, the evolution of the software stops, and the software generally dies because the world keeps changing and so do the requirements on the software. For example, router vendors are continually releasing new versions of router software with new features, fixes and refinements to existing features and performance enhancements. If a vendor stopped evolving its product, it would quickly fall behind and become un-competitive.

Iteration within Releases  The need to support cyclic development may be apparent across releases of the same software. However, cyclic development is generally necessary between releases as well, for several reasons. First, you may not know how well a particular approach works in trying to meet a new requirement until trying it. You try it out by doing a short cycle of development to prototype the feature in a version that is generally far short of a releasable product. It may also be unclear how useful some new feature is or how implementable it is until you have tried it. Second, you may decide to do one iteration to implement the most critical features and then do subsequent iterations to incorporate the less critical features, allowing you to focus on a smaller number of issues at one time. Finally, and related to the above, you should be prepared for sudden release pressure. You may not know how long it will take you or your group to implement all the features and you may not know how long you actually have to implement them. For example, if the marketing department shows up one day panic-stricken insisting that they need to have a new release soon to counter a move by a competitor, you will be under a lot of pressure to complete a release quickly — much sooner than you may have originally planned. Having a short cycle time to each potential release point provides you greater flexibility to respond to these unanticipated market demands. It also allows you to add a new feature quickly that has been identified unexpectedly as a new requirement.

Objectives  In carrying out the software development cycle, you should have three key objectives:

- Identifying and incorporating the next generation of features and functionality.
- Producing these extensions at acceptable cost and time to market.
- Improving the quality of the software as a basis for the current and future generations of the software.

The first two are commonly recognized. The last is critically important yet under-appreciated, under-funded and under-realized. Moreover, it has to fight against the “if it aint broken, don’t fix it” attitude.
1.2 Maintaining Software Quality

In raising horses, maintaining quality is important and it costs. You have to be prepared to give up the immediate profits of breeding or selling every horse and eliminate the weaklings from the herd.

Similarly, in software, you cannot expect that all your efforts will go into new functions and features. As software evolves, new bugs are introduced, original design principles may be compromised and whole new features may be ill-conceived. Without a concerted commitment to maintain quality, the software quality degrades over time, as evidenced by increasing number of crashes or functional failures, longer development and testing costs and decreasing performance.

Also, you cannot hope to achieve quality by throwing out your current software and starting over again\(^2\). First of all, the cost is too high for large software systems, especially to achieve the *maturity* of the existing software. Second, you cannot expect to rewrite a large software system and carry forward all the good ideas of the current system and only add in good new ideas. You only think you can. This is aggravated by the fact that you generally need to write a new system with more features to justify the rewrite, often leading to the *second system syndrome*, where a modest but successful first-generation system breeds excessive confidence, leading to an overly ambitious second system project and consequent project failure.

Cost of Maintaining Quality  One can argue that most of the cost of software development is actually in the cost of maintaining quality because of the cost of testing. Testing is currently a very large part of cost of developing software, both for the development programmers and for the testing organization, if there is one. For example, Bill Gates\(^3\) described Microsoft as primarily a software testing organization because almost 75 percent of the engineering effort goes into testing. In particular, half the engineers are full-time test engineers, plus development engineers spend roughly half their time testing the software they have developed or modified. This investment makes sense when you consider that the primary asset of Microsoft, a $500 billion corporation at the time of writing, is its software. Because software has a low cost of manufacture, especially with the growing distribution over the Internet, the cost is in maintenance, not manufacture, and a key consideration is

\(^2\)Milking our analogy further, a major step in the development of human civilization was the development of the agrarian society, where people invested and maintained an area sufficiently to allow repeated development and farming. This allowed them to improve in one place, rather than always starting from zero after moving to a new area for food. Similarly, hunter-gather software must be replaced by agrarian software that is being refined, developed and extended over time, providing a solid productive software base.

\(^3\)In his talk opening the Gates Computer Science Building at Stanford University, January, 1995.
time-to-market. This accentuates testing or quality maintenance as a key issue in the overall operation of a software-based enterprise.

Furthermore, another key aspect to maintaining software quality is a process of code review. In the code review, a software engineer other than the one that developed the module reads through the source code and convinces him/herself that the code is correct. There is also often a requirement to check that the software adheres to the coding conventions used by the software team. Code reviews are well-known to improve software quality, but they cost valuable engineering time, time that might otherwise be spent developing new software.

Educating new team members on the software team is another significant cost to maintaining software quality, particularly so they understand its structure, conventions, assumptions, etc. well enough to write good new software and not break the existing software. I encountered one organization that determined it was spending 9 months “training” new programmers and yet they typically only stayed with the company for 18 months, so the overhead of bringing someone up to speed was 100 percent (because in part they had their own language and programming environment). This contrasts with the open “C” environment that most programmers already know.

I have never liked the term “training” because it implies something rather menial, more of a motor skill. (We train horses, we train dogs, we train to build up muscles to run a race, etc.) The real issue with getting a programmer up to speed, whether to write new code, review someone else’s code or perform so-called white-box testing in a significant way, is a higher-brain function. A programmer needs to be able to know what are the concepts, assumptions, terminology, structures and dependencies that underly the software. Some authors refer to this as the ontology\textsuperscript{4}. However, I prefer the more concrete term framework.

By the above argument, a good framework is a key part of software quality and a key to reducing the cost of maintaining software quality. So, what constitutes a good framework and how does one design and evolve a good framework?

1.3 Framework

I like to think of a software framework as a pyramid, as illustrated in Fig. 1.2.

At the top of the pyramid are concepts, high-level notions that typically span multiple modules and types. For example, an iterator is a concept, a pointer-like construct

\textsuperscript{4}An ontology in computing is a specialization of a conceptualization, i.e. the concepts and their relationships. We hijacked this term from philosophy where it is defined as the branch of metaphysics that deals with the nature of existence or being, i.e. a misguided attempt to provide a systematic account or explanation of “existence”, yet another reason there is not a big job market for philosophers.
Figure 1.2: Software Framework Pyramid
that maintains a position at a member within a collection of objects, allowing the client/caller to systematically update its position and access the objects in the collection. Each concept has a name, and thus introduces terminology associated with the framework. The concept terminology, their definitions and their inter-relationships are all things that a programmer must comprehend to work within the framework.

At the next level down in this framework pyramid, there are types, defining a specific interface to a concept in terms of other types. For example, a framework may include a Log::Iterator type that corresponds to the concept of iterator but with an interface specialized and specified for accessing a Log, i.e. accessing the collection of log records in the Log. Of course, log is another term/concept to define.

A programmer must comprehend the core types of a framework plus non-core types that correspond to his/her area of focus. For example, every programmer working in the framework may need to know about the Log type and its accessory types, such as Log::Iterator. However, if the framework has a more specialized types such as Quaternion that is only used in specific complex calculations, only programmers working in that specific area of development, review and testing may need to know them. I refer to the portion of the pyramid that a programmer must comprehend as the slice.

Finally, at the bottom-level are implementations (of the types), modules of implementation details. Assuming the types and interfaces are clearly defined, semantically as well as syntactically, only the maintainers, reviewer and tester of a specific module needs to comprehend the internals of the module.

With this framework structure, a programmer needs to comprehend at minimum, a slice through this pyramid that includes much of the concept level, some of the type level (the core types and relevant types to his/her area of effort) and the implementation of the module or modules for which he/she is responsible.

I believe that all software systems have such a framework. It is just that some are far better than others, and some framework are far more clearly identified and documented. Rather than wheel out various platitudes about goodness, let’s try to identify quantifiable aspects of the size of the slice and the cost of understanding it.

First of all, the concept level should be concisely and accurately documented as well as being no bigger and more complicated than necessary. Each new concept is something more for a programmer to comprehend. Moreover, you have to comprehend its relationships with all the other concepts as well, often leading to a non-linear, if not quadratic, growth in cost in terms of the number of concepts. A significant problem is precisely defining each concepts and its relationships. Unfortunately, one often ends up describing a concept in natural language, what I like to call monkey language because it was basically evolved from monkey chatter (if one accepts the Darwin’s story). Monkey language does not have precise semantics, requires manual interpretation and checking and has a very complicated, verbose structure. (I’m
1.3. FRAMEWORK

embarrassed to have to write this text in monkey language! Ideally, concepts are expressed in a *programming language*, a language that has a specific syntax, semantics and can be automatically checked and interpreted/executed by computers. As we will explore later, C++ *templates* provide a means to express some concepts in source code, although they are far from perfect.

Similar requirements arise for the type level of the framework pyramid. Here, the language support is substantially better because one can at least define the type syntax in a programming language. The unfortunate aspect is that these languages do not allow you to define the semantics of the type except by pointing to the implementation. Forcing the programmer to derive the semantics from the implementation dramatically widens the *slice* that you have to comprehend. Ideally, one would like a dictionary that defines the terms used in a framework.

Let’s cut through the fancy talk. What do you think is faster: comprehend 1000 terms or 10,000 terms? What do you think is clearer: pages and pages of English description of interface and procedure or a concise C++ description that the compiler can parse and check, and a computer can execute? Hopefully, the answer is obvious.

With a precise description of a framework pyramid, one should be able quantify the overhead of a programmer coming up to speed on the software, recalling that he/she just needs to learn the relevant *slice* through the framework pyramid, not all the source code. (Thus, measuring in terms of lines of code (LOC) is not completely indicative. Indeed, it may be indicative of the size of the system, but not the difficulty in being able to work on it.)

C++ is, at the same time, not necessarily easy to read even when well-written because it is so dense. However, it expresses the real truth about a software system and it is much shorter in general than a monkey language description of the same thing. Specifications in other engineering disciplines have a similar characteristic: concise yet non-trivial to virtually impossible for the uninitiated to read and understand. Moreover, comprehending an individual module is generally the easiest part of the overall task. Its requirement is simply to implement its interface(s). Most software in my experience uses relatively simple data structures. The real challenge is understanding all the concepts, relationships, types, interfaces, constraints, requirements, interactions that it depends on.

Overall, we are after a *software development methodology* that minimizes the size of the *slice* a programmer has to understand, consistent with the inherent complexity of the overall system and its required efficiency, features, etc. Given the importance of software evolvability across many releases of the product, the methodology needs to be structured to support efficient iteration to better and better software. A discipline or software development methodology is required to structure this process because non-trivial software development requires multiple programmers and large
amounts of code over long periods of time. (In that vein, *husbandry* is defined as scientific control and management as a branch of farming. Software “farming” needs similar scientific control and management.)

1.4 A Software Development Methodology

A methodology provides structure and discipline to software development. The three key aspects of a methodology are:

- **notation** — (re)presentation of the design in a form that is concise yet readable by a programming group, effectively communicating information for agreement and refinement into an implementation and eventually, a released product, including documentation.

- **process** — the steps followed by the group so that the requirements are adequately addressed by each participant according to his or her role, and so the results of each individual’s efforts can be smoothly integrated into the final product.

- **timing** — considerations for scheduling the steps of software development so the process runs efficiently and results of each person’s effort is available and can be used, both by others in a timely fashion.

A clear, standardized methodology also encourages the development of software tools that make the development cycles more efficient. For example, if you can standardize on and popularize a graphic representation for class hierarchies as part of a methodology, then there is a market for programming tools that generate these representations. As in any area of production, *tools* are important in conjunction with methodology.

Let’s consider the need for a methodology in more detail.

1.4.1 Software Development as a Group Activity

A *programming team* (i.e. multiple programmers) is required to maintain a non-trivial software system or application for several potentially overlapping reasons, (the alternative being an individual programmer):

- The sheer number of lines of code may exceed that which one programmer can reasonably maintain in the required time-to-market (and time-to-market can often be more significant than technical quality in determining product success). Development and testing time can be plausibly estimated based on
required lines of code for the application using the general “lines of code” productivity measures for programmers\(^5\).

- A project may entail application-domain and even computational domain expertise that spans a range of areas. Multiple programmers may be required to ensure adequate backgrounds in all the disparate areas. For example, a distributed scientific visualization package may require backgrounds in graphics, networking and the particular scientific discipline. It is unusual to find one person with all the required expertise.

- A system or application riding on one super programmer succeeds or fails based on this one person. A company may want multiple people involved in sustaining the product to ensuring it has the expertise in the presence of personnel turn-over. (Many super-programmers are not interested in continuing work on a project once the product has been released, or even after prototyping it completed.)

Thus, a programming team is necessary for most programming development efforts, and a disciplined and structured approach to software development process is a necessary evil required to make efficient use of these multi-person efforts. The “evil” is the cost of the discipline and structure.

**A Team Needs a Methodology** Programming in a team requires more structure and discipline to coordinate the knowledge, understand the problem domain, and adopt a solution framework with all the participants “buying in”. Similar to that observed with respect to parallel processing with processors, each additional programmer (processor) incurs an additional communication and coordination cost on the rest of the group. Unless this programmer is used efficiently, the overall performance of the group can degrade rather than improve with the addition of new team members (a danger recognized by Brook’s Law — see “The Mythical Man-month [2]”). A good methodology provides a protocol that effectively coordinates and communicates between programmers in the group.

**What about Solo Programming?** Even individual programming efforts can benefit from discipline, just as an artist can benefit from a structured schedule or approach to his or her art, although the discipline may be uncomfortable at first. A single programmer involved in a large effort may find that the time lapse between working on one portion and working on another portion of the software may be

\(^5\)One can use high or low estimates of productivity based on quality of programmers and demands of the applications but estimates can be at least accurate in providing lower bounds on completion times. i.e. the estimate is rarely high.
sufficient that the notational aids, process and tools discussed in this chapter are as useful as if he/she were dealing with the software written by several different people.

**Programming Education** Programmers are generally trained in introductory programming courses to work individually, similar to the training of artists. They learn to value the individual creative process of programming (at the very least, to avoid charges of cheating on assignments). Thus, a software development methodology often appears as an unwelcome intrusion into this personal creative process. With the scale of programming performed by beginning programmers, this intrusion also appears unwarranted. However, programming in-the-large necessarily involves multiple programmers.

You might regard it as unfortunate that we have to accept the restrictions of a methodology with serious programming. However, programming is really an engineering discipline, building software systems, and as an engineering discipline, the design and implementation is really a careful series of engineering decisions making the right trade-offs. The right design seems to follow from this careful decision making, well-founded in a solid knowledge of good programming techniques and appropriate domain knowledge. The evolving nature of software means that there is serious discipline required over time to maintain quality.

### 1.4.2 What Methodology to Use?

There are lots of books on software engineering and programming methodologies. My original plan in writing this book was to focus on programming discipline and style. I thought I would just research this literature on software development methodologies and point you, the reader, at the book(s) and approaches that seemed the best.

Well, I researched to some degree, but every approach seemed to denigrate programming to coding, and elevate some non-programming activity, such as drawing pictures, writing in some specification language, etc. to some higher importance — far higher than I thought warranted. (To be clear, by *programming* I mean writing in a language that can be translated automatically into a form that can be executed by a computer. In contrast, *coding* is just hand-translating from a non-programmed form such as a flow chart into a programming representation with relatively little or no intelligent input.) This direction contradicts the programmer’s precept: *Get the computer to do everything that you can, so you don’t have to do it by hand!* What self-respecting programmer would do some task if he or she can write a simple script to get the computer to do it instead? None, I would hope.

So, the more I read, the more dissatisfied I became with what was out there. Yes, there are some good ideas about developing design, but the programmer’s perspec-
1.5. SOS METHODOLOGY

There are three aspects to SOS, corresponding to its three letters:

The first “S” is for Source code as the primary representation, the notation, of the design. For this book, that means C++ or Java. Therefore, I don’t have to write, and you don’t have to read, a long explanation describing a new notation or graphical representation.

The “O” is for outside-in development, working from the external application environment to interfaces and on to implementation. In the modeling and simulation perspective that I emphasize, this means working from a referent system and its environment, rather than from internal computational structures.

Finally, the last “S” is for short-cycle development for the “timing” in the methodology. Short cycles emphasize iterative development with simple steps, so I don’t have to describe complex analysis and design steps. Moreover, because my methodology is basically programming, the other chapters elaborate on techniques I describe here. In essence, I have developed a PROGRAMMING analysis and design methodology, rather than an analysis and design methodology that eventually leads to programming.

Of course, SOS is also providing salvation from other methodologies, and from ad hoc software development.

The following sections elaborate on these three aspects of SOS. The emphasis is on supporting efficient group programming efforts and on coping with the complexity of large-scale systems and applications. A key focus of the methodology is the work-product at each stage, and of course, the real product. I use the term product to designate the software that you are producing, independent of whether this is a class library, program or application suite.

\[\text{Save Our Software :)\]
1.6 Notation: Source Code Representation

The key idea is simple: Express your design in a programming language: C++ or Java for our purposes, not in pictures and not in a separate specification language. Similarly, analysis can be expressed as use scenarios that the software needs to handle, and those too can be expressed in a programming language, C++, Java or a scripting language.

What does this mean? Well, let’s consider a basic design step. If you determine there is some class of objects in your design of a C++ program called, for example, Cell, then you record that insight as

```cpp
class Cell;
```

You don’t draw a balloon and label it Cell and you don’t write something in some weirdo specification language. If you determine that a Cell object needs to have a mass attribute, you can declare a mass member function\(^7\) in that class. For example, in C++ you would write:

```cpp
class Cell {
public:
    Kilograms mass() const;
};
```

In both cases, you don’t have to fill in all the details at this stage. You can simply add more classes as needed and add member functions to the Cell class as you determine what should be there. Furthermore, you can record implementation steps to the mass function incrementally in a separate implementation file, as those details come to mind. You hide implementation details in a variety of ways. For example, implementation details of Cell can be hidden in an implementation class that is derived from Cell, using the techniques of Chapters 4 and ??.

Similarly, the steps of Cell::mass can be specified as procedure calls, hiding the details in separate procedures, assuming it is a complex calculation. The end result is a set of top-level representations that need contain no more detail than the top-level representations used with picture, flowcharts or specification languages.

1.6.1 Advantages of Source Code Representation

I’d summarize the benefits of source code representation as:

\(^7\)I use C++ terminology throughout this book. The member functions in C++ are called methods in Java, Smalltalk and CLOS.
1.6. NOTATION: SOURCE CODE REPRESENTATION

- Less to learn
- More efficient iterative development
- Less error-prone code

Let's consider each of these benefits further.

**Less to Learn**  Using source code representation of the design minimizes the burden on the programmer. The programmer already has to know the programming language. With any practical language such as C++, it is a significant challenge to know it well. Programmers are also burdened with knowing other numerous notations, such as mathematical, scientific and graphical, as well as numerous processes and tools (compilers, linkers, editors, software management systems, text formatters, profilers, browsers, etc.). Mortal programmers such as you and I struggle to understand the full potential of existing tools as well as learn new tools. There are barely enough hours in the day as it stands to do your job as well as keep up with the programming technology.

Given this existing burden, you should exploit as much as possible the notations, process and tools that are already necessarily part of the programming process, namely source code representation and the associated code development and management tools. Having to write the design in both a design/specification language, even monkey language, and having to learn the associated tools, all in addition to writing the design in the programming language is an unnecessary burden on us software development guys. *Why learn two if one will do?* Especially when monkey language is ambiguous, verbose, not automatically checkable or executable!

**Efficient Iterative Development**  Using source code representation for the analysis and design is more efficient for iterative development because the output of the implementation phase, namely source code, is then the same representation as what you are working on in the next iteration through the software development cycle. In particular, you can work directly with the test scenarios, interfaces and implementation from the previous iteration, extending them to address the new requirements introduced in the new iteration. Because it is source, there is no concern about it being inconsistent with the source, as can arise with pictures and separate specifications.

In contrast, the conventional separation between analysis/design and coding has always been a problem for the iterative development. Once you have coded up the application, how do you reenter the analysis and design process to update the software for a major release? The result of the analysis and design in a previous iteration is in a separate representation than the software and often out-of-date
relative to what the software actually does. It is like trying to maintain two separate versions of the software: twice as much work and limited payoff.

Source code representation also generally means that there is more tool support available, easing the burden on the programmer. As a simple example, being able to compile a design (because it is written in a programming language) provides extensive automatic checking from the compiler, especially if all the warnings are enabled.

Additional non-compiled documentation can also be embedded in the source code, using JavaDoc in the case of Java, and tools like cocoon in the case of C++. I have also just used HTML files in the source tree or embedded in C++ source files so that it can be easily extracted. Then, this documentation is stored and managed the same as the source code, helping to keep it consistent with the software it describes. In particular, one can have a HTML file per directory storing a major module, all linked together into a navigatably tree, of course. Further, one can use a tool such as \texttt{htm2tex} to compile selected HTML into an “off-line” document with all the usual headings, table-of-content, index, etc.

\textbf{Less Error-prone Code} \ Source code representation leads to fewer errors and omissions in the design because you do not have the errors of translation from the specification to the programming language.

Working code allows you to use the computer to get feedback that would otherwise require slow manual inspection or manual simulation, with the attendant inefficiencies. It thus automates a portion of the review of the design. Rather than having to manually review whether a particular design fits together, the compiler and linker do some of this checking for you. Rather than having to review whether your design works, executing the program does some of this checking automatically. Having a tool, the compiler, to check the low-level details, allows you to focus on the high-level aspects of the design, and get that part right. Also, you can and should provide audit code that allows you to automatically check the consistency of your data structures, as described in Chapter 15.

Finally, source code representation puts you closer to working code sooner, even if the software has limited functionality. This gives you an opportunity to try out the basic functionality, to demonstrate it to management, get user feedback, or even to check that the performance is on the right track.

Harkening back to the husbandry angle, this approach says: “If you want to raise horses, raise horses. Don’t draw pictures of them.”

Before going on, let’s consider what I call the fallacy of specification languages.
1.6. NOTATION: SOURCE CODE REPRESENTATION

1.6.2 The Fallacy of the Specification Language

The fallacy of the specification language is its implicit assumption that it is better to specify a design twice, once in the specification language and then once in the programming language, rather than just once in the programming language.

C++ and Java, as object-oriented programming languages, are adequate for specification of the design, if you program well. After all, the “design” of a program is just its interfaces and high-level structure without all the gory implementation details. The class interface/implementation separation in C++ and Java allows you to structure the source code in precisely this form. Conversely, a separate specification language is an indication of an inadequate programming language, or inadequate programmers.

The previous section hopefully made clear the advantages of source code representation over using a specification language. So, why do conventional design approaches emphasize identifying and specifying requirements and the top-level design in another form, either a specification language, monkey language or diagrams, and avoiding the use of “code” until later? Why would you ever do the design twice? It’s just more work! Here are some thoughts on how the specification approach arose.

Specification languages may have made sense in days of yore when you programmed in some low-level language such as assembly or C, where it was difficult express concepts and interfaces clearly and to encapsulate and hide details. Advances in programming language design and compiler technology have rendered obsolete the whole concept of a specification language as distinct from a programming language. Now, a non-compiled language is just a pain. If you cannot parse it, how do you check the syntax? By hand? And, if you can parse it, why not compile it as well? Once you can compile it, it’s basically a programming language so let’s call it that.

Some “specification” languages started out as purely manual languages, ones with no computer parser or translator. Then, some programmer wrote a program to parse the language to provide automatic syntax checking. That step provided the basis for some level of semantic checking as well. Once you can do some level of semantic processing, you are on your way to automatic translation. In the early days of computing, the ALGOL-60 language evolved into a programming language somewhat along these lines. In general, a specification language evolves into a programming language if it can, and if it cannot, it suffers from ambiguities and complexities that make it too defective to warrant using. And, of course, when I think of the number of errors that a compiler catches in my code, a design in any non-compiled (or at least non-parsed) specification language is probably full of errors.

Some groups use specification tools like diagramming tools and flow charts to represent the high-level design before going to source representation. However, programming environment tools can be used to generate graphics and flow charts from
the source code, allowing these “pretty pictures” to be updated as the software is updated. This avoids the out-of-date problem that plagues these other specification approaches.

Some folks seem to feel that forcing programmers to write in a separate non-compilable specification language also prevents them from getting immersed in implementation detail too soon. However, you need to develop the discipline in C++ or Java to refine interfaces before committing to implementation, and that gets you most of the way to this goal. It is far better to develop your own discipline and abilities in one (programming) language than prevent yourself from specifying implementation by using a language that is incapable of such specification. (Moreover, as further heresy, there is merit in recording implementation aspects as you think of them, even as you develop the interfaces, capturing these details in separate implementation classes.)

The source code is the ultimate truth in any case, so that is what needs to be clear for on-going development. You need to learn to structure your software and encapsulate the details so that the overall structure remains clear for the maintenance programmer, and so the poor schmuck who has to revise your software to meet the requirements of the next release can do so with minimal effort and problems. Giving up on clarity at the point of coding is something that should have gone out with assembly language programming and APL. The result is better software, and that is the product you are after in any case. Various tools are available and can be expected in the future that make extracting documentation and diagrams from the source far easier. Pictures can be generated from source code, given some care and restrictions in how the source code is written.

Finally, there is the fallacy that a language that is declarative is better or different from a “procedural” language such as C++ or Java. For example, some specification languages encourage the programmer to make assertions or declarations associated with interfaces rather than specifying the implementation. However, as we know from C, an assert statement is just a front for

    if( <condition> ) { // Take on action on assertion failure
        ...  
    }

So, with a simple macro, one can specify the same assertions in C++ and rely on your discipline to defer the implementation details. Then, the “declarative programming” you do as part of design and analysis can be compiled into working software as you flesh out the implementation in the implementation phase, providing valuable checking in the testing phase with minimal effort.

In summary, I believe that the advances in programming language design and com-
Compiler technology have rendered specification languages obsolete. And, the demand for efficient, cyclic development makes these separate representations too expensive to use. Moreover, a separate representation is no substitute for programming discipline — so let’s develop the discipline!

1.6.3 Socializing The Design

“Socializing the design is more important than documenting it.” Pete McBreen.

The reality is that a design is only implementable and maintainable if the software team responsible for it can understand, communicate in terms of it and really buy-in to it as well as build on it. I refer to this process as socializing the design, in line with the dictionary definition of making something consistent with, or adapted to, the needs of society, the programmer society in this case.

To illustrate the point, consider the names of modules and types. The design will work in theory with a variety of names for these entities. However, people will work much better with the design if these names work well in conversation. Some have referred to this as the telephone test: can you use the term over the phone without confusions? The names and the associated concepts chosen for a design have to enter the vocabulary of the team and be usable naturally for the team to be the most effective.

To some degree, object-oriented design poses an additional challenge for humans because the system is structured in terms of objects that interact and particular functions are accomplished as a result of emergent behavior. However, humans often at the top-level need to understand how the design accomplishes the particular functions it is expected to implement. More broadly, they need to understand the top-level functional behavior that allows a system to meet its requirements.

Similarly, humans are interested in why the design is structured as specified, i.e. the rationale behind the design. Clearly, this aspect is not strictly required for implementation. However, without a clear understanding of the rationale (and associated limitations), the software team is likely to go astray in resolving issues that arise, with great temptation to abandon or compromise the design to solve immediate problems.

The key to socializing is a strategy to selling the team on the design, and in some cases, reselling them again and again. I rather like the term “explain”: You need to explain the system to them. The team needs to know the:

---

8Some specification languages seem to have come into existence because there are people who like to design programming languages without incurring the work of implementing them, or to avoid having to directly compete with existing programming languages. Let’s try to avoid such gratuitous invention and/or being the victims of other people’s inventions.
• what — the top-level vocabulary, concepts and relationships such that they are conversant.

• how — how does the design procedurally meet the design requirements and work overall.

• why — the basic rationale behind the design. Why is it structured the way it is, especially if there are some “obvious” or conventional alternatives whose deficiencies are not obvious.

My best answer here is overview documentation, documentation that describes the design in English as an introduction to the software structure. This should include an overview presentations as well, allowing the overall design to be presented in an interactive setting.

A significant benefit to this documentation and presentation work is that it effectively “debogs” the terminology so that it works in conversation. Even simple things like pronunciation and oral unambiguity relative to other words are going to be important, given that aspects of a successful system will be the topic of a large number of human conversations over the years. As one example, a module might quite reasonably be called “supervisor” when viewed local to the software subsystem design in question. However, if the overall organization into which it is to go also includes other conflicting notions of “supervisor”, conversation with this term is going to be painful. For example, just imagine explaining that: “If the supervisor reports an class 1 error to the operator, the operator should report this error to his or her supervisor.” You may quite happily use this term within the source code but only trip over this problem when you try to explain the system operation to someone else. Moreover, sometimes trying to explain the system points out logical inconsistencies in the naming or relationships, especially relative to the outside world. For example, a “supervisor” logically supervises, not “reports problems”. If the primary visibility of a module to the outside world or the rest of the system is reporting, it would be better named as a monitor or reporter or similar.

A related problem in naming and concepts is the tension between generic names and longer highly specific names. Our supervisor example illustrates some of the problem. Using that name for one module may seem simple and natural until you realize that there are other modules that perform similar functions but in different contexts in the system. One tendency is to come up with additional generic terms to use with these other modules, such as manager, monitor, executive and so on. This is effectively introducing artificial distinctions between English terms within the context of the design. My experience is that these are harder to handle than using longer names that distinguish while using a common generic base name for all modules or concepts that are logically the same. I.e. use FiberChannelSwitchSupervisor, not just Supervisor. For long names like this, acronyms can be used in place of the full
name. Again, in my experience, people can reconstruct the full name and disambiguate working from an acronym better than keeping artificial differences between generic names clear. At least the acronyms provide a hint to work from. Try having someone explain the difference between a process, task, job and context in their system and you will hopefully agree.

In general, the vocabulary and concept level of the system design is debugged by testing it within the context in which it is to function, namely communication between people. Socializing the design serves the purpose of explaining and selling the design to the team as well as debugging it from that standpoint.

Now, let's discuss the process of software development with the SOS methodology.

1.7 The Process: The Outside-In Development (OID)

The outside-in development approach is illustrated in Fig. 1.3. You start with the environment of the application or product and first examine and develop use scenarios. A use scenario is an application, pattern or setting for using the software that you propose to develop. These use scenarios help you to develop the interface

---

9 Some authors use different terms for: i) a very general class of uses, ii) more specific but parameterized types of uses and, iii) fully specified uses, as required to test the software. I use the
design, which then defines the required functionality for the class implementation phase. Finally, conventionally, in the test phase, you test the resulting software, using these coded implementations of the use scenarios to check that the result meets the requirements represented by these use scenarios. However, this testing puts you back into looking over and refining the use scenarios, putting you back into the scenario development phase. This normally leads to identifying the ways in which the software does not meet the requirements specified in the use scenarios, typically leading to further refinement of the class interfaces and their implementation.

Using source code representation with outside-in development means that you develop “code” in each of the phases: test code in the first phase representing the external environment, interface code in the second phase, implementation code in the third phase, and refinement of the test code in the validation phase.

**Documentation Development**  
Outside-in development also integrates the development of user documentation with the development process, as in Table 1.1. Scenario development provides a time to document the typical use scenarios for the end user, such as arises in a tutorial or an introduction to the product. The interface phase is a time to specify the product interface, as required in a reference manual. The implementation phase often introduces refinements and restrictions on the semantics of the product interface. With the documentation source integrated with the program source code, the two can be developed in parallel through these phases, adapting to the necessary changes as they come along. By integrating the documentation into the source tree, the joint evolution becomes even easier to manage.

Now let’s briefly describe each phase of outside-in development.

**Use Scenario Development**

The use scenario phase identifies and develops scenarios in which the product is to be used (and tested!)

---

same term across all three of these, qualifying it as appropriate.
I label these scenarios with use because they describe the key uses of the product. They are also test scenarios for the software but I prefer to subordinate that notion to “use” because use is considered first in the process. The scenarios should first and foremost capture the intended uses of the software. Given that, it is self-evident (I hope!) that the software should work and therefore be tested in these scenarios. For brevity, the term scenario is used unless I need to make special emphasis to specify use versus test or vice versa.

These scenarios are refinements of the mission statement for the product, clarifying what the product needs to do based on specific scenarios of use. Scenarios developed at this stage form the basis for “coded” scenarios that can be used to test implemented classes and the overall system at later stages in its development. The hierarchical organization of the scenarios facilitates incremental testing of these modules, allowing basic behavior to be tested in a basic or partial implementation of a class or set of classes. In this vein, the husbandry of a framework of coded scenarios of use and testing is also important, i.e. it has to be routinely evolved along with product software.

Elaboration of these scenarios into test code means drafting interfaces to the product by which the test codes interacts with the product. There needs to be reasonable agreement across the team on these interfaces. The draft interfaces developed as part of scenario development are a starting point for the class interface design of the next phase, viewing these draft interfaces as code-specified requirements, in effect.

**Interface Design**

The *interface design* phase produces a set of class definitions that specify the interfaces for the product, refined from the interfaces developed in the scenario development phase.

In this phase, you look across the set of interfaces to ensure that they are consistent, complete and encapsulated. You may also need to decompose classes into multiple classes to generalize the functionality or to avoid excessively large class definitions.

The contrast with scenario development is illustrated in Fig. 1.4. In scenario development, you focus on each individual scenario, mapping down to the interface functionality required by this scenario, and ensuring that this functionality is present in the interface. That is, referring to Fig. 1.4, you are mapping “vertically” from the scenario to the class interfaces. In the class interface design phase, you work horizontally across the class interfaces, refining, generalizing and structuring these interfaces.

**Separating Interface from Implementation** This approach results in producing the interface before, and therefore separate from, the implementation. This
Figure 1.4: Scenario development and class interface design as vertical and horizontal views of the product.
1.7. **THE PROCESS: THE OUTSIDE-IN DEVELOPMENT (OID)**

Separation between interface and implementation is a hallmark and critical aspect of good object-oriented design.

Here, *visibility* is a primary consideration in separating interface type from implementation type — who can “see” the type. An interface class is generally visible to all client software. Consequently, client software depends on it being stable. Here, stable means that the file containing the interface class definition does not change. In contrast, an implementation type or class is generally only visible inside the implementation module in which it is contained. Thus, an implementation type can be modified or even deleted without any change to client software, even without it recompiling in most cases.

More generally, the set of interface types define one or more type hierarchies that are visible to clients. The implementation types can extend this hierarchy to a portion that is not visible to clients, as illustrated in Fig. 1.5. In this figure, IA, IB and IC are visible whereas CImp1, CImp11, CImp12 and CImp13 are not visible. Thus, there is a clean separation of interface hierarchy, from implementation hierarchy. Conversely, an implementation type cannot define additional interface that is visible to the client.

**Advantages of Interface/Implementation Separation**

Separating interface type from the implementation type(s) provides several key advantages.

**Client Simplicity and Clarity** The programmer should only have to read and understand interface class definition to be able to use a module that implements this interface. He/she does not have to understand or be dependent on the details of a particular implementation. It further allows the client software to effectively assert that it is only dependent on the exported interface, reducing the amount that a maintenance programmer or code reviewer needs to examine. This separation also avoids revealing implementation details to clients in the client-visible header files. (Keeping the implementation out of the client-visible interface may even be motivated for proprietary reasons.)

**Multiple Implementations** The separation allows there to be multiple implementations of an interface without the client software needing to know. For example, a client application can work with the `Window` type and be insulated from existence of different underlying implementations and specific window systems variants. Conversely, this allows a single implementation of the client software to run on different window systems because the generic interface insulates this application software from the underlying differences. Different implementations also arise to
Client
Visibility
Boundary

Legend:

I* Interface class
CImpl* Implementation class

Figure 1.5: Interface and Implementation Type Visibility
provide different space-time trade-offs for different uses. In a distributed environment, yet another type of implementation arises, namely a proxy implementation, that “implements” the interface by communicating over the network to an actual implementation running on a separate network node.

Minimizing Recompilation  Related to the previous point, client software only depends on the interface class definition, so changes in the implementations do not usually require recompilation of the client software. That is, the implementation hierarchy can be changed without changing or even recompiling the client software, just relinking (possibly at run-time if dynamic linking is supported). Minimizing the need for recompilation benefits productivity in a development setting, but can be essential if the interfaces are provided to a separate group or customer where the need for recompilation can be a major obstacle to acceptance. Note that even “hiding” implementation in private or protected portions of the class definition may not eliminate the need for client recompilation in response to implementation changes. The interface file changes and the size of the object may change, even if the public interface itself is not changed.

In my experience, the interface/implementation separation is worthwhile for almost every client visible class except for abstract data types and a few highly optimized classes.

For major classes, this separation is a version of the powerful divide-and-conquer strategy — using an interface definition divides the problem into that of implementing the interface (on one side of the interface) and implementing the clients of the interface on the other side. These are two smaller more well-defined problems once the interface is well-defined (and hopefully well-designed). Moreover, the programmer of a client of the interface only has to comprehend this interface, not the details of the implementation, assuming the interface is well-specified.

The Hour Glass of Software Development  The interface design phase tends to be the thin part of the hourglass of software development, as illustrated in Fig. 1.6. It requires close coordination and agreement among the programmers involved because unifying the interface may require decisions that have implications or effects across the whole product interface. For example, if there are two attribute names being used for fundamentally the same concept, there should be a top-level decision to use one name or another. In contrast, developing independent scenarios can be done in parallel with less interaction between the programmers. The developers just need to coordinate who does each test scenario as well as the addition of new classes and attributes. Moreover, implementation and testing can also be done in parallel and with less coordination.

A key goal of the interface design phase is to allow different programmers or sub-
Figure 1.6: The Hour Glass Structure to Software Development
groups of programmers to subsequently work independently on implementing assigned sets of interfaces with minimal interaction with other groups. Spending the time to get the class interfaces properly refined pays off in the class implementation phase. The better the interface design, the fewer meetings during the implementation phase, the greater the quality of implementation of the classes and, the fewer problems with integration and validation.

Class Implementation

In the class implementation phase, one or more implementations are provided for each interface, allowing the product to be instantiated and run. If the class interfaces were well designed in the previous phase, each interface should be readily implementable both in terms its complexity and in terms of the functionality provided to it by the other class interfaces. Ideally, the implementation task should also be reduced by the use of standard classes for class templates, smart pointers, collection classes and common abstract data types, such as described in the corresponding chapters (11, 7, 16 and 14). This approach reduces the amount of code that needs to be written and automatically ensures that common classes are used across the different class implementations. For example, if each module uses a standard list class, the list code is shared across numerous class implementations rather than being duplicated or partially duplicated in each. It avoids the proliferation of “quasi-duplicated” or really duplicated software, as arises when every programmer uses his or her own implementations.

The implementation phase should not introduce more new interfaces because the major interfaces should be visible, and thus be generated in the class interface design phase.

Testing

The testing phase checks that the resulting system adequately models the reference system and meets other requirements, such as time and space limitations. It does so by running the product in the test scenarios developed in the scenario development phase.

In practice, it may be necessary to extend, refine and debug these scenarios at this stage as well. Thus, the testing phase is in many respects a return to the scenario development phase. Moreover, this testing typically identifies the (unintended) gap between the requirements identified by the scenarios and that provided by the product, leading to a further iteration through the software development cycle. In this vein, you can regard your initial iteration through the software development cycle, where you have no existing software, as a special case in which none of the requirements are already fulfilled.
I include so-called unit testing as well as full product testing in this phase. Testing by the development group is integral to achieving bug-free software. A separate test organization is not there to cause quality software to be produced. It is simply there to try to ensure that low-quality software is not shipped. It's analogous to meat inspection — a meat inspector does not ensure a company has a good meat product, just that the public are less likely to be exposed to truly poor quality. With meat inspection, normally only a random sample of the shipped product is tested. With software, it is the same. The testing organization can only test a small portion of the possible execution sequences of the code. Thus, the challenge of quality bug-free software really rests with the development programmers. They are the ones that introduce bugs, and they are best equipped to eliminate them early, and use techniques to avoid their introduction in the first place. Finally, let's note that the later a bug is discovered, the more it costs to recover from it. Eliminate bugs early!

Integration Phase

There is no conventional integration phase because, with the careful interface design, integration should not require a separate phase. The modules should just fit together. Integration can also occur incrementally through the implementation phase because each implementation class uses the interfaces of one or more other classes, developed during the interface design phase. Thus, you are using the interfaces of other classes before the testing phase even if you are not using their implementation. Finally, with short-cycle development (described next), there is not a large amount of new software to integrate on each iteration through the software development cycle. In contrast, the traditional waterfall model of software development has to integrate all modules into a working whole at the end of extensive development. This integration was a particular challenge in the old days when there was no language support to ensure consistent interfaces between independently developed modules.

Process Mechanics

In discussing OOD, I presume the standard process mechanics of software development, including:

1. using a source control system such as CVS on the source repository and ...
2. requiring checked-in, committed code to be reviewed, compiled and automated detect/bug tracking, as supported by Bugzilla, for instance.
3. automated nightly (or frequent) build and regression testing

(Note: these are all examples of using computer facilities to ease the job of keeping the software development on track.)
1.7.1 Advantages of OID

OID causes the use/test scenarios to be specified concisely early on, making it more likely the resulting system satisfies the true requirements and that programmers understand them, giving important focus to the subsequent development. The early development of interfaces allows class implementation to largely proceed in parallel across the development group. It also simplifies integration.

Outside-in versus Top-down and Bottom-up Development  Outside-in development differs from both conventional top-down design and bottom-up design by focusing first on the environment of the product, and then on external interfaces, and then on the computational structure. In contrast, top-down design starts with the top-level design of the computational structure and refines from there. For example, top-down design might identify a top-level loop in a flight simulator as

```java
for( ever ) {
    readKeyboard();
    updateFlightState();
    redrawDisplay();
}
```

Bottom-up design starts with the basic computational components and builds up from here. For example, it may cause you to develop a set of primitives for redrawing the display in the above example which are later used to implement `redrawDisplay`. Outside-in development focuses on the objects in the application domain or referent system, without being concerned of how they interface or execute. This allows different configurations of these objects, not just one structure as suggested by the structure for the above flight simulator.

To illustrate, outside-in development of a flight simulation examines what are the objects that are visible external to the system, recognizing objects like airplane, airport, runway, control tower, etc. From here, one identifies the use scenarios, namely a scenario in which an object is found and how it is expected to react in this scenario, leading to suitable tests of this behavior. These tests then indicate the interfaces required. The implementation then follows from the interfaces and their semantics, as required by the use scenarios.

These phases are the major aspects of the process in SOS, which is typically iterated through many times during the lifetime of a software product. This leaves the issue of how much to take on in each iteration through the software development cycle, the timing, as described in the next section.
1.8 Timing: Short Cycle Development (SCD)

The term short cycle development SCD emphasizes using multiple short cycles through the software development process rather than fewer longer cycles. Fig. 1.7 illustrates these short cycles (solid lines) relative to the conventional linear development process (dotted lines).

[Diagram showing the comparison between short cycle development and conventional linear design]

Figure 1.7: Short Cycle Development versus Linear Development

To quantify my notion of “short cycle” somewhat, I see a cycle time of a couple of weeks as achievable within a good programming team. This is long enough to incorporate new features, fix bugs in existing code and yet short enough to keep the pressure on and provide feedback to developers. In fact, the most successful approach I have seen to improving programmer’s estimates of schedule is to break tasks into smaller pieces, and thus operate on a shorter time scale\(^1\).

Drawing analogy to the RISC processor design philosophy, it is better to have a simple short cycle that is repeated rapidly than a longer cycle time that attempts to accomplish more per cycle. For example, if you take a year to develop a new GUI application, you could spend plan 3 months on each of scenario development, interface design and implementation leaving 3 months for validation and redesign. Alternatively, you could generate a simple prototype in 2 months, spending a couple of weeks on each phase, and then iterate based on your experience with this prototype. SCD favors the latter: short cycles that give you several intermediate versions of the product and the associated rapid feedback.

**SCD Versus Rapid Prototyping** Short cycle development can be viewed as similar to so-called rapid prototyping. We use short cycle development because it emphasizes that the cycle repeats many times, rather than viewing there as being a

---

\(^1\) Start on a completely new software system can require longer to get to running code, but such a “new start” does not happen very often.
prototype and then a product. In short-cycle development, the result of each cycle is a prototype in some sense for the next iteration; it is more for the marketing and financial people to decide which “prototype” is a product. To make this feasible, it is critical that the documentation iterate, evolve and improve together with the source code, as should naturally happen with source code representation approach, and that both be of high quality.

1.8.1 Advantages of SCD

There are some clear benefits to short cycles, as described below.

Simplicity of Short Cycles  SCD avoids the ambitious and uncertain steps of a long software development cycle. These long steps often require corrective action when the team encounters unanticipated implications in the implementation phase or when the requirements change. In the short cycle approach, the amount of development taken per cycle is smaller, so there is less likelihood of encountering major surprises. Moreover, it is less expensive to just abort a cycle and restart if such a surprise is encountered, because the cycle time is so short. Many projects end up having to institute “subcycle” structure to handle revised interface design specifications and unanticipated implementation problems that arise in long cycle approaches. Short cycle development avoids the complication of this extra substructure in the “process” to support long cycles, and minimizes the likelihood and cost of unpleasant surprises. The need for subcycles generally indicates that your cycle is too long.

Feedback of Short Cycles  Short cycle development provides quicker feedback to the team on how well the design works. It also provides the reward of seeing working software sooner. A basic prototype gives important feedback to the team about what is important, what is difficult and what is contentious. It also provides management with a better indication of progress and provides greater options in choosing release points. In particular, end user or customer feedback arises again and again as critical to successful development, more important than any amount of traditional upfront requirements analysis.

The short cycles also provide more meaningful milestones in development because they represent real working stages, as opposed to pseudo-milestones like “completion of specifications”, which is impossible to test or demonstrate, except by manual review.

Choice of Product Release Point  With SCD, a product release can be viewed as a point chosen out of many completions of short development cycles, rather than
CHAPTER 1. INTRODUCTION: SOFTWARE HUSBANDRY AND SOFTWARE DEVELOPMENT

just one or a few in a long-cycle development. For example, referring back to Fig. 1.7, there are a large number of cycle completion points at which one could release the product, or at least have a working version for market testing and customer or user feedback. Thus, management has more choices and can more readily respond to a change in market requirements, such as the sudden impending release of a competing product. Of course, this assumes that your software is developed to be bug-free, as advocated by Maguire [8] (and me!)

People Management and SCD Short cycle development allows better people management. The various classes into which a project may decompose at the highest level may take very different levels of effort. A short cycle can allow fairly complete development of the simple classes and the basic, or even just the stub, development of the more complex and demanding classes. On completion of such a short cycle, people can be reassigned to the more demanding portions. The repeat of the development cycle can then serve to further refine and decompose the design of these portions based on experience with the development and implementation from the previous iteration of the cycle. The start of a new cycle forms an excellent time to incorporate new people from other completed portions of the project or even new hires. It also allows current programmers to be reassigned to new tasks to further their own experience and professional growth.

In contrast, with the conventional long-cycle approach, there is a need to coordinate and reassign personnel as one portion completes and others lag behind using some additional microscheduling mechanism. I see this problem as analogous to the complex scheduling of instructions that CISC processors encounter when trying to avoid the inefficiency of instruction issue at the rate of the slowest instruction.

Adaptability to Subsequent Changes Using SCD means that the software process is better designed to respond to new requirements, because you are better positioned to iterate to extend or modify the software to meet the new requirements. For example, adding a new checklist feature shortly before product release can be accomplished by a quick iteration, which might be infeasible otherwise. Conventional approaches take far too long to handle these types of unexpected requirements in a timely fashion.

Iteration to Good Design Cyclic development or iteration represents a process of convergence to a satisfactory solution, rather like buying a pair of shoes. A successful product, like a good pair of shoes, is more characterized by how well it fits, rather than any formal specification of requirements. And a good fit generally arises from trying until it does fit. The iterative development process was succinctly described by Mark Linton as:
do {
    think
design
code and test
} while( !done );

and you are never done if it is a successful product.

The Need for Planned Cycles  At this point, you might be thinking that there is no need for release cycles. One can just have scenario development, interface changes, implementation modifications, etc. take place at random times as convenient for each programmer. After all, you can just walk down the hall, agree with Joe to change an interface, and then go ahead and made the change, right?

Well, yes you can. However, there is a danger that an ad hoc interface change can cause other problems, significantly compromising the design and delay completion of the overall effort. Without coordination, there may be no point at which the whole system can be compiled as a robustly working system. Having a structured cycle means that there is a scheduled point at which you have a running, tested and validated unit of software. Using short cycles means that you have a new version of that software in a relative short period of time, not in a long or indefinite period of time.

1.9 The Challenges of SOS

The SOS Methodology may sounds good, but it has its “challenges”, as the managers say. The challenge of SOS, overall, is to execute it effectively and efficiently to husband high quality software at low cost and time-to-market. Let’s consider the costs and how to address them in more detail.

1.9.1 The Challenge of Source Code Representation

The challenge of source code representation is to make the design as clear in this representation as it would be in some more conventional specification, if not clearer. Conventional methodologies warn against the danger of starting to program too soon for fear of losing the top-level design in implementation details.

Modern object-oriented programming and a carefully disciplined programming style allow a design to be expressed in a far clearer fashion than feasible years ago when many of the software development methodologies were developed. The judicious use of interfaces, encapsulation, inheritance and polymorphism together with conventional procedural decomposition makes this possible. A good part of this book is
concerned with proper structuring of software to hide these details and ensure good modularity. With this structure, a top-level design overview should be sufficient to introduce the software to new programmers.

1.9.2 The Challenge of Outside-In Development

A key challenge of OID is being able to develop coded use/test scenarios before the actual code has been written, and map these scenarios to interfaces without having started on real implementation. Writing test code before you have designed the true interfaces takes considerable discipline plus a notion of where you are headed. Designing interfaces before implementation similarly takes some discipline and anticipation of implementation issues.

Thus, a key aspect is separating interface from implementation to allow this early progress without getting wrapped up in implementation details. The design of interfaces is discussed in Chapters 2, 5 and 7.

1.9.3 The Challenge of Short Cycle Development

The major challenge of short cycle development is the overhead of reanalysis, redesign, reimplemention, redebugging and retesting that are required on each cycle. Drawing one last analogy to RISC processor design, it is important to optimize each stage of the pipeline to make the cycle as efficient as possible if you are depending, as we and RISC processors do, on simple, short cycles to achieve high overall productivity. Object-oriented techniques, used effectively, together with supporting tools make these short cycles efficient and effective.

One issue with SCD is ensuring that version $N - 1$ is a solid base and compatible with version $N$ you are going to produce next. How can you possibly do this without designing it all in advance?

The cost of short-cycle development is reduced by using source code representation. With the appropriate tools, you can automatically (re)generate revised figures and documentation from the source code. Thus, they are easily updated from the updated programming language description of the design as the design evolves through multiple cycles.

Second, outside-in development with its focus on interface design and encapsulated implementation behind interfaces facilitates efficient incremental software development through the multiple cycles. Encapsulation behind interfaces means that the implementation of the interface can change and improve without needing to change, or even recompile, the client code. Moreover, an interface can be incrementally extended using inheritance of the new interface from the old, allowing the software developed with the original interface to continue to work, without change. These
aspects of encapsulation behind interfaces and the evolution of the interfaces and their implementation are discussed further in Chapters ???. A careful discipline in the design of interfaces also facilitates their evolution, as described in Chapters 2, 5, and 6. A key aspect of the techniques in the book is achieving encapsulation without compromising functionality or efficiency.

Automated test cases from the source code representation of the use/test scenarios reduces the cost of rerunning the tests on the software on each iteration. Furthermore, SCD is aided by integrated auditing support, as described in Chapter 15. Finally, the extensive use of standard library abstract data types (ADT) and collection classes together with a careful regime for memory management, as described in Chapters 7, 8, and 13 avoids many of the frustrating time-consuming software errors that cause fragile code and drag out the debugging of new software.

**The Large Project Paradox** There is a paradoxical aspect to large projects — the larger the project, the more time is lost when a cycle really runs into trouble. Moreover, it is hard to cycle a large project quickly. This argues for decomposing a large project into smaller subprojects that are of a manageable size, and ensuring some independence between the subprojects, somewhat like what exists between different companies. For instance, there is a dependence between software products from different vendors that are used together. These interfaces are normally required to be fairly simple, stable and well-documented. The same should be applied in separating a large effort into subprojects.

The rest of the book delves into various techniques for efficient short-cycle development as well as all the techniques applicable to source code representation and outside-in development mentioned earlier, exploiting the facilities of C++.

### 1.10 C++ and Java

C++ is the primary programming language used in this book. C++ descended from C but includes the key language facilities required for object-oriented programming, namely user-defined types (by classes), (user-defined) type hierarchy (by inheritance) and dynamic type dispatch (by virtual functions) in support of polymorphism. Some would say that C++ is just C with the addition of a few object-oriented programming facilities — I’ve heard far more critical descriptions of C++. C++ adopted a similar syntax to C and a similar handling of basic (built-in) types to win over C programmers, but the soul of the language is different. There are enough extensions and modifications, such as exceptions, templates, runtime type identification, name spaces and others to justify calling it a new language. Moreover, the experience of programming properly in C++ using object-oriented techniques is sufficiently different from C that it feels to me like a different language.
Why Use C++  
C++ is the primary language used with this book over other possible object-oriented languages for several reasons. First, C++ is the most popular and widely used object-oriented language, suggesting it is both meeting the current real needs of programming as well as the most likely language for you, the reader, to be using. Part of this popularity is due to the greatest constraint on its design, being as close to C as possible but no closer. Second and perhaps related to the above, C++ provides mechanisms for the highly efficient programming and programmer control that is required to build real systems and applications. For example, C++ includes support for in-lining functions, direct data members\(^\text{11}\) and optimized forms of polymorphism. It also allows direct access to other languages including C, assembly and FORTRAN to use existing code or to take advantage of the particular strong points of these other languages, such as the optimized array support in FORTRAN. C++ also includes critical support for large-scale programming, including static type checking, separate compilation and support for name space management. Last but not least, C++ is the language in which I have written my systems and applications, and in which my colleagues write. I wanted to write this book based on my experiences and those of people I have managed and associated with. Many of the techniques, styles and restrictions discussed in this book are probably equally applicable to other languages, and indeed may have been developed first in other languages for that matter. However, I do not feel qualified to discuss programming in any language other than C++, the ones in which I have actually used the techniques of this book.

Java has also emerged as an important object-oriented language. It is a refinement of the C++ object model, similar in many ways to the restrictions and discipline I advocate here. Therefore, you can program in Java with the same techniques in most part as advocated for C++. Conversely, you might regard some portion of the discussion in this book as how to program in C++ like it was Java! For instance, I advocate a stylized use of smart pointers, interface classes and exceptions in C++, leading to similar programming to Java. However, Java does not provide sufficient control over the hardware resources to be used in many demanding applications.

Learning C++  
This book does not teach C++ in the same sense as the myriad of books on C++ do, Lippman \([7]\) being a popular example. Instead of dwelling on the language in the main text, I have devoted the book to style and technique issues. Complementing this material, Appendix ?? provides a quick introduction and description of the key points of C++ — a condensed version of an instruction book on the language itself. I would expect that any competent C programmer could learn C++ from this book, using this appendix for key points on the language and the main body of the text for examples. You will probably also find it useful

\(^{11}\)Eiffel and CLOS, for example, only allow a record or struct to contain pointers to the objects logically in the record, not the data itself. This is impractical for many programming situations.
to have a copy of the C++ Annotated Reference Manual [15], both to look up the occasional idiosyncrasies of the language as well as to survive nerdfest discussions of the dark corners of C++. I have not bothered to describe all the features and feature interactions in C++. The ones I have omitted should not occur in your C++ programs if you follow the discipline advocated in this book.

**Learning Java** Java is similar to C++ but quite a bit simpler. So, I assume you can learn the language from other texts and understand the examples used in this book by similarity to C++.

**Language Versus Style** Good object-oriented programming style is as important, if not more important, than the choice of programming language when developing large applications and systems. In C++, a disciplined programming style is critical because C++ provides a lot of pitfalls as well as a lot of power. To quote Mark Linton, “C++ provides the power, the programmer supplies the brains”, or not! In this vein, programming in C++ is somewhat of an intelligence test for the programmer, with lots of tempting dark alleys that should be avoided. To quote Bjarne Stroustrup, the language designer: “It matters more how a feature can be used well, than how it can be misused.” This view has significantly influenced the design of C++, and is consistent with the design philosophy of its predecessor language C.12

The C++ intelligence test challenges the programmer with:

- features that should be avoided.
- features that should only be used in a restricted, disciplined way.
- features that need to be used in a consistent, careful fashion with due consideration for hidden costs and anomalous behavior.

A piece of wisdom from Mentor Graphics states, “C++ is a magnifier of programming ability: good programmers get better and bad programmers get worse.” And mediocre C programmers generally become bad C++ programmers. Similarly, a comment from the Ellemtel C++ style guide says: “C++ is a difficult language in which there is a fine line between a feature and a bug”. C++ is a power tool with no clear guards or marking (or agreement) on the cutting edges. This book attempts

---

12Many proponents of other languages object to C++ on the grounds that it is too unsafe, too complicated, too compromised to C and so on. You should read Bjarne Stroustrup's excellent book [7] on the design and evolution of C++ to appreciate the intense effort that went into making trade-offs in C++ design that lead to a broadly usable and widely used language. There were probably mistakes made, but it is an outstanding result compared to the many other language design efforts, and many other ISO/ANSI standards efforts.
to be a good user manual. Java is much more restrictive, simpler and safer but it does restrict some important aspects of programming as well as hides sloppiness of programmers in some ways.

**Simplicity versus Consistency**  A major problem with many C++ programs is over-use of C++ features. With C++, you should use simple constructs whenever feasible, and save the power for when it is needed. There is high payoff (for debugging, porting and maintenance) in keeping it as simple as possible (but of course, no simpler). This view is consistent with the C++ language design philosophy: simplicity is the default behavior; sophisticated language features must be specified explicitly. Using more object-oriented features does not make a program better or more object-oriented.

However, balancing this need for simplicity, “good” C++ programs need to have a highly consistent style, which requires consistent adherence to good programming conventions. There are cases in which consistency does complicate, at least in the sense of causing you to use more mechanism than otherwise. For instance, you might make something a class in a particular situation for consistency, and not because the class mechanism was providing a compelling facility in that particular case. For example, you might define a class for network **Packets** even though a simple structure or even **typedef** might be adequate initially. However, the trivial class that you write today often evolves into a more sophisticated class in the next rev. It’s then you are thankful you didn’t just make it a **typedef**.

Because simplicity “in-the-large” comes from consistency of style and structure, and because we are focused on programming in-the-large, the style and discipline advocated in this book favors consistency over using the simplest construct to meet immediate needs.

Java improves the language support somewhat by simplifying the language compared to C++ and providing additional structuring support, such as explicit **interfaces** in the language. However, most of the same issues remain, relying on programming discipline and style.

### 1.11 The Programming Style to Use

With large-scale software efforts, there is tremendous benefit in adhering to uniform conventions and style. Most significant programming today is done in programming teams. Moreover, the effort spans even more people over time as the set of developers and maintainers change, making a consistent style very important. This book could easily follow the lead of other books and provide lots of alternative styles and conventions from which to choose. Instead, it presents a specific discipline and style because:
• I am interested in identifying the “best” style and conventions. Presenting a lot of alternatives is really just a “cop-out” from making real value judgments. As I argue in various sections of this book, there are strong reasons for one approach over another, and strong reasons to avoid certain combinations of features.

• The style in this book restricts the use of C++ features (and even Java), significantly simplifying, if not eliminating, interactions between certain features. For example, the interaction between defining assignment operators and virtual base classes is dramatically simplified by the conventions used in this book, which actually ensure that assignment operators and virtual bases cases do not arise as part of the same class.

• It is hoped that this book and the Appendix ?? can be a useful guide to those participating in, or managing, such large projects (possibly with a short addendum documenting further local conventions).

So, expect a strong and opinionated style in the rest of the book. Hopefully, this will also make the book more interesting to read, even if you don’t agree with all of my opinions (yet!)

1.12 Overview of the Book

The following paragraphs briefly summarize each chapter.

Software Husbandry

This chapter lays out the problem of “husbanding” software, i.e. the careful management and conservation of resources applied to software and software development. A well-specified analysis and design approach is required for programming-in-the-large when a team of programmers must cooperate.

This chapter discusses some of the issues with such analysis and design for object-oriented programming with particular focus on what I call the SOS methodology. Compared to conventional object-oriented analysis and design approaches, this methodology is far more based on programming than non-programming activities.

This chapter calls for providing a well-designed framework for a software system, structured as a pyramid of concepts, types, interfaces and implementations. This framework needs to provide a uniform, consistent, well-specified and efficient structure for the individual modules and their interactions, reducing the cost to maintaining the software in particular. I hope you feel you got that out of reading this chapter so far (because there ain’t much more to the chapter!).
Attribute-based Interface Design

This chapter tackles the issues with good type/interface design, a key issue with achieving a good software framework. Its focus is on a restricted style of interface design in which only attributes appear in the public class interface. The attribute approach specifies conventions for naming, syntax and semantics for each member function in the public interface of the class, including specifying transactional semantics. The benefits of this approach, some experience with it to date, implementation optimization and some outstanding issues are also covered.

Objects, Entities, Values and Descriptions

The key point of this chapter is to identify the differences in interface between real-world objects or entity types and values types, further recognizing named descriptions. The named description is identified as a named value that corresponds to a common name for a class of objects.

It also contrasts object-oriented programming to other paradigms, including procedure-oriented, functional and value-oriented modular programming.

The Expressability of Object-oriented Programming

This chapter explores the expressability of object-oriented programming exploiting user-defined types, (user-defined) type hierarchy (using inheritance) and genericity including templates, namespaces and inheritance. The chapter describes how these concepts are supported in C++ and Java and how they can be used to express the "knowledge pyramid" that forms the framework of well-designed large-scale software systems. Following a recurring theme, the chapter argues for restricting the use of these facilities so that their use carries extra semantics.

Events Notifications and Callbacks

This chapter discusses the interfaces and mechanisms for event notification and handling. It further illustrates the use of a separate interface for notification. It also explores how to structure programs with event notifications and a framework facility that I call an activity that provides timed callback and processing allocation.

Exceptions and Exception-Handling

Error handling can be major complication in programming and is significantly affected by the error signaling approach used in interfaces. This chapter explores
1.12. OVERVIEW OF THE BOOK

techniques for handling error conditions using the C++ and Java exception facility. It develops a unified framework-wide discipline that reduces the complexity of client code and handles the challenges of object-oriented programming, in particular ensuring that the exception types are meaningful to the client even without the client code knowing the specific function throwing the exception.

Logging is also discussed as a means to record problems and report problems in general, a preferrable approach to communicating with the developer/operator. It demonstrates that exceptions do not need to communicate this detailed information.

I also argue that C++ and Java made the right choice to the support the termination model rather than the resumption model.

Smart Pointer and Reference Management

The so-called *smart pointer class* is a common and powerful use of template classes. This chapter discusses reference management and the use these pointer classes as an example use of class templates and common interfaces using templates. The chapter also defines an approach to *reference management* that provides essentially automatic reference management and resource reclamation, except for a small number of framework classes. The mechanisms for allocating and freeing space — memory management — are dealt with in the memory management chapter. I also argue that the pointer class template is the only compelling use of a pointer class in C++. Java provides automatic garbage collection but there are still a few issues of relevance, given Java can lead to memory leaks.

Automatic Garbage Collection

This chapter discusses reference management using garbage collection. It argues against excessive reliance on automatic garbage collection or storage management except possibly as a fall-back mechanism.

Naming, Managers and Modules

Name-based references to objects are supported by a *Directory* interface, implemented by a manager class in a module that implements this type. Objects other than ADTs are best encapsulated by a separate manager class that handles their creation, execution and deletion. This chapter describes this approach, describing further the use of the directory interface as a generic manager interface, thereby supporting name-based access to interfaces. The result is a loose coupling between client software and the library. The chapter also discusses the notion of *design patterns*, noting the manager class as one key example.
Memory Management

Memory management is a key issue in object-oriented programming. This chapter discusses how to structure memory management with emphasis on a framework-wide scheme that allows classes to be used with different types of memory managers, keeping storage class orthogonal to type. The framework-wide “memory manager” approach should be the common case, restricting the use of class-specific definitions of new and delete to truly compelling performance-critical cases. The memory manager approach illustrates the interface and implementation separation advocated in Chapters 4 and ?? as well as the manager construct presented in Chapter 9. Java does not provide overriding of operators comparable to new and delete. However, one can still program specialized allocators in Java for efficiency.

Auditing

This chapter explores the use of auditing for checking the invariants on classes and modules, separate from the normal processing code.

Templates

The C++ template facility is a key implementation technique for reusing code across different types. The objective is to provide you with a solid understanding of how to use templates so we can get on with modeling and simulation techniques. Templates are an implementation facility, not a modeling mechanism. In particular, templates are a powerful way of expressing common interfaces without using inheritance and the logical coupling that inheritance implies.

Java does not provide templates so this chapter is only relevant to Java programming in pointing out some of Java’s limitations.

Interface and Implementation Classes: Separating Interface from Implementation

This chapter further discusses interface classes in detail, focusing on separating interface from implementation. It also discusses optimizations for interface classes.

This chapter promotes the approach of using multiple interfaces for complex objects, rather than one large interface. Complex objects are best structured as multiple classes for modularity, both in the interface and the implementation.
Descriptions

This chapter covers descriptions as a means of providing framework support for representations of complex objects, particularly named descriptions. The use of shared named descriptions with logically large ADTs is also considered. The framework support for a set of related descriptions and the associated supporting classes constitute a layer.

Inter-Object Relationships: Components and Collisions

This chapter explores the structuring of objects as multiple components, with support for interfacing between the components and the containing super object. It also considers a collision model of discovering inter-object interaction and relationships as the opposite extreme from directly connected components. Finally, it describes how to handle various other relationships.

Run-time Type Information

This chapter describes some aspects of the type structure of programs. It further covers the run-time type information facilities in C++ and the appropriate uses of this facility, minimizing its use with virtual functions and static typing.

Inter-class Relationships and Dependencies

The source relationships and dependencies between classes is a critical aspect to manage in large-scale programming projects. Excessively complex and unstructured relationships can make the software harder to understand, more error-prone to modify, and much more expensive to recompile. This chapter explores techniques for minimizing dependencies between classes, especially across layer boundaries, using the notion of layering and dimensions introduced in the previous chapters.

Inheritance versus Composition

Inheritance is examined from the standpoint of trying to get rid of it. This chapter shows that the conventional notion of “isAKindOf” indicating the use of inheritance is far from intrinsic and can be converted to composition with modest changes. Composition offers far more flexibility and encapsulation than inheritance. This approach indicates how to minimize the use of inheritance, consistent with the rest of the book. It also shows how to avoid the use of multiple inheritance and shows the costs and complexities of multiple inheritance.

The next several chapters are concerned with value-oriented programming.
Value-Oriented Programming and Abstract Data Types

This chapter discusses the design and implementation of simple abstract data type classes. These classes support the value-oriented programming aspects of some C++ programs. ADT classes are viewed as a restricted version of a collection class. Operator overloading and use of references are addressed in this context.

Arrays and Value-oriented Collections

This chapter deals with multi-valued ADT classes, including arrays.

Collections

A collection class is a class that holds a set of values, which may be pointers to objects. Examples include stacks, queues, lists, trees, directories, etc. This chapter discusses the design of collection classes, including the use of templates, specialization and support for iteration.

Appendix: Introduction to C++

This appendix provides a brutally fast introduction to the C++ language, primarily relying on the chapters in the text for examples, and focusing on the language features used in this book with key points to remember about the language. It is provided to make this book more complete. A programmer with a background in C (or other systems programming languages) with this book and the ARM should be well-equipped to learn and program in C++.

Appendix: Programming Style and Conventions

Good style, not good languages, produces good programs — good languages just make it easier to produce good programs. Programming is “precision engineering”. Strong conventions are necessary for programming-in-the-large (and very helpful in the small). This appendix attempts to capture the key aspects of C++ programming style that have arisen in the various chapters as well as more detailed style conventions to provide an effective C++ style programming guide.

1.13 Synopsis

Large-scale software development is best viewed as husbandry, managing the evolution of living entities and their associated resources through numerous iterations, investing in improving its quality as well as features and functions along the way.
Software development is a cyclic process. Each cycle extends and refines the design to adapt to new requirements. Less recognized, each cycle should strive to improve the general quality of the software by improving on the previous design and implementation.

Software husbandry requires a careful methodology, just like that practiced in scientific farming. A good software development methodology must support efficient cyclic development.

A software development methodology specifies notation, process and timing. It is like a protocol among the developers, similar in structure and dynamics to a network protocol. Everyone needs to follow the protocol to stay “in synch” and reliably and efficiently produce quality timely results over many generations of the software.

This book advocates a methodology called SOS, standing for the three aspects of the methodology, namely: “S” for the source-code notation (SCR) notation, “O” for the outside-in development (OID) process and, “S” for short cycle development (SCD) timing.

Source code representation (SCR) means representing the requirements, analysis, design and documentation of the software in source code, rather than separate source representations. With suitable programming languages, discipline and tools, it is more efficient, less error prone and easier for programmers than the alternatives.

Outside-in development (OID), as the process, works from the application environment, developing use scenarios through to interface design, followed by implementation and then testing.

Short-cycle development (SCD) for timing is analogous to RISC processing in computer architecture: It is better to have fast simple cycles than slower more complex cycles. Short cycles are lower risk, provide faster feedback, make people management easier, and provide more flexibility and responsiveness for product releases.

This book presents SOS and object-oriented programming techniques to allow it be to applied effectively and efficiently to husband large software through numerous generations, adapting and responding to new requirements and improving the software quality along the way.

C++ is, as Winston Churchill might say\(^\text{13}\), the worst object-oriented programming language, except for all the others.

1.14 Concluding Remarks

As a programmer, I’ve always felt rather alienated by the strange and unpleasant activities that the conventional software engineering approaches try to impose on

\(^{13}\)Actually, Winston made this comment about democracy as a form of government!
the programming process. In particular, the analysis and specification work generally took me away from programming and also seemed to increase the amount of “manual” work to do. Moreover, these non-programming activities did not seem to necessarily lead to a better result. In contrast, in the company of programmers, one wrote software and tried to automate things as much as possible.

For some, the answer has been to add yet more non-programming activities. For me, the idea is to get rid of the non-programming activities as much as possible, except for sleeping, eating and windsurfing. After all, if you are programming, you are producing software, and if you are not, you are not\textsuperscript{14}.

This chapter builds on this programmer liberation strategy, arguing that good object-oriented programming in conjunction with the right tools eliminates the need for these non-programming activities, and the associated non-programmers, from our work.

Software development has also suffered from questionable measures of productivity, such as lines of code per month. This chapter introduces the notion of software development as husbandry. We need to be as concerned as much about the quality of the herd as the individuals that are sent to market each year. This view leads to an emphasis on software development as a cyclic process, and a software development methodology, SOS, that supports this.

Coming out of this chapter, we recognize the need for a good framework to reduce the overhead of developing programs based on this prospective. A good framework is important for short-cycle development. If the framework provides extensive functionality, the cost of each iteration is reduced because much of the added functionality can be provided by existing code in the framework. The separation of concerns between the application and the framework influences the framework code to be more general and more extensible, allowing cleaner iterations.

Subsequent chapters provide more detailed techniques that elaborate on the general process described here.

\textbf{Exercises}

1. Write a concise specification for a sort procedure and compare its length to that of the source code interface to the sort procedure. Which is longer to write?

2. Develop an \textit{ontology} for a network simulation program, namely a description of the concepts, terminology, types and interfaces.

\textsuperscript{14}In some cases, the impetus for non-programming activities comes from managers who are not programmers, who promote non-programming activities either by their own initiative or after being conned by supposed software methodology gurus, who often also seem to be non-programmers.
Background

Considerable work on so-called software engineering predates object-oriented programming. Much of it seems dated because of this. Moreover, some of it draws the parallel with other areas of engineering where the design is different than the product. With software written using recent high-level mechanisms, the design and the product are one and the same.
Bibliography

[9] Steve Maguire, Debugging the Development Process, Microsoft Press,