Beyond UML:

Advanced Systems Delivery

with Objects, Components, Patterns & Middleware

Graham McLeod

© Inspired Press 2001
Beyond UML:
Advanced Systems Delivery
with Objects, Components,
Patterns & Middleware

Abridged Student Version

Graham McLeod
mcleod@iafrica.com

Inspired Press

P O Box 384
Howard Place
7450
South Africa
www.inspired.org
Cataloguing Data

Publisher: Inspired Press
P O Box 384
Howard Place
7450
South Africa
www.inspired.org

Copyright: 2001, Graham McLeod
mcleod@iafrica.com


All rights reserved. Copyright under the Berne convention. No part of this work may be reproduced, transmitted, stored in a retrieval system or otherwise disseminated without the written permission of the author.
Dedication

This work is dedicated to all the many system professionals, colleagues and students who have taught me, mentored me, and contributed their ideas and experience over many years. These include the folks at NCR Corporation, Comcon, UCT Information Systems and Inspired as well as our many clients and suppliers. Special thanks to James (Jim) Odell (methodologist and OO expert), Ulrich Frank (academic, researcher and Business Modeling pioneer), Terry Halpin (academic, Object Role Modeling author and tool builder) who have had major influences upon my work and thinking.

Thanks are also due to my wife, Hilary, son Matthew and daughter Zoe who have endured my many hours of unavailability during the long labour.

Remembering my colleague Jenny, whom I shall miss terribly.
This page intentionally left blank.
Preface

Intent and Scope

This work is intended to enable the reader to understand and apply turn of the millennium techniques for the rapid delivery of commercial software applications. It is not a text on programming: the reader is assumed to already be familiar with at least one programming language and the general principles of producing a working program in isolation. It covers the production of larger integrated systems of programs, which will work together as an integrated application to solve a business problem, satisfy an operational support need, or in some other way add value to the operation of an enterprise. The enterprise does not necessarily have to be “for profit”, but could include service organizations such as clinics, hospitals, public administration and the like.

We assume that the system to be delivered will have direct interaction with users, either through graphical user interfaces (e.g. Windows, Macintosh) an Internet web browser (Netscape, Internet Explorer) or some other device, such as a Personal Digital Assistant (PDA) or cell phone. Further, we assume that the system will service multiple users concurrently and manage a significant amount of shared data held persistently in a database management system of some sort. This may be relational (e.g. SQL Server™, Oracle™, DB2™, Sybase™...) or object oriented (ObjectStore™, Gemstone™, Objectivity™...). Emerging XML data stores may also be appropriate.

Typical kinds of systems which developers might want to deliver include:

- An operational accounting system such as Accounts Receivable (Debtors) or Accounts Payable (Creditors)
- Inventory Management (Stock Control)
- Personnel Management and Payroll
- Systems to handle bookings, scheduling and the allocation of resources
- Customer Relationship Management (CRM)
- Order Processing
- Financial systems, such as Banking, Investments and Insurance
- Retail Point of Sale
- Standalone personal productivity software, such as project management, personal organizers, user oriented databases etc.
- e-Commerce systems deployed on the world wide web: both Business to Business (B2B) and Business to Consumer (B2C)

The techniques presented in this text will assist you to deliver any of the above. What we do not specifically cover, is other types of systems requiring specialised knowledge or data types. These would include:
- Speech input and recognition
- Video processing
- Data warehouse interrogation
- Interactive learning software
- Software development tools (editors, compilers, linkers, CASE tools and the like)
- Embedded systems for the realtime control of devices

The goals we set for our method include:

- **Comprehensiveness** - the method should provide all the necessary techniques to allow you to move from high level business analysis through the whole lifecycle, including design, to delivery of a quality working application which is easily maintained or adapted in future

- **Economy** - the method should provide the most efficient way to reach the goals of delivering a quality system with minimal effort required of analysts, designers, developers and domain experts

- **Wide applicability** - the techniques should work for the range of system types typically encountered by commercial developers, including client server, Internet applications and mobile client applications

- **Ease of Use** - the methods and techniques should be accessible to practitioners with only a high school education, not requiring an advanced degree or formal knowledge, while ensuring a reasonable degree of rigour in applications produced

- **Effective Communication** - most projects that fail do so because of poorly understood requirements or misinterpretation of the domain: the methods must allow the validation of the specifications and designs by non-IS
domain experts. There should also be seamless communication between analysts, designers and programmers.

**Why “Beyond UML” ?**

The Unified Modeling Language (UML) is currently a standard promoted by the Object Management Group (OMG) for Object Oriented Analysis and Design. It grew out of pioneering work in object methods by Grady Booch, James Rumbaugh and colleagues, Ivar Jacobson (these “three amigos” now all of Rational Corporation) and James (Jim) Odell, an independent author and consultant.

UML brought together the object modeling notations of the various players into a “standard”, specified by the UML documents, currently at release 1.4. You can access the full set of documents at [www.omg.org](http://www.omg.org). UML has some strengths, including:

- **Wide industry support**: The precursors of UML and UML itself are probably used (at least nominally) by as high as 70% of practitioners practicing Object Oriented analysis and design.

- **Wide tool support**: Because UML is a “standard”, it is an obvious choice for support by tool vendors. There are therefore many tools which support UML, in pure or polluted forms. These include: Rational Rose™, Object Team™, Visio™, Select™, System Architect™ and many more.

- **Fairly good static modeling capabilities** for the definition of types, classes, objects and relationships, supporting associations, aggregation and inheritance.

However, in our opinion, the UML also has a number of shortcomings and problems, including:

- **It is just a notation** - Although there is a meta model defined as part of UML which tries to provide a formal semantics for the various model elements, the meta model is not consistent or complete and deviates from accepted meta modeling practice in mixing levels of abstraction.

- **There is no specified standard process** for developers to follow, leading to confusion as to which techniques to use for what purposes, in what sequence etc. Many organizations take on UML assuming it is a methodology, only to find that it provides no guidance to their development teams other than what the finished models should look like when analysis or design activity are complete. This has been addressed by Rational Corporation by the definition of the Rational Unified...
Process, which is extensive but also conservative and traditional in its support for a “waterfall” style lifecycle. Various Rational authors have produced other, conflicting, process books and guides, adding to confusion. There are currently calls for proposals and responses wending their way through the OMG procedures to become a process standard, but these do not seem to hold sufficient promise for our goals.

- **Weak modeling of dynamics at the business and systems level**
  This is one of our main problems with UML. The suggested approach to modeling requirements, viz. Use Cases does not have sufficient or consistent enough semantics to properly capture requirements at the business level, especially where business processes are concerned. We elaborate on this later, but Use Cases encourage a system oriented view too early and do not provide enough support for documenting, evaluating and engineering business processes that are effective and that include computerised and other activities, possibly spanning a variety of systems.

- **Difficult transitions between the user requirements level and the design level.** Many organizations do a great deal of use case modeling, which appears to go reasonably well, only to have significant difficulty in translating the models produced into workable designs and rigorous enough specifications to drive the technical work in the project.

- **No explicit modeling of user interfaces or system interfaces to other environmental entities.** UML does model interaction of users with the system, but at a high, fuzzy level. We prefer to use prototyping to clarify the intended look and feel of the system, the modes of interaction, the dialogues which will take place. We also provide mechanisms to transition smoothly from a high level view with a named but loosely defined interaction, to a logically specified one, to an example of the actual screen or report involved.

- **No explicit support for patterns and reuse.** UML supports object oriented techniques, but does not specify how to use patterns - a key element in achieving high level reuse of designs, architectures, best practice processes and algorithms.

- **No architectural guidance for the design of flexible and rapidly adaptable applications.** UML specifies how to document an architecture, but provides little guidance in how to map from the system specification level to a detailed technical implementation architecture that is robust and extensible.
History of the Method Presented

The Inspired Method has grown up over many years. The author has worked with structured methods from the 70’s, Entity Modeling and Relational Modeling since the late 70’s/early 80’s and advanced integrated methods (mostly information engineering based) through the 80’s. Initial exposure to object concepts, which we immediately found intuitive and appealing, came in 1980 with a landmark article in Byte Magazine which described the Smalltalk 80 environment. At the time, Smalltalk was not easily available in our commercial non-US world. Through the 1980’s the author worked on the development, refinement and practical implementation of advanced I.E. style methods, including SSADM, Advanced Development Method (ADM) and Tetrarch. He was also involved in the development of a complementary Project Management method, called GOLD. With his transition to academia and starting Inspired in 1991, he started working with commercial clients in implementing object oriented methods after learning Smalltalk in 1990.

The early Inspired method drew upon work by James Odell and James Martin who had provided the most comprehensive and practical OO method for commercial use at that time; as well as contributions from Peter Coad and Grady Booch. We enhanced the method of Odell with additional business modeling techniques drawn from GMD in Germany and inspired by value chain analysis and business process reengineering work done by Porter, Hammer, and others. We extended the design notations of Odell into mapping to layered architectures based upon the Model, View Controller architecture developed by Trygve Renskaugh. By 1994, we had a well integrated method spanning business analysis, process modeling and engineering, systems analysis, design and architecture mapping. This proved practical and effective and was widely taught and used in developing client server systems.

When UML emerged in the 1996 time frame, we examined it to see what we could learn and whether we should be compatible, which was obviously desirable if it was practical and effective. UML was also gathering support from tool manufacturers, so there were obvious advantages to using a notation that was well supported.

We concluded that we could happily use the static modeling side of UML (Class and Object Diagrams, Static Structure Diagrams), changing our notation to be compatible. We found UML severely lacking in the dynamic modeling area, though, for the many reasons we cited above. We thus resolved to stay with our techniques and notation in these areas. We later worked on extending activity diagrams, which were incorporated in UML via the OMG, to incorporate the concepts we needed for competent dynamic
modeling. This work was first presented publicly outside our client/student base at UML ’98 in Mulhouse, France.

Our method has continued to evolve and become more refined. Latterly, we have integrated the system level techniques increasingly with corporate level planning and improvement approaches based upon architectures linking business, applications, information and technology infrastructures.

**Major Sections**

We will begin by discussing the objectives of system delivery and what we need to do in order to be successful. Next, we’ll take a look at the problems and difficulties that often arise and try to determine their causes, so we can avoid these pitfalls. We will introduce a method which provides a high level road map of the system delivery process and various techniques and deliverables that we will use and produce along the way. Then we will begin in earnest by looking at how to scope a project and examine its context. Next we introduce you to a host of important concepts related to objects, components and their derivatives.

The following chapter looks at the equally important issue of how we make best use of time and people to try and obtain high quality results and accelerate the project. We introduce joint application development (JAD) facilitation concepts here. From there, we move on to examining and modeling the problem/application domain. This is to gain a thorough understanding of the business or application area we are working in and to lay the foundations for sound modeling of business processes and design of databases later in the project.

We then lift the view to the enterprise level, with the intent of discovering the stakeholders we will interact with, the inputs and outputs that they expect, the business processes that they initiate and the value chains that keep the enterprise in business. This forms the foundation for examining, and potentially improving business processes which comprise the dynamic requirements of the system. Note that we use the word “system” here in the broad sense, including, but not limited to, computerised elements.

Following this, we discuss an exciting new idea: That of using multimedia and scenarios for requirements definition and communication to designers and programmers. This is followed by a discussion of the styles and pro’s and con’s of prototyping as a means to refine specifications and reduce risk. This, and the following chapter, which discusses event modeling, provide the tools to fully describe the detailed functioning of the system in a dynamic way. The latter also provides an explicit mapping to the domain object
model (business objects) mentioned earlier, thus providing a very strong verification of the integrity of both dynamic and static models.

We move on to discussing the design environment, with a view to thoroughly understanding our options, for only in this way can we most accurately translate requirements to a solution. We want to do this effectively, to meet business and user expectations, as well as efficiently, so that we spend minimum effort and time to deliver a quality result. Here is where we will see how we can take best advantage of patterns, component libraries, frameworks and leveraging legacy components. We’ll also examine options for the storage of our business objects, including object and relational databases. We discuss the features available in object tools and environments to assist us in our implementation efforts. Criteria for tool selection are also discussed. User interface options and issues are explored.

With the background gained above, we are in a position to translate our requirements and specifications into a design, using architectural principles. We present some patterns, models and frameworks to assist us with this and a process for achieving a seamless mapping. This produces a flexible design which is suitable for implementation in a multi-tier client server environment. We favour a thin client approach, which could target desktop GUI terminals (e.g. Windows™), Internet style browser clients, or even mobile devices (such as a cell phone using the Wireless Access Protocol - WAP). Business processes are maintained in a discreet layer which might be implemented in an application server.
This page intentionally left blank.
# Brief Contents

<table>
<thead>
<tr>
<th>Cataloguing Data</th>
<th>ii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dedication</td>
<td>iii</td>
</tr>
<tr>
<td>Preface</td>
<td>v</td>
</tr>
<tr>
<td>Brief Contents</td>
<td>xiii</td>
</tr>
<tr>
<td>Detailed Contents</td>
<td>xiv</td>
</tr>
<tr>
<td>Chapter 1 Objectives of Systems Delivery</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2 Systems Delivery Problems and Approaches</td>
<td>5</td>
</tr>
<tr>
<td>Chapter 3 The Inspired Method</td>
<td>21</td>
</tr>
<tr>
<td>Chapter 4 Scoping and Context</td>
<td>45</td>
</tr>
<tr>
<td>Chapter 5 Object Orientation Concepts</td>
<td>53</td>
</tr>
<tr>
<td>Chapter 6 Joint Application Development (JAD)</td>
<td>81</td>
</tr>
<tr>
<td>Chapter 7 Business Domain Object Modeling</td>
<td>93</td>
</tr>
<tr>
<td>Chapter 8 Enterprise Modeling</td>
<td>113</td>
</tr>
<tr>
<td>Chapter 9 Business Process Modeling</td>
<td>121</td>
</tr>
<tr>
<td>Chapter 10 Business Engineering (Process Improvement)</td>
<td>147</td>
</tr>
<tr>
<td>Chapter 11 Multimedia Specifications and Scenarios</td>
<td>165</td>
</tr>
<tr>
<td>Chapter 12 Prototyping</td>
<td>171</td>
</tr>
<tr>
<td>Chapter 13 Event Modeling</td>
<td>177</td>
</tr>
<tr>
<td>Chapter 14 Business Rules</td>
<td>189</td>
</tr>
<tr>
<td>Chapter 15 Object Support in Tools</td>
<td>197</td>
</tr>
<tr>
<td>Chapter 16 User Interfaces and Events</td>
<td>227</td>
</tr>
<tr>
<td>Chapter 17 Additional Constructs</td>
<td>243</td>
</tr>
<tr>
<td>Chapter 18 Persistence</td>
<td>257</td>
</tr>
<tr>
<td>Chapter 19 Libraries and Frameworks</td>
<td>271</td>
</tr>
<tr>
<td>Chapter 20 Architectures</td>
<td>277</td>
</tr>
<tr>
<td>Chapter 21 Using Legacy Components</td>
<td>309</td>
</tr>
<tr>
<td>Chapter 22 Visual Assembly</td>
<td>313</td>
</tr>
<tr>
<td>Chapter 23 Patterns</td>
<td>331</td>
</tr>
<tr>
<td>Chapter 24 Mapping the Models to a Layered Architecture</td>
<td>339</td>
</tr>
<tr>
<td>Index</td>
<td>353</td>
</tr>
</tbody>
</table>
Detailed Contents

Cataloguing Data .................................................. ii
Dedication .......................................................... iii
Preface ......................................................................... v
Intent and Scope ......................................................... v
Why “Beyond UML” ? .................................................. vii
History of the Method Presented ............................. ix
Major Sections ........................................................... x
Brief Contents ............................................................. xiii
Detailed Contents ........................................................ xv
Chapter 1 Objectives of Systems Delivery .................. 1
Systems Delivery vs Systems Development ................... 1
Nonfunctional Requirements ............................... 2
Chapter 2 Systems Delivery Problems and Approaches .... 5
Understanding the Business Problem .................... 5
Gaining and Maintaining User Buy-in .................. 6
  Educate the Users ....................................................... 6
  Focus on High Benefit Features ................................. 7
  Keep User Involvement High ..................................... 8
  Managing Expectations ............................................ 9
  Frequent Delivery ...................................................... 10
  Intelligible deliverables ............................................ 11
Dealing with Complexity ........................................ 11
  Break it down - Miller ............................................... 11
  Manage boundaries .................................................. 12
  Hierarchy ................................................................. 12
  Architecture ............................................................ 12
  KISS (Keep It Simple, Stupid!) ................................. 13
  Objects .................................................................... 14
  Components, Frameworks, Patterns, Libraries, Packages .................. 14
Shifting Sands ............................................................. 14
  Change is inevitable .................................................. 14
  People move on too ................................................. 15
  Repository necessary .............................................. 15
  Shared communications language .................. 15
  Discipline and change management ................... 16
Uncharted Territory .................................................. 16
  New technology ....................................................... 16
Encapsulation and Interfaces ........................................ 57
Classes ........................................................................ 58
Inheritance .................................................................... 59
  Multiple Inheritance .................................................. 62
Polymorphism ................................................................ 63
Complex Objects .......................................................... 63
Collections and Iterators .............................................. 65
Persistence ..................................................................... 66
How does OO affect us? ................................................ 66
History .......................................................................... 67
Components ................................................................. 69
  Components .................................................................. 69
  Class Libraries ........................................................... 69
Building Applications with OO ....................................... 70
Object Oriented Databases ............................................ 72
Object Oriented Models ................................................ 75
An Object Modeling notation ........................................ 77
Classes and Inheritance ................................................ 77
Associations (Relationships) ......................................... 77
Composition .................................................................. 78
References and Further Reading .................................... 79
**Chapter 6 Joint Application Development (JAD)** ............ 81
Background ................................................................... 81
Purpose of JAD ............................................................. 81
How JAD works ............................................................ 84
Where we can use JAD .................................................. 85
Relevance of JAD to Data Management ......................... 85
Participants ................................................................... 86
JAD Process ................................................................... 88
Session Preparation ...................................................... 88
JAD Rules ..................................................................... 89
The Venue ..................................................................... 89
The JAD session ............................................................ 89
Group dynamics ............................................................ 90
Post-session Work ........................................................ 91
Success Factors ............................................................ 91
Results Obtained .......................................................... 92
A Caveat ....................................................................... 92
**Chapter 7 Business Domain Object Modeling** .............. 93
Purpose ......................................................................... 93
Notation ....................................................................... 94
Business Objects versus Controls/Widgets ...................... 94
Drawing the Model ......................................................... 95
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specialization and Generalisation</td>
<td>95</td>
</tr>
<tr>
<td>Association</td>
<td>98</td>
</tr>
<tr>
<td>Association Class</td>
<td>101</td>
</tr>
<tr>
<td>Aggregation and Composition</td>
<td>102</td>
</tr>
<tr>
<td>Navigability of Links</td>
<td>103</td>
</tr>
<tr>
<td>Stereotypes</td>
<td>104</td>
</tr>
<tr>
<td>Resolving Multiple Inheritance Issues</td>
<td>104</td>
</tr>
<tr>
<td>Patterns</td>
<td>106</td>
</tr>
<tr>
<td>Showing Distribution</td>
<td>106</td>
</tr>
<tr>
<td>Power Types</td>
<td>107</td>
</tr>
<tr>
<td>Bringing it all together</td>
<td>108</td>
</tr>
<tr>
<td>References and Further Reading</td>
<td>111</td>
</tr>
<tr>
<td>Chapter 8 Enterprise Modeling</td>
<td>113</td>
</tr>
<tr>
<td>The View from 10 000 Feet</td>
<td>113</td>
</tr>
<tr>
<td>Stakeholders</td>
<td>114</td>
</tr>
<tr>
<td>Value Chains</td>
<td>115</td>
</tr>
<tr>
<td>Extended Value Chains</td>
<td>119</td>
</tr>
<tr>
<td>Identifying Business Processes</td>
<td>119</td>
</tr>
<tr>
<td>References and Further Reading</td>
<td>120</td>
</tr>
<tr>
<td>Chapter 9 Business Process Modeling</td>
<td>121</td>
</tr>
<tr>
<td>Process Orientation</td>
<td>121</td>
</tr>
<tr>
<td>Business Process Modeling</td>
<td>123</td>
</tr>
<tr>
<td>Inspired Architecture Frameworks</td>
<td>123</td>
</tr>
<tr>
<td>Previous Work with Odell notation</td>
<td>125</td>
</tr>
<tr>
<td>Extending UML</td>
<td>126</td>
</tr>
<tr>
<td>Extending Activity Diagrams to Support Business Modeling</td>
<td>128</td>
</tr>
<tr>
<td>Agents</td>
<td>128</td>
</tr>
<tr>
<td>Inputs and Outputs</td>
<td>129</td>
</tr>
<tr>
<td>Simple High Level Example</td>
<td>129</td>
</tr>
<tr>
<td>Triggers</td>
<td>130</td>
</tr>
<tr>
<td>Outcomes</td>
<td>131</td>
</tr>
<tr>
<td>Another Example</td>
<td>131</td>
</tr>
<tr>
<td>Selective Invocation</td>
<td>132</td>
</tr>
<tr>
<td>Either Condition Should Trigger the Activity</td>
<td>132</td>
</tr>
<tr>
<td>Activity is only triggered when all conditions are met</td>
<td>133</td>
</tr>
<tr>
<td>Rules</td>
<td>133</td>
</tr>
<tr>
<td>Resources</td>
<td>133</td>
</tr>
<tr>
<td>Swim Lanes and Bounding Boxes</td>
<td>134</td>
</tr>
<tr>
<td>Getting More Detailed</td>
<td>135</td>
</tr>
<tr>
<td>Instrumenting the Process Model</td>
<td>135</td>
</tr>
<tr>
<td>An Example</td>
<td>137</td>
</tr>
<tr>
<td>Business Engineering (Process Improvement)</td>
<td>141</td>
</tr>
<tr>
<td>Conclusion</td>
<td>144</td>
</tr>
</tbody>
</table>
### References and Further Reading

145

---

**Chapter 10 Business Engineering (Process Improvement)**

147

The Pressures Facing Business

147

Process Changes

148

New Look Organizations

148

- Leaner

149

- Virtual

149

- Fractal

150

Opportunities with Technology

150

- Serial vs Parallel

150

- Escape the Constraints

151

- Disintermediation and Reintermediation

153

Principles of Business Engineering

154

- An Example

154

- Steps to Follow

156

- Another Example

157

- Things to Respect

158

- Balancing Innovation and Improvement

159

- Continuous Improvement - Kaizen

160

- Coupling and Harnessing Both Techniques

161

- Beware! Caveats for Business Engineers

163

References and Further Reading

164

---

**Chapter 11 Multimedia Specifications and Scenarios**

165

CREWS project

165

OLE/OpenDoc

166

Browser with Hyperlinks

167

Persistent Conversations

167

Notation

168

Advantages

168

Scenarios

168

References and Further Reading

170

---

**Chapter 12 Prototyping**

171

Definition

171

Types of Prototype

171

- Requirements Prototype

171

- Technical Prototype

171

- Evolutionary Prototype

172

- Throw Away Prototype

172

Uses of Prototyping

172

Dangers of Prototyping

172

- Prototype becomes Production

172

- Prototyping in a Vacuum

174

- Expanding Requirements

175
**Chapter 20 Architectures** .............................................. 277
Client Server ................................................................. 277
An architecture, not an implementation ............................. 277
Typical Clients and Servers .............................................. 278
Joining the components .................................................. 279
Client Server Drivers ....................................................... 280
The Gartner Model ......................................................... 281
Two and Three Tier ......................................................... 283
Interfaces, Interfaces ....................................................... 285
Middleware ................................................................. 287
Advanced Options, DCE .................................................. 288
Distributed Objects ......................................................... 289
New Generation Tools ..................................................... 293
The Internet as a delivery platform .................................... 294
Enter Java ................................................................. 295
Component Models ......................................................... 299
Competing Component Models ........................................ 300
Containers ................................................................. 301
Server Components ....................................................... 303
Microsoft Alternative .................................................... 306
Wither CORBA ........................................................... 306
Stateless or Statefull ? .................................................... 307

**Chapter 21 Using Legacy Components** .......................... 309
Identifying Useful Things ............................................... 309
Harvesting ................................................................. 310
IDL as a Lingua Franca ................................................... 311
Replacing the components over time ................................ 312

**Chapter 22 Visual Assembly** ......................................... 313
Summary ................................................................. 313
Improvements in the Software Development Process ............ 314
Productivity and Quality ................................................. 316
Traditional Failings of CASE .......................................... 316
Ways of Building a Car ................................................... 317
The Architecture of a Visual Assembly Tool ....................... 319
Commercial Products .................................................... 321
A typical construction scenario ....................................... 323
Early Productivity Indicators .......................................... 326
Implications ............................................................... 326
References and Further Reading ...................................... 328

**Chapter 23 Patterns** .................................................. 331
History ................................................................. 331
Definition ............................................................... 332
Types of Patterns .......................................................... 333
<table>
<thead>
<tr>
<th>Pattern Languages</th>
<th>334</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern Sources</td>
<td>336</td>
</tr>
<tr>
<td>Anti-Patterns</td>
<td>336</td>
</tr>
<tr>
<td>References and Further Reading</td>
<td>337</td>
</tr>
<tr>
<td><strong>Chapter 24 Mapping the Models to a Layered Architecture</strong></td>
<td>339</td>
</tr>
<tr>
<td>Introduction</td>
<td>339</td>
</tr>
<tr>
<td>The Model-View-Controller Pattern</td>
<td>340</td>
</tr>
<tr>
<td>Integrating the Models</td>
<td>343</td>
</tr>
<tr>
<td>An Example</td>
<td>345</td>
</tr>
<tr>
<td>Reusability</td>
<td>349</td>
</tr>
<tr>
<td>Maintenance and Flexibility</td>
<td>350</td>
</tr>
<tr>
<td>Summary</td>
<td>350</td>
</tr>
<tr>
<td>References and Further Reading</td>
<td>351</td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td>353</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Feature Selection Matrix</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Relative Levels of Involvement through the Lifecycle</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Inspired Method Overview</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Increasing Fidelity with Progress Through the Lifecycle</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Sample Context Diagram</td>
<td>27</td>
</tr>
<tr>
<td>3.4</td>
<td>Sample Business Object Domain Model</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>Business Process Model</td>
<td>30</td>
</tr>
<tr>
<td>3.6</td>
<td>Prototyping</td>
<td>32</td>
</tr>
<tr>
<td>3.7</td>
<td>Sample Event Model</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>Layered Application Architecture</td>
<td>34</td>
</tr>
<tr>
<td>3.9</td>
<td>Repository Contents</td>
<td>37</td>
</tr>
<tr>
<td>3.10</td>
<td>The Big Picture</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Project Definition</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>Context Diagram Structure</td>
<td>49</td>
</tr>
<tr>
<td>4.3</td>
<td>Context Diagram Example</td>
<td>49</td>
</tr>
<tr>
<td>4.4</td>
<td>Technical Environment Model</td>
<td>50</td>
</tr>
<tr>
<td>4.5</td>
<td>Technical Environment Model Example</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>Conventional vs Object Structure</td>
<td>55</td>
</tr>
<tr>
<td>5.2</td>
<td>What is an Object?</td>
<td>56</td>
</tr>
<tr>
<td>5.3</td>
<td>Interface is Public, Implementation is Private</td>
<td>58</td>
</tr>
<tr>
<td>5.4</td>
<td>Classes and Instances</td>
<td>59</td>
</tr>
<tr>
<td>5.5</td>
<td>Inheritance</td>
<td>60</td>
</tr>
<tr>
<td>5.6</td>
<td>A Sample Smalltalk Class Definition</td>
<td>61</td>
</tr>
<tr>
<td>5.7</td>
<td>Java Class Definition</td>
<td>62</td>
</tr>
<tr>
<td>5.8</td>
<td>Multiple Inheritance</td>
<td>63</td>
</tr>
<tr>
<td>5.9</td>
<td>Composite Objects</td>
<td>64</td>
</tr>
<tr>
<td>5.10</td>
<td>Collection Classes</td>
<td>65</td>
</tr>
<tr>
<td>5.11</td>
<td>Language Evolution</td>
<td>68</td>
</tr>
<tr>
<td>5.12</td>
<td>Components vs Class Library</td>
<td>71</td>
</tr>
<tr>
<td>5.13</td>
<td>Class Libraries</td>
<td>72</td>
</tr>
<tr>
<td>5.14</td>
<td>Object Database</td>
<td>73</td>
</tr>
<tr>
<td>5.15</td>
<td>DBMS Suitability</td>
<td>74</td>
</tr>
<tr>
<td>5.16</td>
<td>Inheritance Relationships</td>
<td>77</td>
</tr>
<tr>
<td>5.17</td>
<td>Associations (Relationships)</td>
<td>78</td>
</tr>
<tr>
<td>5.18</td>
<td>Composition (aggregation) relationship</td>
<td>78</td>
</tr>
<tr>
<td>5.19</td>
<td>Sample Class Diagram</td>
<td>79</td>
</tr>
<tr>
<td>6.1</td>
<td>Source of Errors in Lifecycle</td>
<td>82</td>
</tr>
</tbody>
</table>
6.2 - Proportion of Effort to Correct Errors .................................. 82
6.3 - Relative Cost to Fix Errors ................................................. 83
6.4 - Conventional vs JAD Approach ........................................ 84
6.5 - The Jad Cycle ................................................................. 88
7.1 - Specialisation Through Inheritance .................................... 96
7.2 - Non-disjoint Subclasses .................................................... 96
7.3 - Disjoint and Non-disjoint Specialisations ............................. 97
7.4 - Using Attributes to Identify a Superclass ............................. 97
7.5 - Using Attributes to Find Specialisations ............................. 98
7.6 - Associations and Multiplicities ........................................ 99
7.7 - Labelled Associations ....................................................... 100
7.8 - Association Class ............................................................ 101
7.9 - Aggregation ................................................................. 102
7.10 - Indicating Navigability of Associations .............................. 103
7.11 - Indicating Stereotypes on Class Diagram ............................ 104
7.12 - Multiple Inheritance Implied - Many Permutations ............. 104
7.13 - Resolving Multiple Inheritance Permutations ..................... 105
7.14 - Multiple Inheritance Resolution Pattern applied to Fiction Work . 106
7.15 - Bounding Boxes show Distribution .................................... 107
7.16 - Power Type ................................................................. 108
7.17 - A Banking Domain Model ................................................ 109
7.18 - Class Box Expanded ....................................................... 110
8.1 - Stakeholder Model .......................................................... 115
8.2 - Porter’s Value Chain Model .............................................. 116
8.3 - Value Chains and Business Processes Cut Across Functional Boundaries ..................................................... 117
9.1 - Inspired Frameworks© Process Architecture ....................... 124
9.2 - High Level Process Model - Equivalent to Use Case ............... 130
9.3 - Simple business process model ......................................... 132
9.4 - Rules added to Models ..................................................... 133
9.5 - Modeling resources and consumption ................................ 134
9.6 - Swim Lanes and Bounding Boxes ...................................... 135
9.7 - Current Business Process .................................................. 138
9.8 - Reengineered Business Process ......................................... 142
10.1 - Flatter Organizations ....................................................... 149
10.2 - Technology makes concurrent work possible ...................... 151
10.3 - Perceived Constraints and Remedies ................................. 152
10.4 - Business process before reengineering ............................... 154
10.5 - Business process after reengineering ............................... 156
10.6 - Another business reengineering example ........................... 157
10.7 - Characteristics of Innovation ........................................... 160
10.8 - Kaizen - Slow continuous improvement ............................. 161
10.9 - Sustainable Innovation Model .......................................... 162
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1 - Comparing Prototype and Construct with Incremental Development</td>
<td>174</td>
</tr>
<tr>
<td>12.2 - Dangerous Prototyping</td>
<td>175</td>
</tr>
<tr>
<td>12.3 - Useful Prototyping</td>
<td>175</td>
</tr>
<tr>
<td>13.1 - Relationship between dynamic models</td>
<td>178</td>
</tr>
<tr>
<td>13.2 - Activities affecting multiple object types are decomposed</td>
<td>180</td>
</tr>
<tr>
<td>13.3 - Example of a precondition rule</td>
<td>181</td>
</tr>
<tr>
<td>13.4 - Example of a view definition</td>
<td>182</td>
</tr>
<tr>
<td>13.5 - Referencing IO Specification on the model</td>
<td>183</td>
</tr>
<tr>
<td>13.6 - Event model with logical transaction boundary</td>
<td>185</td>
</tr>
<tr>
<td>13.7 - Common logic in event model</td>
<td>187</td>
</tr>
<tr>
<td>14.1 - Rules linked to models</td>
<td>191</td>
</tr>
<tr>
<td>15.1 - Evolution of OO Technology</td>
<td>198</td>
</tr>
<tr>
<td>15.2 - Comparison of some object languages</td>
<td>203</td>
</tr>
<tr>
<td>15.3 - Abstraction Level of Languages</td>
<td>206</td>
</tr>
<tr>
<td>15.4 - Classes provided in some popular tools</td>
<td>207</td>
</tr>
<tr>
<td>15.5 - Continua for overall decision making</td>
<td>209</td>
</tr>
<tr>
<td>15.6 - Different tool options vs team skill/capacity</td>
<td>210</td>
</tr>
<tr>
<td>15.7 - Development Tools Architecture Environment</td>
<td>211</td>
</tr>
<tr>
<td>15.8 - Development Environment Architecture - Choices Populated</td>
<td>216</td>
</tr>
<tr>
<td>15.9 - Typical CASE tool</td>
<td>217</td>
</tr>
<tr>
<td>15.10 - Categories of tools</td>
<td>218</td>
</tr>
<tr>
<td>15.11 - An integrated CASE environment</td>
<td>219</td>
</tr>
<tr>
<td>15.12 - Code generation - CASE “Holy Grail”</td>
<td>221</td>
</tr>
<tr>
<td>15.13 - Reverse engineering via CASE</td>
<td>222</td>
</tr>
<tr>
<td>15.14 - A fool with a tool is still a fool...</td>
<td>225</td>
</tr>
<tr>
<td>16.1 - A graphical user interface</td>
<td>230</td>
</tr>
<tr>
<td>16.2 - Modal processing</td>
<td>231</td>
</tr>
<tr>
<td>16.3 - An event handler</td>
<td>232</td>
</tr>
<tr>
<td>16.4 - Event Processing</td>
<td>233</td>
</tr>
<tr>
<td>16.3 - Lifestreams Prototype</td>
<td>237</td>
</tr>
<tr>
<td>17.1 - New data types</td>
<td>245</td>
</tr>
<tr>
<td>18.1 - Image File</td>
<td>258</td>
</tr>
<tr>
<td>18.2 - Persistence via Operating System Files</td>
<td>259</td>
</tr>
<tr>
<td>18.3 - Parking the CAR object in a Relational Garage</td>
<td>261</td>
</tr>
<tr>
<td>18.4 - Using a relational database for object persistence</td>
<td>262</td>
</tr>
<tr>
<td>18.5 - Object Database</td>
<td>266</td>
</tr>
<tr>
<td>18.6 - ODMG’93, ’95 and SQL standards compared</td>
<td>267</td>
</tr>
<tr>
<td>20.1 - Client Server Concepts</td>
<td>278</td>
</tr>
<tr>
<td>20.2 - Typical Servers</td>
<td>279</td>
</tr>
<tr>
<td>20.3 - Typical Topology</td>
<td>280</td>
</tr>
<tr>
<td>20.4 - Application Components</td>
<td>281</td>
</tr>
<tr>
<td>20.5 - Simple Gartner Client Server Model</td>
<td>282</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>20.6</td>
<td>2 Tier vs 3 Tier Client Server</td>
</tr>
<tr>
<td>20.7</td>
<td>Application Programming Interfaces</td>
</tr>
<tr>
<td>20.8</td>
<td>Need for Middleware</td>
</tr>
<tr>
<td>20.9</td>
<td>Middleware Concept</td>
</tr>
<tr>
<td>20.10</td>
<td>OMG Object Management Architecture</td>
</tr>
<tr>
<td>20.11</td>
<td>CORBA 1 and CORBA 2 Compared</td>
</tr>
<tr>
<td>20.12</td>
<td>Distributed Object Standards Evolution</td>
</tr>
<tr>
<td>20.13</td>
<td>Java Environment in Internet Context</td>
</tr>
<tr>
<td>20.14</td>
<td>Java Applets Provide Internet Platform</td>
</tr>
<tr>
<td>20.15</td>
<td>Java Portability allows creation of Orbletts</td>
</tr>
<tr>
<td>20.16</td>
<td>Internet as a Client Server Platform with Java ORB</td>
</tr>
<tr>
<td>20.17</td>
<td>EJB Container Options</td>
</tr>
<tr>
<td>21.1</td>
<td>Identifying good wrappering candidates</td>
</tr>
<tr>
<td>21.2</td>
<td>Wrapper Turns a Legacy Application into a Component</td>
</tr>
<tr>
<td>21.3</td>
<td>IDL Provides a Common Interface Definition Language</td>
</tr>
<tr>
<td>22.1</td>
<td>Improvements in software development due to tools</td>
</tr>
<tr>
<td>22.2</td>
<td>Architecture of Visual Assembly Tool</td>
</tr>
<tr>
<td>22.3</td>
<td>VisualAge Visual Construction Editor</td>
</tr>
<tr>
<td>22.4</td>
<td>Sample Application User Interface</td>
</tr>
<tr>
<td>22.5</td>
<td>FP per staff month (vert) vs Language Level (horiz)</td>
</tr>
<tr>
<td>24.1</td>
<td>Model, View, Controller Design Pattern</td>
</tr>
<tr>
<td>24.2</td>
<td>Controller or Business Logic Layer Marshalls Services of Business Objects</td>
</tr>
<tr>
<td>24.3</td>
<td>Extended MVC Architecture</td>
</tr>
<tr>
<td>24.4</td>
<td>Mapping Models to the Architecture</td>
</tr>
<tr>
<td>24.5</td>
<td>Domain Objects in Class Diagram</td>
</tr>
<tr>
<td>24.6</td>
<td>Event Model for Sale Process</td>
</tr>
<tr>
<td>24.7</td>
<td>Domain Class Responsibilities</td>
</tr>
<tr>
<td>24.8</td>
<td>Controller Logic</td>
</tr>
</tbody>
</table>
Chapter 1

Objectives of Systems Delivery

Systems Delivery vs Systems Development

Conventionally, we have talked about system development - the analysis, design, programming and testing of application software from scratch either on behalf of a paying client, or to deliver a generic product for sale to users with similar needs. We prefer the term system delivery, which is about delivering the same result, but not necessarily from scratch: There are important differences:

- We may take advantage of existing complete systems (packages or industry solutions) which either form part of our final product or are modified to become the required system
- We may need to incorporate existing applications already in place in the organization ("legacy" or, more kindly, "heritage" systems)
- We certainly want to save ourselves work by using
  - Predefined industry designs in the form of class libraries
  - Pre built components developed on previous projects or acquired from specialist suppliers
Reusable "patterns" which are recognised algorithms, approaches, frameworks or structures which give us a generic solution to similar problems, or the starting point for our own extended solution

Standards which allow us to mix and match components from different sources with minimal interfacing work

We may deliver an application by simply providing an integration layer on top of a variety of existing facilities, for example:

Using a word processor, spreadsheet and business graphics program together, under control of our own graphical user interface and exploiting Object Linking and Embedding (OLE) to control the applications to provide a system which uses figures from the spreadsheet, integrates these into tables and graphs which are dropped into a presentation structure, and then embeds slides from the presentation in a document

Putting a work flow layer on top of systems which capture an order, issue a product, bill a customer and schedule delivery of goods

Customising an ERP package from a vendor

Delivery is thus about delivering an effective solution rapidly and cost effectively. When we talk about the cost, we are concerned not just with the development or implementation cost, but the lifetime cost - or cost of ownership over the useful life of the solution. To achieve low lifetime costs we need to meet a variety of other criteria (sometimes referred to as nonfunctional requirements).

Nonfunctional Requirements

The application must be easy to use so that training and support costs are kept low

The application must be efficient in its use of resources so that machine, network and supplies costs are controlled

The system must be flexible and easy to modify so that costs of accommodating new requirements and changing conditions are curtailed

Development costs should not be excessive - implying that we should take maximum advantage of what we already have as well as work that others have done and that effort expended in development should produce results quickly

The system should be easy to integrate with future applications, requiring that we adhere to standards and provide an open solution
• The system must be **accurate and reliable** so that we can depend upon it and not suffer any data or other consequential business losses

• System and data should be **secure** to prevent unauthorised access or tampering

• System must be **legal** and **ethical**

There are other requirements which may be present in some circumstances:

• We may need the application to be **portable** to allow its deployment in multiple environments (e.g. When we are providing a solution to clients where we cannot dictate the platform in use)

• Increasingly, we want to create solutions which are **open**, which means that they should adhere to standards and expose their capabilities through a published Applications Programming Interface (API)

• It is often an advantage to develop applications so that they are **scalable**, meaning that they can be deployed in a small, relatively inexpensive environment to begin with, and moved with minimal change to larger, possibly multi-tier and multi-processor environments as demands increase

Meeting the functional and non-functional requirements together, as well as the business constraints on cost and deadline, can pose significant challenges. If we meet some without others, we will not be successful. If we build a system with wonderful features, but it runs so slowly it is unusable, we have failed. If we build something ultra-reliable which produces incorrect results, we have failed. If we deliver something which meets the functional and nonfunctional requirements, but too late and over budget, we have failed. To quote Tom Gilb, we need to fix all holes in the boat simultaneously, otherwise we sink!
Systems Delivery Problems and Approaches

In this chapter we’ll review the situation in typical system development projects. It is not a rosy picture - essentially, we want to highlight problems that occur and try to identify the sources of these problems so that we can avoid them in our own efforts.

Understanding the Business Problem

At all times, we must realise that we are running the project and building the solution to address a business opportunity or problem. The project exists and is funded to achieve this purpose. If we did not have a business need or goal and a suitable sponsor, we would not have a project, and maybe not a job either. So, we need to keep the customer and their needs firmly in mind.

Further to the above, we need to keep an eye on the broader picture. Modern organizations face a very competitive environment and must be nimble and customer focussed in order to survive and prosper. If we keep our heads down too much and ignore what is happening around us, we risk having our project cancelled when priorities change, or missing the target in terms of the
requirements for the system. Be warned, these can change significantly during the project - the longer the project, the greater the risk.

When we tackle leading edge challenges, we may enter territory that is very nerve racking for traditional technical people. We might like to have all the details specified up front, all the i's dotted and t's crossed. That way we know how to go through a structured lifecycle (requirements, analysis, design, build, test, deliver) and produce a result. Unfortunately, the real world will not allow that luxury any longer. We are almost guaranteed to have some requirements that will only become clear during the project, or even after the first production release. We will probably change at least one of the technologies we use (GUI, language, database, operating system...) during the project. We need to accept this and devise techniques that allow us to "surf" these waves, not get dumped by them.

Sometimes we will need to use some unfamiliar and "fuzzy" techniques to get to creative solutions. These may involve brainstorming, mind mapping or "out of box" thinking. Once we have generated ideas, we can try out their viability by rapid prototyping. More on this topic later.

We should be ready to accept ideas from any sources, then submit them to critical review. So, look in the magazines, trade journals (both I.S. and in the domain of the client), examine competitors systems (as far as you can legally and ethically). There is nothing to stop you opening an account with a competing bank, for example, and checking out how their systems work. Packages can also be a rich source of ideas. Be careful not to try and make your system a cornucopia of everything you like from everything you see, though, this will tend to inflate scope to unmanageable proportions and dilute a clear philosophy and consistency through the system.

We should clearly identify exactly what we really want the system to do, and for who. The roadside is littered with the abandoned corpses of systems which the technical staff thought were great (and may indeed have been wonderful designs) which nevertheless did not satisfy basic user requirements. Often, what the sponsor, business and user community really needs and wants is much simpler than what we expect. Get the goals clear. To paraphrase Tom Gilb again: It is hard to hit an unknown target, except by accident!
Gaining and Maintaining User Buy-in

Educate the Users

It is our job as systems professionals to educate our customers, i.e. The user community and sponsor. We need to share with them our understanding of what computers and systems do well (usually repetitive routine tasks requiring efficiency and accuracy) and what they do less well (usually tasks requiring innovation, lateral thinking, judgement etc. Which are not routine). Some things will appear simple to users, but may be very difficult to deliver through systems. Other things may appear complex to users, but actually be very easy to deliver. We need to guide our clients to apply automation in those areas where it will deliver high benefits at reasonable costs.

High Benefits, Low Effort

In one bureau situation, we found a user of a stock control system who spent days each month ploughing through several thousand pages of stock item detail printouts, just to isolate items which were below or near their reorder levels. Each stock item on the master file was printed out in full over about a half a page of computer listing paper. There were some 23 000 stock items. The user used a steel ruler and a craft knife to cut out the items which needed reordering. It was enormously frustrating for him, but that was how he believed computers worked! Of course, a couple of lines of code in the report program the following month did this automatically for him. Deleting information not required for his purposes from the report cut the information down from half a page to one line per item. This is a classic case of a high yield change.

Unnecessary Development

In another situation, we saw a project team spend several months (about 36 staff months) building extremely complex logic and options into a large annuities administration system. These options were only needed for one of the 12 000 clients which the system served, and only then each quarter or once a year. It would have been far more cost effective to exclude these from the system and do this one exceptional case manually!
Focus on High Benefit Features

One way to determine which features to include in a project is to map them into a matrix where the axes are the relative cost and relative benefits of each feature. I stress *relative*, it is not necessary for this comparison to have absolute costs (which require difficult to achieve accurate estimates): likely cost/benefit of one feature relative to another is sufficient.

<table>
<thead>
<tr>
<th>Benefit</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td>Low</td>
</tr>
<tr>
<td>Low</td>
<td>Exclude these features</td>
</tr>
</tbody>
</table>

2.1 - Feature Selection Matrix

Keep User Involvement High

We also need to keep users informed and involved throughout the development process. In traditional approaches it was not uncommon to see involvement levels for user and systems personnel as shown in figure 2.2.
2.2 - Relative Levels of Involvement through the Lifecycle

The systems personnel are intensely busy in the middle of the project, but the long gap between user requirements definition and implementation can lead to problems, where

- The user or business requirements change and are not communicated to the team
- The user personalities themselves change as people are promoted or move into other roles
- Technical staff make inaccurate assumptions which do not reflect reality when they expand the high level requirements to detailed designs

A variety of mechanisms can be used to prevent these problems, including:

- Prototyping to refine the user requirements and sustain involvement beyond the high level requirements
- Joint user/technical teams to keep domain skills on board throughout the project
- Joint Application Development (facilitation) to ensure high user involvement, shorten the project lifecycle and reach higher levels of consensus across user areas
- Iterative project lifecycles, where smaller parts of the full solution are delivered incrementally, none more than a few months apart
- Joint user/IS steering bodies to review progress and quality and to set priorities for application of resources
• **Intelligible deliverables**, preferably in graphic or multimedia form, which can be reviewed by business personnel without difficulty

• **Open communication** channels, participative style and team building beyond the immediate task at hand

**Managing Expectations**

One of the most critical things in delivering systems and having satisfied clients is managing expectations. People do not necessarily have to get the best system ever, but they must at least get what they have been sold, or expected based upon our promises. As systems people, we often envisage what *could possibly be built* (given no constraints on time, skills or budget) and naively promise this to the client. We should, of course, be professional and take all the constraints into account, then only promise the client something *less* than what we realistically can deliver. That way, if we deliver something better than promised, we will have a delighted client. The text box puts this in an everyday context.

---

**The Gronking Noise..**

You take your car to a garage because it has developed a horrible Gronking noise. The mechanic listens and says: “It has a Gronking noise on the left front - we’ll need the car for half a day.” You ask: “How much will it cost?” And are told “About $400”. Well, you can’t live with the Gronking noise, so you drop the car the next morning. In the afternoon you call to pick it up and find that not only is it not ready, but it looks like the car is stripped and distributed around the workshop floor. “What is happening with my car?” you ask. “Ah.” Says the mechanic, “This is a class B Gronking noise, so it will take two days to fix, if we can get the parts”. When you finally get the car back, after several days, trips and much anxiety, you are presented with a bill for $1200. Are you happy? Hardly. Now consider how often systems projects do this to **our** clients... I.e. Delivering what we said (fix the Gronking noise) but very late and way over budget..

You would have been much happier if you had initially been told that it could cost up to $1300 and take a week, and then found that it took 3 days and cost $1200. It may even have turned out to be the lesser class A Gronking noise which is fixable in a day at a cost of $400, in which case you would have been delighted.
**Frequent Delivery**

The best way of keeping users involved is to have frequent deliveries. They do not have to be the final product, or even code which can go into production, but they must be relevant and add value for the user. Best is if we can follow an incremental delivery lifecycle, where we are able to deliver meaningful parts of the application at frequent intervals. We may also follow an iterative development approach, where we take smaller chunks through a specify, design, build, test cycle in a matter of weeks keeping users actively involved in at least the specify and test activities. These chunks may not actually go into production, but may await an integration test and release which constitute an incremental delivery. Even where this is not possible, we can provide interim deliverables to users to verify that our understanding is correct and that what is built accurately reflects business requirements.

**Intelligible deliverables**

The most intelligible deliverable for a user is obviously a working part of the system desired. Next would be a prototype which exhibits the look and feel of the desired solution. Following this, we can also educate users to read and understand, and therefore verify, systems models that we use to help us understand the domain and the requirements. These would include items like Business Domain Object Models (Class diagrams), Business Process Models and Event/Activity models. Throughout this text, we will try to use models which are comprehensive, efficient (only as much as needed) and effective for both systems professionals (developing them) and users (reading, understanding and verifying them).

**Dealing with Complexity**

Systems and technology have become increasingly complex. While this can serve us well by providing sophistication and subtlety in solutions, it can also be the death of a systems project. We need ways to cope with complexity. Fortunately, a variety of techniques for doing this have evolved from different backgrounds over the years. Below, we discuss some of the more widely adopted approaches.

**Break it down - Miller**

In a landmark article several years ago, “The magical number seven, plus or minus two” the psychologist Miller outlined a principle based upon the short term memory structure of the human mind. While our long term memory is essentially limitless, our short term memory is, relatively, very limited. This
means that we can, on average, deal with no more than seven discrete concepts at one time. Beyond that number, and the average person will lose track of something. Gifted individuals may be able to extend this to nine, while those less fortunate may only cope with five, hence the title.

The implication of this principle is that we should restrict analysis and modeling techniques to accommodate these constraints. This may lead us to a requirement to keep a process model to seven or less high level steps, decomposing some of these steps to related models where necessary. The same principle will apply to functional decomposition, where we break a higher level function down into constituent components - there should be no more than 5-6 children below a parent function. The whole model may be much larger, but the focus at any one time should not include more than seven items (e.g. Parent box and six children).

Manage boundaries

Another very important mechanism is the management of boundaries, or interfaces. We can deal with very complex things as “black boxes” without needing to know what goes on inside them. A simple real world example is a calculator. It is filled with complex circuitry that the average person has no concept of - yet even young children can use a calculator effectively due to a simple interface provided by the buttons and screen. Putting a variety of components together, we can concern ourselves with just the connections rather than the content. An example would be connecting up a component hi-fi system comprising a tuner, CD player, amplifier and speakers. We need to know which cable and plug goes where, but we can be unconcerned with the plethora of complex electronics within each of the boxes.

To make the whole job even simpler, we can take advantage of standards. There may be one shape for speaker plugs, one for unamplified signals from the tuner and CD player and a third for the mains voltage supply. Clear interfaces and standards make it possible for even novices to hook things together effortlessly and correctly. Later, we will see how these principles apply to systems and technology in object oriented techniques and distributed object standards.

Hierarchy

One of the most powerful organizing patterns is that of hierarchy. It is something which we understand intuitively and can be used to great effect to see things in relationship to the whole. This principle is used in work breakdown, as part of defining the tasks within a project, as well as in functional decomposition to determine requirements, as seen in the Information Engineering approach.
Architecture

Architecture is about the overall structure or big picture i.e. The relationship of parts to each other and the whole to the environment. We must be intensely aware that our solution must fit into an environment within the technology, the business and the organization. It will probably have to interact with a variety of data sources and other applications. It will also have an internal architecture - which is the structure we chose to use to organize the parts or functions of the application, both at a logical and at a physical level. Using the principles of separation of function, well defined interfaces and standards, architecture helps us to deal with complexity by:

- Keeping track of the roles of each component
- Identifying where components “touch” i.e. Interface or impact on each other
- Specifying what the interfaces look like or which standards they adhere to
- Allowing us to deal with each component in turn, without dealing with the complexity of the whole

Choosing a good architecture will make our applications more robust, as well as easier to adapt to future needs and demands.

Within Inspired, we have developed a set of frameworks which help organizations to manage architectures at the enterprise level as well as between and within applications. These cover four major interrelated areas, viz:

- Business
- Applications
- Information
- Technology

In later parts of the text, we will see extracts from these frameworks used to elaborate upon specific topics, including:

- Business Processes
- Application System Architecture
- Application Programming Interfaces (APIs) to technology, data and other applications
**KISS (Keep It Simple, Stupid!)**

An acronym which reminds us not to get too clever or complex. Einstein is quoted thus: “Things should be only as complex as they need to be and no more”. We should always look for the simplest way to achieve the result. This extends to architecture, design, programming and user interfaces. Simple things are much easier to conceive, to communicate to other team members, to design, to code, to test and to verify as working. We also have a much better chance of understanding them later when we need to extend or modify them. Often, it may take longer to come up with a simple, elegant solution than to adopt the first (often complex) one we think of. This initial investment will pay off handsomely in all the downstream activities.

**Objects**

Object orientation is a fundamental foundation for new generation information technologies and systems. It is also finding a home within analysis beyond IT, extending into business modeling and strategy. The more powerful modeling capabilities in object methods allow us to capture more subtle and accurate specifications, mimicking the behaviour of real world entities and systems. OO includes many concepts which allow us to deal with systems which solve more complex problems reliably. It is the basis of component based development. We will devote a chapter to understanding these core ideas and technology.

**Components, Frameworks, Patterns, Libraries, Packages**

Objects can be delivered as components, adhering to a standard for their interaction with the environment and publishing the services which they offer. This important technology offers radically new ways of rapidly assembling complex, capable systems of pre-built building blocks.

Libraries of object definitions or components allow us to leverage the work of others to enhance our own productivity and quality.

Frameworks are sets of object definitions or components designed to work together to provide a solution. They are normally delivered in libraries.

Patterns are established ways of solving a class of problems, which we can adapt to produce tailored designs.

We will explore all of these concepts in early chapters as well as seeing how they are used through the system delivery lifecycle in later chapters.
Shifting Sands

Change is inevitable

The pace of technological innovation, business change and sociopolitical development is accelerating. The one constant we can rely upon, is change. Why then do we persist in using techniques and methods which are grounded in a belief in stability? We may just be setting ourselves up to fail before we start. An example is to stick rigidly to a waterfall system development lifecycle, where each defined phase proceeds logically to the next, assuming that when we have done analysis, for example, the requirement is completely captured and we can go on to design, code and implement without revisiting any of the analysis or requirements activities again. This simply does not reflect reality. What we need is techniques that recognise the dynamic world in which we live and work and which are competent to deal with a high level of change during specification, development and even after implementation.

People move on too

People move on to new roles as well. This can involve changes in emphasis or requirements. It can also mean a loss of key skills or knowledge. The latter can be particularly damaging, as many products today embody a high knowledge component. We need ways to capture knowledge, to make it explicit and sharable. This can involve powerful models, recording technology and knowledge management tools. We also need agreed standards for the representation of concepts. We will try to adhere to industry standards wherever possible.

Repository necessary

To manage all the diverse information, models and knowledge in a system delivery project is a challenge indeed. This is exacerbated by frequent change, the need to support multiple perspectives and team members concurrently, and the need for all critical items to be in congruence. It is possible to do this manually for small projects and teams, but it quickly becomes impossible as we scale up. We highly recommend the use of an automated repository (possibly forming part of a CASE tool) to support development and to insulate the project and organization from loss of key knowledge.
Shared communications language

Communication consists of transferring concepts, ideas, knowledge or other information from one person or group to another. The difficult part is in ensuring that the receiver’s perceptions come to match those of the sender. I could say to you:

“We need an alarm system to protect the premises”

and you could nod agreement, BUT

I might have in my head a large Rotweiller, while you have an infrared motion sensing system with a radio link! We think we have communicated, but I am going to be very surprised by your implementation. It is extremely important not only to agree the words, or the symbols that we use, but also the semantics, or meaning that we attach to them. These shared meanings need to be documented in the repository we mentioned previously. As far as representation of models is concerned, we should use industry standard symbols and models to enhance the level of communication and consistency. To this end, we will adopt core representations from UML, the Unified Modeling Language, promoted by Rational Corporation and the Object Management Group as a standard for object oriented modeling in the systems field.

Discipline and change management

Systems work is creative. Unfortunately, it also needs to be disciplined. We can all recall war stories of apparently minuscule changes to software or specifications which had disastrous unforeseen consequences. One of my favourites is the exclusion of a checking routine in the “undercarriage up” code for a new bomber aircraft. This omission allowed the undercarriage of one of two prototypes to be retracted while on the ground, wrecking a multi-billion dollar prototype.

We will have change, but it does need to be managed and tracked. We need good tools to do this consistently well on a large scale. We recommend tools that keep track of versions, releases and perform automated builds of associated components using the correct versions of each.

Uncharted Territory

New technology

New technology is fun. It also breaks a lot. If we are using it for mission critical projects or applications, this is not fun. We should use technology
that is appropriate to the level of risk which we can tolerate. This, in turn, relates to the following factors:

- The business or safety criticality of the solution we are required to deliver. Personally, I do not want to be operated on by a robotic probe running beta test software
- The level of skill of the people who will develop (and, even more important, maintain and operate) the solution
- The impact of delays in the delivery of the solution. If I am asked to deliver quickly under pressure, I will use a tool that I am very familiar with (this implies that it has been around for a while and is not the latest release or version). New tools often promise features that would be ideal for our project, but they may actually delay delivery because we have a learning curve to go through and the technology may need to stabilise before it can be reliably used in production

**New business ideas or paradigms**

Technology brings with it new possibilities and business models. We should be open minded but critical. By this I mean the following:

- Do not exclude ideas because they have not been done before - they may now be possible
- Do not adopt new ideas simply because they are new or your competitors are pursuing them - think through the pro’s and con’s and the risks and implications of all available options carefully before committing to a new approach. Many companies we see are pouring millions into e-commerce projects in knee jerk “me too” responses to activity by their competitors. Unfortunately, they often have no idea of a clear business model or goal, much less how they will actually recoup investment and make a profit from these ventures

On the other hand, many new models are possible and desirable. We can remove many layers of inefficiency in organizations, many unnecessary steps in business processes and achieve much higher quality more cheaply and quickly by the creative, critical and competent application of new technology and approaches.

**Prototyping**

Prototyping is the production of working models to try out and refine ideas. It is a powerful technique properly employed, but is not a substitute for the discipline of thorough design and testing. We will devote a later chapter to the advantages and disadvantages, and the appropriate use of prototyping.
Iterative lifecycles

Iterative lifecycles allow us to deal with high levels of change, both in requirements and technology. They give us multiple, frequent learning opportunities. By learning, we can correct our course. They also facilitate user involvement in the specification, development and testing of solutions - a very good thing in ensuring that we meet requirements and produce practical results.

Dealing with Size

Scope management

One of the biggest challenges facing us, is to keep projects from growing out of control. Research done into measuring the anticipated size of systems projects at the budget commitment stage and the delivered size at the end of the project, indicates that there is frequently a three fold increase. Some of this is due to over optimism at the beginning, some to lack of detail in early deliverables and some to so called scope creep, which is the addition of new requirements or features not originally within the scope of the project. This results because it is much easier to add requirements to an existing, running project than it is to justify a separate new one. We have to guard against this if we are to have a hope of delivering within time and budget. One essential is understanding the scope and size to begin with. If we are not clear on this, we will not know when it changes or be able to predict the impact. We will advocate some techniques for quickly getting a handle on scope in such a way that we can monitor creep and prevent it, or at least renegotiate deadlines and resource commitments.

Does it need to be so big?

A fundamental question to ask ourselves is: Does the system need to be as big as we are planning? We should examine what is the minimum that could actually do the job and deliver meaningful results. We should distinguish mandatory features that are essential from those that are cosmetic or desirable. Often, the latter will not be justifiable and should be dropped in favour of doing the essential for a different set of business requirements.

Do we need to do it all?

Some requirements may prove to be completely spurious. We have seen situations where users request things because they have been told that computers can do them or because they believe that that is what the analysts
want. We should go to the fundamentals and establish *why* things are needed - what is the essential business purpose, and then try to satisfy that in the most direct and simple manner. For example, we may be asked to produce a comprehensive and complex sales analysis report. We could accede to this request, spend several person months programming it and have it run routinely in production. It would probably require several days worth of user time per month to use the output. If we probe deeper, we may find that what is really required is an alert of which major customers are reducing their business with us, so that sales attention can be focussed to improve customer relationships. This could be delivered easily with a bit of analysis on the database and the ad-hoc sending of an e-mail to the sales director. We may save a lot of development effort, and the client could save a lot of manual report reading.

A further danger is that some clients will try to make the system under development do everything for them. This is just human nature. If someone is promising to take work off our hands, we will happily transfer as much as possible! As analysts, we are often only too happy to please in an attempt to demonstrate how competent we and the technology are. In truth, we may be sabotaging the project by growing the scope as well as taking richness away from the job of the client.

**Teams**

Teams are a mixed blessing. An individual working alone can often be much more productive, providing they have all the required skills and the task is not too big to achieve in the required time frame. On the other hand, there are few individuals with all the required business, technical, project management and people skills to pull it off on their own. The organization can also be at risk if the one super competent individual disappears. In most cases we will need to resort to teams. As the number of people grows, so do the communication and coordination overheads. The bottom line is that we should have a small team of competent individuals with the required complement of skills, shared goals and clear responsibilities.

**Build a bit that works**

Grady Booch and many others make the point that no successful complex system was built from scratch in one single attempt. We will always find that successful complex systems evolved over several iterations and releases from simpler ones that worked. The message is clear: We should break up large problems into smaller manageable ones. We should identify a core of useful functions and build them, test them and take them into production, then refine them. Only then do we have a foundation and set of skills to add
major new features. We will see this theme recurring in our discussion of lifecycles and deliverables.

**References and Further Reading**

DeMarco, Tom, 1982, Controlling Software Projects, Yourdon  
DeMarco, Tom & Lister, Timothy, 1987, Peopleware: Productive Projects and Teams, Dorset House  
Gilb, Tom, 1988, Principles of Software Engineering Management, Addison-Wesley  
McLeod, Graham & Smith, Derek, 1996, Managing Information Technology Projects, International Thomson Press  
Taylor, David, 1995, Business Engineering with Object Technology, John Wiley
The *Inspired* Method

**Introduction and scope**

This chapter is a brief overview of the *Inspired* system delivery method. It aims to provide a high level view of the major concepts and their interrelationship. The scope of this discussion is limited to the activities which would normally form part of a system delivery project. We are assuming that a range of strategic activities have preceded this stage, including the definition of the business objectives, prioritization of projects, and initial definition of scope. In some cases, there will be a need to perform extensive enterprise modeling and business engineering (or process improvement) before the computer systems stage of the project is reached.

This overview does not aim to be comprehensive and there are many subtleties within individual steps and processes not addressed here. Treatment of these will be left for further detailed chapters. Figure 3.1 shows the major processes and steps in the method. This gives the impression that the method is linear, whereas it is, in fact, iterative. You should read the diagram as a series of iterations, which occur a limited number of times until a given deliverable has reached a certain *state* or level of completion, at which point the ensuing activity will commence. Even then, it is entirely possible to loop back to perform earlier analyses in more detail or to clarify issues which were not completely resolved in earlier operations. The idea is that by the end of each “phase” the deliverables are at a certain minimum level of completion.
Various deliverables and models will be created at a high level and then expanded in terms of detail as the lifecycle progresses. There are no major discontinuities: users should experience a smooth increase in the level of detail, accuracy and fidelity of the models as analysis progresses.

3.1 - Inspired Method Overview

The first step involves the definition of the project and the scoping of the business activity of interest. Project Definition is performed in accordance with the guidelines in the book *Managing Information Technology Projects* (McLeod and Smith, Course Technologies (International Thomson Press), 1996). We discuss scoping the business activity briefly in this method overview and in more depth in a later chapter. Output from this step includes a *project definition* and a *context diagram*. The project should also be managed in accordance with other guidelines in the McLeod/Smith text related to estimating, risk assessment and control, change management and quality assurance.

The following step involves the definition of the initial *business domain object model* detailing the entities in the business or organizational domain. This model will be progressively expanded later to include full attributes of a complete object model for the entire application. At this stage, it will include
only the business objects, their relationships and their attributes. Behaviours are deferred until later stages. Examples of business objects would be Customers, Products, Orders etc.

Enterprise and business process modeling can commence in parallel to the business object modeling. This involves examining and modeling the key business processes within the scope of the project. Analysis will include functions which are currently computerised, as well as those which are not. It seeks to build a complete picture of how key business events in the environment are handled by the organization. In the process, we often examine the “value chain” or "value network" Involved - i.e. how the various activities add value in transforming the inputs from the environment into desirable outputs from the organization. Examples of business processes include the processing of a sale in a retail environment, or the processing of a policy application in an insurance company. The outputs from this step involve a stakeholder model and a collection of business process models.

Should it be felt that the current business processes are ineffectual or that they do not take sufficient advantage of opportunities provided by technology, then clients may opt to undertake a redesign of the business processes. The techniques employed allow easy assessment of a variety of competing scenarios. Other enhancements also encourage the building of business processes which are self-optimizing once implemented.

3.2 - Increasing Fidelity with Progress Through the Lifecycle

During the preceding two steps (Business object modeling and business process modeling), a start is made on the building of a dictionary or
repository containing details of objects, their attributes and the characteristics of these attributes. For example, we may know about a Customer object that has a telephone number attribute which is an integer type. As we discover this type of information it will be captured and maintained in the *dictionary/repository*. The repository will be updated in all subsequent steps, gradually becoming more comprehensive, detailed and accurate.

Once we have a business object model and the business process models, we can commence prototyping and event modeling. Prototyping involves building representative interfaces to the system, including screens and reports, as well as considering the flow of the system under various *scenarios*. It allows users to see what the system will look like and how it will behave without the full investment in development of the business logic and production system controls. The prototype should clarify requirements, validate the system “look and feel” and capture more detail in the object attributes for inclusion in the repository. We will often only prototype ‘kinds of things’, not every instance - for example, one domain object maintenance function, one report, one interactive query etc. We can also make use of existing or newly defined patterns to rapidly create consistent functional interfaces which are standards conformant.

A further use of prototyping is to determine the feasibility, complexity or best approach to follow when using new technology. This can reduce risk and save effort in later stages of the project.

*Event (or in UML terms, activity) models* are built which are similar in concept to the business process models, but now concentrating on the computerised aspects of the proposed system and linking these tightly to the business object model. Events are recorded which show the changes in state of underlying objects. For example, a withdrawal operation may affect a customer account object, causing its state to become overdrawn. These models help to express the exact behaviour required of the new system, and to verify the ability of the business object model to support these requirements. During the process, the object model will normally be extended and enriched. Outputs from this stage include the *prototypes* and *event models*. During this stage, we may also capture various business policies and algorithms in the form of *business rules*, which can be attached to the other models.

Once requirements are well understood, we can move to the design mapping task. This involves the apportioning of responsibilities for the various behaviours required (as identified in the dynamic event models) to the classes in the business object model. Also, we will translate the overall logic of the event model and associated rules to reside in a layer of the software
application, generally termed the controller or business logic layer. This separation allows business logic to be expressed at a higher level, making for easier maintenance in future, as well as facilitating distribution of function (e.g. in a client server environment).

User interface requirements (as expressed in the prototype) are mapped to a third layer - the user interface layer (sometimes also called the view). Keeping these in a separate layer allows flexibility in having multiple interface styles for a single application e.g. for in-house use via a GUI such as Windows, or for use via an Internet browser. The design step can also involve creating interfaces to existing computer systems or data files. Outputs from this stage include a three layer application model which can be expressed in an expanded object model or using UML package diagrams. This will include the business objects (now with behaviours as well as attributes), objects in the business logic layer which manage transactions and business events, commensurate with the business rules, and objects which create the user interface.

This design can be directly implemented in object oriented tools such as Smalltalk, Java or C++ coupled with an object oriented database. Where the data will reside in a non-object database (e.g. Oracle or Sybase relational systems), business object structures must be mapped to relational tables. This involves additional work and can impact performance of the resulting system negatively. Designers can perform this activity themselves, but since it is nontrivial and fairly generic, they could also make use of a growing number of frameworks available from vendors to accomplish the mapping effectively. Examples include TopLink™ and IBM’s Object Extender™.

The design model is implemented by writing code as well as customizing and assembling components sourced from outside the project. These could include various interface widgets, such as Microsoft ActiveX Controls™ or Java Beans™; or business level objects such as Enterprise Java Beans™. This is followed with comprehensive testing.

Documentation of the system (in hard copy or electronic form) is built incrementally during development and is finalised during testing.

**Project Definition and Scoping**

The objective of this step is to properly define the project including the following aspects:

- Title
- Sponsor (who is paying for it and will derive main benefits)
• Overall goal
• Objectives
• Relative priorities of quality, schedule and cost
• Assumptions
• Terms of reference
• Business deadline
• Proposed budget
• Related Projects
• Any legal, moral or ethical issues

The Mcleod/Smith text mentioned previously provides a Project Definition Form to assist with the above, as well as providing guidance in the respective items and their completion.

A *context diagram* is usually prepared. This gives us a concise way to express the relationship of the proposed business activity to its environment. It includes the proposed business activity as a single central box, surrounded by other symbols representing parties and things with which it interacts. These may be various stakeholders, people, organizational units, or other systems. They can be internal to or external to the organization. Connecting the proposed business activity and parties are arrows indicating the movement of data/documents and other artifacts into and out of the business activity. These are labeled to indicate what information or item is moving. A boundary is often drawn to show which things are considered to be under the control of the business activity designers, and which are not. A node on a flow line outside the boundary would indicate a format which the team would be obliged to adhere to, while one inside the boundary would be one that the team has the discretion to design. Figure 3.2 shows a simplified context diagram for theatre reservations.

We can use symbols familiar to the user community, as is done in presentations or the rich pictures technique of soft systems modeling. If we are using CASE tools, we may restrict ourselves to symbols available, such as UML Actors from Use Case Diagramming.

The context diagram is a valuable starting point for analysis, and is also useful to perform quantitative estimating of the likely effort required for the project. This can be done quantitatively at later stages of the project, using Function Points, for example.
3.3 - *Sample Context Diagram*

Where the project will result in changes to, or requires the installation of, new technology infrastructure, we will also prepare a *technical environment model*, which depicts graphically what the various components are in the technical environment, including hardware platforms (e.g. PC or mainframe), software (e.g. Operating system, network control, database) and how the various components are interconnected (communications network and protocols). The diagram will normally also indicate numbers of relevant equipment types and the physical geographic distribution of the components. These models are also described in the McLeod/Smith text. Sophisticated organizations may also be using an architecture based approach to the management of their business, applications, information and technology infrastructure. *Inspired* supports such planning, management and evolution by providing comprehensive architecture frameworks and tools which address these issues. These will be the subject of a forthcoming text on Business Advantage through Architectures. In the interim, consult the web site at: www.inspired.org for further details.

**Initial Business Domain Modeling**

This is done to gain an understanding of the objects inherent in the business domain, without regard for functionality required. Business objects provide a stable base for the creation of flexible systems. We use a technique derived from the Unified Modeling Language (UML) devised by the “three amigos”
- Rumbaugh, Booch and Jacobson now all at Rational Corporation. UML has been adopted by the Object Management Group (OMG) as a standard. We include some extensions based on advanced work by James Odell. The technique is easy for commercial developers with an information engineering background to grasp and we have found that it scales well in practice. It is also supported by a large number of CASE tools.

3.4 - Sample Business Object Domain Model

The object model comprises rectangles representing objects or classes of things. These are interconnected by relationships. Three types of relationship are shown:

- **Inheritance**, where one type is derived from another e.g. Salesperson is derived from Employee

- **Containment** or embedding e.g. where the Employee object contains an Image (which could be a digital photograph of the person)

- **Associations**, where one object simply knows about another one. e.g. Employee knows which Office he/she works in

Relationships are normally named to make the model more meaningful, and to distinguish between relationships occurring between the same two objects. The last two types of relationship will also have multiplicities or ratios. These are indicated by the notation at the end of the line. For example, a City has 0 or more Offices. It is possible for object types to be related to themselves. This normally occurs where we are building a hierarchy: as in a
reporting structure within an organization, or the bill of materials which makes up a product.

Relationships are inherited along with attributes and behaviour. Subclasses can participate in relationships in addition to those inherited from the parent. Generally, circular relationships should be avoided. The modeling technique will allow you to express them, but most software environments (particularly databases) will have problems with them, and they are very rare in the real world, so you should check that it is actually an accurate model if you have one.

**Business Process Modeling and Reengineering**

We normally start this activity by drawing a single stakeholder model. This depicts the enterprise or business unit under examination from the perspective of all the agents in the environment that have an interest in the continued functioning and well-being of the unit. These normally include customers, staff, shareholders, and suppliers. Next we identify the value chains or networks which link the resources provided to the enterprise with the value derived as output by the various parties. We examine the way in which inputs are transformed to outputs and, in particular, how the value generated by the enterprise is added. This stage uses techniques popularised by Michael Porter.

Each discrete business event can become the focus of one or more business process models. These detail the processes which occur along the value chain (both value adding and non-value adding) before the output is generated. We distinguish in the model between three types of processes:

- **Manual processes** which are not accomplished through any automation. The receipt of a telephone call by a clerk could be an example
- **Computer supported processes**, where there is computerised support, but human intervention is required. An example could be the logging of a support call into a tracking system by an operator
- **Fully automated processes**, where a computer system operates without intervention. An example would be the batch printing of statements at the end of the month, using information already present on the client accounts

Processes are linked to show their dependencies. They may occur in parallel if this is possible in the business. Boundaries can be used to show where activity is physically taking place e.g. at head office or a branch. Groupings could also be logical, showing departments within an enterprise. Resources,
volumes and timings may be added to the process models to gain an appreciation of the processing demands and performance requirements of technology.

3.5 - Business Process Model

Where existing processes are deemed inadequate for whatever reason (inefficiency, ineffectiveness, poor quality, cost etc.) we may embark upon reengineering. This examines the outputs required and the processes to derive new, innovative, creative and often technology-supported ways to achieve these outputs more effectively, more efficiently, or with higher quality. In making radical process design changes, we can draw on the work done by Michael Hammer and later adherents.

We should be cautious, however, since reengineering involves considerable risk for the organization and stress for those affected. Great care must be taken to involve all participants, to thoroughly examine existing processes to understand the real problems, and to examine alternatives to the extent that we can prove that they are superior and obtain the commitment of all parties involved to change to the new way of doing things. Change cannot be done too quickly or too often without serious consequences for morale, productivity and core organizational competencies. Reengineering should therefore be an infrequent occurrence. Where it is done, we encourage the use of simulation and prototypes/pilots to prove viability before widespread deployment.

We can gain much from a less radical approach whereby we instrument processes so that we can measure their operation and pinpoint problem areas. In this way we can make small incremental improvements over time, which
can cumulatively yield large results. This is much less traumatic for the organization. Our approach encourages building this capability into all processes which are automated in the project and into any processes which are reengineered. The implementation involves the use of techniques of statistical process control and the Japanese *kaizen* (continuous improvement) philosophy.

**Prototyping and Event Modeling**

Prototyping is a technique where we “mock up” the system to see what it will look like before we undertake the real construction. Like an architect’s paper model of a building, it is something which allows you to validate requirements and design choices. It assists us in refining functional requirements, in generating a good user interface, and in discovering more attributes of objects for inclusion in the object model. It can also lead to the discovery of business rules which can be recorded for later implementation.

Like the architect’s model, however, the prototype does not have the strength or plumbing of a real system. There is little of the validation, recovery, archival, help facilities etc. which a real system will require. Users must be educated to understand that the prototype is an opportunity to explore requirements and design options at relatively low cost before major resources are committed to full scale construction. They should also understand that the prototype is not a real system and that there will be a delay between seeing this and having the real system available. They should understand that changes after the system is built for real will be much more difficult and costly than during the prototyping stage.

Concurrent with prototyping, we build event models for key processes in the system. Processes of a generic nature (e.g. the maintenance of tables (add, edit, delete, query, print)) can be designed just once and reused for all similar requirements. *Pattern libraries* can assist us in finding many generic solutions. The event models show the triggering external operations (which can be transactions or time related), the sequence of operations which occur within the system to respond to these stimuli, and the outputs generated. In addition, events are recorded where the operations change the state of underlying objects. This explicitly links the event model to the previously generated object model. In this way the object model is verified and, where necessary, extended.
3.6 - Prototyping

The look and feel of the event models is very similar to the higher level business models, but with more rigour. It is well suited to implementation via event driven software, the three tier design architecture and/or work flow packages. Processes can be designed so that operations can execute in parallel and that asynchronous operations can occur (the initiator does not have to wait for the requested activity to finish before continuing). Event models can be enhanced with rules which embody business policy or important algorithms. Where necessary a narrative specification or reference can be recorded for an operation box.

3.7 - Sample Event Model
Design Mapping

Layered Architecture

In mapping to the technology available, we will normally seek to implement a model with at least three layers:

• The **business object layer (model)** which reflects as accurate as possible a model of the objects found in the problem domain. These objects should, as far as possible, have little awareness of each other. This increases the reliability of the implementation and the flexibility in the face of change. Typical objects in this layer will include Customers, Products, Employees and the like. This layer is normally made persistent via storage in a database.

• The **business process or logic layer (controller)** which is responsible for coordinating activity between the user interface(s) and the underlying model. This will normally have two sub-levels. The first deals with the manipulation of the interface itself for actions which have no lasting impact (e.g. shrinking or expanding a window on a GUI screen). The second deals with the logic to handle a business event (e.g. handling a client payment once the data is validated and the user clicks the OK button). Logic in the latter layer may be invoked from multiple points in the technical layer. This allows different user interface elements to drive the same business logic. For example, we could have a menu selection, a tool bar icon and a text command trigger the same processing.

• The **view layer** consists of objects implementing the user interface. It normally comprises predefined components such as windows, sliders, buttons, menus and the like. These are normally obtained as components which are then dropped onto a canvas to paint the desired interface. They may be customized in terms of size, font, colour etc. An alternate view may provide a text/command interface or a batch interface to the same underlying business logic.

We may need additional layers depending upon the technical environment. For example, the business logic may need to be split between a local and a remote machine or we may need a layer which wraps a legacy database or system and makes this accessible to the model layer.
3.8 - Layered Application Architecture

Object Technology

The cleanest mapping is to pure object technology such as represented by a language such as Smalltalk, Eiffel or Java in combination with an object database such as Gemstone™, Versant™ or Object Store™. In this case the mapping proceeds as follows:

- Prototypes are evolved into the view layer, making use of available components and frameworks from vendors.

- Rules and the high-level logic from the event models are implemented in the business logic (controller) layer. This consists mainly of obtaining activity from the view and sending messages to appropriate objects in the model then passing results back to the view, managing transaction integrity and security as appropriate.

- The business object portion of the object model is implemented in the business object (model) layer. This normally contains the persistent objects which will remain accessible over time and which will be sharable between users of the operational system and between systems. This persistence is best achieved through an object database management system.
Relational Technology

Where the mapping is to relational technology, it is common to have the business object layer stored in the relational environment, while still making use of objects in the client software (i.e. in the business logic and view layer). Depending upon the capabilities of the tools available, we may have only a view which is object oriented, with a procedural language behind it.

The best strategy for these environments is to convert the object model to the closest relational equivalent before doing detailed design work. This will allow the use of components which understand the relational model (e.g. PowerBuilder™ data windows or Delphi™ Master/detail controls). Users should be aware that there are considerable compromises in implementing the object model on relational technology. The accuracy of the simulation declines and performance can be problematic for complex data structures. Where the data is simple and well understood, the overhead is tolerable. Middleware products such as TopLink™ and UniSQL™ can provide a fairly object view of relational data. Frameworks such as the Object Extender™, part of the IBM Visualage™ family, provide a fairly seamless and competent mapping between object processing and relational storage.

Where it is planned to migrate to object database, the implementation should aim to provide a consistent and stable messaging interface to the data, which may currently reside in relational or conventional technology, but which will in future move into an object database. This will allow legacy components to be selectively and easily “unplugged” and replaced over time without disrupting the applications which use them. A good degree of encapsulation can be achieved in relational environments by the use of stored procedures, but the languages provided by the various vendors are fairly proprietary, limiting portability. An emerging solution to this problem is the growing support for stored procedures written in Java, making them portable.

Business Rules

Starting early in the lifecycle, and continuing until well into design, we may uncover and need to document a variety of business rules. We have a flexible approach to these, considering a rule to be anything which we need over and above the graphic models to fully define the system requirements. Examples of rules include:

- How certain decisions are reached based upon business policy e.g. When to order stock, who should receive discount - these are normally expressed as IF [condition] THEN [action] style rules
How values are calculated or derived e.g. How to raise interest on an outstanding loan. These may be expressed as a formula or algorithm.

References to recognised procedures, laws or standards e.g. The prescribed method for calculating a VAT submission, or determining a medical aid deduction.

Integrity rules which guard the accuracy and integrity of our domain knowledge e.g. Ensuring that an item can only be ordered when there is a valid product code and a recognised supplier, or that dependents information may only be captured where there is a valid employee.

Rules can be expressed in a near English pseudocode, making use of domain class names and attribute names to enhance clarity. Alternately, we can use a formal language such as Object Constraint Language (OCL) sanctioned as an addendum to UML. In either case, the rules are cross referenced from the models and should be stored in the repository or implementation rule base.

Repository and CASE

Where the applications become complex (most places!) and there are multiple project teams, it can become problematic to keep models up to date. We should, as a minimum, keep the data dictionary or repository in an electronic form which allows easy updating and flexible query and Reporting. Various computer aided systems engineering (CASE) tools are available to support the modeling activity. These range from fairly simplistic tools which are essentially diagram editors and do not understand the models which they hold and will not integrate changes across models or views, to very sophisticated tools which may integrate multiple perspectives, support multiple users and projects, manage changes and generate production quality database schemas and code.

Really high-end tools will even support partitioning across a variety of platforms and the deployment of the production application. Key requirements for a suitable tool would include the following:

- It should support the modeling approach chosen
- It should have a comprehensive and integrated repository
- It should provide consistency across multiple model types and views
- It should be open - the repository should be in a form which is intelligible to other products, or provide export facilities.
3.9 - Repository Contents

- It should ideally integrate with the development tools providing
  - Database schema generation
  - Code generation from business rules
  - Prototyping (where the downstream tools do not provide this)
  - Round trip engineering - the ability to alter the generated code and feed these changes back into the CASE models

With CASE, you tend to get what you pay for. Some tools are cheap but are frustrating in their lack of facilities, while others are comprehensive and capable, but prohibitively expensive. We also caution that CASE in itself does not lead to high productivity and quality, unless it is accompanied by suitable investment, commitment, training, skill and discipline in use.

Hypermedia and Multimedia Specifications

Just as new technology makes possible new business models and innovative ways of working for our client organizations, it also affects what we can do in crafting systems. A recent innovation is the use of hypermedia and multimedia specifications. Within our method, we encourage the rich interlinking of models via hyperlinks. This can, with standard Internet
technologies, now be accomplished across tools and distances. It is quite feasible to have a prototype screen in CASE Tool A in Johannesburg, which has a hyperlink from a field on the screen to an object model residing in a repository in New York. There are some serious integrity management challenges, of course!

Often, in the past, subtle requirements expressed by users in interviews with analysts were lost in the translation to models passed on to designers and developers, only to re-emerge at the user testing or implementation stage, with attendant high cost. A technique which can overcome this is to videotape interviews with key users during requirements definition (with their permission, of course) and store these videos digitally, hyperlinked to the relevant models. This provides a unique opportunity for a programmer, for example, to replay the specification process and clarify concepts. It has been used to great success in the development of new air traffic control systems in Europe.

Facilitation

At many points in the lifecycle, we will make use of facilitation (otherwise known as Joint Application Development (JAD)). This is a technique of holding intensive facilitated sessions with all relevant parties present to achieve the following aims:

- Reach consensus on important decisions, models and approaches
- Increase ownership of the results
- Improve accuracy of requirements and efficacy of design
- Compress the lifecycle in terms of calendar time

Facilitation is suitable for project scoping, object modeling, business process modeling, prototyping and event modeling activities. It can also be used for resolving design questions and planning testing and implementation. JAD will be covered in a later chapter.

Existing Systems

We seldom have a “green field” situation with no previous systems or data to worry about. More often than not, we will need to interface to older systems capabilities and access data created by such systems. The preferred technique to deal with these is to create a “wrapper” which provides an object oriented messaging interface on the outside, while communicating
with the legacy system or accessing the legacy data in its own preferred way on the inside. The actual interface to the product can include calls, passing parameters, executing programs or jobs, simulating screen transactions etc.

The creation of such wrappers is not trivial, but it can make functionality available which would be difficult or expensive to re-implement. Again, if the interface is maintained consistently, the problematic components can be gradually phased out and replaced without impact on the newer applications.

**Workflow**

Workflow can be both a concept and a software product. In the case of the former it is a concept whereby work flowing through an organization can be understood, scheduled, monitored and made more efficient. As a software utility, workflow packages provide facilities to control the routing of transactions, movement of files and data between processes, and the invocation of the necessary services to perform the next required step in the overall process. Sophisticated products also provide extensive capabilities for gathering statistics, analysing these and balancing workload across resources.

Using the design model which we described earlier, it is fairly easy to use a workflow engine to implement the high level logic of a business process or event model. Some packages will also allow us to leverage legacy technology, treating it as a step within the overall process.

**Internet Deployment**

The Internet (and use of Internet technology on private networks, an *intranet*) offers great opportunities but also challenges. The design mapping discussed above can be easily used to target these delivery platforms. The three layer architecture maps well to client-server, which the Internet represents. Essentially the browser and forms become the view layer of the application. If the browser supports the loading of applets (e.g. via Java Bean™ or ActiveX™ controls), then some business logic can also execute on the client machine. In theory, data could also reside remotely on the client platform. With the current state of these technologies such data would be very insecure and this should probably not be contemplated except for personal data maintained by the workstation operator.
Components and Frameworks

The method promotes the use of components and frameworks obtained from external vendors and internal sources as follows:

- A suitable framework which implements the three tier design model should be found and acquired or built. This will provide application programmers with a standard way of handling the user interface, communications, layering, handling events, and packaging of the application for delivery.

3.10 - The Big Picture

- Vendor graphical user interface, communications and database libraries should be used. Specialised components for other aspects of the technical infrastructure, such as communications, security or interfacing to special devices, can be added.

- Application class libraries can be incorporated into the business object model at the requirements stage, providing a rapid leg up to comprehensive and viable models.

- Legacy application wrappers for popular products can be obtained from vendors or third parties.

- Application developers should, as far as possible, concentrate on the delivery of excellent user interfaces and rapid implementation or adaptation of business functionality. Most of their work will be with the
view and controller scripts. A second group can engineer components for reuse at a corporate level, or obtain suitable components from outside the organization.

**Quality Assurance**

At each step of the project, there are defined deliverables which should be refined or finalised. The team should adhere to guidelines for the content, format and presentation of these. Good examples should be collected and catalogued so that new staff will be able to see what is expected and what good work looks like. It is beyond the scope of this chapter to detail the quality philosophy here, so we will make this the focus of a later chapter. Additional information can be found in the McLeod/Smith text mentioned earlier.

**References and Further Reading**


Coad P, Yourdon E 1990, Object-Oriented Analysis, Yourdon Press 1990

Colter M A 1984, A Comparative Examination of Systems Analysis Techniques, MIS Quarterly Vol 8 No 1 March 1984 pp 51-66


Graham, Ian, 1993, Object Oriented Methods, Addison Wesley


Infomet, 1988, The Infomet Methodology, Infomet (Pty) Ltd, 1988


42


This page intentionally left blank.
Chapter 4

Scoping and Context

As discussed previously, this is one of the most crucial areas to understand and manage carefully. We need to establish early in the project just what is and is not included in the scope of our work and objectives. We need to understand the context within which our system will operate i.e.

- Who are its users?
- What databases and collections of information will it use or manage?
- What other systems will it communicate with?
- What devices must be supported for interaction with the users and the environment?
- What inputs will be obtained from the environment and what outputs are expected?

In addition to the business context, we also need to have an idea of the technical environment in which we are working. For example: is the system centralised on a large mainframe computer, supporting hundreds of users on a wide area network via text mode terminals; or perhaps intended for deployment as a shrink-wrap application to individual PC users? These considerations will drastically affect the correct answers to many questions we will face along the way.

Generally speaking, everything that is required of our system that is not provided by an existing part of the infrastructure will have to be acquired or built - thus having a direct impact on the scope of the project and hence...
effort, delivery date and cost. Lack of proper understanding of project scope and/or management of the scope as work proceeds is one of the major causes of project failure.

**Understanding the Project Objectives**

The whole area of project choice, project definition and project design is beyond the scope of this text, but it will be useful to have a brief look at some pertinent points. For a more complete treatment see *Managing Information Technology Projects* (Course Technologies, ITP, 1996). As a minimum, we should have some sort of project definition, with at least the kind of information on the form shown below. This may be an actual form, or a document with similar headings. It could be a single sheet for a simple project, or a 5-8 page document for a large expensive project. The important thing is that someone should have asked these questions and obtained agreement from all concerned to the answers.

**PROJECT DEFINITION**

| Company ____________________ | Project Manager _____________ |
| Division ________________ | Originator __________________ |
| Date ____________________ | Sponsor ____________________ |
| Project Title ___________ | Project Code ______________ |
| Project Goal ________________ | |

Priority: Quality __ Cost __ Schedule __ (Rank as 1, 2 or 3)
Terms of Reference ____________

Business Deadline ___________ Budget ____________
Assumptions __________________

Related Projects
Moral/Ethical/Legal Issues

---

4.1 - Project Definition

Some items are self explanatory, others are detailed below:
• The originator is the person starting off the project, or with whom the idea arose

• The sponsor is the person or business area that is paying for the project. They will have to agree budgets, approve deadlines and be kept informed of progress

• The priorities allow us to determine which of the three aspects (Quality, Cost, Schedule) to optimise or trade off when things are tough

• Terms of reference details things like what resources may be used, what technical environment the project must operate in, policies that must be adhered to and other important terms under which we are sanctioned to proceed

• Business deadline is the date by which the capabilities of the system must be available in production use

• Budget is normally quoted in staff time units (e.g. person days) to avoid effects of inflation or currency

The System Scope

This is largely a matter of understanding the things in the environment that will interact with our system, the inputs that they will provide or requests that they will make, and the outputs which our system is expected to produce. This information is concisely captured in a context diagram. This shows:

• The proposed system at the centre as a rounded corner box (consistent with the notation for a process: for now we are viewing our system as one "black box" process)

• Around the system are the things we will interact with:
  • External Entities, shown as a square, represent users (which may be people, devices or organizational units)
  • Other systems, shown as rounded corner boxes

• Arrows show information flowing from and to external items. These can be input only (towards the system), output only (away from the system) or input-output (bi-directional). Each arrow should be labelled to indicate what is entering or leaving the system. Multiple arrows can be drawn between an external item and the system
• We draw a dotted line boundary to show the division between the system and the outside world. Data flows crossing this represent *interfaces*.

• On each interface, we draw a small node (circle). If this is drawn inside the dotted line boundary, it indicates that our system will have ownership of the data in that interface. It will be responsible for the design, including the list of elements, the types of these elements and the format and sequence. In other words, we have choice regarding this interface - we will specify it. If the node is drawn outside the boundary, this indicates that we will have to adhere to an externally dictated interface specification. An example would be where we want to send a transaction to a bank network: the bank will dictate what data they need and the format they expect.

The nature of interfaces and ownership dramatically affects system scope. If we own the data, it is likely that we will have to have all capture, validation and maintenance (add, change, delete, enquire, print) functions to work with this information as part of our system. If the data is not owned by our system, chances are we will be providing limited functionality to either generate a predefined transaction, or to accept an incoming request.

A further refinement of the nodes is to code each node with a character indicating the nature of the interface from a technical perspective. E.g. we could use the following letters:

- **M** Passed in memory as a parameter, message or queue item
- **O** Sent or received online over a network via a message
- **H** Hardcopy, e.g. a printed report or an input document that we scan
- **S** Sequential media e.g. a Magnetic Tape
- **R** Random media e.g. Disk or diskette
4.2 - Context Diagram Structure

A sample diagram for a theatre reservation system is shown below:

4.3 - Context Diagram Example

In this case we can see the following:

- Our system will support customers making reservations, and box office clerks. It must respond to requests for bookings, acceptance of bookings, capture credit card details, or record cash receipts.
The system will generate deposits to a bank transfer system and will also pass credit card details to this system. It will generate tickets for customers.

Theatre management will query the system to determine booking levels and may adjust prices to encourage sales.

The system can dictate the format of all interfaces, except the ones to the existing banking system.

**Technical Environment**

To choose a viable solution, we also have to have an idea of the technical environment in which our system will function. We can represent the technical context of our system using a technical environment model. This details the platform(s) on which the system will run, the supporting software, the networks involved (if any) and the user population expected. A typical structure is shown below.

4.4 - Technical Environment Model

The user type identifies sets of users with similar requirements. In the case of our theatre booking system, these might be the box office clerks, the customers and the theatre managers. We are interested in how many of each type we are likely to support as well. This will have an impact on volumes and the scalability that we must build into our system.
Equipment types refer to the types of machines which the users will use to access the system. These could be PCs, text mode terminals, or other special devices, such as a security card reader at a doorway.

Nodes generally are ways of connecting to the system. They could be a LAN server, a communications processor in a Wide Area Network, or an Internet Service Provider (ISP).

4.5 - Technical Environment Model Example

Inside the platform where our application will execute, there are often supporting software products, including:

- Operating system and Graphical User Interface
- Database Management system
- File system
- Communications software
- Middleware, Transaction Processing Monitors (TPMs) and Object Request Brokers (ORBs)
- Co-resident applications
- Containers which host specific object models, for example Enterprise Java Beans (EJB)
We need to identify these, since our software will often have to adhere to specific interfaces to make use of these facilities.
Chapter 5

Object Orientation Concepts

What is Object Orientation?

Object orientation is a different way of looking at systems and data. Over the years we have become used to the separation of processes (represented by systems and programs) and data (represented by databases and files). This is not a natural division, however, and has largely arisen because of the limitations of previous hardware and software technology. We traditionally had enough memory to load the program code, but not enough to load our data in its entirety. This meant that we had to leave the bulk of our data on secondary storage (e.g. disk) and load only small portions of it into real memory to work on at a time.

Typically a block of records would be moved between disk and memory with one input or output operation. This also burdened the programmer with devising the logic to make files accessible (by opening them), reading in the necessary records required, and matching the position on various related files. Unfortunately, this meant that the programmer (and the program) had to know a lot about the implementation of the files and their structure. This is problematic from a maintenance perspective: since programs access and modify the same data, the possibility arises that they do so in different ways, thus corrupting the structure or content for other programs.
With larger real memories, and good virtual memory management techniques which can make storage limitations transparent to a programmer, the separation of code and data is no longer necessary. Figures from James Martin indicate that the amount of real memory available per MIP (Millions of Instructions per Second) of processing power has grown from .1Mb in early computers (60's) to around 20Mb in today's mainframes and PC's.

**Reasons to Use Object Orientation**

**New types of applications**
One of the major factors driving the rapid adoption of object technology, is the desire for new types of applications which new hardware capabilities make possible, but which bring with them complexities that prior approaches could not handle. Examples of these types of applications include:

- Business Simulation
- Client/Server
- Distributed Applications
- Graphical User Interface (GUI)
- Multimedia
- Internet/Intranet
- Complex: CAD, CAM, CASE
- Knowledge-based, AI

**Project and Business Reasons**
There are also many other, more business oriented, reasons for moving in the OO direction. These relate to the following issues:

- Productivity
- Reliability
- Reusability
- Integration
- Maintainability and Flexibility
- Handling Complexity
**What is an Object?**

Good structured, modular programming dictates that we should have small, highly cohesive modules which have minimum interaction with other modules (high cohesion, low coupling). Object orientation takes this concept to the ultimate conclusion by combining the data structures and the related code into a formally defined object. The object contains both the data and the code which will manipulate the data. Thus, instead of a customer file and several programs that may access and update it, we will have a customer object which will contain the customer attributes (data items) as well as methods which will perform the necessary processing of customer data.

![Diagram of conventional vs object structure](image)

**State and Identity**

Objects represent things which have properties (or attributes). The current set of values of the data items within the object represent its state. For example, a person may have attributes age, gender and nationality. These may result in interpreting the person's state as being a "child" or an "adult", "male" or "female", "citizen" or "foreigner". An object representing a bank account may have an attribute balance and another dateOfLastTransaction. The values of these may mean that the object could be in the states "Active", "Overdrawn" or "Dormant".
A structure containing both data and methods which operate on that data

- It has state and identity
- Only methods within the object can manipulate the data within the object

5.2 - What is an Object?

Each object has an identity which is unique within the system (not just within a class). Unlike in relational systems, this "key" does not have to be present in the object's attributes.

Business Objects and Technical Objects (Controls/Widgets)

Objects can occur at varying levels of granularity. We can have large objects with many attributes and complex behaviours, representing things of interest to the business, such as Customers, Products, Orders, Agents, Suppliers and so on. These are normally unique to the organization and are defined by its own developers. We term these business objects. Reuse of these objects is non-trivial, but may be achieved within the applications of the organization with careful design and management. Over time, traditional package vendors will begin to offer their products increasingly in the form of purchasable libraries of business objects, which client organizations will be able to tailor without affecting the base definition.

Alternatively, we can have small, relatively simple objects which help us to build systems quickly. These include objects which we will use to build a graphical user interface (GUI), such as Windows, Menus, Buttons, Scrolling Lists etc. Other examples may provide for access to certain kinds of files, or communication protocols. These are normally generic, and are often provided with programming environments. Examples include ActiveX™ controls in the Microsoft environment and Java Beans™ in the Sun/Java™
world. They are termed technical objects, controls or widgets. Since they are so generic, we can normally achieve very high reuse of these components.

**Methods and Messages**

Methods are small pieces of code (similar to functions) which are written to accomplish a single, highly focused purpose. For a customer, we may have methods such as: `setCreditLimit`, `debitWith`, `deductPayment`, `closeAccount`, `changeAddress` and so on. Just like normal modules, methods can be passed parameters containing the data necessary to perform their function. In a pure object oriented environment (like Smalltalk) they are not called, but are invoked by sending a message to the object in which they reside. If I wanted to process a sale transaction, I might write:

```
CurrentCustomer debitWith: SaleAmount
```

This sends a message to the `Customer` object whose identity is contained in `CurrentCustomer`, invokes the method `debitWith`, and passes to it the `SaleAmount`. The method will use this to perform the function of updating the customer balance.

In a hybrid environment such as C++, although we still conceptually work with messages, they are implemented as calls. The above example would translate to something like this:

```
CurrentCustomer::debitWith (SaleAmount)
```

Here the `debitWith` function in the `CurrentCustomer` object is invoked passing the parameter `SaleAmount`.

**Encapsulation and Interfaces**

So who sends the messages? Other methods, of course! A system is composed of cooperating sets of objects, communicating by sending messages to each other. A key concept here is that each object operates independently, and is responsible for its own implementation details, which are hidden from the rest of the system. This concept is called *encapsulation.*
For example, if we decided that the Customer data structure had to change, this would not affect anything in the system except the Customer object. The change should keep the interface to the object stable, however, since altering this would affect other objects and methods. The interface is formally defined in terms of the method names and the parameters which are passed. We are free to change the way the method works or the data item types within the object without impact on other objects.

![Diagram showing Interface and Implementation]

**5.3 - Interface is Public, Implementation is Private**

**Classes**

In conventional programming and design, we have the concept of a record structure, defined to the database and/or program, and the actual records which contain individual data. In object oriented systems we have a similar concept. The structure of similar objects (objects which share the same data attributes and which require the same behaviour defined in their methods) is described in a Class. For example, I can describe the class `Customer`, which would define what attributes a customer has (e.g. `customerId`, `customerName`, `customerBalance` and so on) as well as behaviours which a Customer exhibits (e.g. `buyGoods`, `closeAccount` and so on). Individual customers are instances of this class. An instance could be:

- `customerId` = HerriotJ
- `customerName` = James Herriot
customerBalance = 100.00

......
......

Obviously there could be many instances of a single class. Each instance has its own set of data (sometimes referred to as instance variables) but methods are held once, with the class definition. Objects are aware of their own type, and each has a unique identity in the system.

- A class is a definition of the structure of an object
  - Data items (Instance Variables)
  - Behaviour defined in Methods (Code)

- An object is an Instance of a class
  
  e.g. Employee (Class) has instances Joan, Bill and Colin

- Each instance has its own copy of the data structure
  Methods are kept once with the class

5.4 - Classes and Instances

Inheritance

O.K. so this just sounds like modular programming taken to a logical extreme. Is that all there is to it? No, there are still several other exciting aspects to come. On of these is inheritance. We frequently find that we need something which is similar to an object we already have, but slightly different.
5.5 - Inheritance

An example would be having Employee defined as a class. We find that this is fine for everyone except sales people, who have a quota assigned. In all other respects, they behave exactly the same way as other employees. We can define SalesPerson as a subclass of the Employee class. It will inherit all characteristics of Employee. In the SalesPerson class, we define the additional attributes required (quota, salesValue) and the new methods to implement the additional behaviour (setQuota, recordSale, calcCommission). A SalesPerson instance will understand any message that an Employee would, plus the additional ones. This is a very powerful concept and allows modeling the real world with much less effort than conventional approaches.
"Define the Employee Class"

Object subclass: #Employee

InstanceVariableNames:
  'name joinDate salary'

classVariableNames: "

poolDictionaries: "

"Define the Behaviour of an Employee"

"Calculate Bonus, Pro-rata"

calcBonusAt: bonusRate for: month
| bonus |
bonus:=salary * bonusRate * (month / 12)
^bonus

"Get and set methods"

"Get the salary"
salary
^salary

"Set the salary"

salary: aNumber
salary:=aNumber
...etc

5.6 - A Sample Smalltalk Class Definition

The sample class definition shown as figure 5.6 illustrates some key principles:

- NOTE: In Smalltalk, all items between double quotes are comments

- The top section indicates the creation of a new class Employee by asking the Object class to make a subclass of itself. This is pure object orientation - in Smalltalk classes are themselves runtime objects with behaviour so we can ask the Object class to make a derivative. The # in front of Employee just means that the name must be unique (a Smalltalk symbol)

- The next section defines the instance variables for the class. These are the properties or attributes that each instance of the class will have. This is analogous to a record definition or structure definition in a conventional programming environment. Not that in Smalltalk, we simply name the variables, without specifying any types (integer, string etc.) This is because Smalltalk is dynamically typed. Everything in the language is an object and our variables will actually just point to a value, taking on whatever type we store into them at run time. Ignore the classVariableNames and PoolDictionaries for now

- The remaining text represents several methods which provide the objects with behaviour. Each method has a name, following the comment. Some methods will take parameters, which are named following parts of the
method name with colons. Other methods do not require parameters, since they merely request the object to return a value it already has, e.g. Salary. Note that we have shown all the code together here. This is not how a programmer will typically work with it. We would first define the class and its inheritance, then define the variables, then each method in turn.

A second class definition example, with three sections (inheritance, behaviour, structure of variables) in a different sequence this time, is given below for Java. Note that Java code is much more “programmer style” with lots of brackets and punctuation, as well as key words. Note the strong typing of variables in Java and the specific designation of items as private or public.

```java
// Define the Employee Class
class Employee {

    // Public Methods
    // Create an Employee instance
    public Employee(String n, double s, Day d) {
        name = n;
        salary = s;
        hireDay = d;
    }

    // Increase salary by a percentage
    public void raiseSalary(double byPercent) {
        salary *= 1 + byPercent / 100;
    }

    // Get the year of hire
    public int hireYear() {
        return hireDay.getYear();
    }

    // Get the name
    public String getName() {
        return name;
    }

    // Private data variables
    private String name;
    private double salary;
    private Day hireDay;
}
```

### 5.7 - Java Class Definition

#### Multiple Inheritance

It is possible for a subclass to inherit from more than one superclass. For instance, I may want the class `BranchManager` to inherit from `BranchStaff` as well as `Manager`. Conflicts can arise where each of the superclasses has the same method defined e.g. there is a `calculateLeave` method for `Managers`, and one for `BranchStaff`. In this case we would have to identify which one `BranchManager` should use, or define a new one entirely within
5.8 - Multiple Inheritance

Polymorphism

Another important concept is that of polymorphism. This is achieved by defining methods with the same name for several classes. This will allow different classes to respond to the same message, each in an appropriate way. An example would be a document composed of text portions and diagrams. Each piece of text, and each diagram could be an object. I could send the message “print” to each of them in turn. Each would print itself in an appropriate way.

Complex Objects

Objects can contain simple data types as their instance variables (data attributes) but also other objects. For example, I can define the object Computer as consisting of the objects Processor, MemorySystem, I/OSystem, Peripherals, PowerSupply and Casing. Each of these, is in turn an object, with its own instance variables and methods.

This capability allows us to handle rich data types such as sound, pictures, vector graphics, animation, video and time series (multi-dimensional) data easily and as a matter of course in our business applications. OO techniques underlie the new multimedia products. We could, for example embed a
scanned image of a product in the product attributes along with the stock code, price and quantity on hand; or annotate a document with a sound sample. A key thing to realise is that an object is usually contained by \textit{reference} not physically held within the composite object, so there is no redundancy and an object can simultaneously be contained in several composite objects.

- Objects can contain atomic attributes eg. number, string, logical value
- Objects can also contain other objects as attributes - these may have arbitrary complexity

\textbf{5.9 - Composite Objects}

The diagrams would use graphics printing while the text sections would use normal text printing. The important thing is that this detail is hidden from the requesting code which simply asks them to print: it does not even need to know that they are text and diagrams. This concept is extremely powerful and is exploited extensively in design of graphical user interfaces, for example. Polymorphism can be further supported by dynamic binding where the decision of which method is invoked is determined at run time, rather than compile or link time. This is supported in interpretive or incrementally compiled environments such as Smalltalk, but not usually in 3GL based environments such as C++.

\textbf{Collections and Iterators}

Another interesting concept is that of collections. Object oriented languages support a variety of these as standard structures. Smalltalk, for example, provides bags, sets, sorted collections and dictionaries amongst others. Bags
contain an arbitrary collection of objects and allow duplicates. Sets contain a collection of objects, and prevent duplicates. Sorted collections maintain their content in a default or user specified sorted sequence. Dictionaries allow the association of a key value with a second value or description. To define a lookup table in Smalltalk is thus a single instruction requesting a new object of type dictionary!

The Smalltalk hierarchy (Pruned a little)

Collections are extremely powerful. We could regard a document, for example, as an ordered collection of text and diagram objects. Iterator methods are provided to address the same message to all objects in a collection. These remove the burden of loop control from the programmer. Thus to print the document, I would simply send the message print to the collection. This would cause the print message to be sent to each element in the collection causing it to print itself in an appropriate manner.

Collection iterators include methods that allow “walking over” the collection, selecting objects from the collection based on user-defined criteria, and the ability to collect the result of evaluating an expression for each member of the collection. Collections can also be converted from one type to another (e.g. I can convert a dictionary to an array for use in a drop down list).

Persistence

This is all very interesting, but how is the data stored between executions? Here we draw upon the concept of persistence. This means that the values of instance variables must be preserved from one session of working with the system to the next. This can be achieved in a variety of ways:
• The entire set of instances for the whole application can be written out to storage as a single “image” file. This approach is used by single-user environments on personal computers, such as Visual Smalltalk™ from ObjectShare.

• The data for an instance, or set of instances in a class, can be written to a conventional operating system file and loaded back in from there the next time the application is run.

• The data can be written to relational tables in a relational database management system, often using SQL as the interface mechanism. This approach is fine for relatively simple data (such as is typically used in transaction processing systems), but can become a performance overhead when more complex data (such as that in a CAD system) is involved.

• The data can be stored in an object oriented database, specifically designed to support object oriented applications. Examples are Gemstone (Gemstone Corp.), ObjectStore (Object Design), Jasmine (Computer Associates).

More on object databases and persistence via object to relational mapping later.

How does OO affect us?

Fundamentally. There are a host of applications now being delivered by a wide variety of vendors which could simply not be done without object oriented techniques. These include the Graphical User Interfaces (GUIs) such as used on the Apple Macintosh, Amiga, Microsoft Windows and OS/2; Computer Aided Drafting (CAD) packages; Simulation software; Multimedia authoring systems; CASE tools and many others. All of these exploit object oriented technology to cope with the complexity involved. Virtually every major player which manufactures software for a living has already adopted, or is busy adopting, object oriented techniques. These include Microsoft, IBM, NCR, DEC, Borland, Sun, Next, Computer Associates, Hewlett Packard and others.

Object oriented techniques pervade virtually every aspect of Information Systems. They appear in object oriented programming languages, they provide more powerful design techniques, allow closer modeling of the real world during analysis, and are even being used in enterprise modeling. They affect systems software such as operating systems (e.g. NextStep), communications, databases and other user interfaces. They affect the way that we structure applications, and the manner in which users will interact
with them. There is likely to be an object oriented operating system in your cell phone, and object oriented programmes in your video camera. In short, OO is a much more significant and widespread phenomenon than structured programming and the structured techniques of the seventies and eighties.

**History**

The concepts originated in programming, as far back as Simula 1967. This was a language designed to allow development of simulation applications. As such, it had to have facilities to easily define real world objects, and to model their behaviour. It introduced the ideas of objects, classes and inheritance. An early artificial intelligence language, LISP (List Processing Language) contributed some powerful syntactical ideas and easy manipulation of collections.

The next major development was Smalltalk, developed at Xerox Palo Alto Research Centre (PARC) beginning in '72 and reaching a widely used standard in 1980. This is where the graphical user interface (GUI) and the desk-top/windows metaphor of user interaction was born. Developers created Smalltalk as a rich language to use for the development of systems software. Smalltalk incorporated the concepts of objects, classes and inheritance from Simula, conventional programming constructs from the Algol family of languages, and collections from LISP. Smalltalk was the first “pure” object oriented language. In it, everything is an object: A single character, a string, a number, a bitmapped image on the screen, even the compiler! It is probably the most unified development environment created to date.

Smalltalk was used to develop the early windowing software on the Star workstation. These concepts were later commercialized by Apple with the Lisa, and later the Macintosh. They were incorporated into a multi-tasking system by Commodore in the Amiga, and later brought to the IBM PC world in the form of Windows by Microsoft. Today, GUIs are found on virtually all platforms, including Presentation Manager under OS/2, and Motif under Unix/XWindows. All of these use object oriented concepts.
Object Oriented Languages

- Simula-67 introduced objects, methods and classes with inheritance
- LISP, early 60's, introduced list and collection handling
- Smalltalk 72 thru 80 developed at PARC introduced GUI, full OO IDE
- Ada defined as realtime OO language for defense agency
- Object Pascal, C++, CLOS OO variants on conventional languages
- Eiffel, Actor
- Clipper 5 + SuperClass, Visual Objects as Database OO 4GL
- Object/1, ObjectVision Frameworks
- Object COBOL
- Forte, Dynasty
- Java

5.11 - Language Evolution

On the language front, a number of object oriented languages were developed in the wake of Smalltalk. These included Actor (fairly similar to Smalltalk with a more C-like syntax, and using Windows as a GUI) and Eiffel (an object oriented language developed by Bertrand Meyer to support formal specifications and contracting). In addition, object oriented features were added to many conventional programming languages: Thus C++ and Objective C were developed from C; CLOS and Flavors were derived from LISP; and Object Pascal was developed from Pascal. Object Pascal underlies the very successful Delphi development environment from Borland. Ada, a language commissioned by the U.S. Department of Defense for use as a standard for realtime and embedded systems, was developed using some object oriented principles. Even COBOL has undergone a painful specification development process to deliver an object oriented version in 1995. The first commercial implementations of this are now available from MicroFocus and IBM.

It is important to realise that Smalltalk, and other products like it, are not just a language, but a complete integrated development environment (IDE). This includes editors, browsers, file management, virtual memory management, compiler, and a very rich set of functionality in the form of predefined classes. Smalltalk, for example has some 300 classes which are standard
across various vendor implementations, and a high end commercial version normally has 1000+ classes as shipped. The environments normally operate with a GUI, and provide data, object and instance browsers.

Languages such as C++ allow the definition of classes, but only include a few of these as part of the product. Class libraries are available from vendors and third parties, however, to address a wide variety of technical and application requirements. For example, we can purchase a library to handle Windows™ easily, or to implement the algorithms and data structures necessary to handle production scheduling. The explicit definition of the classes, methods and attributes, together with the power of inheritance, allows us to rapidly adapt the provided libraries to meet our specific requirements. Thus we can buy the functionality which is generic, and only write that which is unique to our situation.

Components

There has been quite a bit written in the last few years to the effect that “objects have failed” but “components work”. This is amusing, since components are simply another way of distributing objects! We could not have components without objects. It is simply a maturation of the technology into a more “packaged” form, which is, incidentally, much easier for novices or commercial developers without a formal programming or computer science background, to grasp and use. So how is a component different from an object? Not much, since a component is an object. The major differences in approach lie in the way in which libraries are distributed and in the execution environment expected by the runtime object. We should really ask, how are components different to the distribution of designs as class libraries? The following table will demonstrate:

<table>
<thead>
<tr>
<th>Components</th>
<th>Class Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Like</td>
<td>Building something from plans or blueprints provided by a designer</td>
</tr>
<tr>
<td>Distribution</td>
<td>In source form as class definitions</td>
</tr>
<tr>
<td>Use</td>
<td>By modifying original code or via subclassing</td>
</tr>
<tr>
<td></td>
<td>In object form - i.e. precompiled</td>
</tr>
<tr>
<td></td>
<td>By filling in instance values to customize</td>
</tr>
<tr>
<td>User can alter</td>
<td>Properties which the designer has predetermined as parameters which may be modified</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Expect</td>
<td>A predefined runtime environment conforming to some standard</td>
</tr>
<tr>
<td>Usually conform to</td>
<td>An accepted industry standard “component model”</td>
</tr>
<tr>
<td>Require</td>
<td>Low to average skills to use safely</td>
</tr>
<tr>
<td>Reuse</td>
<td>Achieved relatively easily</td>
</tr>
<tr>
<td>Mental image to illustrate concept</td>
<td>“Lego” blocks</td>
</tr>
</tbody>
</table>

**Building Applications with OO**

Conventional development performs an analysis of the problem, designs new structures in the form of processes and data models, then implements these by writing new program code, which must be tested and documented from scratch. This is an extremely labor intensive and error prone process. This, in turn, is reflected in the inordinate amount of time spent testing and debugging applications (typically about 50% of development effort) as well as in the high levels of maintenance effort experienced by organizations (typically 70-80% of total effort). We clearly need to do things differently if we are to successfully build more complex applications in future, with quality levels that will ensure that they are both reliable and easily maintainable.

Object orientation offers such an alternative. OO encourages the construction of applications by the use of predefined, tested components obtainable from specialist suppliers. An automobile manufacturer today
would never build a vehicle from the raw material (iron, steel, rubber, copper wire etc), but obtains the components from specialists (tyre manufacturers, electrical component manufacturers, instrument makers etc) and assembles these into a completed vehicle.

- Components are like ready made building blocks
- Class Libraries are like an expert design

5.12 - Components vs Class Library

This is much more time and money efficient, and provides a superior product. OO encourages re-use of predesigned and tested components in this manner. This has the potential to vastly improve development times, as well as enhancing quality and reducing maintenance. This will not be achieved by buying an OO compiler though, but requires a long-term cultural change in our way of approaching development.
• Collections of classes designed to work together within a problem domain (e.g. Engineering) or technology environment (e.g. GUI)

• Packages of the future

• Enable the creation of bolt together applications

5.13 - Class Libraries

The promise of OO analysis, design and construction is that we will be able to build information systems as increasingly accurate simulations of the business process.

**Object Oriented Databases**

Databases have evolved over the years through simple file systems, to indexed and multiple indexed random access files, to hierarchical databases, network databases, and the relational model. The last of these has become very popular since it has a rigorous mathematical foundation, and promises minimal redundancy and the elimination of processing anomalies. Unfortunately, these benefits are achieved at the expense of performance, particularly in applications where very rich data relationships are involved. Two of these applications are CAD/CAM applications and repositories for CASE. Designers of these products have run into severe problems with relational technology. The problems arise because things that the application wants to deal with at an assembly level are stored physically at an atomic level, and have to be laboriously assembled into usable structures via extensive input/output and processing. Consider what has to happen to load a 3d design for a jet engine stored as a set of relational tables!

A solution has been forthcoming in the advent of Object databases (or OODBMS). These are persistent stores for data and code modeled in objects. They have some very interesting properties and implications:
Their modeling abilities are superior to those of both network and relational databases, particularly as regards the ability to support specialisation through inheritance.

Their performance is as good as that of network databases, without the associated overheads normally incurred by network systems.

They do not lose the benefits of the relational model i.e. o redundancy and anomalies.

They are capable of storing complex objects, including such rich data types as bitmaps, images, structured drawings, digitized voice and video.

They can retrieve composite objects efficiently, since they see them as a whole, and do not require application logic to reconstruct the object - for example the 3D picture of the jet engine could be retrieved by the application with a single request.

---

5.14 - Object Database

They can store the methods with the objects upon which they operate, thus the concept of a "system" as we know it today disappears. There is no separation of data and systems. This has major benefits for integrity - if the only way to access data items is through the methods in an object, no other process can corrupt the integrity of the database. Also, the...
implementation of a data structure is localized within an object. We can change its implementation without affecting any other part of the system, provided that we keep the interface (message and parameters) the same.

- Since the database becomes the repository for code, and methods are activated there, we can refer to the OODBMS as an active or intelligent database. It is also very easy to build in monitors and triggers which will accomplish certain processing when the value of a variable assumes a particular value. For example, if our EmployeeLeaveBalance exceeds the maximum which can be accumulated, we can trigger a method which will generate a letter to the individual requesting them to take or cash in some leave.

### 5.15 - DBMS Suitability

Commercial OODBMS have evolved from several camps. First there are the OO language proponents seeking to provide persistent storage for their data, then there are vendors building object databases from the ground up and finally, there are the relational vendors trying to protect their market share. Notable OO databases are Gemstone from Gemstone Corporation which interfaces to Sybase, ObjectStore from Object Design, ODBMS from Ontos, Versant from Versant Object Technologies, Open ODB from Hewlett Packard, and Object/DB from DEC. These systems offer rich modeling and work well with OO applications, but still lack the maturity and production utility support of the more traditional products.
A body called the Object Data Management Group (ODMG) has defined standards for OODBMS, including the object modeling capabilities, Object Definition Language (ODL) and Object Query Language (OQL). As vendors implement these, commercial acceptance of object databases will grow rapidly.

Relational vendors working hard to provide richer data types, inheritance and code storage include: Oracle, Software AG, IBM (with DB/2), Sybase, Ingress (OO version is called Postgress) and Progress. Informix has taken an interesting approach by buying in the Illustra Object Database and integrating this with the traditional Informix engine.

The relational vendors are extending support by supporting complex objects (a column can be the name of another table); Inheritance of schema definitions down a hierarchy; and storage of methods within a table containing the instances as relational rows. Time will tell if it is possible to graft object oriented capabilities onto a relational base or whether a complete new architecture is required. The network system Vendors were less than successful with adding relational capabilities to their systems.

There is an ANSI committee working on an object oriented version of Structured Query Language (SQL) to be known as SQL3.

**Object Oriented Models**

To model the real world, we need to understand what things we are dealing with in the environment (or problem space). This is similar to the concept of entity relationship modeling, but we are also interested in behaviours and communication between objects. We may also need to be sensitive to timing issues and dependencies. If we can uncover these things, then we can construct an object model which is an accurate description of the real world.

There are various notations for Object Models. These include techniques from Grady Booch; Shlaer and Mellor; Yourdon and Coad; James Martin with James Odell; Ivar Jacobson and Jim Rumbaugh. All of these have certain characteristics in common:

- The ability to depict graphically the inheritance between classes (a *kind of* hierarchy)
- The ability to depict the composition of complex objects (a *part of* hierarchy)
• The ability to show the collaboration of objects by message passing to achieve the objectives of the application

• Some also have mechanisms to depict timing and dependencies required for real-time or event driven applications

• The Martin/Odell method uses an extension of entity modeling and can also depict normal data relationships between object types (classes) e.g. One to Many, Many to Many etc.

• The Booch, Rumbaugh and Jacobson approaches were merged in the Unified Modeling Language (UML) notation which has been adopted as a standard by the Object Management Group (OMG)

Of the techniques you have used to date, entity modeling is probably the most useful as a starting point for object modeling. You can begin with entities, think of them as objects. Specialization of attributes would indicate a need for a subclass. Aggregation of entities can lead to abstractions represented as superclasses. For example, we may decide to group Chairs, Tables and Photocopiers under the superclass OfficeFurniture. We would then add behaviours (methods) as far up the class hierarchy as possible (i.e. make them as generic as possible). Second and third normal form thinking is useful in deciding where an attribute or method should be defined. First normal form does not apply - where an attribute or set of attributes recurs, define a list on the parent object consisting of a collection of child objects. These can be treated as though they physically resided there. The parent object thus becomes a complex object. There is no actual redundancy, however, since the repeating objects are contained in the parent object by reference. They actually exits in the system only once.

Object modeling may proceed as follows:

• Identify application objects

• Find abstractions and define class hierarchy

• Identify external events to which the system must respond

• Detail interactions between objects (collaborations)

• Specify methods

This will produce an application object model suitable for developing an object oriented database. The designer will also build two other types of models, to describe the user interface objects (the view) and the communication of these with the application model (the controller). These will not be detailed here, since they do not normally involve persistent storage.
An Object Modeling notation

We are going to introduce the basics of a notation for object modeling. This is based on the Unified Modeling Language, now an OMG standard. It is not very different to entity modeling, with which most commercial developers are familiar.

Classes and Inheritance

Classes are shown as boxes, containing the class name. Inheritance (the kind of relationship) is shown by a line with an open ended arrow pointing from the subclass to the superclass.

Class Hierarchy

Associations (Relationships)

These can be shown as we normally would on an Entity Relationship Diagram (ERD), using a simple convention for indicating multiplicities.
- Normal data relationships are supported

- Notation

<table>
<thead>
<tr>
<th>Multiplicity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>One</td>
</tr>
<tr>
<td>0..n</td>
<td>One or More</td>
</tr>
<tr>
<td>1..n</td>
<td>One or More</td>
</tr>
<tr>
<td>0..1</td>
<td>One or More</td>
</tr>
</tbody>
</table>

One Department has one or more Employees
An Employee works in one Department
One Post has 0 or one Employee occupying it
An Employee always occupies one Post

5.17 - Associations (Relationships)

Composition

Composite objects are shown using an aggregation relationship. This is shown by a filled diamond at the end of the line connecting the composite object class to the contained object class. The diamond is on the composite end.

An organizational unit contains exactly one manager, and one or more employees

5.18 - Composition (aggregation) relationship

Note that the contains relationship is a part of relationship. Also note that we can still represent multiplicities to indicate whether the contained objects occur once, or many times.

Putting it all together, we can see how the technique allows us to represent all the types of relationships on a single diagram:
If the models are developed in a CASE tool, then we can choose to hide or show various types of relationships easily. We could, for example, show just the class hierarchy diagram to discuss the accuracy of this with a user. We could show just the data relationships for verification with a data architect.

**References and Further Reading**

Harmon, Paul & Taylor, David, 1993, Objects in Action, Addison Wesley


Taylor, David, 1992, Object-Oriented Information Systems: Planning and Implementation, John Wiley

Fowler, Martin, 1997, Analysis Patterns - Reusable Object Models, Addison Wesley
Chapter 6

Joint Application Development (JAD)

Background

Joint Application Development (JAD) is a facilitated approach to problem solving and reaching consensus, whilst making use of a wide range of inputs and participant skills. It was originally called Joint Application Design and arose in IBM Canada around 1979. There is now a standard published by the world-body GUIDE (an IBM user group). The approach has evolved way beyond the original design focus to include the whole of the systems development lifecycle and the change in name to joint application development reflects this. It has also been adapted and used successfully to support strategy formulation and planning activities. In this guise it is known as Joint Requirements Planning (JRP).

Purpose of JAD

Research by Martin, DeMarco and others has shown that a high proportion of system errors are introduced at the specification and design stage of projects. Relatively few errors are due to the later project stages such as programming and testing. This is aggravated by the fact that errors at these early stages tend to have high impact and many ramifications. This means
that an inordinate amount of the cost of correcting errors is due to problems in these early stages.

6.1 - Source of Errors in Lifecycle

The major issues were further identified as getting sound communication between developers and users, and reaching consensus amongst the user community. JAD was developed in response to these insights.

6.2 - Proportion of Effort to Correct Errors
6.3 - Relative Cost to Fix Errors

The traditional development approach involved a series of interviews of users and management by systems analysts, who then went away and drafted specifications. These were often large and arcane. The users may have been asked to approve them, but in effect did not have a hope of knowing if they were right or not until they eventually saw the resulting system. When they did, more often than not, it was not even close to what they had in mind. The evolution of structured techniques and the graphical modeling methods they promote went some way toward making the models unambiguous and the communication more precise. There was still the problem of time, however, since the interviews, collating the results, constructing the models, reviewing them with interviewees and the iterations through this cycle produced very lengthy projects. JAD was seen as a way to achieve two very important objectives:

- Higher quality specifications and designs, which better represent the real user requirements
- Accelerated development by eliminating the lengthy cycle of interviews, collation, drafting, reviewing and adjusting
6.4 - Conventional vs JAD Approach

How JAD works

JAD involves facilitated, structured meetings where all parties with a contribution to make, or a significant stake in the outcome, participate in one session, or series of meetings. Essentially, it aims to get everybody involved together in one place, at one time, in a focused way, so that plans, requirements or decisions that represent the consensus view of the group and carry their commitment can be produced. There are several important aspects here:

- Participants should include all those with a major contribution in terms of skills and knowledge. This will include a mixture of business and technical people, operational and management staff.

- The JAD session is held on a continuous basis until a result is produced. This can range from sessions as short as an hour, to some that can last several days.

- A trained facilitator leads the meeting. This person is independent and does not have a vested interest in the outcome of the session. His or her role is to ensure that the session runs smoothly, and that the required results are produced.
Conflicts are resolved and consensus is reached through brainstorming meetings. Thus, in a JAD meeting, the development of user requirements is carried out in realtime in contrast to the traditional and slow approach of individual user interviews. JAD works because the right people are involved and can make decisions by contributing to a decision-making process and reach consensus on solutions with their peers. This ensures better executive commitment and improved user input.

Automated JAD takes the concept still further with the introduction of CASE tools. CASE automates the use of modeling techniques generally used in systems analysis and design. The "automation" of the meeting increases productivity and produces more rigorous deliverables - provided the person operating the tool is fully conversant with it and the technology does not intrude on the group dynamics.

Where we can use JAD

JAD can be used for a wide variety of purposes, including:

- Determining IS strategy
- Prioritising projects
- Determining system requirements
- Domain modeling using Entity Relationship or Object Modeling
- Business Process Modeling, or Business Engineering
- Producing a technical design
- Choosing technology
- Review of technical deliverables
- Implementation planning

In short, anywhere where a number of different parties can provide input and a better result and higher commitment would be obtained from reaching a high level of consensus.

Relevance of JAD to Data Management

The Data Administrator and Data Base Administrator sit at the centre of networks of communication in the organization. They have to balance and
satisfy requests from competing groups of users, and maintain a balance between the users and software professionals.

A great deal of the data manager's time is thus spent in the areas of communication, planning, analysis, design, verifying models, prioritizing tasks and reaching consensus between areas. Facilitation skills are thus an extremely valuable part of the data managers toolkit.

JAD can be usefully employed for:

- Developing data/object architectures
- Developing corporate level data/object models
- Developing entity relationship/object relationship models
- Developing relational models
- Implementation planning of a new technology or software version
- Verifying physical database/object designs

Participants

The participants in a JAD session include managers, users and IS staff. The group should be able to understand the business and technology whilst having the authority to make decisions.

Typically participants in the JAD session include the following:

- An **executive sponsor** who is a senior executive who can make project decisions regarding resources and user selection
- **User managers** who take final responsibility for the system and are involved in the entire project
- **Users** who understand the business in detail and will eventually use the system
- The **project manager** who is responsible for the project
- **IS professionals** who are part of the project team
- The **JAD leader (facilitator)** who runs the session and ensures action is taken on outstanding issues
- The **JAD scribe** who documents sessions, often using an automated CASE tool
• **Observers**, who cannot participate in the session unless requested to do so, but attend to learn about the JAD process. Maximum observers in a session is normally limited to two.

In some organizations some of the above roles are combined. This has been found to be detrimental to the process. For example, if the project manager acts as JAD leader, there may be a bias, because the project manager is often an IS professional. Similarly, if a senior user manager facilitates, then technical issues may be ignored. The JAD leader also requires a different set of skills. The JAD leader should ideally be a person from outside the organization (or at least outside the project) who has been specifically trained in facilitation skills, including:

- Active listening
- Conflict handling
- Reading body language
- Probing
- Modeling techniques to be employed
- Data/Object Modeling
- Function/Event Modeling
- Data Flow Diagrams/ Process Modeling
- Prototyping
- Presentation skills
- Team building and group dynamics
- Situational Leadership
- Quality orientation

The JAD leader should also possess more subtle characteristics which are difficult to produce purely by training. These include:

- Business acumen, or a sense of the organization/industry
- Maturity
- Honesty and integrity
- Confidence
- Ability to see all facets of a problem and think laterally
JAD Process

JAD requires a receptive environment to work successfully. An example is the requirement that participants have equal status and are all expected to contribute and allow others to participate fully during the session. If the company has traditionally had great disparities between the power and privileges of managers and other staff, this will require a special effort on the part of senior people and the building of trust by other participants. There is thus a marketing process to “sell” the concept to the organization and senior management. Only once management are committed to the approach and agree to abide by the JAD rules, can we proceed to run successful sessions.

Following the initial organizational introduction, the JAD process works in 3 distinct phases: Session preparation is followed by the JAD session proper, in turn followed by Post-session work.

- Marketing
- Senior Management Commitment
- Participant Education
- Session Preparation
- JAD Session
- Post Session Follow up

6.5 - The Jad Cycle

Session Preparation

Proper session preparation is imperative to ensure that the valuable time of participants is not wasted. When we consider that a session may last two days and include 10 senior people, the costs can easily be calculated in the tens of thousands of Rands (several thousand $US). Preparation involves:

- Reviewing JAD documentation (output from previous sessions)
- Developing JAD controls (session charter & agenda, identifying applicable deliverables and standards for their production)
- Selecting participants
- Review by the Executive Sponsor
• Interviewing the participants to inform them as to their role in the session and to deal with any concerns
• Providing JAD training (where necessary)
• Review agenda
• Organizing venue and facilities
• Distribution of documentation prior to the session

JAD Rules

Participants, including managers and sponsors, must agree to the following principles:
• There is no seniority in the session. All participants operate as peers. The facilitator acts as the arbitrator and chairman
• There is only one person speaking at a time. The facilitator will determine who speaks next
• Stay focused on the issues at hand.
• Everyone attends full time, arrives on time, and interruptions are not permitted

The Venue

The venue should be clean, convenient and quiet. Interruptions must be controlled. There should be no telephone. Cell phones should be switched off. There should be ample white/black board space, pens/chalk, erasers etc.; an overhead projector, and possibly a data/video projector if I-CASE tools are employed. Flip charts are a valuable way of capturing information permanently as consensus is reached. A data board is an extremely valuable facility. This is a white board which, at the touch of a button, produces a paper copy of its surface. A digital camera can also be used to capture information from the board and transfer it to computer rapidly.

The JAD session

The JAD session normally starts with introductions and the executive sponsor explaining the purpose of the session. This is to lend authority and
importance to the proceedings. The JAD process is then reviewed, followed by a presentation of the agenda after which the session gets under way. In the session, the JAD leader ensures that:

- There is only one conversation at a time
- Every participant is given a chance to contribute
- The subject focus is not lost
- Consensus is reached
- All ideas are considered

**Group dynamics**

The leader tries to influence the group to behave as a team. Some ways of achieving this are:

- Emphasizing common goals
- Bringing in and highlighting the contribution that each person makes
- Coaching slower or shy individuals to make their contributions (including them in the group)
- Distributing responsibility to team members
- Maintaining focus on the topic and goals of the session
- Shared experiences - intense sessions
- Providing opportunities for participants to socialise

Research shows that groups will tend to adopt a higher risk tolerance than individuals. If this is appropriate to the situation, it can be a good thing, since more adventurous decisions can result. Where quality or safety are at stake, however, the tendency must be countered by being more thorough, and emphasising these aspects.

It may sometimes be necessary to include a maverick in the sessions to disturb the comfortable "group think" that people who routinely work together have. This allows new options and alternatives to surface and be considered on their merits. Another strategy is for the leader (an outsider with nothing to lose) to throw these ideas in.

It is vital that full consensus be obtained if a high degree of commitment from participants is to be achieved. This sometimes means that a discussion will be prolonged long after you know what the answer will be. This is the
price we have to pay for the people who are still not with us to "come on-board". It is time well invested.

At the end of the session the leader summarises the results including:

- The deliverable(s) produced in the session
- Future activities
- Responsibilities assigned and deadlines
- Decisions taken

The scribe documents the decisions, models, requirements and other deliverables produced by the session. This can be done manually, or using a CASE product.

**Post-session Work**

Post-session work comprises a review of the session followed by the completion of neat versions of the deliverables by members of the project team. It is crucial that any outstanding deliverables be completed quickly and circulated to participants. This maintains momentum and provides an opportunity to trap any errors which may have crept in.

Care should be taken to review the quality of both the deliverables produced, and the JAD process as experienced in the session. Ways of improving both should be documented and distributed for future application. This is part of an ongoing quality improvement process.

**Success Factors**

Good JAD sessions are run by leaders who can influence people without being dictatorial. They should be people-oriented with a humorous touch whilst being persistent enough to get results. JAD is not a panacea and can fail if incorrectly used. To be successful, there must be senior management commitment, the session must be well prepared, and the right participants must attend.

A successful JAD session requires:

- Senior executive support
- Company wide use of JAD (or at least support for its use)
- A well organised session
• A good venue, free from interruptions
• A strong JAD leader
• A competent scribe
• The right participants
• A quality deliverable

JAD succeeds because everything is visual and the JAD leader continually seeks clarification and commitment.

Results Obtained

Companies, such as CNA Insurance and Carrier Corporation are reporting up to a 40% reduction in time in the SDLC phases from planning to external design. Other users such as Texas Instruments report significant benefits in quality due to better user involvement. Successful JAD sessions happen because of good group dynamics. The effectiveness of a group relates to how well the facilitator and project manager build the team.

A Caveat

JAD is a useful and successful technique, but it is not a substitute for the other techniques, especially modeling. It should rather be seen as a better way of reaching the deliverable, which may well be a model, using a structured or object oriented technique.
Business Domain Object Modeling

Purpose

We model the business domain objects to obtain a comprehensive view of the information required to support the activities and proper management of the enterprise. Experience has proven that information (data and objects) are a more enduring base upon which to build information systems than functionality. The latter may change with alterations in business processes, policy and the whims of management, but the business objects remain constant so long as we stay in the same fundamental business.

A full object model would contain the objects and classes, relationships between them (inheritance, aggregation, containment, association), attributes (data items) and the behaviours (expressed as methods). At the early stages of domain object modeling, we are likely to concentrate on the objects and relationships, later adding attributes as we discover more detail, and finally, behaviours from the dynamic models we will build later. This section will cover the initial stages, leaving the behaviours for later.

Notation

We have chosen to use the Unified Modeling Language notation of Booch, Rumbaugh and Jacobson (all now at Rational Corp.). This is a
comprehensive notation and is widely supported by CASE vendors. It has been accepted by the OMG as a standard for the industry. Another very capable notation that we have used successfully for many years is that of James Martin and James Odell. We finally made the switch to UML when we felt that this had incorporated all of the concepts of the former that we needed.

**Business Objects versus Controls/Widgets**

We can differentiate between business objects and widgets as follows:

- **Business Objects** are usually coarse-grained (they have many attributes) and represent something in the real world of significance to the enterprise. They act as simulations of things in the real world. Examples would include Customer, Product, Order, StaffMember, Branch, Agent. They are normally persistent i.e. we require that the values of their attributes are preserved across time and space, being available to multiple applications on multiple occasions.

- **Controls/Widgets** by contrast are fine grained and provide components for the construction of interfaces or technical aspects of systems (e.g. the graphical user interface (GUI) or relational database). Examples would be a Window, scrollbar, buttons, menus, relational table wrapper etc. They are normally contained within one executing program image and transient i.e. they are destroyed when the program finishes.

Although the technique of object modeling we will use can accommodate both types of objects, for the moment we are really only interested in business objects - the others will just add noise now.

A third kind of object is an abstract data type (ADT). These represent complex data structures that we may wish to embed in business objects as attributes. Examples would include: Name, Address, Date, Point, List, BitMap and so on. Many of these will be provided as standard in the class library of object development tools, but we will note the need for additional ones as we proceed with domain modeling. We will normally not include them formally in the model to keep the model more manageable. The ADT’s will be specified fully at the design stage.
Drawing the Model

Business objects can be found by looking for nouns in the domain (problem space) under consideration. This is natural, since objects represent things and concepts. Examples of objects found in this manner are:

- For a retail food seller
  - Customer, Product, Store, Employee, Supplier
- For a University
  - Student, Faculty Member, Course, Degree, Venue, Result, Prescribed Work, Faculty, Department

In addition, we will need to record business events or transactions which take place between the enterprise and the stakeholders or other external parties. These include:

- For a retail food seller
  - Sale, Payment Received, Return, Refund, Supplier Order, Supplier Payment, Staff Payment
- For a University
  - Enrollment, Examination, Graduation, Fee Payment, Salary Payment, Promotion

Specialization and Generalisation

Some types of objects may have more specialised forms. For example, our retail customer may choose to make a payment by cash, credit card or cheque. Each has slightly different requirements, but they are still all kinds of payment. This represents the Generalisation/Specialisation (GenSpec) relationship. It is expressed as an inheritance relationship between classes. In UML it is shown as follows:
7.1 - Specialisation Through Inheritance

The open arrow head indicates inheritance by subclasses of attributes, relationships and behaviours of the parent class. The way we have combined the lines from the subclasses in the example is used to indicate that the sub-types are mutually exclusive. I.e. if an object is a Cash payment, it will have only those characteristics: it cannot simultaneously be a cheque or credit card payment. There are other cases where the sub-types are not mutually exclusive. An example:

- Employee
- Manager (inheriting from Employee)
- Temporary (inheriting from Employee)
- Temporary Manager (inheriting from both)

While this is a convenient modeling shortcut, we need to take care with multiple inheritance since it is not available in many environments, and because it can lead to design problems with an explosion of permutations.
We will discuss this problem after dealing with some of the other types of relationships.

We can combine the above two notations to indicate quite rich differentiation, as below:

![Diagram of Student and Foreign relationships]

7.3 - Disjoint and Non-disjoint Specialisations

Sometimes we will identify the need for a generalisation to hold common properties of subtypes which we already have. An example would be the following:

![Diagram of Office Equipment and Vehicle attributes]

7.4 - Using Attributes to Identify a Superclass

Here we see that there are common attributes between Office Equipment and Vehicle, viz. Description, BookValue and DatePurchased. We ask, are Office Equipment and Vehicles kinds of something else that has a BookValue and DatePurchased? The answer, of course, is yes, they are both kinds of Asset. We must be careful to apply the kind of test, though, before assuming an abstraction. We may find common Branch attributes between Order and Customer, but this does not mean that Orders and Customers are kinds of Branch!
In other cases, attributes may help us to identify required specializations. This is particularly true where we find “type” attributes or optional attributes. Consider this example:

**DEPOSIT**
- DepositType
- Client
- Amount
- BankDrawnOn
- ChequeDate
- DepositBranch

(Cash or Cheque)

**DEPOSIT**
- Client
- Amount
- DepositBranch

BankDrawnOn and ChequeDate are only required for Cheque deposits. The DepositType field indicates what kind of deposit was received. We would specialise, creating two sub classes. The fact that the Cash Deposit subclass has no new attributes is not a problem, it will inherit the needed attributes from Deposit. Notice that the “type” field has disappeared to become implicit in the type of object that we create.

### 7.5 - Using Attributes to Find Specialisations

Association

Objects can also be connected without inheritance, particularly where they are related or know about each other. It is important to realise that while inheritance relationships are between classes or types of objects and a specification/design/programming convenience, association relationships are actual references at run time between real objects (instances of classes). Associations exist where an attribute of one object holds the identity of another. Examples would be:

- A Customer has a number of Orders
- A Product is stocked in a certain Store
- An Employee works at a certain Branch

In UML, associations are shown by a simple line between the two object type symbols. They can be enhanced with multiplicity, which indicates how many of the target object can be associated with one instance of the other
object. Multiplicity is shown at the target end of the line using the following conventions:

- One
- * OR 0..* Zero or more
- 1..* One or more
- 0..1 Zero or One (Optional)
- 2..4 Specified minimum and maximum

Consider the following model using associations:

- A Department can have one or more Employees.
- An Employee works in one Department.
- An Employee is appointed against one Post.
- A Post may or may not have an Employee occupying it.
- An Employee can have zero or more Dependents.
- A Dependent may only be associated with one Employee.
- A Qualification can be held by zero or more Employees.
- An Employee can have zero or more Qualifications.

7.6 - Associations and Multiplicities

Associations can also be labeled. This is particularly useful where there is more than one association between two object types. An example will illustrate this. Association labels are written above the line reading left to
right, and rotate with the line in the direction of reading. This means that reading top to bottom, the label would be to the right.

![Diagram]

**7.7 - Labelled Associations**

It is also perfectly acceptable for an object to have a relationship with other objects of the same type. E.g.

![Diagram]

This creates a hierarchy of reporting relationships with an indeterminate number of levels. A similar structure will be found within a bill of materials.
**Association Class**

Sometimes the association will have attributes of its own. For example, the association between a Student and a Course, viz Enrollment, would need to record details such as the Date of Enrollment, Result and Fees Paid. In these cases we can link a class symbol to the association as shown. Alternatively, the Association Class can be modeled as a normal class, resolving the many to many association into two one to many’s.

![Diagram of Student, Course, and Enrollment relationships](image)

*Sometimes the association will have attributes:*

- Enrollment records the links between Course and Students
- e.g. Date of enrollment, result of attendance

*Can also be shown as a normal Class:*

![Diagram of Student, Course, and Enrollment relationships](image)

**Aggregation and Composition**

There are two stronger types of association. These are created in similar ways to normal associations (i.e. by holding an attribute which refers to another object) but we wish to constrain the relationship further. The methods experts are not in full agreement about this topic, but a practical distinction is given below:

- **We speak of aggregation** when the associated objects are regarded as sub-components of a larger part. We would like to be able to refer to the composite part and imply that it includes the sub-parts. This would be the case with a project team and its members (employees). When we refer to the team, we take it to include all the members. Sub parts are permitted
to be part of more than one aggregation. An employee could be part of more than one team. They can also come into being independently and can persist after the parent part has been destroyed. If we disband the team, we do not want to lose the employee!

![Diagram](image)

7.9 - Aggregation

- **Containment** is a stronger form of inclusion. It implies that the sub-part has meaning only within the context of the whole part. The sub-part would only be contained within one whole part and would not exist before or after the parent part. An example would be Dependents for an Employee, where we are solely interested in these details for an Employee, and they are meaningless outside this context. Containment is shown with a filled diamond in UML.

![Diagram](image)

A further example would be the controls within a GUI window. They have no existence except as part of the window and should be moved when the window is moved. If the window is destroyed, they should disappear too.

**Navigability of Links**

If associations are navigable (i.e. They can be traversed by a program), we can show this with an open arrowhead at the target end of the link.
If associations are navigable, we can show this with an open arrow head at the target end of the link.

![Diagram showing associations between Product, Customer, and Order]

Product is not aware of which Orders it appears on. Customer holds a collection of Order references. Order has identities of the Products it contains and the Customer to whom it belongs.

**7.10 - Indicating Navigability of Associations**

**Stereotypes**

This is a standard mechanism built into UML to identify sub-categories of the basic symbol types provided. Stereotypes can be indicated textually by enclosing the category name within guillemots written above the name in the class box, or visually by altering the symbol shape or including an icon. Typical stereotypes we would encounter in UML Class Diagrams include:

- <<Type>>
- <<Class>>
- <<Object>>

![Diagram showing stereotypes on Employee and Post, Joe and Cleaner]
Resolving Multiple Inheritance Issues

We mentioned earlier that we could run into some problems with multiple inheritance where the permutations of non-disjoint subtypes become excessive. Now that we have the tools to deal with an alternative, we can examine the problem in more detail. Consider the following model which allows for a Person to be a variety of things to a business:

![Class Diagram]

This would allow a person to be none of the subtypes, any individual sub-type on its own, or any combination of the subtypes. For 4 subtypes, there are 16 permutations, from the Person type with no additional characteristics, through all combinations, to one with the combination of all four roles. This is clearly a major problem when we come to implementing the model in a design and actual classes. It is also a problem when a Person object acquires another role, since that would require it to be destroyed as an instance of one subclass and recreated as a new combination subtype.

We can solve these problems by abstracting out the common properties (viz Person) and including this in the subtypes by aggregation, rather than inheritance. This looks like this:
Here the Person attributes occur once on a Person object, which is included in the other roles as an attribute. The same person can thus assume a variety of roles as necessary without having to define many permutations and without requiring recasting to a different object type. If we have variations of Person in another dimension (e.g. whether this is a real or legal person), this can be accommodated by Person having subtypes. The attribute which holds a person would be *polymorphic*, allowing us to contain either a real or legal person in the role as necessary. This is very flexible and powerful. We should ensure that all subtypes which will be polymorphically contained in this manner provide a common method protocol (same set of method names and parameters) so that including objects can reliably use them without worrying about implementation. An example would be to include an ID method on real and legal persons. This may return an individual’s Identity Number for a real person, but a company registration number for an enterprise.

Patterns

What we have just seen in the preceding section is an example of a *pattern*, that is a defined way of solving a particular problem or meeting a certain kind of requirement. Patterns allow us to capture experience in an explicit way and pass it on to other people. The power of a pattern is in the expression of a principle, which can be applied to, and customised for, a
variety of situations. The *multiple inheritance resolution* pattern we used above can work wherever we have the same object participating in multiple *roles*. We saw this above for a person acting in various roles in relationship to an enterprise. It could be adapted for use where a literary work can appear in a number of guises, which are non-disjoint. For example, a fiction story can appear as a book, a tape, a video and a film.

![Diagram of multiple inheritance resolution pattern applied to fiction work]

7.14 - *Multiple Inheritance Resolution Pattern applied to Fiction Work*

We will explore patterns in more detail in later chapters.

**Showing Distribution**

Distribution is not a standard feature of the UML, but we have found it a useful extension. It allows us to discuss where objects are physically located, and the issues surrounding synchronisation of data across locations. We use a bounding box with dotted lines to show distribution.
7.15 - Bounding Boxes show Distribution

Power Types

Again, not present in UML, but an innovation from James Odell. These allow us to model the situation where a type has attributes or behaviour of its own, discreet from the instances of the type. In these cases, the type itself is shown as an object/class on the diagram. It would have one instance per subtype of the generalised object type. An example would be in an insurance setting where we have different types of policies. The normal model would look like this:
We can add a Power Type to hold the information related to the type of policy, e.g. acceptance limits, rules, rates etc.

This is best implemented as a class object, which is provided in pure object languages (e.g. Smalltalk). Where classes are not objects, a class will need to be created for the type.

**Bringing it all together**

The following example will illustrate most of the principles which we have discussed. It provides a model for a bank.
7.17 - A Banking Domain Model

Properly used, Business Domain Object Models can express a great deal of information very concisely and with considerable subtlety. They can later be expanded with full definition of the attributes of each object type. Where this is desired, we use the following notation (extended from the UML recommendations) to show more detail for an object/class. This box can be substituted for the name-only boxes we have used in the diagrams till now. Of course, a CASE tool will allow you to look at the overview diagram and then “drill down” to the detail level.
Method patterns can be expressed in the 3GL-ish style of UML:

```
methodName (param1, param2, ..):return type
calcInterest (startDate, endDate, rate):decimal
```

or the more English-like form used by Smalltalk:

```
methodName: param1 extension: param2 ..^return type
calcInterestFrom: startDate to: endDate at: rate ^decimal
```

Of course, too much detail on presentation models can also become confusing or detract from a good review with users. We should selectively show what is important for a given purpose. For example, we can suppress subtypes to simplify the diagram, or look just at inheritance or associations at any one time.

In later modules we will map the behaviours to the classes as dictated by appropriate responsibilities. The necessary behaviours will be discovered by modeling the business processes and prototyping the interaction of the system with the environment.
References and Further Reading


Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick & Lorensen, William, 1991, Object-Oriented Modeling and Design, Prentice Hall

Martin, James & Odell, James, 1996, Object Oriented Methods: Pragmatic Considerations, Prentice Hall

Various, 1996/7/8/9, UML Notation Guide, Rational Corp., OMG
This page intentionally left blank.
 Enterprise Modeling

The View from 10 000 Feet

If we have an effective systems delivery method, then it is reasonably straightforward to turn a known requirement into a working system. We can thus be efficient - “doing things right”. More difficult is to be effective - “doing the right thing”.

Enterprise Modelling helps us to:

- Keep projects and developments aligned with Corporate Goals
- Identify high payback applications to apply scarce resources
- Prioritise development from the perspective of business need and technical imperative
- Scope projects and systems

There are a variety of techniques including:

- Critical Success Factors (CSF’s) [Rockart]
- Functional Decomposition [I.E., Tetrarch, Infomet, Method/1]
- Value Chain Analysis [Porter]
- Use Cases [Jacobson and UML]
Based upon our experience, we have found that the value chain approach is powerful, especially when combined with stakeholder analysis and business process modeling. We use these in a manner that is a superset of what can be achieved with Use Cases. Elements of Critical Success Factors are brought in by examining attributes of processes, such as their goals and critical variables to monitor.

**Stakeholders**

Stakeholders include anyone (or any enterprise) with an interest in our organization’s continued existence and success. Generally they derive some value from their interaction with us. They typically include:

- Customers/Clients
- Suppliers
- Employees
- Shareholders
- Business Partners
- The State (in the guise of the Receiver of Revenue/IRS at least)

Stakeholders usually provide some form of input to the enterprise and expect some kind of output, which for them has added value over the input. A summary of the inputs and outputs might be:

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Input provided to us</th>
<th>Output expected from us</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>Money</td>
<td>Goods or services</td>
</tr>
<tr>
<td>Supplier</td>
<td>Goods or Services</td>
<td>Payment</td>
</tr>
<tr>
<td>Employee</td>
<td>Effort, intellectual contribution</td>
<td>Remuneration, benefits</td>
</tr>
<tr>
<td>Shareholder</td>
<td>Capital</td>
<td>Return on Investment, Capital Growth</td>
</tr>
</tbody>
</table>
We like to draw a stakeholder model, which shows the organization at the centre and its interaction with the various stakeholders. This helps our analysis to be outward looking rather than inward. We want to do things that enhance the ability of the organization to add value for stakeholders, thereby ensuring its prosperity.

### Value Chains

The processes involved in using the inputs (possibly from more than one stakeholder) to produce a valuable output are referred to as a value chain. An example would be the sequence below:

- A Client gives us money and requests goods or service
- Some of the money is used to buy raw material from suppliers, some to pay production workers (Employees)
- The employees transform the raw materials into a finished product
- The product incorporating the *added value* is supplied to the client
- Surplus funds generated may be distributed to the Shareholders as dividends or retained within the enterprise

Porter defined a value chain model, depicting a primary value chain and a set of support activities.

### 8.2 - Porter’s Value Chain Model

In this model, the primary value chain has the following steps:
- Inbound logistics - how we obtain our basic stock in trade. For a manufacturer, this will be acquiring raw materials and designs. For a retailer, it will be purchasing goods from suppliers
- Operations - how we transform the inputs into something of higher value e.g. Assembling raw materials into finished goods, or investing money on behalf of clients to produce dividends
- Outbound logistics - how do we deliver the products and services to our clients
- Marketing and Sales - how do we chose which products or services to pursue, how do we segment our market, price the product and conclude sales
- Service - how do we look after our product and clients after the sale
The model shows a variety of support activities which are there to support the primary value chain. E.g. Human Resource Management should ensure that we have appropriately qualified and motivated employees.

The Enterprise should prosper to the extent that it can optimise the value chain and this is represented by the margin to the right of the figure. If steps in the value chain can be performed more quickly, to higher quality or consuming less resources, this will increase the margin, enhancing the organization’s competitive position.

Unfortunately, the value chain is often obscured or at odds with the traditional functional structure of the business. Consider Figure 8.3. This represents a retail furniture seller organized along traditional functional lines. The primary split is between branches which deal with customers and make sales and deliveries, and head office where central purchasing is done.

Within the branch there is a Sales department, a Managers area, a Collections section and a Delivery department. Head office includes Purchasing and Dispatch of purchased goods to stores. A client may conclude a sale with a sales person in the Sales area. This then moves to a Manager for credit approval, following which Collections will take an initial deposit and record banking details for credit agreement (hire purchase) payments. Following this, Deliveries are notified so that the goods can be delivered to the client. This can only take place if they have first been purchased at Head Office and dispatched to the store. Finally, when the client has the goods, we hope to routinely process monthly payments.
It looks like a familiar scenario. Unfortunately, it is fraught with many problems. What will happen if a client pays a deposit and is promised delivery, but documentation is not successfully passed on between Collections and Deliveries? We could have an irate client who feels that a promise has been broken. When the client calls us to enquire or complain, they are likely to get a run around while they are passed from one Department to another, none really knowing what the status of the order is or what the position is. We will only become aware of the problem when the client is already annoyed. How could we change this scenario?

Here are some suggestions:

- Recognise that there is a business process which is triggered by the request to purchase goods, which only concludes when the last payment is made
- Appoint a single responsible party to monitor the overall status of the process, so that the client has one point of contact with the organization and is not expected to know our internal functioning
- Instrument the process, so that the movement of the order from one stage to the next is recorded
- Establish some norms for what is reasonable in terms of time in each stage
- Implement exception reporting to the process manager so that we can proactively recognise that there is a problem, before the client knows
- Organize the business along process lines rather than functional hierarchies
- Integrate systems to optimize the value chain and the experience for stakeholders

We will take a look at how to achieve some of these after discussing business process modeling in more detail in the next few chapters.

**Extended Value Chains**

In today’s business climate, organizations often have to “stick to the knitting” and concentrate on what they do exceptionally well. This means outsourcing with other organizations for non-core activities and partnering to deliver some elements of the value chain. For example, we may be a national retailer, but could outsource our store planning to an architecture firm. We may contract with a courier company to handle customer deliveries. The message is that the boundaries of the organization are by no means as fixed
as they once were - they are more grey, fuzzy and flexible. There are system and technology implications for this:

- We will often need our systems to interact with those of our business partners
- We cannot define stable interface points between our (internal) and their (external) systems. This brings with it a need to implement architectured solutions with multiple layers or points of opportunity to mix and match.
- We should also look to conform to industry or international standards wherever possible, so as to minimize the effort in interworking and hence the time to implementation from a business perspective

Identifying Business Processes

A useful way of identifying business processes, is to consider each stakeholder and the likely stimuli they will generate to which our enterprise must respond. Each of these will normally trigger a business process. Some of these will be routine, such as updating information about a business object (e.g. Customer address change, supplier price change ) and can be handled easily by a standard process based upon a pattern. Others will be core value adding processes on the primary value chain (or in support areas where we do innovative and strategically advantageous things). These require our focussed attention to examine them in detail and, where necessary, improve them. In chapters that follow, we will look at how we can document these processes at a high level, analyse them in detail and improve them through a variety of techniques, including the application of technology.
References and Further Reading

Comcon, 1986, The Tetrarch/1 Strategic Information Planning Methodology, Comcon (Pty) Ltd.

Frank, Ulrich, 1994, Memo: A tool Supported Methodology for Analyzing and (Re-) Designing Business Information Systems, in Ege, R; Singh, M; Meyer, B (Hg): Technology of Object Oriented Languages and Systems, Prentice Hall pp 367-380

Jacobson, Ivar; Ericsson, Maria & Jacobson, Agneta, 1994, The Object Advantage: Business Process Reengineering with Object Technology, Addison Wesley


Chapter 9

Business Process Modeling

Process Orientation

All who work in or with industry are aware of the increasing pace of change and the global competition which challenges organizations to become more efficient and nimble in their responses. Taylor identifies three key areas where organizations must excel to remain viable and competitive in the face of these pressures:

- they must do things faster
- they must do them better
- they must do them more cheaply

He points out that this implies that we need business processes which are more efficient, produce higher quality products and services and consume less time and resources. Similar trends are noted by many other authors, including [Jacobson, 1994], [Hammer, 1990]. In our observation within corporate consulting clients, these requirements for business process modification now affect up to 70% of systems projects. This implies that any competent systems analysis and design method should provide constructs to facilitate rethinking of business processes and expressing modified processes
in a form which can flow smoothly into later stages of a systems development lifecycle.

The Unified Modeling Language (UML) [Rational, 1997], produced by the collaborators (Booch, Jacobson and Rumbaugh) at Rational corporation is emerging as an industry standard for the expression of system level models using object oriented techniques. It is an amalgam of earlier work by the three authors, but also includes innovations from other contributors. UML was approved (late 1997) by the Object Management Group (OMG) as a standard. UML specifies a notation for analysis and design models. It does not prescribe a method or process for how these should be used. UML has powerful techniques for expressing many facets during the analysis and design stages of a project including:

- Domain knowledge required to support the application (Static Structure/Class Diagram)
- Interaction of a user with a system to achieve a single goal (Use Cases)
- Sequence of collaboration between objects (Sequence Diagram)
- Responsibilities for dynamic behaviour within the system (Collaboration Diagram)
- Changes in state of an object over time (State Diagram)

UML latterly also includes an Activity Diagram. This is derived from the work of Odell [Martin & Odell, 1993] where the equivalent is referred to as an event diagram. The notation differs, but the concepts are essentially similar. This is capable of showing processes in a powerful way, including parallel operations, triggering of operations by state changes affecting objects (events) and synchronization. UML recommends that this be used in specifying the implementation of a responsibility within a class, or the detailed implementation of an operation within a use case. There are also component and deployment diagrams which express specific implementation choices.

UML is thus richly endowed with techniques to express the results of systems level modeling and design choices. There is little, however, to express high level views of the organization or processes and to assist a business engineer in choosing among competing alternative processes. [Jacobson, Ericsson and Jacobson, 94] have described the use of use cases in support of business process engineering, but the approach still lacks the semantics to facilitate the above fully. The next section looks at what we require to facilitate such support.
**Business Process Modeling**

Work by [Frank, 1994] and others at the Gesellschaft für Matematik und Datenverwertung (GMD) on the “virtual government” project incorporated many state of the art business engineering and enterprise analysis techniques. Specifically, stakeholder models and value chain analysis were adapted to allow powerful business modeling, with extended capabilities to assist the analyst in understanding the organization, its interaction with the environment and the issues and alternatives in choosing between competing methods of achieving the business of the organization. The concepts were incorporated in a methodology (Memo) and implemented in a set of advanced tools, developed in Smalltalk on Sun workstations. These can guide us in listing the requirements for a competent approach supporting business engineering.

Additional semantics which we require to support competent business engineering include:

- Ability to identify the *stakeholders* in the environment and how they interact with the organization
- Ability to identify *external stimuli* from the environment which trigger business processes to commence and the source of these, which is usually an actor, but could also be another process
- Competence in expressing *parallel and asynchronous activities*
- Identifying activities which are performed *manually, with automated support, or in a fully automated way*
- *Volume* of transactions (or stimuli per unit time)
- *Responsibilities* of organizational units for activities within the process
- *Durations* of operations
- *Resources and costs* of operations
- *Probabilities* that various alternative paths are chosen

**Inspired Architecture Frameworks**

In our work with architectures and strategy within our corporate client base, we have evolved a set of architecture frameworks over the years. These address the business, application, information and technical aspects of creating an enterprise architecture. This architecture approach is detailed in a forthcoming work by this author to be published in 2000. In the meantime
details are available from the author via e-mail. Within the business architecture, there is an architecture for business processes, which views the process from an external perspective and facilitates a high level discussion of the objectives and constraints of the process before detailed modeling work is undertaken. The process architecture is shown below:

9.1 - Inspired Frameworks© Process Architecture

In this picture, we can see the following elements:

- The **business process** is seen as a sequence of steps and decisions. It may include sub-processes as steps
- The process is triggered by a **stakeholder** (which is anyone who interacts with the enterprise) initiating a business event. Examples would be requesting a new account, ordering a product, requesting payment for goods provided..
- The business process is governed by **rules and policies** which the enterprise may establish, or which may be mandated by law
- Various parties (internal **organizational units** and external **partners**) are responsible for the performance of particular steps
- Steps in the process will occur at a variety of **locations** (physical or organizational)
- The process should support one or more expressed **business goals**
- The process effectiveness can be measured by one or more **key indicators**
- Various **business communications** may be produced as a result of the process e.g. Statements, contracts, letters etc.
The process may reference or update the state of various **business objects**, such as customer, product and agent information

- The process is likely to be supported by one or more **application systems** (which is how it normally makes use of business objects)
- The process will usually produce a **product**, a **service**, or combinations of these
- The process is likely to consume **resources**, such as staff time, money, raw materials, etc.

This framework provides a powerful model for discussing the relevant business processes with business executives. It is non-technical and allows us to focus on the essential issues related to the process, the objectives and the context of the process, without becoming bogged down in the detail of how the process is done now, or should be done in future. Once these issues have been clarified, we can move on to detailed modeling of current processes and engineering of new processes where necessary or appropriate. This more detailed modeling is the focus of the following section.

**Previous Work with Odell notation**

The author previously incorporated the ideas from GMD, Jacobson and [Taylor, 94] into a method based upon the Martin/Odell OOA/D method [McLeod, 92-97]. This extended the method into business engineering with the use of stakeholder models, value chains and business process models. The latter were evolved into standard Odell style event models, providing a seamless transition from the business engineering view, to a systems view, and finally, a design view. At the design level, a multi-tiered architecture mapping based upon Model-View-Controller concepts (but extended for integration with legacy environments and competent for client server and Internet environments) ensured a further smooth transition to implementation level. In teaching, consulting and project work, this has proven to be a very successful approach over a period of some four years.

The approach supports the following analyses:

- Building a current and desired view of the business process for comparison purposes
- Distinguishing between manual, computer supported and automated activities
- Determining the relative duration and costs of process alternatives
• Mapping operations within the processes to organizational or geographic locations
• Elimination of non-value adding activities by abstraction of essential effects of operations and recasting these as value adding activities which are performed efficiently
• Analysis of parallel activities and asynchronous activities with proper respect for dependencies and preconditions

The approach uses a simple notation and has proven easy to use and effective in Joint Application Development (JAD) session with users. Minimal introductory education is required (usually about an hour) for participants to feel comfortable with the technique. Some devices and notations are introduced and explained as needed. One limitation has been the lack of a suitable CASE tool to capture the models. These have to date been captured in a graphics package with no knowledge of the semantics, and later transferred into more formal tools, including Aris™. In other environments, we have first translated the business models into event models and then captured these in tools such as Paradigm Plus™ and Object Team™.

With the prevalence of the UML and its increased support in industry and widespread support in tools, the time was right to attempt a UML version of the approach. A paper was presented at UML’98 in France, June 1998. This chapter is derived from that paper which focussed on how to extend the UML to support business engineering, as described, with minimal impact to the existing UML definition. It also incorporates ideas developed subsequently and presented as a tutorial at Tools 99 Europe.

**Extending UML**

The first consideration in extending the UML is to ensure that we are not duplicating functionality which already exists. Secondly, we must chose devices present within UML which most closely resemble the necessary constructs to express the required business process models. Finally, we must ensure that our extensions are sensible, introduce minimal distortions to the UML, are consistent with the overall philosophy and integrate well with the other techniques, particularly refinement into later models.

Jacobson *et al* have demonstrated that use case diagrams can be used to express similar knowledge to that in a stakeholder model. Within such a picture, we can identify various discrete interactions that external parties may have with the organization. Using use case notation, these can be shown...
as ellipses within the boundary of the organization. It would be possible to decompose any of these bubbles into a set of equivalent processes using the use case technique. While useful, this does not support many of our requirements as previously detailed. For this reason, we chose to use the use case model only at the highest level and then to use other devices for further analysis work.

We examined existing models within UML to determine the most appropriate vehicle for expressing our business process/value chain models. The alternatives included Sequence Diagrams, Collaboration Diagrams, State Diagrams and Activity Diagrams. These are discussed below.

We rejected:

- Sequence diagrams, since they are not easy to use with end users in JAD sessions, and do not easily show parallel and asynchronous activity. They also are optimised for showing message sequences between objects, which is a much more detailed “design” view than we often have during business modeling
- Collaboration diagrams, for some of the same reasons as sequence diagrams. They are best for expressing object interaction at a design level after responsibilities have been chosen for classes
- State diagrams, since they show the changes in state experienced by a single object type. Business processes inherently affect the state of many different types of objects

We chose Activity diagrams for the following reasons:

- They easily give a sense of the flow and dependencies of steps in the business process
- They can show parallel and asynchronous activity easily, as well as synchronization where required
- They can support the definition of where operations take place organizationally or geographically through judicious use of the “swim lane” concept
- They easily represent processes that require or generate state changes across multiple domain objects
- They are closest to the event model technique which we extended previously in the Odell approach

We are aware that we are deviating from the UML advice to use Activity diagrams “for internal design”, but we feel that we have valid reasons for so
doing. There is as much a requirement to design business processes and to choose amongst alternatives as there is to design internal processes.

**Extending Activity Diagrams to Support Business Modeling**

We make use of the UML Stereotype mechanism which allows adding notation to the model. Stereotypes may be expressed as a text within *guillemots*, thus: `<<text>>` or as an icon. We use these mechanisms to show which activities within a diagram are performed manually, with computerised support (but still requiring human interaction) or in a fully automated fashion. In text form these are shown thus: `<<manual>>`, `<<supported>>` and `<<automated>>`. As icons, they may be shown thus:

```
  +
```

Manual         Computer            Fully               Embedded
Supported      Automated

The overall activity diagram has properties detailing the volume of this activity per unit time, the current experienced duration (best, average, worst case), the current cost per invocation and the desired target duration and cost.

**Agents**

We borrow the Use Case stick man for human actors interacting with the process. We extend this to two forms:

```
  +
```

```
  +
```

Internal         External

These can represent individuals, classes of users, or organizational units.
Inputs and Outputs

Borrowing from earlier methods of data flow and event models, we document inputs from and outputs to Agents. On high level models we simply name them on the directional link.

On more detailed models, we show them as a large bidirectional arrow, and define their attributes in detail. We sometimes refer to these collected attributes as a “view” to distinguish them from domain classes. It is sometimes also useful to show the type of input or output medium e.g. Form, Online Entry, etc. These are shown with an IO type symbol which is similar to the IO symbol, but shaded. If we do not need the visual clues, then we can simply record the type of the IO as an attribute of the IO symbol.

On system level models following prototyping, the IO ID will identify a prototype artifact, such as a screen, report or record layout for another type, such as a magnetic card reader.

Simple High Level Example

Putting some of the above together, we can see what a simple high level “context” or “use case equivalent” model might look like:
9.2 - High Level Process Model - Equivalent to Use Case

Triggers

Steps or activities in a process can be triggered by various means, including:

- Input from Agent

  ![Capture Order Diagram](image)

- Outcome from another activity

  ![Raise Interest and Print Statement Diagram](image)

- Time (reached or elapsed), where we show a clock face, and the relevant condition

  ![End of Month Diagram](image)

Arrow heads on links are optional if the flow direction is the default left to right and top to bottom.
Outcomes

Activities often produce outcomes. On high level models, we express these as an annotation emerging from an activity. If there are multiple possibilities, we can show them and what ensuing activity they trigger. As with Inputs, we simply label them on high level models. They will become more rigorous later.

![Diagram of Assess Credit Risk activity with outcomes OK and BAD]

Where we know the relevant numbers and want to do detailed statistical analysis of the models, we can include probabilities of various outcomes being reached. These are shown as a decimal fraction between 0 and 1, with 1 being certainty. Outcomes can be disjoint, in which case they are shown as emerging from a synchronisation bar. As we did with implied multiple inheritance on class diagrams, we follow the same principle with the outcomes. If they are not joined by a sync bar, then they can occur simultaneously. Thus in the following model, we can Accept or Decline and simultaneously request more info or not as required.

![Diagram of Check Medical activity with outcomes Info Request and Accept 0.8 Decline 0.2]

Another Example

Using the above devices, we can now look at a slight more comprehensive example, dealing with a simplified university registration process.
9.3 - Simple business process model

Selective Invocation

Entry to some activities will be optional, or based upon certain criteria or conditions being met. These can be shown as follows:

Either Condition Should Trigger the Activity
Activity is only triggered when all conditions are met

![Diagram showing the relationship between Update Account, Statement Request, Produce Statement, and the conditions for trigger]

Although default is "AND", the control condition can contain any evaluation required (see Rules)

Rules
We will explore rules in more detail in a later chapter. We can link them to our process models as guard conditions, specifications for how actions are performed or general policies which apply to the process overall. See the examples that follow:

- Can be specified anywhere
- As simple text on high level models
- Or identified by diamond with reference to rule base
- Class names are highlighted
- All data items mentioned should be defined in domain model

If StockOnHand < ReorderLevel and no PURCHASE ORDER issued then ...

StockOnHand = PhysicalStock - CommittedStock

When QuantityOnHand < TotalDailyOrders for PRODUCT issue to CUSTOMERS with PriorityStatus 1 first, place BACKORDERS for CUSTOMERS not satisfied

9.4 - Rules added to Models

Resources
If we want to model the resources consumed by a process, or see the impact of resources as constraints or on costs, we can add resources to the model.

133
Generally, resources are consumed or produced by activities. We can specify the resource usage in the properties of the activity, or we can show them graphically on the model. We also have resource types, shadowed, much as we did for IO Types earlier. The type allows us to specify constraints on the number of a given resource type available, e.g. We have 10 Assessing Clerks. We can also use it to hold a cost rate per unit of consumption.

![Diagram of Counter Assistant and Person with 5 mins Fill Form activity]

9.5 - Modeling resources and consumption

Swim Lanes and Bounding Boxes

UML offers the concept of swim lanes to group activities which affect the same class of objects. We extend the idea to allow us to group things for a variety of purposes. We also allow the use of bounding boxes for similar purposes. We use them to show:

- Organizational responsibility
- Geographic location
- Logical Transaction Start and Commit
- Business Objects Affected (system level models)
- Platform for deployment (design level models)

Could have several layers or overlays per model.
Getting More Detailed

Instrumenting the Process Model

We can add properties to the activities to express the following:

- Minimum, average and maximum *duration* of the activity (current, target)
- *Lead time* before the activity can commence (e.g. waiting for external activity). Minimum, average and maximum can be expressed if desired
- Organizational *responsibility* (which department, section, business unit performs it)
- *Resources consumed* (type of resource, unit of measure and consumption minimum, average and maximum)
- *Number of servers*. This details the number of replicatable resources available to perform the activity. This allows us to gauge the effect of adding or subtracting resources without changing the model structure
- *Geographic location(s)* (where the activity can be performed)
- *Cost* of performing the activity once (current, target)
Where multiple triggers emerge from activities, we document the state for each path, and the probability that that path is followed. The latter is expressed as a fraction of 1, or as a percentage e.g. .5 or 50%.

All of the above extensions are optional. We may use the model to simply express the process and the flow, or add as much detail as desired or available. Obviously, the more detail we add, the richer the understanding of the business process. A fully attributed model can allow sophisticated analyses of competing business process alternatives including:

- Determining the duration of the overall business process using standard critical path techniques as used in project management. Briefly, these sum the longest duration between any two points, giving the minimum time in which the overall process can be completed. This can be done using minimum, average or maximum times to give an overall best case, average or worst case duration

- Project Evaluation and Review (PERT) techniques can be used to determine most likely times and probabilities of meeting various time benchmarks if required. A full treatment of these is not possible here, please consult [Kerzner, 92]

- The cost of performing the process can be calculated by summing the costs of all activities traversed. Where activities are traversed more than once (e.g. picking stock for each line on an order), the sum of these occasions would be included. Where activities are optional, the probability of performing the activity times the cost will be included if we calculate an overall average scenario; or we can calculate the cost of various scenarios by computing the cost of specific paths

- The resources consumed can be calculated in a similar manner to costs

- Queuing effects can be brought into play. If the arrival rate of new requests for the process is such that a new request will arrive at an interval shorter than the processing time, then queuing will be experienced. We can use standard queuing network analysis techniques to determine the effect of such queuing on experienced duration by the initiating actor. We can also evaluate the effect of reducing processing duration for various activities (e.g. by replacing a manual operation with an automated one) and/or of increasing the number of servers of a replicatable resource (e.g. by having more clerks performing the operation manually)
An Example

To illustrate the approach, we will develop models for an insurance application process. The current business process is initiated by an agent or broker with a client; proceeds with submission of the application via a branch office, and concludes with the processing of the application by the head office, which responds with an acceptance or declination letter. Several simplifying assumptions have been made:

- The application requires a medical examination, and the medical report will be submitted independently by the doctor to head office
- The agent/broker will be credited with a partial commission immediately upon acceptance of an application
- No reassurance has been shown. We assume that this is done on a batch contractual basis rather than for an individual contract
- Bank details for payment are submitted with the application form
- The client will accept a reasonable loading of premium (in reality this would almost always be referred back to the client for acceptance before proceeding)
- Payment of doctors for performing examinations is outside the scope of the model
9.7 - Current Business Process

Selected properties for the overall and sub-activities are as follows:

(NOTE: for simplicity and brevity, we will work with averages in all cases other than for duration)
Overall Process

Volume: 1000 per day

Complete Application Form

| Duration | Min: 10 min | Avg: 25 min | Max: 90 min |
| LeadTime | Avg: 0 min  | Avg: 100    |
| OrgUnit  | Sales       | Location: Customer Premises |
| Cost     |             | Avg: 100   |
| Resource | Agent       | UnitOfMeasure: mins Used (Avg): 25 |
| NoOfServers | 1000         |

Application Completeness Check

| Duration | Min: 2 min | Avg: 5 min | Max: 10 min |
| LeadTime | Avg: 4 hrs | Avg: 10    |
| OrgUnit  | New Business | Location: Branch Office |
| Cost     |             | Avg: 50   |
| Resource | Clerk       | UnitOfMeasure: mins Used (Avg): 25 |
| NoOfServers | 500         |

Application Capture

| Duration | Min: 10 min | Avg: 25 min | Max: 90 min |
| LeadTime | Avg: 0 min  | Avg: 50    |
| OrgUnit  | New Business | Location: Branch Office |
| Cost     |             | Avg: 50   |
| Resource | Clerk       | UnitOfMeasure: mins Used (Avg): 25 |
| NoOfServers | 500         |

Poll Applications per Branch

| Duration | Min: 1 hr | Avg: 4 hrs | Max: 6 hrs |
| LeadTime | Avg: 24 hrs | Avg: 1    |
| OrgUnit  | I.S.       | Location: Head Office |
| Cost     |             | Avg: 1    |
| Resource | Mainframe  | UnitOfMeasure: secs Used (Avg): 15 |
| NoOfServers | 1          |

Capture Medical

| Duration | Min: 2 min | Avg: 5 min | Max: 30 min |
| LeadTime | Avg: 2 days | Avg: 10    |
| OrgUnit  | New Business | Location: Head Office |
| Cost     |             | Avg: 10   |
| Resource | Clerk       | UnitOfMeasure: mins Used (Avg): 5 |
| NoOfServers | 100         |

Assess Medical

| Duration | Min: 5 min | Avg: 15 min | Max: 90 min |
| LeadTime | Avg: 0 min  | Avg: 30    |
| OrgUnit  | Medical Assess | Location: Head Office |
| Cost     |             | Avg: 30   |
| Resource | Medical Ass. | UnitOfMeasure: mins Used (Avg): 15 |
| NoOfServers | 10          |

Underwrite

| Duration | Min: 10 min | Avg: 15 min | Max: 30 min |
| LeadTime | Avg: 0 min  | Avg: 30    |
| OrgUnit  | New Business | Location: Head Office |
| Cost     |             | Avg: 30   |
| Resource | Underwriter | UnitOfMeasure: mins Used (Avg): 15 |
| NoOfServers | 10          |
### Load Premium
- **Duration:** Min: 10 min Avg: 20 min Max: 30 min
- **LeadTime:** Avg: 0 min
- **OrgUnit:** New Business
- **Location:** Head Office
- **Cost:** Avg: 40
- **Resource:** Underwriter
- **NoOfServers:** 10
- **UnitOfMeasure:** mins Used (Avg): 20

### Check Bank Details
- **Duration:** Min: 1 min Avg: 2 min Max: 5 min
- **LeadTime:** Avg: 0 min
- **OrgUnit:** New Business
- **Location:** Head Office
- **Cost:** Avg: 4
- **Resource:** Clerk
- **NoOfServers:** 100
- **UnitOfMeasure:** mins Used (Avg): 2

### Issue Contract
- **Duration:** Min: 10 min Avg: 15 min Max: 20 min
- **LeadTime:** Avg: 0 min
- **OrgUnit:** New Business
- **Location:** Head Office
- **Cost:** Avg: 30
- **Resource:** Clerk
- **NoOfServers:** 100
- **UnitOfMeasure:** mins Used (Avg): 15

### Advise Client
- **Duration:** Min: 5 min Avg: 10 min Max: 15 min
- **LeadTime:** Avg: 0 min
- **OrgUnit:** New Business
- **Location:** Head Office
- **Cost:** Avg: 20
- **Resource:** Clerk
- **NoOfServers:** 100
- **UnitOfMeasure:** mins Used (Avg): 10

### Credit Commission
- **Duration:** Min: 10 sec Avg: 10 sec Max: 10 sec
- **LeadTime:** Avg: 24 hrs
- **OrgUnit:** I.S.
- **Location:** Head Office
- **Cost:** Avg: 1
- **Resource:** Mainframe
- **NoOfServers:** 1
- **UnitOfMeasure:** secs Used (Avg): .5

By performing the recommended analyses on the current process, we can determine the following:

- The best case duration, given by summing the minimum performance time and (for this example) average lead time on the slowest path between synchronisation bars, yields 3 days 38 mins and some seconds
- The average duration, given by summing the average performance time and the average lead time (as above), yields 3 days 1 hr 12 mins
- The average cost of processing a successful application is 326 Rands (about US$ 72)
• By far the longest delay is caused by waiting for the medical report to come in
• We would not achieve major time savings by speeding up head office processes
• Fully automated functions are vastly cheaper that computer supported ones with clerical staff still involved. We could achieve major savings in cost by higher levels of automation
• A great many other insights can be uncovered by further analysis. We may find additional delays in the head office processing due to inadequate resources of certain types (e.g. Underwriters). We can calculate optimum levels of various resource types to ensure availability without queues, but without wasted capacity

**Business Engineering (Process Improvement)**

After business engineering, we may have a very different process, for example:

**Overall Process**

| Volume: 1000 per day |

**Complete Application Form**

| Duration: Min: 10 min Avg: 25 min Max: 90 min | LeadTime: Avg: 0 min |
| OrgUnit: Sales | Location: Customer Premises |
| Cost: Avg: 100 |
| Resource: Agent | UnitOfMeasure: mins Used (Avg): 25 |
| NoOfServers: 1000 |
9.8 - Reengineered Business Process

Capture Medical
Duration: Min: 2 min  Avg: 5 min  Max: 30 min
LeadTime: Avg: 2 hrs
OrgUnit: New Business
Location: Head Office
Cost: Avg: 10
Resource: Doctor Staff
NoOfServers: 500

Assess Medical
Duration: Min: 5 min  Avg: 15 min  Max: 90 min
Lead Time: Avg: 0 min
Org Unit: Medical Assess
Cost: Avg: 30
Resource: Medical Ass.
No of Servers: 10

Cost: Avg: 30
Resource: Medical Ass.

Lead Time: Avg: 0 min
Org Unit: New Business
Cost: Avg: 30
Resource: Underwriter
No of Servers: 10

Cost: Avg: 40
Resource: Underwriter

Lead Time: Avg: 0 min
Org Unit: I.S.
Cost: Avg: .2
Resource: Mainframe
No of Servers: 1

Cost: Avg: .2
Resource: Mainframe

Lead Time: Avg: 0 min
Org Unit: New Business
Cost: Avg: 2
Resource: Mainframe
No of Servers: 1

Credit Commission
Duration: Min: 10 sec Avg: 10 sec Max: 10 sec
Lead Time: Avg: 0
Org Unit: I.S.
Cost: Avg: 1
Resource: Mainframe
No of Servers: 1

Check Bank Details
Duration: Min: .1 sec Avg: .2 sec Max: 1 sec
Lead Time: Avg: 0 min
Org Unit: I.S.
Cost: Avg: .2
Resource: Mainframe
No of Servers: 1

Load Premium
Duration: Min: 10 min Avg: 20 min Max: 30 min
Lead Time: Avg: 0 min
Org Unit: New Business
Cost: Avg: 40
Resource: Underwriter
No of Servers: 10

Issue Contract
Duration: Min: .1 sec Avg: .2 sec Max: 1 sec
Lead Time: Avg: 0 min
Org Unit: I.S.
Cost: Avg: .2
Resource: Mainframe
No of Servers: 1

Advise Client
Duration: Min: 20 sec Avg: 30 sec Max: 1 min
Lead Time: Avg: 0 min
Org Unit: New Business
Cost: Avg: 2
Resource: Mainframe
No of Servers: 1
Here, we have eliminated the branch office and equipped the representatives with notebook computers, which allow direct capture and completeness checking of applications and then transmit these via cellular telephone link to head office, eliminating the daily polling run. Doctors reports have been streamlined by accepting them via the Internet, with a paper version following later as a confirmation. Contract issuing, advice letters and bank checking have been fully automated. Credit commission has been changed from a daily batch run to a real-time update.

These changes yield the following improvements:

- Best case duration is reduced to 2 h 27 m
- Average duration is reduced to 2h 56 m
- Average cost per successful application is reduced to R213 (about US$ 47)
- There are substantial resource usage savings

We must be careful to balance the system, though, since the arrival rate of transactions at internal services may now increase as a result of removing external “buffering” and delays. There are also many social, political and ethical issues to consider in real-world scenarios. Our purpose here has been merely to demonstrate the efficacy of the proposed technique in identifying areas for analysis and expressing alternatives in such a way as to allow informed decision making.

The models produced in this way can be easily and seamlessly expanded to design models. We simply eliminate manual activities and expand computer supported or fully automated activities. For computer supported activities, human-computer interfaces can be prototyped and linked to the models. We can annotate the models to indicate which objects reach which state as a result of each operation. We can also change the swim lanes to reflect class, geographical or platform partitioning of the responsibilities, if required.

**Conclusion**

Our earlier approach has proven useful in many organizational contexts. With the wide acceptance of UML and the growing support for the notation in CASE environments, we hope that the extension of UML into business process and enterprise modeling will find support. This, coupled with capable tools and good training will equip analysts and business people to examine business operations in rich and multidimensional ways. Ultimately
this can assist organizations in becoming more efficient and effective in better serving their customers and other stakeholders.

We hope to further extend the work into providing quality measures for the outputs of business processes and the reengineering of work for better results in terms of products and services in addition to better control of costs, duration and resource usage. We would be interested to correspond with (and perhaps collaborate with) other parties working in similar areas.

References and Further Reading

Frank, Ulrich, 1994, in Ege, R; Singh, M; Meyer, B (Hg): Technology of Object Oriented Languages and Systems, Prentice Hall pp 367-380


Jacobson, Ivar; Ericsson, Maria & Jacobson, Agneta, 1994, The Object Advantage: Business Process Reengineering with Object Technology, Addison Wesley


McLeod, Graham, 1992-1997, Advanced Systems Engineering with Objects, Inspired, Box 384 Howard Place 7450 South Africa


Taylor, David, 1994, Business Engineering with Objects, John Wiley
This page intentionally left blank.
Chapter 10

Business Engineering (Process Improvement)

The Pressures Facing Business

Businesses, and indeed all organizations, today face massive pressures to operate effectively and efficiently. The time when it was good enough to be a price leader, or a high quality high price supplier have gone. With increased use of technology, open markets, powerful trading blocks, rapid transportation and the increasing knowledge component of goods and services, competition has become fierce. To simply survive in this marketplace, organizations must simultaneously deliver high quality, low prices and rapid delivery (or innovation).

Systems have become far more than back office record keeping. They are integral to the operation and service delivery of the business. Consequently, they must match the business goals and processes intimately and must be robust and reliable. In addition, they must be adaptable to match the rapid innovations in the business arena, or else they will inhibit the flexibility and agility of the organization to take advantage of opportunities or react to competitive thrusts.

To meet these challenges, we need to adopt a process view of how organizations transform inputs into value added outputs. These processes may be found to be inadequate. This chapter looks at opportunities to
improve processes, reshape the business and employ technology in innovative ways to deliver more value, more reliably and do it quickly. Later chapters dealing with architectures and mapping requirements to the design environment will build on this theme, providing ways to abstract the business processes into an easily flexed architectural layer, creating responsive organizations.

Process Changes

To success, processes will often need to deliver:

- Cheaper - using less resources to produce the required result. Resources include staff, skills, plant, machinery, capital and information
- Faster - producing more items of delivery per unit time (e.g. more products per day, more units of service delivery per month etc.) And/or being able to introduce new product or service offerings more quickly
- Better - delivering better quality products and service. “Engineering customer delight”, as the saying goes.

New Look Organizations

Often, the organization structure itself will need to be transformed. We still live with organizations whose shape reflects constraints dating back to the feudal times when the primary means of communication was person to person, or via a physically conveyed letter. The model was popular in the military, where privates were grouped into troops, which in turn were grouped into regiments, brigades, armies etc. with a lengthy and rigid chain of command from the commander down and upward reporting back to the top.

With modern communications, the constraints of physical communication and attendant delays over distance do not apply. Also, technology can perform the distributing and summarising functions of much middle management far more quickly and efficiently. Consider the relative costs of moving everyone in a large geographically distribute organization into physical venues to communicate a message to them versus sending a broadcast e-mail. Both to reduce costs, but also to increase efficiency and eliminate delays and distortions, organizations are becoming “flatter” with less layers between top management and the coal face. One oil company we worked with went from 12 layers to 4 over a period of some 18 months.
They increased profits and turnover some 40%, while reducing staffing levels by about 20% - obviously not without some trauma.

**The changing shape of the organization**

![Diagram showing changing shapes of organizations from slow, distorting, inefficient to lean, flexible, effective.]

**Leaner**

Other changes are making organizations leaner. Non-core competencies are outsourced to specialist companies, customers become an integral part of the business process and technology enables individuals to do more. All of these reduce the people intensity of organizations. This is potentially good for profits, but can, of course, be bad for people displaced from jobs. A goal should be to seek win-win situations where jobs may disappear, but be replaced by others which are “richer” and use the unique capabilities and potential of humans, rather than menial work or rote performance.

**Virtual**

More and more virtual organizations are springing up. These do not rely on the “bricks and mortar” (building, facilities, premises etc.) of earlier generation business, but instead thrive on flexibility, adaptability and organic forms which grow and shrink in response to market conditions and needs. These organizations may have no employees and no premises. They may meet in cyberspace over the Internet and use a convenient coffee shop anywhere as a boardroom. Yet, they can have high value in terms of collective skills, intellectual capital or brand. Our own company, *Inspired*, is such an organization. Some global clothing brand names, with turnovers in the hundreds of millions of dollars, operate on a similar basis. They do not
have any manufacturing capability or own a single store. What they do have is excellent design skills, manufacturing contacts in the East and developing world and a very strong brand which they franchise to local operators. Corporate identity and brand management via advertising and other activities is jealously nurtured by the very small corporate staff.

**Fractal**

An interesting new organization structure is the fractal. Fractals are a branch of mathematics where a certain shape or pattern repeated at different scales leads to large scale structure with considerable subtlety. It is evident in the curving edges of clouds, the branching structures of trees and the indentations of Norwegian fjords. Fractal organizations use similar principles to organize small work groups of individuals. Repeating the structure on a larger scale with the work group as a participant in a larger structure creates the “fractal” nature of the organization. They can provide many of the benefits of small, focussed teams, while still managing the scale of a large organization with considerable resources.

**Opportunities with Technology**

Technology provides us with many new possibilities. Part of our role as IT professionals, is to see how it can best be employed to help the business reach its goals. We should seek to do this in a manner that will enhance the experience of our clients and other stakeholders, while simultaneously improving the lot of internal stakeholders such as employees.

**Serial vs Parallel**

One opportunity is to remove sequential flows and make them parallel. We are no longer constrained by a paper rendition of a document which can only be in one place at one time. Via databases and communications technology we can view or update information in any location concurrently. Technology is available, for example, to allow collaborative authoring. I could be working on the document using a word processor in Cape Town. Open on my screen would be a page view with the text. Embedded in the page are a table created in a spreadsheet, which lives on a machine in Johannesburg, and a graphic rendered by an artist in New York. I may be editing the text, while the artist is revising the drawing. If she saves the graphic, it could instantly be refreshed in the document I am viewing. This kind of technology has been available in selected environments from the mid 90’s, but is becoming more widespread with the adoption of industry standards for the inter operation of distributed objects and compound documents.
Escape the Constraints

There are a great many constraints which the appropriate use of technology can remove. Some key ones are summarised in the accompanying table.

- Databases and networks allow many users, perhaps with different purposes, to share data regardless of location.
- Complex work typically requires experts, but work is not always as complex as it looks. We are aware of one situation where a company bought and sold oil on the spot market in Europe. The transactions are large and margins shifting constantly based upon supply, demand and political events. They had a room full of experts from various disciplines (marketing, finance, oil, shipping, logistics, pricing etc.) Who decided what and when to buy or sell and at what price. These decisions were then conveyed out to the field. Frequently cargoes are resold before they
reach their destination and are redirected.

<table>
<thead>
<tr>
<th>Perceived Constraint</th>
<th>Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information can only be in one place at a time</td>
<td>Shared Database Comms</td>
</tr>
<tr>
<td>Only experts can perform complex work</td>
<td>Expert Systems</td>
</tr>
<tr>
<td>Must choose between centralised and decentralised</td>
<td>Comms Networks</td>
</tr>
<tr>
<td>Managers make all decisions</td>
<td>DSS Empowered Staff</td>
</tr>
<tr>
<td>Field offices required to support personnel and forms</td>
<td>Wireless Comms</td>
</tr>
<tr>
<td>Personal contact with clients is vital</td>
<td>e-mail, tele / video</td>
</tr>
<tr>
<td>Manual tracking of progress/ movement</td>
<td>conference, EDI</td>
</tr>
<tr>
<td>Plans revised on fixed infrequent cycle</td>
<td>Ongoing adaptation</td>
</tr>
</tbody>
</table>

10.3 - Perceived Constraints and Remedies

An expert systems vendor felt that their product could facilitate the decision making. They proposed this to the organization, but were dismissed with the complexity of the problem cited as the reason. Undaunted, they asked the company to provide the information which went into the room and the decisions emerging from the room for a period of two weeks. They planned to use this to tailor an expert system capable of supporting the decision making. They configured their system and ran it alongside the expert group, provided with the same information. Within a couple of weeks, it was equalling their decisions, with some better calls. The shocking thing here is that the expert system contained just seven rules which they had distilled out of the data patterns.

- We can have systems which are centralised from a logical or control perspective, but decentralised from a technical perspective; or vice versa: A fully centralised system can support distributed decision making.
- Increasingly, organizations tend to put decision making at the coal face, where the most detailed situation knowledge is available. This can be backed up with data drawn from back office systems (such as payment history), external databases and directories, and systems which ensure that corporate policy is followed with respect to things like discount, pricing, purchasing limits, transfers etc.
In some cases we will find that the traditional branch or field offices are no longer required. Think how much smaller bank branches are these days: When did you last see an assurance company build a new branch or regional office? Forms can be distributed and collected electronically. There is no waste when they change, there is no delay for printing, there is no delay to get the new version into circulation - it can simply be downloaded onto the representative’s notebook when required.

Personal contact is not always vital - many of us would prefer to talk to our ATM than the bank manager!

Manual tracking of progress and movement, with human capture of information is not always required. We are aware of systems in mines where production is automatically monitored by recording the signals from radio equipped ore carriers descending into and coming out of an open cast pit. Other examples include cargo tracking systems that are fed by data from transponder equipped containers that report their position via satellite.

As for planning - this used to be a cyclical, high level, formal affair. The annual strategy session which was attended by senior managers, then conveyed to the troops, to be revisited again the following year. This can no longer be tolerated in an environment that innovates continually, runs on Internet time and has nimble global competitors. Planning is an ongoing dynamic process, which is as much about reacting to real time data and communicating ideas and responses, as it is about formal models and planning documents. We have worked with companies in the telecoms sector to implement “electronic strategy” systems which exploit web technology to provide an emergent, constantly adapting strategy which is immediately communicated company wide.

**Disintermediation and Reintermediation**

New mechanisms allow new models of interaction. Traditional middlemen such as wholesalers, importers, agents and the like are being squeezed out of their roles. Clients can access the vendors, products and services they want to directly, often more quickly and at lower cost. An intermediary who adds cost and not value has a very limited life span. On the other hand, the huge choice available to consumers and clients on the web in a global marketplace creates opportunities for new kinds of intermediaries. These may offer value by doing the legwork, comparing prices and offerings on behalf of the buyer and facilitating payment or other parts of the transaction, such as payment. Sometimes the commodity is information itself, when the new intermediary might be termed an infomediary - such as a portal that brings buyers and sellers together.
Principles of Business Engineering

An Example
Ford in the US is a very large manufacturer of motor vehicles. They had a traditional purchasing process, which went something like this:

**Ford Purchase Management - Before**

1. **Head Office**
   - Request
2. **Purchasing**
   - Purchase Order
3. **Accounts Payable**
4. **Supplier**
   - Goods
5. **Receipt**
   - Invoice
6. **Payment**
7. **Materials**
8. **Plant**

× 500 staff
× Complex reconciliation
× Very slow

**10.4 - Business process before reengineering**

- The plant personnel would determine a need for purchasing raw materials, components or services. They would request the purchase by communicating with a purchasing department at head office.
- Purchasing would make up a Purchase Order which would be sent to one or more selected suppliers, with a copy to Accounts Payable.
- Supplier would deliver goods to the plant. Plant personnel would check the received goods and reject any problematic deliveries (usually only in part). They would then generate a receipt advice to accounts payable.
- Suppliers would invoice Ford, and these invoices would go to the Accounts Payable department.
- Accounts Payable would try to match up original purchase orders, receipts advices from plants and invoices from suppliers. This required extensive and complex reconciliation - remember that there will be complicating factors such as back ordered goods where a supplier is out
of stock, substitutions of newer items (with different codes) for components originally ordered, incomplete deliveries or deliveries where some of the items are rejected, etc.

- Eventually, suppliers would be paid

The process was so convoluted and slow that suppliers, regardless of volume, were threatening to cease doing business with the organization. Serious indeed!

What Ford did to remedy the situation was to radically alter the process, supporting it with information, systems and policies implemented via these systems. In the new process:

- A computer system at head office taps into the production schedule database to determine what volume of production is required of various products. Next, using a bill of materials, the system determines what components and materials are required to produce the products. The system accesses the inventory levels for the various plants, also held on a database. The difference between inventory and requirements provides a reasonable order level for each item within a given plant. Having full inventory records can also prevent ordering by transferring items across plants

- Plant personnel generate purchase orders directly with the assistance of the system mentioned above. There are policy limits in place to determine who can order at what level and checks to ensure reasonableness against the production schedule and inventory. Purchase orders go directly to suppliers

- Suppliers deliver to the Plant. The plant personnel receive the goods and check them immediately against the purchase order that they generated. They record and short deliveries, substitutions etc. And authorise payment for all correct items received. This generates a cheque or electronic payment to the supplier immediately

The results were dramatic. A saving of 75% in staff consumed in the process, a dramatic improvement in delivery quality from suppliers and greatly reduced stoppages and outages at the plants. Ford went from being last on the list for suppliers to first - they quickly learned that if they delivered the right stuff quickly, they would get paid immediately. This, in turn, allowed Ford to reduce raw material stocks in the plant, cutting storage costs, capital tied up and damages and losses.
10.5 - Business process after reengineering

Steps to Follow

So how do we examine a business process and improve or reengineer it?

Michael Hammer suggests the following Principles:

- Organize around outcomes, not tasks - look at what we are trying to achieve at a fundamental level, rather than on what is done now
- Have those who use the output of the process perform the process - they are the most motivated: Think of how banks have successfully trained the populace to become unpaid tellers via the introduction of ATM’s
- Subsume information processing work into the real work that produces the information. Integrate the IT component into the work without introducing extra steps or capture processes. Use technology to capture information automatically as a by product wherever possible. Ensure that information need only be captured once and will then flow reliably into all processes, systems, decisions and reports that require it. Consider innovative use of technology such as bar coding, transponders, mobile devices etc.
- Treat geographically dispersed resources as though they were centralised
- Link parallel activities instead of integrating their results - this allows for a higher level of coordination and early warning if one process is delayed or problematic
• Put the decision point where the work is performed, and build control into the process - as was done in the Ford scenario discussed above. The people on the spot have the best chance of knowing all the relevant facts and making a good decision, provided they are supported by good information and empowered to take the decision. Their discretion level and important corporate policies can be implemented by business rules in the support systems.

• Capture information once, and at the source

Another Example

In the case of Mutual Benefit Life, a new CEO was brought in by the board to rescue the business, which was losing market share quickly and becoming unprofitable. Their old process of dealing with an application could go through as many as 35 steps in some 15 departments, taking several weeks.

BPR at Mutual Benefit Life

The “new broom” radically altered the approach with the following innovations:

• The old sequential processes where every application went through the same steps, regardless of whether it needed them or not was discarded

• Case Managers were appointed with the sole mandate to accept good business. They are rewarded based upon business accepted and staying in force (not being abandoned by clients or lapsed for non-payment) and penalised when claims are too high. Many of these personnel now earn
2-3 times what they previously did. Case Managers have the final say in whether a case is accepted or not

- Case Managers can draw upon whatever experts they deem necessary from a pool of these skills. They can chose to consult a physician if they are not sure about medical issues; or an underwriter for further information on risk or statistics. Many of these specialists came from departments which previously reviewed every case
- An expert system was implemented which could draw on information from existing back office systems related to clients, contracts and products. This could assist the Case Managers in making good decisions.

The results were interesting:
- Turnaround improved dramatically, to where routine cases now take between 2 - 5 days and special cases can be processed in a matter of hours
- 100 field staff positions were eliminated
- Double the amount of work could be processed

Given all the above, BPR looks very attractive, but there are some other issues to consider too.

**Things to Respect**

Davenport & Short, Sloan Management Review, 1990 suggest:

- Develop business vision and process objectives - we do not want to alter things in undesirable ways or ways that conflict with business objectives. Also, having a *shared* vision is important in getting everyone moving in the same direction.
- Identify processes to be redesigned - We need to understand what we have now before we meddle. Often, processes that appear illogical or convoluted and ripe for reengineering are not so illogical as we might think. When we get down to the real issues at the coal face, there may be very good, very subtle reasons for doing things the way they are. We advocate modeling and measuring before you meddle. This has two further advantages:
  - We can confine our efforts to high payback areas (processes that are very ineffective now, or where major improvements can be realised with relatively little negative impact)
  - It reassures people who may be adversely affected by the restructuring that due diligence is applied and that no rash decisions will be made.
• Understand and measure the existing processes to establish a benchmark against which you can judge proposed and newly implemented processes

• Identify I.T. Levers - Where can technology help? This is often through automation of capture, sharing of information, rapid analysis of data to gain insights, high speed computation, or effective communication

• Design and Prototype the new process - Building new processes is a challenging design activity, requiring innovation, creativity, analysis skills and rigour. Prototyping is important to prove whether the ideas will work in practice.

To which we add:

• Ensure consultation and buy-in of affected parties. Successful reengineering is as much a marketing and communication process as a technical one. If we do not gain the support of people in the functions affected and their management, we are very unlikely to be successful

• Measure effectiveness of new process as "proof of concept" - This takes emotion out of things. If we can demonstrate that the new approach is clearly more effective or efficient, then we have a strong case for acceptance

• Ensure scalability and repeatability to ensure that the solution is viable in production environments at full volume and with tight time constraints. Check too whether the skills required to operate the solution are available where you will need them

• Market and routinize - this is about getting people involved, excited and psychologically ready for change; then setting things up so that the new approach becomes the norm

Balancing Innovation and Improvement

A key issue in improving the performance of an organization is balancing innovation and continuous improvement activities. Innovation is a very “Western” idea. It is characterised by new ideas which usually involve radical change. When successful, it can achieve dramatic improvements: say 2 or 3 or more times the productivity or output of a previous technique. This is very attractive, but there are also problems: Innovation comes from creative thinking and is inherently unreliable - we cannot demand that someone has a brilliant idea every Monday morning! Innovation is also invasive - because it is radical it normally turns our existing processes upside down, often invalidating infrastructure investment, current processes and existing skills in the process. Because it requires new processes, equipment, investment, training etc. Innovation is also usually high risk. Examples of
innovation based approaches include Business Process Reengineering (as originally promoted) and Rapid Application Development.

**Innovation**

- New Ideas
- Radical
- n x change
- Unreliable
- Invasive
- High Risk
- "Western"

**Continuous Improvement - Kaizen**

Continuous improvement, or what the Japanese call Kaizen, is a different approach. It does not rely on major breakthroughs or innovation, but rather continuous, small, incremental improvement of existing designs, processes and skills. It is about doing things just that little bit better every time. It is not invasive, is reliable and it is low risk. On the other hand, it will typically result in small improvements: 1%, 5% or even .1%. Applied on a continuous basis, though, little bits add up to a lot. Kaizen thinking is behind approaches such as Total Quality Management (TQM) and the software productivity improvement models of the Capability Maturity Model (CMM) and SPICE.
10.8 - Kaizen - Slow continuous improvement

**Coupling and Harnessing Both Techniques**

Ideally, we would like to harness the power of both techniques, using each where appropriate in a complementary way. Within *Inspired*, we have determined a model for achieving just that.

You may need to do an initial review to determine which processes (or other items considered for improvement) are satisfactory and which are problematic. Where there are serious problems, we may make immediate interventions, sometimes using an innovation approach, but also drawing upon best practice from other situations. The initial intervention may involve a major change and high risk to begin with.

Our examination and suggested changes should be the basis of a framework for understanding the performance of the organization. The framework needs to cover the activities, the products (deliverables), the resources (human and other), the goals and the current performance levels. It is also very useful if we can find industry benchmarks against which to compare our performance. The framework should allow us to see how tasks and deliverables impact upon each other.

We can use the framework to determine the extent of proposed changes. Something which affects just one deliverable within one process and a limited set of resources will not have a major impact. Things which affect multiple products across multiple processes will obviously have high impact and must be approached with caution. We generally want to apply innovation in cells and Kaizen both in cells and to the overall processes.

Where innovations extend beyond one area, we need to approach them in a structured way. We should do this with an affordable fraction of our
resources, and pilot it to prove viability before rolling it out on a large scale. Below we provide a model for achieving this.

**10.9 - Sustainable Innovation Model**

- As an example, let’s discuss making improvements in a software development process.
- At the core of the model (centre row) is the definition of the current process and product, as well as norms and historical performance information. For our example, defined processed would include analysis, design, coding, testing etc. Defined products would include things like object models, process models, prototypes etc. Norms and history would include things like the productivity and error rates achieved on previous projects. This information is managed by a team of people responsible for Process and Methods support. They should represent about 8% of our resources.

  This group provides the current methods and deliverable standards to project teams. They also collect results and measurements from ongoing project work to enrich the database, to validate quality and to provide early warning of negative deviations. They can also glean valuable learnings from the project activities and update the method and model definitions for future use. This is a continuous improvement cycle.

  - The project teams (lower row) use the established methods and processes and produces deliverables conforming to the established standards. Their work will be evaluated with respect to past performance, and benchmarks
from inside and outside the organization. People working in this mode should represent about 87% of the total.

- Where innovations hold potential for improving the delivery of the organization, they should be investigated by a research group (top row). These are highly skilled individuals, representing about 4-5% of your resource who will develop and evaluate innovations. They should research, stay up to date, try things out and determine which innovations have major potential for the organization. Ones that do should be piloted and tested for applicability to the organization. This group would have a responsibility to see how innovations can be integrated into the established approaches and deliverables of the methods and processes group. They should, together with the latter group, update the standards and norms and support project teams in early adoption.

The model shown provides a way to capture innovations and leverage them with minimal lost productivity and trauma to the organization.

**Beware! Caveats for Business Engineers**

Despite the many success stories, a great many BPR efforts fail. Studies conducted by Davenport and others have highlighted some of the major causes:

- Lack of management commitment and follow through
- Targeting too many areas
- Lack of measurement
- Jobs are threatened
- Loss of core competencies
- No change from command and control to empowerment
- Lack of adequate or timely I.T. support
- Lack of involvement by stakeholders at all levels
- Internal rather than external focus
- Not doing the necessary training, counselling
- Not changing the incentive systems

There are major benefits to be achieved if business re/engineering is done intelligently, well and sensitively. There are major disasters in waiting for those who rush in ill prepared.

163
References and Further Reading

Comcon, 1986, The Tetrarch/1 Strategic Information Planning Methodology, Comcon (Pty) Ltd.


Frank, Ulrich, 1994, Memo: A tool Supported Methodology for Analyzing and (Re-) Designing Business Information Systems, in Ege, R; Singh, M; Meyer, B (Hg): Technology of Object Oriented Languages and Systems, Prentice Hall pp 367-380


Jacobson, Ivar; Ericsson, Maria & Jacobson, Agneta, 1994, The Object Advantage: Business Process Reengineering with Object Technology, Addison Wesley


Taylor, David, Business Engineering with Object Technology, John Wiley
CREWS project

The Esprit research programme in Europe often sponsors innovative research into methods of improving software project effectiveness. One of the earlier initiatives resulted in the production of the Euromethod. A more recent project (CREWS: Customer Requirements Elicitation With Scenarios) has resulted in a new way of capturing specifications which takes advantage of the convergence of entertainment technology with computing.

One of the historical problems encountered with the capture of user specifications has been the recording of user input by analysts. This has traditionally been done in text or diagrams. The former can specify policies well and the latter is able to capture much information succinctly, but both can lose the subtleties of what the user really said. There is also much non-verbal or structured communication which takes place when humans communicate: facial expressions, emphasis in speech, hand gestures etc. These cues can alter the sense of what is said or impart differences in priority or shade of meaning which may not be translated by analysts to the written specifications.

It is therefore advantageous to capture the real conversations with users in the form of audio, or better yet, video. With current multimedia technology, this
need not be an expensive or difficult task. Inexpensive video camera’s or web cams can be used to digitise conversations. These can be stored on computer in standard formats, such as Quicktime (from Apple) or AVI (from Microsoft). In this form, they can be attached to documents or graphical models as annotations. A simple double click of the mouse can load and run the original interview. This can enable a designer, programmer, tester or future maintainer to gain a quick and exact understanding of an original requirement.

**OLE/OpenDoc**

Microsoft desktop technologies, in the form of Object Linking and Embedding, make it possible to link or embed recorded material into narrative or structured specifications. This can be done in two ways:

- **Embedding** where a copy of the material is inserted into the file which contains the text or diagram and stored with it
- **Linking** where a reference to the file is held in the text or diagram file, while the multimedia content is held elsewhere in its own file

Each has advantages and disadvantages, as follows:

<table>
<thead>
<tr>
<th>Embedding</th>
<th>Linking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pro</strong></td>
<td>Only one copy of the content is stored, saving disk space</td>
</tr>
<tr>
<td>Can distribute a document and its embedded content as a single file</td>
<td></td>
</tr>
<tr>
<td>Content cannot easily get “lost” by being moved from where it was originally located</td>
<td>As the content can be included at various locations by reference, updating the one copy updates it wherever it is referenced</td>
</tr>
<tr>
<td>Extra disk space is used if the same content is embedded in several “host” documents, since a copy is stored within each</td>
<td>Must be careful to distribute the content files along with any that reference them</td>
</tr>
<tr>
<td>If one copy is updated, others are unaffected. There is thus higher maintenance effort if content is subject to change</td>
<td>Links between “host” and “target” files can be lost if files are moved around</td>
</tr>
</tbody>
</table>

Similar effects to OLE can be achieved across heterogeneous platforms (and even with documents spanning multiple network locations) by using OpenDOC, an object oriented distributed document management technology
produced by Apple, IBM and collaborators. Some implementations of this kind of technology are very sophisticated. For example:

- The Lotus desktop applications (Freelance Graphics, WordPro, 123 etc.) Have extensive team collaboration utilities and features, allowing documents to be jointly authored, reviewed, consolidated, and so on
- The Sepia multimedia authoring system allows simultaneous distributed authoring, editing and viewing of documents

**Browser with Hyperlinks**

We can achieve significant multimedia linking on a distributed basis quite easily with judicious use of the Internet (or web technology on a LAN, an Intranet). Most browsers will handle links to files, which they will open natively, or with the help of “plug ins” or helper applications.

**Persistent Conversations**

Persistent conversations is a concept where collaborators (in our case usually analysts, designers, architects and business domain experts) engage in conversations to establish shared understanding. These conversations are recorded (i.e. Made persistent) through a variety of approaches, including:

- Audio recording, possibly with transcriptions produced later
- Video recording, possibly with editing and annotation later
- Conversations performed through a medium such as e-mail, or online relay chat on the Internet, or another group support system such as Groupwise or HyperBoard where extensions to the mechanism of communication can record the information conveyed electronically in real time

Persistent conversations can be useful from a variety of perspectives

- We can replay them to clarify detail that might otherwise have been lost forever
- We can annotate them or edit them to add meaning or enhance the clarity of communication
- We can distribute and share them readily with people who were not part of the original discussion
- We can analyse them at leisure to extract more detail and meaning
• We can use them as a record of decisions, even as a witness if things get legally difficult
• They encourage quality of interaction, since participants know that their contributions will be preserved
• They can be linked into pertinent parts of specifications or points in models to convey a sense of “being there” or “getting it from the horses mouth” often not available to detail level developers

Notation

Within our approach, any model can contain a multimedia artifact. They are represented by a CD symbol inserted into the model. This will have a hyperlink to the relevant rich content. They should be attached to the most relevant model element, much as we do with rules. Several symbols can link to the same multimedia file.

User Interview
Jan 2000

Advantages

Multimedia specifications can greatly enhance communication. Improved communication improves quality and business fit of the delivered solution.

Scenarios

Scenarios are different possibilities that may come to pass. In system terms, they relate to ways in which a user may approach a task in interacting with the system, or performing a business process. Taking a customer order may have several scenarios, for example:
• The normal case where the customer is known and the product is in stock
• A situation where the customer is new, but stock is available
• A situation where the customer is known, but there is no stock available
Of course, there could be many more permutations related to credit and other factors.

We can discuss scenarios via persistent conversations. We can also document them via Business Process Models. Finally, we can mock up what they will look like in system terms and what the user will experience by building suitable prototypes and providing walk throughs which present the various interactions in the scenario sequence.
References and Further Reading

Crews Project Home Page: Cooperative Requirements Engineering with Scenarios [http://www-i5.informatik.rwth-aachen.de/lehrstuhl/projects/crews/]

Papers from the Persistent Conversations Minitrack of the Hawaii International Conference on System Sciences, January 2001:

- Radicals of Presentation in Persistent Conversation, by A. Bregman and C. Haythornthwaite
- Backchannel: Whispering in Digital Conversation, by S. Cogdill, T. Fanderclai, J. Kilborn, and M. Williams
- System Features of an Inhabited 3D Virtual Environment Supporting Multimodality in Communication, by I. Naper
- Profiling Turns in Interaction: Discourse Structure and Function, by S. Condon and C. Cech
- ‘Making Conversation’: Sequential integrity and the Local Management of Interaction on Internet Newsgroups, by D. Reed
- In and Out Scream and Shout: An Internet Conversation about Stock Price Manipulation, by J. Campbell
- Supporting Verification of the Comprehensive Nuclear Test Ban Treaty through a Persistent Conversation Interface, by S. Guerlain, D. Brown, J. Green, B. Hashemi, R. Rauschenberg, C. Stacy, R. Mason, and J. Bohlin
- Spaces, Traces and Networked Design, by M. Perry, R. Fruchter, and G. Spinelli
- Conference Scribe: Turning Conference Calls into Documents, by P. Wellner, D. Weimer and B. Arons

Prototyping

Definition

A prototype is a model of the real thing which will exhibit some (but not all) of the characteristics and behaviour of the final product.

Types of Prototype

Requirements Prototype

In systems work, requirements prototypes normally take the form of screen layouts (sometimes with logic and populated with representative data), reports, interfaces to other systems and scenarios or dialogues which link various elements together.

Technical Prototype

Technical prototypes may include new hardware, new software, new architectures or combinations of these. They are frequently built to prove the viability of an idea or proposed solution.
Evolutionary Prototype

This is where the prototype is intended to evolve into the finished article. We might use this approach where we are developing an application in a spreadsheet, for example. The prototype will include all the essential data items and formulae, but would not have all the validation, error handling, formatting and other sophistication of the final product. The core functions could be built and tested (maybe even used in production) and the other facilities will be bolted on later. We should take great care with evolutionary prototypes. They are relatively manageable in a declarative development environment like a spreadsheet, or in a high level object oriented and dynamic environment, like Smalltalk, but they can quickly lead to a product with a bad architecture going into production in more procedural/traditional environments where design is still a critical activity.

Throw Away Prototype

This is where we use the prototype just to understand the problem better and elicit good requirements. We have no intention of evolving it into a production system.

Uses of Prototyping

Prototyping can be used (or abused) for a wide variety of purposes. Good ways and places to use prototyping include:

- Reduce risk by proving technical viability
- Obtain more detailed requirements
- Improve the user interface
- Evolve functionality in "fuzzy" situations
- Try out new development approach / tools on subset of the problem

Dangers of Prototyping

Prototype becomes Production

Unfortunately, prototyping can also become a problem. It is fine if we realise that the prototype is just that - a mock up of the real thing, but not the real thing. It is like an architect’s model of a building. It looks much like the real thing but is not fully functional. Just like we would not want to live in a
cardboard model of a building with no plumbing, insulation, security or electricity, we would not like to live with a prototype system lacking validation, robustness, scalability and full functionality.

The problem is that, while just about anyone can tell the difference between a cardboard house and the real thing, it is much more difficult to tell the difference between a software prototype and the real thing. A good prototype, to all intents and purposes looks the same! It is very hard to resist pressure from users who have seen what they want to take the prototype into production. Beware of doing this, as those things you intend to “fix later” will likely never get done. This leads to unstable systems, lack of scalability, problematic support and maintenance and difficulties in adapting applications to future requirements. The bottom line is that, if we intend to use prototyping, we should educate our sponsors and user community to understand what a prototype is, why we are doing it and what its limitations are. They should also be preconditioned to expect that there is still a major amount of work to do from having a successful prototype to having a finished system.

If you intend to evolve a prototype into production, it is much better to rather do incremental, iterative development where:

- We have a reasonable idea of the full eventual requirement up front
- We build a small subset of the system using good modeling and prototyping techniques, test it thoroughly, expose it to users, and every so often release it into production
- We then build another set of functionality to expand the system and repeat the process, growing functionality with each increment.

The difference between releases and increments is that releases go into production. We may have an increment say every two to four weeks. These go through the whole lifecycle, including integration and user testing, but are not deployed into production. Releases will be less frequent, maybe every three months and will go all the way to production.

The following figure illustrates the difference in principle (of course there are many shades of grey, not just A or B in practice - we want to illustrate a principle)
12.1 - Comparing Prototype and Construct with Incremental Development

Typical prototyping concentrates on the interface, look and feel and what functions the user will have access to, populates the business logic thinly with enough dialogue management to string it together and maybe puts a little demonstration data behind the scenes. Over time the business logic and domain model, as well as back end services are developed.

Incremental development does all layers for a small scope first, then over time expands the scope from the core. This will generally result in a much more robust application, which has evolved according to real world needs and experience.

Prototyping in a Vacuum

Some practitioners have embraced prototyping as a substitute for structured methods and modeling, which they perceive as too slow. There are major dangers here, especially in not adequately understanding the scope and functionality of the application. If we just sit down and begin defining screen layouts and attributes and reports, we run the risk of getting too much depth in some (usually well understood) areas and too little coverage of other less well understood areas. The coverage is spotty and inconsistent.
12.2 - Dangerous Prototyping

On the other hand, if we first understand the scope and functionality required, through doing a context diagram, possibly supported by functional decomposition or an analysis of the various interactions of the proposed system with its environment, then we can prototype to very good effect. Prototyping used in this way can verify requirements, flesh out detail and provide a quick method of evaluating alternatives.

12.3 - Useful Prototyping

Expanding Requirements

Scope management of projects is hard enough. Prototyping can make it even more difficult, if we do not define a scope up front and manage / limit any expansion actively. Once users get excited by what a proposed system could
do for them and see the possibilities, they will tend to try and get it to do everything. Clearly, this is a recipe for disaster, where the prototype and the scope keep expanding, while little solid engineering is done and no workable, robust system emerges.

**Summary**

We are pro prototyping. We are also advocates that it should be done for the right reasons, in the right way and with a clear understanding of the pitfalls. Properly used within an incremental and iterative lifecycle it can materially enhance the quality of requirements and proposed solutions while reducing risk and shortening delivery times.

**References and Further Reading**

Event Modeling

Origins and Purpose

We originally used the Martin/Odell event models as described in the book “Object Oriented Analysis and Design”. The definition of an event used is: “The change of state of an object”. These proved very effective at capturing the detailed processes to be implemented at systems level. We integrated the approach with mapping to an architecture based design model in the downstream part of the method and with business process models from analysis. When UML became an accepted standard, we adopted the UML activity diagramming notation for our system level event models, and extended it for business process modeling as described in earlier chapters. What we present in this chapter is thus an expansion and formalisation of what we covered earlier in business process modeling.

The relationship between the levels of dynamic model is shown in the figure below. At the Enterprise level, we created a stakeholder model. There is just one of these per business unit, although we could of course produce subset views of this for particular scope projects where some stakeholders are not relevant. From the stakeholder picture, we identified external events which trigger activity (business processes) within the organization. We documented these business processes via the business process architecture and more detailed business process models. There is typically a business process model per external initiating event.
Where necessary, we reengineered or improved the business processes, to produce suggested new processes, expressed in the same modeling notation. Frequently this would involve increasing the level of automation and systems support. Once we are happy with the proposed business process, and have a domain model in the form of a class diagram, we are in a position to begin system level modeling. This activity can precede, follow or run in parallel with prototyping. We usually prefer to do them in parallel.

**Linking Business and Event Models**

13.1 - Relationship between dynamic models

**Selecting Computerised Components**

The notation used in the business process models allows us to easily identify computerised support - viz. All activities which are shown as computer supported or fully automated. In addition, we can easily see which agents/actors in the environment will interact with the system, and what inputs will be received as well as what outputs are expected. From this, we can easily extract the computerised elements requiring detailed design. The next step is event modeling, which expands the system subset models to include a more formal notation for how activities affect underlying business objects (in the domain model). There will normally be one event model for each interaction of a user with a supporting system to accomplish a single purpose. This may involve several physical input screens, i.e. A dialogue with the computer. The idea is not dissimilar to the concept of a Use Case in UML where these are drawn at a system level.
Outcomes

On higher level models, we showed outcomes fairly informally. E.g. We might just say that something was checked by a process and the result was OK or NOT OK. We noted these as two possible outcomes triggering different ensuing activities in the model. To make our specification more rigorous and to validate the dynamic model against the static model (domain model), we now become more formal in the specification of outcomes. Specifically, we document the effect of the activity on a domain object type (class) by recording the state that the object will reach. The notation is as follows:

\[ \text{Object Type} \ [\text{State}] \]

E.g.

CUSTOMER Updated
ORDER Created
STOCK Decreased

Outcomes on the graphical models are updated to reflect this more formal view.

As with the less formal ones on business process models, the outcomes may be independent, as above, or mutually exclusive, as below.

Typical Outcomes (Event Types)

Typical event types that we will be interested in include the following:
• An object is created
• An object is deleted
• An object is reclassified
• Instances of collections are added or dropped
• The state of an object is changed by updating attributes
• An external event is processed or initiated
e.g. We get input from the screen or change the state of a device (e.g. close a communications port)

Decomposition and Expansion

We will find that certain activities result in multiple events, i.e. They alter the state of several types of underlying business objects. Where this occurs, we should decompose those activities further, until we reach a level where the activities affect only one type of object (class).

This has several advantages:
• It provides an exact guide to the modeler as to when to decompose and when to stop, which was often lacking in previous methods (e.g. Functional decomposition or data flow diagramming)
• If an activity affects the state of only one object type (class), then surely it is a candidate for a method? We have the beginnings of our mapping of responsibilities to domain classes. We will return to this in a later chapter on mapping to an architectural design pattern.
Pre- and Post-Conditions

We may wish to specify conditions which “guard” access to the activity. They must be met before that activity will run. They are shown as a rule, with a diamond symbol, at the entry to the activity, as below. We will discuss rules in more detail in a subsequent chapter. A post condition would be specified on exit from the activity, and could say under which conditions the activity will complete, or could specify under what conditions the activity will notify the system of an event of interest to other parties. We can also have derivation rules specified within the activity for how results will be arrived at, e.g. A calculation formula or algorithm. Finally, we can have a rule which is attached to the diagram as a whole, to express more complex business policies which may affect the overall process.

![Diagram: Request, Preconditions, Result]

13.3 - Example of a precondition rule

Design Level Models

As we transition our logical level business oriented models to design, we need to consider a number of issues. We discuss these below.

Technical Objects and Events

We may need to add extra activities and events dealing with issues such as security, integrity and auditability. Examples would include logon procedures, logging of updates, writing to an audit trail and so on. In some cases, we will need to add new items to our static model e.g. We may need a class for Authorisations to link users to permitted activities. We would add these to our class diagram. This is now evolving to be more than a domain model, to include system level classes which may not necessarily form part of the business domain. Nonetheless, the classes added will use the same notation and adhere to the same principles as domain classes. It is often useful to distinguish domain and more technical classes in some way, so that the latter can be suppressed when we want a higher level model to discuss with sponsors and user personnel, without the clutter of more technical objects.
At a detail level, we may want to wrapper some of our system components, possibly for distribution across platforms. In this case, we might see the remote interface component as an object which will generate a variety of events that the logic will respond to.

**Input and Output Specifications**

On high level models, we were content to name the input or output. Now we must become more rigorous. At a minimum, we must know the details of the data items which are input or output, their types and the medium through which the input or output will occur. We may document these details in the form of a view, as shown below. A view is not unlike a class, but usually without behaviours. In some tool environments, we have documented these views in UML static model diagrams, using a stereotype of `<<View>>` to distinguish them from domain classes. The data items mentioned should, of course be in the dictionary or repository where they can be typed, described and have legal values and ranges specified.

```
<<View>>
Student Course Result (SCR1-GUI)

StudentID:aStudent
CourseID:aCourse
MarkAttained:aPercentage
Date:aDate
Remarks:aText
RecordedBy:anInstructor

Medium:online capture from graphical user interface
```

13.4 - Example of a view definition

The view should have a unique reference, which we can use to link it to the flow on the graphic model.

```
GUI
  ^
  |
---\---
SCR1
```

13.5 - Referencing IO Specification on the model
Where prototyping has preceded this stage, or is happening in parallel, the view can be equated to a prototype artifact, such as a screen, report, or other layout. One example of the latter would be the list of elements generated when a magnetic access card is swiped at a doorway.

**Frameworks and Infrastructure Components**

This stage of our development process is a good place to look for and identify frameworks and infrastructure components which we can include in our solution. For example, we may chose a framework which provides a lot of the menu system and user interface management for us in a consistent way. This could save a lot of work in building the system. Examples of other components that we may want to incorporate include:

- *Security* related components
- *Encryption* and decryption facilities
- *Compression and decompression* of rich media
- *Interfacing* components that allow us to address specialised devices at a high level of abstraction e.g. Interactive Voice Response technology to access account balance information; speech recognition software; controlling a manufacturing robot
- *Persistence* frameworks to deal with long term storage of our domain objects (we cover this in a later chapter)
- *Middleware* that can assist us in implementing multi-tier solutions in heterogeneous technology environments (also addressed in later chapters)
- Dealing with *complex protocols*
- *Best practice algorithms* for problems such as scheduling, least cost optimization, best fit algorithms for cutting components from stock and the like
- *Workflow* technology to create a process view on existing applications
- *Reporting and enquiry* functionality which can eliminate much development work while enhancing system flexibility

**Logical Transactions**

For integrity reasons, we normally want a business transaction completed as a whole or not at all. Consider a *Sale* process which will affect objects as follows:
• Create a Sale object
• Update the Customer Balance, adding the sale amount
• Decrease Stock on Hand for the Product sold
• Credit an Agent with a Commission payment.

We do not want a situation where, due to hardware, power or other failure, some of the above happen, but other parts are not completed. Database management systems provide facilities to ensure that we can manage this problem, provided we mark the beginning and end of logical transactions. After we start a logical transaction, updates are recorded, but not made permanent in the database, until a corresponding end transaction is reached, at which point the set of updates is committed together, making the changes permanent. If the system should fail for any reason after the start transaction, but before the end, the changes will be lost, but the database will remain in a stable state with integrity preserved.

The program logic could also choose to roll back a partially completed transaction, thereby asking the system to forget what was done since the start of the transaction. We add logical transaction boundaries to our models to indicate which activities should complete as a unit.

**Event Model - Process Sale**

*13.6 - Event model with logical transaction boundary*
**Geographic or Platform Distribution**

A similar technique can allow us to indicate platform allocation (e.g. Client, Application Server, Corporate Server) or geographic split (e.g. Branch, Head Office).

**Capacity Planning and Performance Estimating**

If we fully specify activities and detail resources and volumes, the models can be used for capacity planning and performance estimating.

**Accommodating Existing System and Data Assets**

We seldom have a completely new, “green field”, application with no interfaces to existing systems or data. Much more common is that we will already have some data resident in some form of database, as well as a variety of functionality in a mix of technologies. Now is the time to identify which of our domain objects already exist in some form, and what that form is, as well as to identify which elements of our processes (activities) are already available in some automated form. Following this, we will need to devise ways to interface to these elements. We deal with this in the chapters on Persistence and Mapping to a Layered Architecture.

**Achieving Reusability**

One of our goals is to improve productivity and enhance quality through reuse. This is as much a cultural as technical issue. We need to actively seek elements to reuse, as well as identify opportunities within the elements that we will create, both to reuse components within the current development, but also to solve problems and create elements in a generic way, such that they may be reused in future. To facilitate reuse, a few principles should apply to each activity (method):

- Ideally:
  - An operation (method) does not know, and does not need to know what triggered (invoked) it
  - It does not know what operations are triggered by its events
  - i.e. Its knowledge is limited to receiving input passed, and changing the state of one object.

This allows us to reuse the operation from many different places in the application. It is sometimes called “contracting” since the operation “contracts” to provide a predictable result to those who wish to use it and
who provide the necessary input. There is a somewhat different and more formal mechanism to support contracts within the Eiffel programming language.

A further way to achieve reuse, is to identify common processing elements which can be common to a variety of subprocesses. In the example shown, there are common issues in processing all kinds of transaction (e.g. Checking authorisation, logging). These common elements can be held at a generic level and “inherited” by sub-processes. The hierarchy in the event diagram should match that of the domain model classes, allowing us to map the common issues into methods on the parent class, while mapping the specifics to methods on the subclasses.

13.7 - Common logic in event model

Here, all types of transaction will result in a transaction being recorded, but only some will result in additional events.

Summary

Event models provide an extremely powerful means of showing business processes, from a high level business view, right down to the detailed operation of the final system. This is important since there are no abrupt transitions of approach between the logical and physical dimensions of the system. This promotes user/analyst communication and ultimately results in applications which better match real world requirements.

Unlike previous flow diagramming methods (e.g. data flow diagrams) and decomposition charts, event models have a natural way of factoring functionality to match the corresponding data structures (held in the object model). Decomposition is controlled by the stipulation that each activity should affect the state of just one object type. This translates naturally to operations which become methods of the appropriate class. In a later chapter
we cover the translation of the higher level models into scripts in a layer of the design architecture called the controller. There is thus a natural and easy transition from high level business models to the architecture of the actual runtime design.

**References and Further Readings**

OMG, UML Specification Revision 1.4, [www.omg.org](http://www.omg.org)

Odell, James; Engineering Artifacts for Multi-Agent Systems, Presentation obtained from author, 1999

Dean Leffingwell, Don Widrig, Edward Yourdon, Managing Software Requirements: A Unified Approach, The Addison-Wesley Object Technology Series, 1999
Chapter 14

Business Rules

What are Rules?

There are many definitions for rules. We adopt a very pragmatic one:

*Rules are any statements over and above the graphicale specifications that we need to fully specify the required system.*

Where are Rules Relevant?

They can apply to dynamic (e.g. Business Process, Event), interface (Prototype, View) and static (Class Diagram) models. They help to provide clarity, rigour and detail to our specifications.

- Normally rules that apply to integrity management and relationships between domain objects, or enduring business policy issues, will be relevant to the static model (Class diagram, class definitions and glossary)
- Rules that affect processing, calculation, choice of action based upon outcome of a previous activity, time related triggering, or algorithms will normally apply to the business process or event/activity models. An
exception to this would be the calculation rule for a derived item on a domain object

- Rules may apply to the interface (user or other) when they specify validation, interdependencies between choices, dialogue flow and other items which directly affect the interface to people or other systems

Types of Rules

There are a variety of types of rules, including:

- **Integrity Rules** which help to maintain the accuracy of information and relationships
  
  Example: Order may be added only for an existing Customer

- **Derivation Rules** which specify how we derive the value of something
  
  Example: Stamp duty = Greater of (Initial Premium * 0.05) or $5

- **Behaviour Rules** which determine how something will behave under certain circumstances or based upon particular outcomes
  
  Example: Trigger RaiseInterest at End Of Month for InterestBearing Accounts

Linking Rules with the Models

We allow the linking of rules with any model using the diamond symbol viz:

Which can optionally contain a unique rule id. Rules can appear in a variety of places on models as illustrated in Figure 14.1.
On Static Structure Model

Employee -> Medical Aid Plan

EMPLOYEE may only be on MEDICAL AID PLAN if permanent and with service > 6 months

On Event Model

If StockOnHand < ReorderLevel and no PURCHASE ORDER issued then ...

StockOnHand = PhysicalStock - CommittedStock

When QuantityOnHand < TotalDailyOrders for PRODUCT issue to CUSTOMERS with PriorityStatus 1 first, place BACKORDERS for CUSTOMERS not satisfied

14.1 - Rules linked to models

Scope

The scope of rules varies greatly. It can be from entire business (global) to a calculation within an individual program or even method. Scope should be specified for each rule.

A Formal Rules Language - OCL

UML provides for the use of a formal rule language called Object Constraint Language. This is based on work done initially within IBM. It is a language with a formal syntax and rigour, but is not an implementation language. It is based mainly upon concepts from Smalltalk. It allows for the specification of operations (arithmetic, logical, string manipulation etc.), working with classes, objects and instances (create objects, destroy objects etc.) and dealing with collections (initialize, add members, locate members, delete members etc.) It is high level but can be parsed to generate executable statements in a target implementation language.
Specification or Executable Code?

If our technical environment supports this, then we may use rules in our models which will become actual runtime code or be interpreted by a rules engine or inference engine. Where we do not have these facilities, rules are still valuable as a concise and unambiguous way of specifying requirements and design decisions. They are then treated as specifications by programmers.

Structure and Notation

Within our approach, we allow quite a bit of flexibility. You can choose to treat rules formally or informally. You may want to use rules which are written in your target implementation language, or in a formal language such as OCL. Alternatively, you may want to treat them purely as part of the specification. We find that the latter is the most common for business systems. In this approach we still want to achieve a reasonable degree of rigour. What follows is a minimalist set of guidelines to achieve this.

Business rules are expressed as text expressions, where any nouns used must be defined in the domain model and should reflect a business object or attribute of a business object. All normal comparison operators (e.g. < > = etc.), logical operators (e.g. AND NOT OR..) and generally understood functions (e.g. Sum Of, Difference between, Product of etc.) may be freely used. Any additional operators required must be explicitly defined in terms of standard or previously defined operations. Business Object names (Class Names) are capitalised. Parentheses can be used if required. Rules may be nested. Rules may be referenced by a unique identity, which can be included on graphical models where the rule is applicable. This can include stakeholder models, domain models, process models and interface models.
Example

<table>
<thead>
<tr>
<th>Rule ID</th>
<th>Scope</th>
<th>Rule</th>
<th>Status</th>
<th>Implemented in</th>
</tr>
</thead>
<tbody>
<tr>
<td>OrderFullfillment1</td>
<td>Order Processing</td>
<td>When QuantityOnHand &lt; TotalDailyOrders for PRODUCT issue to CUSTOMERS with PriorityStatus 1 first, place BACKORDERS for CUSTOMERS not satisfied</td>
<td>Production</td>
<td>ORD101, ORD110</td>
</tr>
<tr>
<td>Discount2</td>
<td>Sales</td>
<td>Allow Discount only for CATEGORY 1 and 2 CLIENTS on PRODUCT with Margin &gt; 25%</td>
<td>Analysis</td>
<td></td>
</tr>
</tbody>
</table>

Quality Standards

Rules should have unique Ids. No nouns should be used that are not defined in the Domain Model. Business Object names should be capitalised. All operators and functions should be generally understood. Any unique functions required must be explicitly defined. All concepts should be clear to all participants in the analysis.

Associated Tasks

Rules are discovered at many different points in the analysis and design processes. Tasks which may uncover rules include (but are not limited to):

- Business Process Analysis
- Business Domain Analysis
- User Interface Definition and Prototyping
- Interface Definition
**Prerequisite Deliverables**

A high level Domain Model should be in place before defining rules, otherwise the vocabulary and common understanding of concepts is lacking.

**Tool Support**

Rules may be defined in a simple spreadsheet or database. They could be part of a CASE repository. They should be accessible across projects and their status should be clear. E.g. Is this rule in production or the current best understanding of an ongoing analysis activity? Production rules should be referenced to where the rule has been implemented in software if applicable. This will allow impact analysis and change control where rules require adjustment in future. If desired, rules can be held in the Inspired knowledge management tool, Archi, and linked to various elements of the architecture e.g. Platform, domain objects, data collections, application systems etc.
References and Further Reading

This page intentionally left blank.
Chapter 15

Object Support in Tools

Object Characteristics

In Chapter 5 we introduced the basic object orientation concepts. You will probably recall that these included:

- Encapsulation, Interface Definition
- Definition of types or classes
- Inheritance
- Polymorphism
- Collections and iterators
- Components
- Patterns
- Frameworks

Analysis, design and programming tools support these constructs to greater or lesser degree and with different levels of purity. If we want the advantages of object orientation, the better support we have in the tools we use, and the more pure they are in their implementation of these concepts, the better. In this chapter, we take a look at these and a couple of additional concepts, and discuss how they are (or are not) supported in various popular languages and tools. Some of the additional concepts we will come across include:
• **Contracts** which is a formalisation of what a module agrees to do and what it expects as input to allow it to perform correctly

• **Visual Construction** which allows the visual assembly of applications from visually represented classes or components, thus eliminating much traditional programming while ensuring that only compatible items are combined in legal ways. Chapter 22 will explore this concept in greater detail

---

15.1 - Evolution of OO Technology

**Hybrid Languages**

Hybrid languages are typically third generation languages that have had object oriented (or object based) features added to them. Because these features are added after the language design was complete, the implementation may not always be very clean.

**C++**

C++ is the most widely used hybrid language - primarily because the C language on which it was based was the most widely used third generation language for the development of software. It was developed concurrently with the Unix platform and spread widely throughout both academic and commercial organisations. Good implementations on the PC and its
promotion by Microsoft furthered the popularity of the language. There were two competing object versions of C, viz Objective C, which was used on the Next platform developed by Steve Jobs after his departure from Apple, and C++ developed by Bjorne Stroustrop. The former was actually better and cleaner, but the latter triumphed in the marketplace since it was implemented as a precompiler for normal C compilers, allowing vendors to quickly jump on the object bandwagon.

C++ strengths include its widespread availability, many mature tools from a variety of vendors, efficient runtime performance (it compiles down to machine code), portability across many platforms, and cheap distribution (since no runtime support environment is required).

Its shortcomings include a complex syntax, inconsistencies in implementation across different vendor’s implementations, low level of abstraction (meaning that we have to write a lot of code to get things done), and poor support for safe memory management: it allows pointers, expects the programmer to allocate and deallocate memory and does not support garbage collection of objects no longer used. The language also has no standard class library - although many vendors provide a variety of these. Finally, C++ has no inherent database support, but can achieve this through calls or libraries.

C++ is an effective language in the hands of highly skilled and experienced developers. It can be productive if you have a rich class library, but can be very tedious and low level if you do not. It is recommended for professional shops producing shrink wrap software, who can spread their development costs across many copies of the software and who have the skills and can invest the time to build rich class libraries.

Basic / Visual Basic

The most object oriented variant of basic in widespread use is Visual Basic from Microsoft. This is more accurately described as an object based language, since it does not properly support inheritance and polymorphism. Microsoft has steadily improved object support in the language, including features to support binary objects (e.g. Images, audio clips), definition of types, and basic collections.

Strengths include ease of use, cheap tools, widespread teaching and availability of literature. Visual Basic has good support for graphical user interface development in the Microsoft Windows environment. It can interoperate easily with other Microsoft languages. Database access is easily achieved through built in components. Variants allow the development of
client side (browser - VB script) and server side (ASP) applications logic for Internet-style applications.

Problems with Visual Basic include: inefficiency of execution (it is tokenised and then run interpretively - although compilers are now available), poor support for definition of real classes with inheritance, and a programming model which encourages binding visual controls tightly to the database layer. If the latter can be resisted (which requires much more discipline and hard work), then large scale systems can be successfully written. Use of data bound controls will typically inhibit scalability of typical VB applications. VB is bound to the Microsoft platform and is generally not portable outside the Windows operating systems.

Ease of use and teaching makes VB a popular commercial and first language. Much more skill and discipline in design is required to successfully write large scale, scalable applications with good performance and easy maintenance.

**Delphi/Object Pascal**

Pascal is one of the more recent third generation languages. It is a cleanly defined language with good constructs which encourages maintainable code. The most popular object oriented variant is implemented in the Delphi tool from Borland, which previously produced Turbo Pascal. Delphi uses extensions to Pascal to implement object orientation cleanly and supports this with a productive visual development environment, including good component libraries.

Strengths include a good development environment, good performance from native compiler, rich component library and enthusiastic support groups sharing their work. Delphi interfaces easily to databases and includes an object request broker (Visibroker from Visigenic). It is a good environment for commercial developers with a good balance between ease of use, productivity and flexibility.

Delphi’s main weakness is the fact that it is a single vendor product, leading to a strategic risk for organization’s that commit to its use for mission critical applications. That said, the technology is pretty solid and their is a fanatical user base, so if the vendor did go down, it would likely be bought and survive it’s parent.

**CLOS**

CLOS is the most popular object oriented variant of the Lisp (List Processing) language, popularised in the 1960’s for work in artificial intelligence. It is a small, powerful language with very good features for
managing complex data structures. CLOS requires good skills and abstract thinking, as well as a basic understanding of mathematics and set theory. With these in place, it can be a highly productive language.

Weaknesses include lack of widespread support in industry, no standard class library, and no standard database support.

**Object COBOL**

COBOL was for decades (and probably still is) the most widely used language for commercial applications. It was devised in the 1960’s to simplify business programming (Common Business Oriented Language). It is verbose, but relatively easy to write and maintain. It has good support for text mode interfaces, batch and online transaction processing, file management and database access. It is widely implemented by many vendors, standardised by ANSI and executes efficiently due to native compilers.

Object oriented COBOL was slow in coming. This is a pity since the language provides excellent support for the basic OO concepts, certainly better than C++. It is much better suited to commercial programming and is suited to easy writing of reliable applications with lower skills. There are two main variants, produced by Microfocus and IBM respectively. Its late entry to the market unfortunately meant that COBOL lost much ground to other languages in the OO space. It is still an excellent language. While verbose (as presented to the compiler), modern tools can hide this from the programmer.

Strengths include: ease of use, reliability, portability across platforms, runtime efficiency, potential of migrating many existing COBOL applications, and specific features to support commercial work.

Weaknesses are few, but include the lack of a large standard class library and lack of support from large number of vendors.

Object COBOL is recommended for those who already have a large COBOL skills and applications base. It can also be productively used to create reliable commercial applications where productivity is required without insisting on very high skills.

**Java**

Java grew up from a language named OAK, developed originally at Sun for use in embedded processor applications. It was designed to be smaller and more reliable than C++, while preserving much of the syntax and flexibility of the latter. A key element was the implementation of a virtual machine,
which would allow the partially compiled bytecodes to run on a variety of
platforms unchanged - the “write once, run everywhere” philosophy.
Vigorous promotion by Sun, and the ability of Java to be downloaded to run
in a web browser, catapulted the language to prominence. It is cleaner than
C++, highly portable and widely supported by many vendors. It has rapidly
become the second most widely used object language and may overtake C++
for the crown. Key changes for the language include the virtual machine,
explicit definition of interfaces, elimination of multiple inheritance and
pointers, and the inclusion of automatic garbage collection. These features,
together with its good development tools and a growing standard class
library, have made Java the language of choice for many commercial
developers, particularly those working in the Internet environment.

Sun has maintained the pressure and market growth by surrounding the
language with many additional facilities, including:

- Standard IDE, which, although basic, provides a working environment
  for development with each new release
- GUI and collection class libraries
- Standard database access in the form of JDBC (similar to ODBC)
- Java to Java communication over TCP/IP networks via Remote Method
  Invocation (RMI)
- A visual component model - Java Beans
- A server side component model - Enterprise Java Beans (EJB)
- Various other elements and standards which allow Java applications to
  richly interact with their environment. E.g. JINI, which allows the
  automatic unassisted networking of devices which are plugged into the
  same network, or find each other on a wireless network

Java has many strengths, including: Wide support, good tools, reasonable
efficiency, implementation of most OO features, suitability to Internet
applications, and a growing standard library. With Java 2 Enterprise Edition,
there is standard inclusion of a CORBA ORB.

Java has suffered from incompatibilities between releases and
implementations in the early days. The situation continues to improve as
both specifications and tools mature. Java will feel familiar to ex C or C++
programmers, but represents a significant learning curve for typical
commercial programmers. It is lower level than either COBOL or Smalltalk.
C#

Pronounced “C sharp”, this is a relatively new language (2000) from Microsoft. Microsoft originally wrote nearly all its own software in C or C++. Later they flirted with Java and produced a variant known as J++. This did not adhere fully to the specification of the language according to Sun. Microsoft argued that it had needed to alter certain implementation details to achieve high performance. Sun felt that Microsoft was breaching the terms of the agreement under which organizations were allowed to use the Java technology. They felt that Microsoft was trying to “hijack” the Java technology and sought remedy in the courts. Sun won the battle, but not necessarily the war. Microsoft rapidly lost interest in supporting Java thereafter and has now launched C# which it sees as a good language for implementing its own software. This, like Java, is heavily based upon the C syntax and produces byte codes for execution by a virtual machine. C# though, is also heavily tied into Microsoft’s tools and Internet architecture plans, with reference especially to .net.

Time will tell how C# is received by the market. It is likely to have many of the same strengths and weaknesses that Java has, with the exception that C# is single vendor and supports Microsoft communications and middleware technologies, rather than open ones.

15.2 - Comparison of some object languages
Pure Languages

Simula

Simula was the one that started the whole OO trend, back in the 1960’s. It was implemented as a preprocessor for the Algol language and was designed to allow the efficient definition of simulation software, where real world objects, and types (classes) of objects could be easily defined and set in motion in simulations. It was standardised by 1967. Objects were endowed with properties and behaviours and the language supported inheritance between similar types, as well as the concept of classes and instances. The community responsible for Simula has continued to work on advanced object tools, most recently producing the [Beta language].

This family of languages still offers good facilities for the definition of simulations and development using object models. It is not considered mainstream though, for database, transaction processing and other commercial type processing.

Smalltalk/Squeak

Smalltalk is the third most widely used object language after C++ and Java. It is still one of the purest and richest. It was developed in the 1970’s at Xerox Palo Alto Research Centre (PARC) by Alan Kay and colleagues. Dual goals included

- the creation of a language which would allow non technical people to interact with a computer using symbolic or domain concepts, rather than the technical detail required by the machine
- the creation of the first graphical user interface (GUI) environment to promote ease of use and high productivity for end users through the use of a paradigm based upon familiar objects and behaviours, represented by visual icons

Syntax came largely from Algol, Simula and Lisp. OO concepts came from Simula, but were extended further, to include the notion of Classes as first class runtime objects in their own right. Lisp provided the ideas of collection processing and iterators, to simplify working with groups of objects. A very powerful, consistent and clean language with a small syntax was developed. Smalltalk used a virtual machine to ensure compatibility of code across platforms. It was standardised in 1980 and has changed very little since. The class library has grown large and rich over the years. Modern Smalltalks typically ship with 600-1200 classes in the base product. Many libraries can add to this total.
Early versions were expensive, proprietary and performed relatively poorly. Later versions are available from a variety of vendors and range from free to reasonable in pricing. Smalltalk products typically have very sophisticated development environments which are tightly integrated.

The language has a significant learning curve, more for the class library than the syntax, but will reward the persistent with very high productivity and good support for reuse one proficiency is achieved. In recent years, the major class library (about 300 core classes) has been standardised across most popular variants. Performance has been optimised with the provision of just in time and native compilers.

Best current commercial products come from IBM (Visual Age for Smalltalk); Objectshare (VisualWorks & Visual Smalltalk) and Cincom.

Smalltalk strengths include: maturity, pure OO implementation, support for dynamic binding, rich collection classes, excellent class library and development tools, high level of abstraction and productivity, support from multiple vendors.

Weaknesses include the onslaught of Java for the mindshare of organizations and developers, causing the market share of Smalltalk to slip behind, while its growth in number of users still continues.

Some of the original developers have continued work in the spirit of the original Smalltalk at a new home - Disney. The result is an open source product called Squeak, which is a small, tight implementation supporting some 20 platforms and with powerful support for multimedia and Internet delivery. More information at: [squeak address]. The power of this environment is seminally brought home by one of the provided demonstrations, which, in one line of code will:

- Open a TCP/IP port
- Connect to a web site
- Download a MIDI music file
- Play it

Despite all the hype and promise of Java, Smalltalk is still our preferred language for rapid commercial development, as well as writing tools, including Internet work.

**Eiffel**

Eiffel is a language designed by Bertrand Myer from the ground up to support object orientation and contracting. It is unique in tightly defining
requirements and expectations in the language, along with the operational code. Early implementations were implemented as preprocessors, but later versions provide native compilers.

Strengths include the rigour of the language, tightly tying requirements to implementation, and good development tools.

Weaknesses include that it is a “one company” product for all practical purposes, although several variants exist. It has a relatively small, but fiercely loyal, user base.

Eiffel can be a good choice if you are not nervous about being on the leading edge, and have disciplined developers. Finding skills may present problems.

<table>
<thead>
<tr>
<th>Language</th>
<th>Lines/FP</th>
<th>Reuse %</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>128</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>COBOL</td>
<td>106</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>4GL</td>
<td>40</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>C++</td>
<td>29</td>
<td>40</td>
<td>4</td>
</tr>
<tr>
<td>Visual C++</td>
<td>18</td>
<td>40</td>
<td>6</td>
</tr>
<tr>
<td>Java (Visual IDE)</td>
<td>18</td>
<td>40</td>
<td>6</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>21</td>
<td>80</td>
<td>8</td>
</tr>
<tr>
<td><strong>Visual Smalltalk</strong></td>
<td><strong>8</strong></td>
<td><strong>80</strong></td>
<td><strong>24</strong></td>
</tr>
<tr>
<td>Excel</td>
<td>6</td>
<td>80+</td>
<td></td>
</tr>
</tbody>
</table>

*Caper Jones Data, with Interpolation*

*15.3 - Abstraction Level of Languages*
15.4 - Classes provided in some popular tools

### Choice of Tools

A good set of tools is so much more than just a language. A modern integrated development environment (IDE) should provide language tools such as editor, compiler and linker, but may also provide additional integrated features such as graphical modeling, class libraries, frameworks, debugging tools, version management tools and more. In our opinion, you should consider broad principles before choosing specific tools. Think about the following issues:

- **Purity vs Pragmatism**: How important is OO to you? Do you want to truly use features such as inheritance and late binding? Pure OO environments tend to provide great support for modeling concepts and dynamic behaviour at run time, coupled with good abstraction, but this may be achieved at the expense of performance or having to use a language which is not that well known.

- **Performance vs Power**: Languages with high levels of abstraction (e.g. Spreadsheets, declarative languages) allow us to achieve a lot with relatively little code. On the other hand, they restrict what we can do to what the language designers considered as necessary. Low level languages such as assembler and C permit us to do almost anything, since we can control the machine very explicitly. Their power comes at
the expense of effort, since a lot must be written in great detail to achieve relatively small results.

- Maturity vs Features. Often we have to go without the latest fancy innovations if we want an environment that is mature, tried and tested and therefore robust and reliable. If we want the latest features, we often have to put up with more errors and faults. Exceptions to this rule are environments which are both mature and feature rich, due to having extensive, well tested class libraries.

- Flexibility vs Ease of Use. Assembler provides ultimate flexibility, but requires high skills and long training, as well as much discipline and effort. On the other hand, a spreadsheet requires little training, but is only suited to certain kinds of problem.

- Multiplatform Support vs Features. If we want to operate across many different target platforms, we will encounter the “least common denominator” effect. In other words, we will only be able to make use of features and capabilities that are supported by ALL the platforms we are targeting, which is likely to be a subset of those offered by any one platform.

- Application Development vs Application Purchase. Writing applications is a risky and expensive business. Small wonder that, from a business perspective, we would rather buy than build. It is often frustratingly difficult, though, to find applications that really match the business and the problem. Frequently, the implementation of a large package can be just as much effort as developing the portions that we will use would have been anyway. The two activities require different skill sets.

- Tactical vs Strategic. Are we building applications merely as a support or are they vital to achieving our business aims? How much control do we need over our destiny and how much can we afford to spend to have that control? In similar vein, are we looking at small applications for point solutions, or something that must support multiple lines of business and integrate with a large applications portfolio?

- Proprietary vs Open. How important is it to us to remain “open” and be able to connect to or interoperate with systems and data which is not of our own making. Given the fluid nature of today’s enterprises, we suggest that openness is very important. It comes at a price, though, of not being able to use the latest innovation from a specific vendor until there is broad industry support. Proprietary solutions are often well integrated and effective, but tend to lock us into using a specific vendor’s offerings.
Choose Directions Before Products

<table>
<thead>
<tr>
<th>Purity</th>
<th>vs</th>
<th>Pragmatism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>vs</td>
<td>Power</td>
</tr>
<tr>
<td>Maturity</td>
<td>vs</td>
<td>Features</td>
</tr>
<tr>
<td>Flexibility</td>
<td>vs</td>
<td>Ease of Use</td>
</tr>
<tr>
<td>Multi-platform Support</td>
<td>vs</td>
<td>Features</td>
</tr>
<tr>
<td>Application Development</td>
<td>vs</td>
<td>Application Purchase</td>
</tr>
<tr>
<td>Tactical</td>
<td>vs</td>
<td>Corporate</td>
</tr>
<tr>
<td>Proprietary</td>
<td>vs</td>
<td>Open</td>
</tr>
</tbody>
</table>

15.5 - Continua for overall decision making

Strategic Choice

We should select an environment with multiple sources (ANSI languages, Java) OR very stable backing from a single source (Visual Basic, Forte). Skills should be portable to other tools (c-ish, Smalltalk, Java). Tool should interoperate with other parts of the infrastructure e.g.

- SQL, ODBC Data Access
- CSV, WKn, DBF Spreadsheet and PC Database
- OLE/OpenDOC Compound documents
- DDE, RPC, RMI Runtime data exchange
- .xls, .123 Spreadsheets
- DLL's Runtime loading of infrequently used code
- CORBA/COM/COM+/RMI Distributed object messaging
- VBX, OCX, ActiveX Microsoft components
- Java Beans and EJB's Java Components
- HTML, Java Inter/Intranet deployment
15.6 - Different tool options vs team skill/capacity

If you are developing relatively few copies of an application then it is cheaper and more sensible to use a high level tool. The savings in development will almost always outweigh the cost of the tool and the runtime overheads. Suitable characteristics of a toolset for this type of application:

- High level of abstraction
- Well integrated IDE with good source level tools and support for code and version management
- Integrated modeling tools
- Availability of libraries, frameworks and patterns for the environment
- Tolerable learning curve and skill availability

If you have the skills and capacity to do it, and the user base to spread it, then you can afford to build your own environment using low level tools, thus avoiding expensive runtime licenses and machine requirements. For developing shrink wrap applications which you wish to market inexpensively to a large user base, you may look for these characteristics in the toolset:

- No runtime deployment fees or licenses
- Efficiency of generated object code
- Support for a variety of delivery platforms from a single code base
Good availability of components and libraries

What comprises a comprehensive development environment?

Here, we will again draw upon the Inspired Architecture Frameworks, and discuss the model that deals with development tools and their interoperation. The diagram illustrates the various elements, which are then explained below.

15.7 - Development Tools Architecture Environment

This is probably one of the most complex areas in the whole architecture. It encompasses the gamut of tools used by modelers, analysts, designers and developers to specify, design, build, test, debug, tune and deploy applications.

We may need to define several variants of the architecture, depending upon the target platform on which the development will be done and the applications will run. Various tools are discussed in several categories below:
• In the center we have the information, libraries and repositories around which the whole process and various activities revolve. Its elements are discussed first:

• Runtime Libraries are compiled code which will be invoked at runtime. This may take the form of Dynamic Link Libraries (DLL’s) for example

• Component Library refers to compiled components which will be used at development and run time. They adhere to a recognised standard in terms of how they present their services and interface to the world. Examples would be IBM DSOM (CORBA compliant) or Microsoft ActiveX components

• Class Libraries hold source definitions of classes (object types). These could be business objects previously defined and unique to the enterprise, bought in generic business objects, or most likely, technical classes provided by a third party. An example of the latter would be the Microsoft Foundation Classes for dealing with event driven programming in the Windows environment

• Model Libraries are normally held in a special purpose repository environment, or in a CASE tool repository e.g. Rational Rose. Models are normally present for Object Definitions (Class Diagrams); Business Processes; Sequence of Operations; User Interface Prototypes and the like

• Behind the above is the Repository Manager, which looks after repository management and integrity. It may translate between various tools sharing a repository. It can also provide version management and control over sharing

• Tools to the left of the central repositories represent the primary development tools related to the selected programming language or integrated development environment (IDE). They include:

• Class Browser which allows a developer to peruse the available classes and their implementation to select ones useful to the current project

• Editor which allows the writing of new code and the modification of existing code

• Compiler which allows the translation of program source code to machine code for execution on the selected runtime platform. Alternately, or additionally, a language interpreter can be used to translate the program at (or just before) runtime. The compiled
approach provides better runtime performance but is more cumbersome during development, while the interpreted approach is convenient for rapid development, but incurs a performance penalty at run time. Some environments provide both options, or just in time compilation to achieve the best of both worlds

- The Linker (or Link Editor) allows the linking together of individually compiled modules into a single executable program. It typically also includes system or third party modules not written by the current developer

- Debugging tools assist programmers in locating and eliminating errors from programs

- Profilers and monitors allow us to observe or instrument the running of programs so that we can find errors or eliminate performance or efficiency problems

- IDL editors/compilers allow us to define the interfaces to our modules/components independently of the modules themselves

- To the top and left of the repository group, are modeling and automated process support tools. These include:

  - Computer Aided Software Engineering tools (CASE) which typically support modeling, prototyping and code generation. They assist analysts, designers and programmers by providing validation of models and across models, version and change management, team collaboration and other facilities

  - Application Frameworks provide pre-written frameworks and logic to handle certain types of applications, allowing developers to concentrate on the unique aspects for this project while making use of previously tested patterns, algorithms or application design elements

  - Computer Assisted Reengineering Tools (CARE) allow us to automate some of the cleaning up and redevelopment of old software. They can recover data definitions and models from file structures and application logic from program code. This is normally fed into a repository for amendment and forward engineering by CASE code generation

  - Process Modeling tools may be part of our CASE environment, but could also be independent or tightly tied to Workflow analysis and definition tools. They generally allow us to represent business process flows graphically, linking this to their effect on underlying
business objects, or to the invocation of various software modules or packages to accomplish the overall business process

- Other useful development tools are to the right of the repository column. These include:
  - Workflow tools allow the specification of business processes and the invocation of underlying application functionality. The tools allow the routing of work to responsible parties in the organization, load balancing and the monitoring of work for service, performance and productivity
  - Wrappers are a means of creating a desirable interface to legacy or incompatible software components. We can use them to ease our migration to object or component based technology. They allow us to continue to use (leverage) existing applications, while creating a new, cleanly architected environment. By providing an interface compatible with the new, they allow us to slowly and seamlessly replace the old without undesirable side effects
  - Component Assembly tools allow us to build (construct) application solutions rapidly from predefined components. This may be accomplished in a visual environment (e.g. IBM VisualAge product)
  - GUI Painting tools allow us to visually create a user interface by dropping predefined components onto a palette
  - Internet and Web development tools are required to create web interfaces and web standard content which makes use of web standards such as HTML, GIF and JPEG.
  - Database related tools are lower right, second last column. They include the following:
    - The database management system (DBMS) engine itself. This provides an application programming interface to the developer, and provides facilities to help manage storage, access control, performance, recovery and navigation
    - Query and Report Generation tools may assist by eliminating the need for add-hoc programming for non-routine reports
  - On the right, there are optional development tools including:
    - Rules Editor which may allow us to define business rules independently of individual applications or processes. These rules would be actioned at run time by an inference engine or expert
system. They may be interpreted by sophisticated user written applications as well

- Prototyping tools allow the creation of mock up reports and screens to verify business requirements and user satisfaction with the solution under design before detailed construction and testing

- On the left, are some support and productivity tools less intimately tied to the development activity, but nevertheless vital and commonly used. These include:
  - Project Management tools which allow the definition of the project tasks, available resources, dependencies, allocations, deadlines and monitoring of progress towards targets
  - Office automation tools including work processing, spreadsheets, business graphics, e-mail etc.
  - Groupware tools may allow the management of shared knowledge and facilitate collaboration between workers, both local and remote
  - Below and to the left are tools typical of mature development shops, including:
    - Version and change management tools. These can ensure that correct versions are included in each release and allow the management of concurrent development and production versions.
    - Automated Test tools allow the repetition of tests in a structured way to ensure that applications improve in reliability
    - Packaging and Deployment tools assist with the assembly and distribution of finished applications to their installation points

A sample architecture, with tool choices populated is shown in Figure 15.8.
15.8 - Development Environment Architecture - Choices Populated

As can be seen from the populated architecture, our lives are considerably simplified by obtaining large sets of functionality from an integrated toolset. The more of the framework a toolset covers, the less problems we will have in integrating the total set of tools we use.

**CASE Tools**

**Goals**

Integrated CASE environments aim to

- Support the full lifecycle including code generation
- Support sophisticated modeling with cross checking and a central shared repository
- Support multiple developers, multiple concurrent projects
- Target multiple delivery environments
- Accelerate development and simplify maintenance
15.9 - Typical CASE tool

Typical Functions Supported

- Diagramming
  - Data Flow Diagrams
  - Entity Models
  - Relational Models
  - Function Charts
  - Module/Structure Charts
  - Class/Object Diagrams
  - Use Cases
  - Sequence Diagrams
- Dictionary/ Repository management
  - Data Definitions
  - Process Descriptions
  - Prototypes
  - Relationships and Cross References
- Prototyping
  - Screen design
  - Dialogue design and playback
- Report design
- Checking
  - Rule enforcement
  - Integrity checking
  - Cross-model checking
  - Lifecycle sequence compliance & validating deliverable presence/correctness at checkpoints
- Completeness checking

**Types of Tools and Investment**

With CASE, you tend to get what you pay for. The tools range from those which essentially support diagramming by one person with little semantic understanding of what the pictures mean and little or no integration across models, to those that will support multiple teams working concurrently on multiple projects, integrating all their models and model types across the full lifecycle and generating code to boot. Figure 15.10 shows one classification and the typical characteristics of each category.

![Diagram showing categories of tools](image)

*15.10 - Categories of tools*
CASE Successes and Failures

Record shows some spectacular successes, many spectacular failures. Success requires:

- Management commitment - payback is in the long term and requires major investment
- Significant investment (>R35 000 ($5 000) per developer seat). Costs are not just in software, but also in consulting support, training and mentoring
- Strong discipline and consistent methodology
- Technically superior and well integrated product
- High technical skills - good tools are no substitute for the skill of the person using them

Some High End Products

- Sapiens, Ideo, Object Pool (fairly OO)
- Texas Instruments/Stirling COOLGen, COOLSpecs, COOLJex (More component based than OO)
- Cayenne Object Team
- IBM UML Designer
Middle tier products

- Paradigm +
- Rational Rose
- Select
- System Architect
- ProVision
- Meta Edit+

Most popular methods

Other methods most widely supported in current tools are (in order of popularity)

- OMT, Booch, Jacobson (UML)
- Martin/Odell
- Schlaer Mellor
- Fusion

Advanced Features

Code Generation

This is the “holy grail” of CASE. The idea that we can, from sufficiently detailed and well defined requirements, automatically generate error free code. In practice, real reliable code generation is achieved by very few tools. By successful code generation, we mean that 100% of code is generated from the analysis and design models and that no intervention is required at the code level to produce a fully working system. To our knowledge only the CoolGen family and its predecessors have really delivered here, though there have been many valiant attempts by the likes of Software AG, McDonnel Douglas and others.

Most middle to high end tools will support code generation to some degree. Most popular target languages are:

- C++
- COBOL
- Java
- Smalltalk

A less ambitious approach which takes a pragmatic view and realises that we will seldom be able to generate 100% of code appears in the form of round trip engineering. This allows us to modify the generated code in such a way that we can preserve the changes made from one generation cycle to the next. In some cases this can be very impressive, allowing us to make changes at the code level which are then reflected back accurately in visual models.

Other tools use a component based approach, where libraries of components are used and what is generated is more like “glue” code to link together all the bits we need to complete an application. This can be an effective approach and is partly how CoolGen achieves its relative success.

**Reverse Engineering**

This is the extracting of meaningful specifications and models from existing code. Basically, source code and data definitions are parsed and the tools try to recreate as accurate and useful a set of analysis models as possible. In some cases these models are then amended by analysts before forward generating a new, cleaner version of an old application, sometimes in a new language. McDonnel Douglas did a lot of this from COBOL to their fourth generation language.
CASE is fine for new applications, but what about the old ones?

15.13 - Reverse engineering via CASE

Of course, you cannot reclaim what is not there: if the original software used horribly cryptic variable names and program and procedure names, you will still have these in the recovered models. Normally we really need the manual step to make sense of the recovered information. Nevertheless these types of tools can be valuable in conversion and renovation work.

**Supporting Multiple Developers and Projects**

Most tools will support one developer relatively well. Some will support a team working on one project well. Very few will support a number of teams working on a number of projects simultaneously. The pricing of tools tends to reflect the relative sophistication and complexity required to achieve these different ways of working. Low end tools often run on a single PC with little in the way of a real database or repository. Middle tier products will often have a database or repository sharable in a LAN environment with multiple client modeling workstations. High end products will have true server, multiuser repositories which will support concurrent access by many developers, manage versions, provide federation of repositories and check-in check out facilities etc. Pricing tends to range from up to $1000 for the low end, to $10 000 for the middle tier, to $100 000 for the high end. Of course, the larger tools will be supporting more users per license, so they may actually work out no more expensive in the end.
Commercial Tools

Popular tools with good object support include:

- Rational Rose from Rational Corporation. This is from the organization that now employs the three originators of UML. It has a lot of credibility as a result and is widely used. Earlier versions were clunky, but later versions are increasingly sophisticated and capable. There is now serious support for integration with Microsoft desktop products and emerging support for several flavours of process using the UML as a notation.

- Paradigm Plus has traditionally provided a solid multi-user modeling environment and integrates with other tools from the vendor in the methods management area.

- Select is widely used in the UK and is a good tool if you want to marry business modeling with back end partial code generation.

- System Architect from Popkin marries rich modeling tools, at reasonable prices, with strong architecture support.

- Visio was traditionally just a drawing package, but later releases have strong support for UML static modeling and growing support for dynamic modeling as well. A good choice if you mainly want diagrams and are not too concerned about a large integrated repository.

CASE Realities

- About 5-10% of CASE implementations are successful i.e. achieve the benefits planned at the outset.

- Successful sites spend upwards of R55 000 ($8 000) per developer for
  - Hardware (workstations)
  - Software
  - Training
  - Mentoring and Consulting

- Successful sites achieve about 35% improvement in productivity on average.

- Sites which do not adopt a method and disciplined approach usually fail
  - Sites adopting a method without CASE show similar success!
  - CASE may be a way to persuade developers to follow a method.

- Many CSF's have to be satisfied simultaneously to succeed
CASE Critical Success Factors

- Senior management support and commitment (It's expensive)
- Staying with it (Payback is not short term)
- Skills (A power tool will not make a monkey an engineer)
- Discipline (A formal method must be followed)

Tool characteristics

- High integration across components and models
- Strong Repository
- Model validation and cross checking
- 100% Code Generation AND/OR round trip engineering
- Openness and adherence to standards for interoperation (CDIF, OpenCASE)
- Multiple developer, multiple project support

Vendor Support and Mentoring

Open CASE

Started as an alternative to IBM's AD-Cycle, which never really flew. Collaboration between vendors including: Intersolv, CADRE, DEC, Oracle, LBMS, Software AG, Sybase. There are a number of relevant standards, including:

- CASE Data Interchange Format (CDIF) - Electronic Industries Assoc.
- 1175-STL std from IEEE specifying non-graphical interchange protocol
- PCTE - Portable Common Tool Environment
- European Computer Manufacturers Assoc. (ECMA)
- Information Resource Dictionary System (IRDS)
- ANSI, ISO and National Institute of Standards and Technology (NIST)
- XML rapidly emerging as an inter tool data interchange standard

Bottom Line

CASE tools can help competent developers become more productive. It requires discipline, commitment, investment and sustained effort as well as choice of suitable well integrated tools. It will not solve the problem if we do
not have suitable skills or if we do not understand a proper development process.

15.14 - A fool with a tool is still a fool...
References and Further Reading

McLeod, Graham, Choosing Object Oriented Tools 1994, CSSA National Conference, Cape Town

Usability is Vital

The interface is what the world generally sees of your application. If the interface is good, but underlying functionality is lacking, people will still be interested. If the underlying functionality and data is superb, but the interface makes is unavailable or unusable, you have lost the battle.

The interface must also be appropriate to the user community and the circumstances under which a system is used. I frequently see systems which have been designed by developers in an office environment but deployed into a factory or restaurant or other vastly different context. What works well for a power user with a large monitor in an office setting is not the same thing that works for a casual staff member rushing by a shared workstation in a busy restaurant, or a turner working on a production line lathe in an assembly plant where wearing of heavy gloves is the norm.

None of us likes ineffective, frustrating or inconsistent interfaces - we will tend to stop using those products which force us to, even if they are otherwise excellent. Would you put up with a door handle that has spikes which puncture your hand, or a shirt that scratches your skin and makes you itch all day? Just like a well designed artifact or piece of equipment is a constant joy to use, good software should provide a similar positive
experience. Good interfaces tend to share some common characteristics. They are:

- Functional, allowing us to achieve our chosen purpose with the product with a minimum of effort
- Intuitive, so that we need no or minimal training or assistance to figure out how to use them
- Ergonomically sound, meaning that they fit human capabilities well. This refers to how our senses will perceive them and how our bodies will act upon them to indicate our wishes
- Aesthetically pleasing, exhibiting taste and style that pleases a wide variety of people
- Customisable or heuristic, allowing the user to set his/her own preferences, or learning from usage patterns what a given user normally wants to achieve and helping them do it
- Predictable and consistent - users should be able to develop models of expected behaviour and not be disappointed
- Helpful, providing easy access to online assistance and tutorials, as well as worked examples, integration of help with task wizards, etc.
- Secure - ensuring only authorised users are allowed to do only legal things
- Standards compliant, so that it is familiar even before we use it and that learning curve already expended in the environment can be leveraged
- Responsive, providing good feedback and quick response times
- Appropriate to the user community, context in which application is used, type of application and technology employed
- Logical, fitting the normal work pattern or mapping familiar paper forms in terms of sequence
- Supportive rather than prescriptive or arcane
- Intelligible so that intended users will find information required easily and be able to interpret it correctly with minimum effort and delay
- Efficient in terms of resource usage
Text Interfaces

One of the oldest, but still in widespread use. Text interfaces occur in two major types: Structured page mode displays (typical of mainframe applications) and command based systems which can use either page mode or teletype terminals. The latter is typical of minicomputers and pre-GUI PC systems. Command based interfaces can be very effective for expert users who will perform similar or related tasks with high frequency over extended periods of time. They may require initial training and provision of online help capabilities to make them easier to learn. Typical examples of where they might be appropriate include: a programmer’s editor for source code and a system which is used by reservation clerks to capture bookings and payments. Advantages include wide support across platforms, relatively low cost and very efficient use of bandwidth.

We seem to feel these days that we must have a GUI interface, otherwise our application is old fashioned. We should resist this, there are many situations where a text interface is the most appropriate.

We should also not let the medium prevent us from using other mechanisms such as menus, pre-filled forms, heuristic menus and short cut command names. Online help can be provided by providing for help menu items, having a help selection or link on each screen and, in command based systems, a help command and the ability to submit other commands with a help modifier which simply causes the return of the required syntax.

Graphical User Interfaces (GUIs)

Graphical User Interfaces were first developed at Xerox PARC, then popularised by the Apple Macintosh, high end workstations on Unix and other minicomputers, and then, in the PC space, by OS/2 and Microsoft Windows.

They use the familiar WIMP (Windows, Icons, Mouse and Pointer) paradigm we are all pretty much familiar with now. The mouse was an innovation of Doug Englebart at Stanford University, down the road from Xerox.

GUI’s provide many benefits for users, including:

- Easy direct manipulation of objects representing real world entities (e.g. Projects, folders, drawers, trash cans etc.)
- Standardisation across applications on a given platform, reducing learning time and effort
- Non-prescriptive nature, puts the user in control, allowing user to chose what to do when, rather than locking them into a predetermined dialogue

Unfortunately, GUI interfaces provide much greater complexity for programmers. We will contrast the working of a traditional interactive application with a GUI example in the discussion that follows.

**Modal Processing**

In a modal interface, the system works with the user in a question and answer mode, where the application directs the dialogue. This is how most beginner programs are written. In this mode, the system is either processing a response, presenting a new request to the user, or busy processing the response. A typical scenario is shown in Figure 16.2.
16.2 - Modal processing

The key thing to notice here is that the program is in control and determines what the user can do next. Once we move into a GUI environment, there are additional challenges:

- We do not know what the user will do next - they can activate any control on the display, even moving between our application and others that we know nothing about.
- We do not control the sequence in which things happen - this is entirely user driven. What we can control is how to respond to each separate user action, given the current state of the system.

The nature of an event driven application

The structure of our application must change from one with an essentially fixed path that we control, to one that creates an interface, activates it, then responds to each individual user action while preserving the system’s current state as a context in which to interpret each request.

Event Handler & Event Processing

The main logic of an event driven application typically consists of a routine that

- waits for something to happen
- determines what happened (mouse click & position, keyboard entry, time reached, arrival of information at communications port..)
- Determines who the event belongs to (there are typically multiple applications running concurrently)
- Despatches the event to the appropriate code to deal with that occurrence within a specific application.
- When the code has run and performed the required action to respond to that event, the main loop regains control and waits for the next event

16.3 - An event handler

Note that our application, instead of being a long dialogue, is a collection of small modules (or methods) that each perform one coherent action in response to a given user stimulus. This enables us to have more maintainable code which is relatively independent of other parts of the application. On the other hand, we must preserve the state of the application somehow across all the related modules/methods to properly interpret the user actions. For example, it may be significant what was clicked on in a list before a button is pressed - these will be processed as two separate events, but the button (lets say a “Delete” action) is required to cause the deletion of an item (lets say a Task in a personal organizer) which was selected by a previous click on a scrolling list. The application must either:

- Store the CurrentlySelectedTask ID associated with the item clicked on in the list OR
- Be able to interrogate the list object to determine which item to work with when the Delete key is clicked
If we are writing our application in a low level language, such as C++, we may actually code a main event loop such as that shown above. More typically, we will be running in an environment that supports multiple concurrent tasks (most modern PC Operating Systems and pretty much any server operating system). In these situations, it is more typical that the basic event handling is dealt with by the OS which traps the event in a driver appropriate to the device, then determines who it belongs to and passes it to relevant user code. To do this, it must know who is using what devices, or parts of screen display area etc. To assist with this, our application will typically inform the system of its device needs and use operating system (or GUI subsystem) routines to manage display interaction. Our application will also need to set up its interface controls (user interaction objects such as buttons, lists, menu items...), inform the operating system which code to invoke when these are activated and initiate the interface when it is built. Figure [] illustrates how this is achieved via a fragment of Smalltalk code, belonging to a Controller class. We will talk more about the large scale architecture of applications later when we map our models to a design pattern.

16.4 - Event Processing
Using events to decouple logic

Events can also be used to decouple methods from a knowledge of other methods or classes. Systems often provide a standard event handling mechanism which allows:

- User code to trigger events
- Interested parties to register their interest in an event

In this way, the code triggering the event does not need to know about the other class or method. We can thus alter the system very easily to trigger additional or different reactions to particular events without altering the original code.

Handling a situation procedurally: Using Events:

**in Customer Class:**
```plaintext
debitWith: anAmount
balance:=balance + anAmount
(balance > creditLimit)
ifTrue: [Report sendRudeLetter: self]
```

**in Report Class:**
```plaintext
sendRudeLetter: aCustomer
```

**in Controller Class:**
```plaintext
sendRudeLetter: aCustomer
when #CustomerNaughty
Report: sendRudeLetter: eventData
```

Yes, this is slightly more work, but, like we found with “get” and “set” methods which provided encapsulation, this indirection provides insurance for the future.

Browser as a Client

A major new interface technology has burst on the scene in the last five years, namely that of the Internet or World Wide Web browser. This is an application that receives information comprising text, graphics and hyperlinks to other content, usually in the form of a Hypertext Markup Language (HTML) data stream representing a “page” and renders this
information into a display comprising content and controls, including hyperlinks. There are many browsers in the market, but the Big Three are:

- Microsoft Internet Explorer
- Netscape
- Opera

Current generation browsers support a wide range of capabilities:

- All support HTML (page description) protocol and Telecommunications Protocol/Internet Protocol (TCP/IP). HTML mixes content (text and graphics, usually as .gif or .jpeg files) and instructions about formatting and placement (e.g. Alignment, Bold, Text Style) which is embedded in the form of “tags”. Tags identify chunks of text as having a special meaning, such as being the address of another page (a hyperlink), or an instruction as to how to render the content.

- All current generation browsers also support HTML Forms. These allow the creation of a “fill in the blanks” form within the browser which the user completes and then submits to the server. The web server will normally pass this data on to a user application to process via a protocol called Common Gateway Interface (CGI) or one of its derivatives.

- Newer browsers also support eXtensible Markup Language (XML) which allows definition and exchange of meta data definition plus content. Unlike HTML, XML does not determine how the information should be displayed, but rather informs the recipient what the data represents (e.g. A name and phone number) and leaves it up to the recipient to determine how to render that kind of information. A companion language XSL allows communicating rendering information for XML data. XML provides some very powerful possibilities for interoperability of systems from different environments without prior collaboration or coordinated development. For example, I could generate from a database an XML data stream that provides a list of Products and their respective % Market Share. One client could choose to render this as a table, another as a pie chart and yet another to announce it via a speech synthesiser.

- Java Script is supported well by all major browsers. This is a scripting language derived from Java by Netscape which can be embedded in web pages and allows a limited amount of programming within the browser environment. It is normally used to handle events on web pages, validate locally entered data without communicating with the server, or perform calculations or rendering not supported by the browser.
A Java Virtual Machine (VM) is well supported by non-Microsoft browsers. Microsoft also has a Java VM, but it is not conformant to the current Java standards. A VM allows the browser to receive, validate and execute small applications (applets) embedded in HTML and downloaded to the browser. This allows the runtime extension of the browser environment with a very wide range of additional capabilities. This is obviously exciting, but problems include the bandwidth to download the applets (and classes that they use) and incompatibilities between VM implementations. Microsoft would rather you used its VB Script and ActiveX technology than the Java equivalents.

VB Script is a scripting language for the browser derived from the popular Visual Basic language. It is only supported directly by I.E., although there are “plug in” helper applications for the other browsers that allow them too to use VB Script.

Forms from a browser or data from applets or other embedded programs can communicate to the server, requesting various forms of processing. Various back end interfaces are used (e.g. CGI, ASP, ISAPI, NSAPI etc) - more on this later when we discuss architectures, design mapping and distributed objects.

Mobile devices (e.g. Cellphones and PDA’s) have quickly led to the development of so called microbrowsers. These are optimised for limited display sizes and less powerful processors and typically make use of the Wireless Application Protocol (WAP) for data transfer and Wireless Markup Language and Wireless Style Sheet (WML and WSL) to control the rendering of the received information. These basically offer a subset of the capabilities of their bigger brothers, optimised for the limited capabilities of the target devices. The attraction is that they make the World Wide Web accessible to a mobile community.

Increasingly, applications are written from scratch to target the Internet (or Intranet) as a deployment platform. Consequently, we will spend significant time in later chapters detailing how we accommodate this at the design level.

Other Interfaces

The glamour of the Internet and mobile devices should not be construed to mean that these are the only important interfaces, though. We are also living in an increasingly “wired” world where devices and environments become smarter and increasingly networked. Drivers of BMWs (and other top end cars) have a small army of microprocessors attending to their needs and
quietly communicating over their own in-car LAN. The engine management computer talks to the transmission computer, which talks to the anti skid braking computer, which receives information about driver intentions from the steering system. Most of this happens without our awareness, except as we experience the sophistication of an intelligent cooperative device. Our systems will increasingly interface with a plethora of interface units, many of which will be unobtrusive and may communicate with us via the Internet. These additional interface possibilities include:

- Card Readers, Sensors, Scanners which monitor physical presence or movement of people or things
- Mobile devices such as cellphones, PDAs and pagers
- Gestures which can interpret written input into commands or text
- Voice input and output
- Touchscreens and position sensing in new 3D variants. A public phone in Germany has a keyboard with no moving parts. It is etched onto a metal plate representing the key tops. Sensors at the edge of the keyboard interpret the position and movement of the user’s fingers to determine what keys are pressed and in what sequence.

16.3 - Lifestreams Prototype

- David Gelertner and colleagues at Yale have a fascinating interface model as part of a broader concept called LifeStreams™ which makes data available to a user based upon important concepts to the user, filtered by context and sorted by time. This information, public and
private, will live on servers in cyberspace and be accessible to its owner from any suitable connected workstation worldwide. The concept is now being commercialized by MirrorWorlds.

When we map our designs to technology and architect our solutions, we will pay particular attention to creating applications that can support a variety of interface technologies simultaneously while sharing business logic and domain knowledge.

**User Interface Design Principles**

We suggest that we follow some important user interface design principles to help create the kinds of interfaces we would like:

- Follow industry standards for appearance and behaviour of controls
- Make interface appropriate to user and application
- Use business oriented terminology, avoiding jargon and cryptic codes
- Consistency is vital
- Predictability is important. Confirm user intent under dangerous situations
- Provide feedback, but avoid modal dialogues
- Map business process, sequence, forms
- Minimize levels/modes, provide short cuts
- Handle errors gracefully
- Allow user to customize (colours, toolbars, window placement, etc.)
- Provide online Help and Tutorial facilities
- Heuristic modification based upon use
- Consider portability implications

**Help and Tutorials**

Systems should be as intuitive to learn and use as possible. Consider the paradigm of a car. It is a very complex device and driving is a challenging activity (just think about learning to drive for the first time). Yet, those who
travel frequently can arrive at an airport, pick up a rental car which is unfamiliar, and be driving it down the freeway in traffic reasonably competently within minutes. This is achieved by having a powerful paradigm and consistent model for how a car works. I press the accelerator, it speeds up. I press the brake, it slows down. Turning the steering wheel anti clockwise will cause the car to turn to the left.. And so on.

We need similar powerful shared models of how software will respond to our actions so that we can jump into a new application easily and quickly make use of its facilities. It is not too long ago that each machine, each operating system and each application had its own unique, often arcane interface. We are already at a point where most users can discover the major features of a newly presented application within a short time and unaided, just by browsing the menus and toolbars. We need to support users in becoming self sufficient by having

- Comprehensive, context sensitive help as an integral part of the application
- Tutorials, either online or provided on CD via multimedia, which convey the principles important in the use of the product, rather than the nitty gritty
- Online assistance in the form of a web site that allows updated help and topical information to be provided and for users to communicate directly with developers and support staff

**Error Handling**

Errors must be anticipated as far as possible. In all cases, we should provide meaningful, intelligible feedback to the user. It is meaningless to display a dialogue that says:

```
Internal Error No: Abc103-z
At: A010H002
OK
```

No, its not OK! Just what is the problem and what can I do about it? How do I recover the work I just did? Far better something like this:
You have insufficient space on disk F:
Please save your work to another drive
or free some space on F: and try again
Current space available: 2.1 Mb
required: 3.5Mb

We could also have buttons that would take us directly to utilities to do something constructive, like clear temporary folders, compress files etc. Now at least I know what the problem is and can do something about it.

Some principles for dealing with errors:

- ALL anticipated errors should be catered for and dealt with explicitly, providing as much user feedback as possible in a non-technical way
- Unanticipated errors should still be trapped by a generic routine that at least reports the maximum amount of information available. Try to eliminate as many of these during testing as possible
- Wherever possible, the application should preserve the integrity of data and initiate recovery
- Consistency in error handling is just as important as it is in the user interface
- Keep as much working as you can.
- Build logging capabilities into the product so that you can replicate the problem exactly to trace and eliminate it
- Allow users to submit problems directly via the Internet, possibly with the log information mentioned above
References and Further Reading

Gershenfeld, Neil, When Things Start to Think, Henry Holt & Co, 2000

HTML4 Specification: at the World Wide Web Consortium:
http://www.w3.org/TR/REC-html40/

Latest web standards: www.w3.org

Lifestreams home page at Yale:
http://www.cs.yale.edu/homes/freeman/lifestreams.html
This page intentionally left blank.
Chapter 17

Additional Constructs

Untyped Vs Typed Variables

In conventional languages, we have built in types e.g. integer, floating point, string, Boolean, etc. Modern languages such as C and Pascal defining new types through STRUCT and TYPEDEF. In pure object orientated languages (e.g. Smalltalk) all variables (and even the characters which comprise them) are objects.

Some provide a "typeless" environment where a variable can contain various types of data/objects at different times during program execution. This can only be supported in an interpretive (or dynamically linked environment. Types are said to be polymorphic, since they will allow similar operations to be requested of them, but may implement these differently. All OO languages, both compiled and interpretive, allow user defined types of arbitrary complexity.

Through definition of appropriate methods, user supplied types can use the same notation and operators as built in types. For example, I could define a + operator on Policy objects to allow me to sum the various elements of policies into a portfolio total. Viz:

ClientPortfolio:=PolicyA + PolicyB
“Rich” Data Types

In conventional languages we have built in data types e.g. integer, long integer, floating point single, floating point double, logical, string etc. but these are pretty dumb... Consider how many versions you have in current applications of:

- date
- name
- address

More modern languages e.g. C and Pascal allow user-defined types defined in terms of built in ones via (e.g.) STRUCT and TYPEDEF. These user defined types can then be passed around as though they were fundamental data items, but the modules using them must understand their structure and/or manipulate them via appropriate functions.

    DaysDifference:=DateDifference(StartDate,EndDate)

The content of the structure is totally accessible to routines using it and therefore corruptible.

OO languages provide important extensions: new types of arbitrary complexity are defined as classes, inheritance is therefore possible. Overloading of operations is supported by method definition. It becomes possible to define a meaningful operation for newly created user types that is represented by a standard symbol, for example:

    daysDifference:=startDate - endDate

This change means that the object can maintain its own integrity. It also provides a higher level of abstraction, which allows developers to work at a domain, rather than technical, level, thereby increasing productivity.

Rich media

The definition of classes with their own internal representation and behaviours also allows the definition of rich types which can handle very complex data easily. We can deal at a logical level with structured data such as name/title, dates, addresses; we can handle spreadsheets, tables and matrices; and we can handle both vector and bitmapped graphics. The former is typical of engineering drawings, business graphics and models, while the later is typical of scanned images, digital photographs and images rendered to displays and printers. We can also handle multimedia types, such as
animations, audio clips, music (either in a structured notation such as MIDI or in a sampled form, such as is recorded on CD, or compressed form, as is found in MP3 files), and video. Because of the encapsulation of the representation, and the hiding of the complexity of dealing with the internal representation, we can work with these items as easily as normal commercial data types.

17.1 - New data types

All data types can behave intelligently: They may know how to save, read, compress, expand, print and "display" themselves. Types can also be designed to behave polymorphically, allowing programmers to deal with them in a consistent way. For example, a word processing program can implement just one set of logic for dealing with bitmapped graphics. Provided suitable types are defined for the various options, adhering to a common method protocol, the logic will then work with a whole array of types, such as .bmp (Windows and OS/2), .jpg (Internet and Joint Photographic Experts Group) and .gif (Compuserve).

Abstract Data Types

We should extend the types available in the development tools to cater for those that are commonly found in our domain. This should be done in a standard way for the organization, across all applications. It may even be
done for an industry where suitable collaboration or standards exist. These types can then be added to the repertoire of standard types available to requirements specifiers and designers, allowing them to work with a higher level of abstraction and consistency, and allowing programmers to achieve higher levels of reuse and reliability. Good candidates for ADTs include:

- Address
- Name & title
- Currency
- Date
- Time
- Percentage
- Temperature
- Blood Group
- Social Security Number
- Etc.

**Collections**

Collections allow easy manipulation of collections of objects. They provide high functionality without worrying about implementation details. Most collection class libraries are based on the rich set provided by Smalltalk. They typically include:

- Strings (Collections of characters)
- Arrays (Collections of objects of any kind) e.g. Collections of screen objects
- Sets (Unordered collections of objects, no duplicates allowed)
- Ordered Collections (Maintain members in a specified sequence)
- Dictionaries (Collection of objects accessible by key, no duplicates)
- Various space or performance optimizations of the above, plus standard data structures such as queues, linked lists and stacks
Iterators

Iterators are predefined methods that work on collections of objects. They remove the logic to perform the iteration from the programmer. Some examples:

**do:** iterates over the collection, sending the code block specified to each object in turn

E.g. "Print the names of all employees within a given business unit"

```plaintext
Bus Unit employees do: [:emp | emp name printOn: Transcript]
```

Equivalent to procedural code:

```plaintext
Read BusUnit
Read first employee
Do until End Of Set
  Print employee name
  Read next employee
End Do
```

**select:** iterates over a collection returning objects for which the block evaluates as true

E.g. "Count the vowels in a string"

```plaintext
('Now is the time' select: [:c | c isVowel]) size
```

"Find customers who have a positive balance"

```plaintext
PositiveCustomers:=Customers select: [:cust | cust balance > 0]
```

**reject:** Answers the objects of the receiver for which the block returns false

**collect:** Iterates over the collection, passing each object the block, and returning the value resulting from the evaluation. This can be a new collection or can update the original one

E.g. "Square elements of an array"

```plaintext
#(1 13 7 10) collect: [:i | i*i]
```

"Put customer names in an array for a scrolling list"

```plaintext
ListArray:=(Customer allInstances collect:
  [:cust | cust name]) asArray
```

Simple Collections (Bag & Set)

Bags are the simplest kind of collection and store objects of any type, allowing duplicates. No sequence of members is maintained.
Methods available include:

- new (class method)
- aBag add: anObject
- aBag add: anObject withOccurences: anInteger
- aBag do: aBlock
- aBag includes: anObject
- aBag occurrencesOf: anObject
- aBag size

SETS are similar to Bags, but do not allow duplicates

Methods for sets are as for bag, plus:

- aSet adjustSize
- aSet find: anObject ifAbsent: aBlock
- aSet grow

**Arrayed Collections**

Arrays inherit from Collection and Indexed Collection. Indexed collections are those where members can be identified by their position within the collection. They store elements at a given integer location and return them provided with the same index for retrieval. Arrays can be polymorphic, storing different kinds of objects at different locations. This is generally supported by late binding languages (Smalltalk, Visual Basic) but not by compiled languages.

Easy to define in Smalltalk:

```smalltalk
list:=Array new: 10
list:=#('one' 'two' 'three')
```
Methods

same as bag plus
“Store objects to the array”
anArray at: anInteger put: anObject
“Retrieve object from array”
anArray at
“Concatenate contents of two arrays”
anArray, anArray
“Test equality”
anArray = anArray
“Locate index of first item matching criteria”
anArray findFirst: aBlock
“Locate index of last item matching criteria”
AnArray findLast: aBlock

Sorted Collections

Sorted Collections are maintained in a key sequence. This is achieved by specifying a sort block. The sort block may be arbitrarily complex, but must return a value of True or False. It is designed to compare the receiver with another object passed as a parameter. A default sort block is provided which compares a <= b, this will give normal alphanumeric sorting if no user defined block is provided.

In Smalltalk, the sort block can be changed during the existence of the collection. Obviously this has some processing/performance implications for large collections of course. Below are some samples working with sorted collections.

“Define the Collections”
| anArray aSortedCollection |

“Load some members”
anArray:=#('peter' 'john' 'mary')
“Sort the array members into the sorted collection”

\[
\text{aSortedCollection:=anArray asSortedCollection}
\]

“Take a look at the sorted collection”

^aSortedCollection

\[
\text{SortedCollection(‘john’ ‘mary’ ‘peter’)} \quad \text{(returned)}
\]

“Change the sort sequence”

\[
\text{aSortedCollection sortBlock: [:a :b | (a at: 2) <= (b at: 2)].}
\]

^aSortedCollection

\[
\text{SortedCollection(‘mary’ ‘peter’ ‘john’)} \quad \text{(we sorted by the second character, just to show that the sequence can be arbitrary.)}
\]

Dictionaries

A particularly useful type of collection is the Dictionary. This is a collection where the member objects are stored according to a key, which can itself be any object. Members are retrieved by providing the key. For the traditionalists, this is like having an instantly created indexed file resident in memory. Some samples illustrating dictionary use follow:

“Define a new dictionary”


“Load some members keyed by Department code”


“Query the dictionary”

^deptDictionary at: ‘fin’.

Finance Department \quad \text{(returned)}

Can store any object as the value e.g. a lookup table for postal codes where each town has a street and a boxes code:

postCodes:=Dictionary new.

postCodes at: ‘Cape Town’ put: #(8000 8001).

^(postCodes at: 'Cape Town') at: 1
8000

Conversion

Collections may be converted from one form to another:

aSortedCollection:=aCollection asSortedCollection

Both collections are maintained

Other options:
- asBag
- asSet
- asArray
- ...

Often used to get things in right type for user interface: e.g. items in a list box need to be an array, etc.

Interfacing and Invocation

We also need to be aware of the many ways in which parts of applications can communicate with each other. Each method has advantages and disadvantages. These options include:

**In memory call**

Where one module invokes another directly in memory within the same machine and address space. They will normally share a common thread of execution on the processor. Parameters can be passed to the called module and a value is normally returned from the called module. The first module will not proceed until the called code returns control to it.

The modules will normally have been combined into a single executable file by a link edit step following compilation.

**Dynamic link library**

Similar to the above in effect and operation, but the module being called is not bound to the calling module prior to execution. It resides on a storage
medium (normally in a library) until it is invoked, when it is instantiated in memory and control is passed to it.

**Remote procedure call**

Similar to a normal in-memory call in principle and syntax, but the called module is not in the same machine or address space. The local module calls the remote module in a similar way to a normal call, using a logical name and passing any necessary parameters. The call is intercepted by a stub module which does reside in the local machine. The stub communicates via a network with a container module on the remote machine. The stub formats the parameters in a form suitable for transmission over the network and sends them to the remote container via the network. The container formats the module called on its machine and passes on the parameters. When the called module returns control, the container receives the returned value, sends the information to the stub on the calling machine. The stub returns control to the original calling program, which has remained suspended, awaiting a return on its original call.

Remote procedure calls of this type are the basis on which distributed services such as Distributed Computing Environment (DCE) are built. They add sophistication in the form of protocols to manage the distributed functionality, directories to look up physical locations from logical names and other features. Distributed object protocols, such as object request brokers, may in turn call upon these services.

**Messaging**

Messaging is where modules communicate by the exchange of messages, usually, but not necessarily, over a network. One of the most common forms of this is a mechanism known as named pipes. It grew out of a facility provided in the Unix operating system to allow output from one command to be fed to further commands via a “pipe”. By having named pipes, we can designate logical channels for communication between programs, where one is designated a sender and another a receiver on a specified pipe. Another popular protocol is the sockets protocol. Here, programs connect to a “socket” or port at the end of a communication link. Sockets underlie much of the Internet communications, hence the Windows Sockets Layer or Winsock. These protocols have much to recommend them. They can be reasonably efficient while offering insulation from the physical locations or characteristics of receivers. With a little extra plumbing they can run across networks in a fairly transparent way.
File or Database

Application modules can also communicate by writing information to a file or database and having the receiving application read the information from this source.

Queuing

Queuing is similar in concept to messaging, but the messages may be delayed in their delivery to the receiver. The receiver will also request the next item to deal with from the queue when it is ready. This is particularly useful if the receiver is not always available, or needs to deal with such volumes that it may not be available due to still being active processing previous requests. If the receiver is not available, the message is held in a queue which can be memory or disk based. Disk based queues are slower, but have the advantage of persistence, meaning they are permanently recorded and would not be lost should we have a failure, such as a power outage. Disk queues can also be much larger than can be accommodated in memory. Queues can support sophisticated models where

- A single server handles requests from many clients
- Requests are prioritised within the queue, giving preference to certain types of request
- Requests are handled First in First Out (FIFO) or Last In First Out (LIFO)
- Multiple servers process requests from a common queue or set of queues, participating in a load balancing scenario
- Work is distributed over multiple servers on multiple physical configurations

Queues are frequently used in high end transaction processing systems. A very capable product which is available in flavors for all popular platforms is IBM’s Message Queue Series.

Asynchronous vs Synchronous

Synchronous invocation is where the caller/sender suspends while the called party/receiver services the request and resumes when it receives a response. It is typical of call type interfaces.

Asynchronous working is where the caller/sender does not suspend while the request is serviced but continues execution itself. It may later be informed of
the outcome of the request by a separate message being returned. A sort of synchronous working can be achieved in this way if desired by sending an asynchronous request but immediately beginning to monitor the receiving path for a response.

Synchronous working is simpler, but less reliable, especially in distributed systems where the state of the recipient is unknown or the communication itself may fail. Asynchronous working tends to be a little more complex to implement, but provides more flexibility and is usually more robust in practice. It is easier to achieve with messaging and queuing protocols.
References and Further Reading

Open Group Portal to DCE:
http://www.opengroup.org/dce/

IBM MQ Series:
This page intentionally left blank.
Persistence

The Need for Persistence

Certain objects must be "persistent" i.e. must endure across time and space to be available whenever the associated methods are activated. Examples would be data related to Clients, Products, Accounts, etc. - traditionally considered "Master Files" or the information that would be stored in a database. These are normally our domain classes.

Some objects are ephemeral - they are created anew when we run the application and destroyed when we have done with them. This would include things like windows, menu items, temporary collections and so on. These will not be mapped to persistent storage.

Some options

There are various ways of achieving persistence, stemming respectively from making programming language objects persistent and from making databases object oriented. These include:

- Image files
- O/S files
- Relational (SQL) Databases
Object Databases

We will take a look at each of these options in turn.

**Image files**

Are probably the simplest method. Essentially, we simply take a copy of the memory “image” of the running application and store it on a persistent medium, usually disk. When we want to restore the system to its previous state, we load the image back into memory and give control to the executable portion of it. This can preserve an exact snapshot of the system at the time the image is saved. When we restore it, it will come back with everything just as it was - even down to the same windows in the same places.

Image files are used in a variety of development tools, and we do something similar when we save a document in a word processor. Smalltalk uses an image file to save the state of the development environment, and can do the same for runtime systems.

- **Entire state of the system is written to storage including all objects, screen contents, instructions, etc.**
- **Known as an "image" file**
- **If image is too large to be held in real memory, paging techniques are used at runtime**
- **Works well for:**
  - systems supporting individual users
  - managing a limited volume of data
- **But:**
  - Not as effective where users must share objects, or where data volume is large

18.1 - Image File

Problems with image files appear when

- The amount of data to store becomes very large - too big for real memory and too much to save and restore in acceptable times. Smalltalk overcomes the physical memory issue by paging parts of the image into real memory as required
• We want to share the data across several simultaneously running applications, where it needs to be separated from the executable and have proper contention and locking mechanisms etc.

Image files can work well for situations where the application supports a single user and the collection of information managed is not too large.

**OS files**

In this approach, we use operating system files to store object data, which is “flattened” to a string or serial representation, or stored as a binary representation with one file per class.

- Provides great control
- Can support sharing
- Can complicate programming
- Generally no support for recovery, integrity management, security, indexing,

18.2 - Persistence via Operating System Files

Advantages include the ability to have a lot of flexibility in what we store when, and consequently the potential for convenience and high performance is good. On the other hand, this approach requires complex programming and maintenance and makes our applications platform specific. Unless the management of the files is done very carefully, this approach will also not provide any integrity management or recovery.
Relational Databases

By far the most popular way of holding information for commercial systems is the use of relational databases, such as Oracle, DB2 and Sybase / SQL Server. These are typically accessed through either Structured Query Language (SQL) or Open Database Connectivity (ODBC). The former is an ANSI standard Data Definition and Data Access Language based on Relational Calculus. The second is a call level interface to the database engine, which was originally defined by Sybase, but standardised with the Microsoft port of Sybase to become SQL Server. Both interfaces are widely supported across a great many languages and development tools. Unfortunately, relational database structures (normalised tables) and object structures are not very compatible.

The relational model insists on very simple structures, viz tables with no repeating rows or columns. SQL also supports only a limited number of data types. Most relational implementations are using the SQL 2 standard which has no real support for object principles. There is now an SQL 3 standard which begins to provide support for object features, but which is only partially implemented by major vendors at the time of writing.

Despite the limitations of relational databases for the storage of objects, they are the dominant platform for corporate data storage and are not about to be unseated. Many organisations will simply insist that that is what they have and new fangled object systems will simply have to find a way to store their objects there.

This can be achieved by specifying a mapping between the object structures and relational tables, using an object relational database with extended features, or abusing the relational database and storing denormalised structures there. We will take a look at each of these options.

Object to Relational Mapping

This is a non-trivial exercise. The relational model does not allow:

- Any data items which are not of the simple types provided by SQL
- Any embedding of types within a column such as we have with composite objects
- Unique object identities across the various types defined in the schema
• Inheritance in schema definitions such as required to support inheritance in class definitions

So, to map to a pure relational database, we have to effectively undo all the extra richness that we achieved with class modeling (at least at the physical schema and storage level)! If we want to preserve the benefits of our object oriented design and business logic we must perform the mapping at run time. This requires complex code and can negatively impact performance. It is possible to write this code ourselves, but it is a non-trivial undertaking, especially for large volume, multi-user, transactional systems. A nice analogy is having a car (the object we want to use) which we have to store in a parts store in its smallest components between drives. Obviously, this is a tedious process, can take a lot of time and will require high skills.

Requires either:
- Specialists to assist the driver
- An extremely technically competent driver
- And lots of time...!

18.3 - Parking the CAR object in a Relational Garage

The extreme case (like the car example) is where our object model is complex. An example of this would be the model supporting a 3D CAD system used for engineering design. A more viable scenario exists if our data is less complex. This would be like having a bicycle which we also dissemble at each journey end and reassemble at the start of a new journey, but where it is equipped with clip on wheels making this a simple task that only takes a few moments. This is more akin to the situation we face in making typical commercial data represented as objects compatible with a relational database. It is a chore, but it is not too bad. In essence, the thickness and complexity and performance “hit” of the mapping layer will be proportional to the complexity of the object model.
While the relational database is not ideal for object storage, there are some persuasive advantages of using this approach:
- Most organizations already have this technology and will not need to make new investments in technology or skills
- The technology is mature and offers good performance and reliability
- If we preserve the normalisation of the data at the schema level, then relational systems and query tools can access the information

### SQL3
Relational vendors have seen the need to extend their products to cope with objects. The SQL community has been working for a long time toward SQL 3 - a new standard that will support important enhancements over an above the SQL2 model. These include:
- User defined types (composites of standard SQL types) which are then usable as column types
- Inheritance in Schema definitions
- Ability to store binary large objects (BLOBS) which allows for storing rich data types and multimedia
- Support for object serialisation and reconstruction, including the maintenance of unique object identities

With these extensions, object/relational databases can do a reasonable job of supporting object systems in the commercial applications space. They will
still be challenged in supporting systems with very complex object models, where object database technology will be a better option.

**Using Stored Procedures to Provide an Object View**

One option in the use of a relational database to provide object storage is to store the data relationally, but to assemble it into objects before working with it in the application. The latter can be achieved by stored procedures which are invoked by the object application and return a composite object after doing the necessary relational access.

While reasonable from an architectural viewpoint, this approach has a flaw, in that the stored procedure languages are not standard across databases, leading us up a proprietary garden path. An alternative is to use Java to write the stored procedures, accessing the SQL database through JDBC or JSQL. A growing number of databases support Java as a stored procedure language. If you are contemplating this, you may rather go to the next level and use the Java server technology to store your domain objects as Enterprise Java Beans (EJB).

**Object to relational mapping using a persistence framework**

Since the object to relational mapping is non-trivial, but nonetheless desirable for many organizations, a number of products have been developed to fill this gap. They generally take the form of frameworks of classes to create the mapping between a chosen object language and back end SQL database. Three products in this space are Toplink, UniSQL and Object Extender. We have experience with the first and last.

Toplink proved to be very effective in making existing relational data available as objects to VisualWorks (Parc Place Systems Smalltalk). It was not very successful in taking an object model and creating a working relational version of it.

Object Extender from IBM is available for VisualAge Smalltalk and Java IDE’s. It is an industrial strength product offering a comprehensive and easily used framework, as well as integrated modeling tools. You can use the object modeling tool within Object Extender itself or IBM’s UML modeler, generating its model into Object Extender. Our experience has been that it is easier to map your object model directly into Object Extender. From there you can automatically generate a working relational schema for DB2, Oracle, Sybase or ODBC. OE also generates all the necessary access and transaction management code. We found a few snags, but no show stoppers. It does a huge amount for the developer fairly painlessly.
Enterprise Java Beans

Sun’s Java 2 Enterprise Architecture includes the concept of server side components in the form of Enterprise Java Beans (EJBs). These are components written in Java that reside in a server side “container” application. The container must provide a range of services from object instantiation, life cycle management and persistence, to load balancing.

The EJB model provides two kinds of beans:

- Session beans, which are not persistent and are intended to provide the processing part of an application. They will typically hold business transaction or process logic and will manage transactions.

- Entity beans, which are persistent and are designed to hold domain object data in a permanent store where it is sharable and managed with transactional integrity.

We can thus use Entity Session Beans to store our domain objects persistently. To do this, we must have a container that supports a good database (or is itself one) and that will host EJB’s. The interface to the EJB is in the Java language using user defined functions within the bean plus a standard API and protocol to access standard services such as those mentioned above. An emerging variant on this approach is the so called “CORBA BEANS” within the new version of the OMG’s CORBA which uses the same component model as EJB’s, but allows the beans themselves to be written in any CORBA compliant language.

If you are committed to Java and/or CORBA, these approaches are a fairly attractive option.

Object Databases

So what of fully object databases? Do they exist and are they viable? The answers are as much commercial and political as technological. There is certainly viable OODB technology and some excellent products. These map data at the schema level in a way which is entirely consistent with the object oriented modeling we have discussed in earlier chapters. This is extremely attractive, since it can remove one of the major flaws in our current programming model, the so-called “impedance mismatch” between the way we work with in-memory and secondary storage data.
Impedance Mismatch

In electronics, the term refers to components which are not compatible because they have different expectations about the context in which they should operate (the impedance or experienced resistance of the circuit). The term was borrowed to describe the problems programmers face in moving between the memory view of objects and the storage view of records. In conventional programming, we have effectively two different languages for these: Possibly C++ and SQL, or COBOL and SQL, or Java and ODBC. The programmer is forced to switch mental models and syntax each time we move between these paradigms. Object databases with object languages can entirely eliminate this problem.

An example would be the GemstoneS object database, where the schema is defined in Smalltalk classes. There is no difference between the programming view and the schema view of an object. The responsibility for whether an object is resident in memory or not is removed from the programmer and becomes a system or database concern. The programmer simply sends a message to the object irrespective of its loaded or unloaded status. The DBMS will load the object if it is not there. This is similar in concept to demand paging in a virtual memory system. Similarly, when objects are updated, it is a system responsibility to ensure that the persistent copy reflects these changes. Eliminating these problems frees programmers from much burdensome and error prone work and lets them get on with solving business or domain problems at a higher level of abstraction. GemstoneJ does much the same for a Java programmer. Another product, Object Store, from Object Design, allows C++ programmers to work in a similar seamless fashion.

Characteristics of OO Databases

Object databases store objects natively i.e. The schema view and the object model are synonymous. They must have facilities to move objects efficiently between memory and storage. This is often accomplished by demand paging of a very large address space. Being new, ODBMS products typically take good advantage of modern computer and operating system architecture. They can be very efficient for complex model data - some benchmarks in these domains have shown performance advantages over pure relational systems of 100 to 1 or better. Their advantage declines as the complexity of the models becomes less and the sheer data volumes increase. ODBMS are technically suited to a very wide range of applications, but have won most of their success in areas where relational databases have not fared well. These include:
• Multimedia and rich media (including publishing, movie making, broadcasting)
• Scientific and other modeling (with very large complex multi-dimensional data sets)
• Image processing and storing geographical data
• Complex applications such as CAD/CAM, CASE and repositories
• Medical applications

18.5 - Object Database

ODMS Standards

One thing which initially held back the takeup of ODBMS was the lack of standards. Since the products were linked to various languages, there was no common standard across products. This created a situation where corporates were nervous to commit since if the vendor they chose disappeared, they would have no strategic safety net for their valuable data. By contrast the relational world had the SQL lingua franca which was widely supported by DBMS tools, development tools, languages and query tools alike.

The object database vendors and the OMG addressed the problem. The result was the ODMG, a separate but friendly organization promoting object data management standards. There major product has been the ODMG standard,
which specifies a standard Object Definition Language and Object Query Language, which are a superset of the SQL capabilities.

18.6 - ODMG’93, ’95 and SQL standards compared

The figure compares the ODMG and SQL 2 standards. It can be seen that the ODMG standard provides for much richer facilities in schema definition, transaction management and support for object oriented concepts. Most major ODBMS vendors have embraced the standard and implemented it at the release 2 level.

The major difficulties now faced by the ODBMS vendors are more commercial and strategic than technological. The largest ODBMS vendor is less than one tenth the size of the major relational players, viz Oracle and IBM. Clients can buy relational secure in the knowledge that at least one or two of the big vendors will be there for them well into the future. No such assurance exists for the object vendors.

ODBMS Vendors and Products

Major object vendors include Object Design (Objectstore), Gemstone Systems (GemstoneS and GemstoneJ), UniSQL, Matisse, Ontos and Versant.
ODB Strengths and Weaknesses

Strengths of object databases include:

- Allow inclusion of more semantic information in the database - better "real world" modeling
- Better support for complex objects e.g. CAD,
- Multimedia, Repositories
- Extensible types increase functionality and allow
- higher abstraction
- Use technological improvements in computers
- (fast CPU's, memory, caching - very high performance)
- Versioning easily supported
- Reusability - faster, high quality development; easier maintenance
- Inheritance eliminates design redundancy - speeds development
- Can implement relational view as one of many models
- No impedance mismatch between programming and database language

And the weaknesses follow:

- Not all products standards compliant.. ODMG 2
- Vendors relatively small - strategically risky
- Lack of formal theoretical model
- "Pointers" come in for criticism (not really valid)
- Only recently acquired standard ad-hoc query language (OQL already there, SQL/3 draft complete)
- Steep learning curve - inhibits commercial use
- Lack of pool of skills in the marketplace
- Relational "sabotage" - FUD (fear, uncertainty and doubt)

XML Stores

An emerging category of products is the growing number of XML data stores. These may be object or relational behind the scenes, but offer facilities to define or query data using XML and XML data type definitions
(DTDs). We predict that these will be very successful in support of system integration and middleware, as well as in the document handling, modeling tool and office productivity space.
This page intentionally left blank.
Chapter 19

Libraries and Frameworks

Class Libraries

Class libraries provide a source of predefined designs, usually produced by experts in a given technology or domain. They may have been produced by other projects in the organization, by a tool vendor (usually for a specific technology) or by a vendor that specialises in the provision of libraries in a certain domain, say retail, financial or medical.

Class libraries provide class definitions in a source language, so you need to select from those available for your chosen development language/environment. There are many available for C++, Smalltalk, Java and Delphi. Visual Basic library availability is likely to grow with support for definition of classes, although most vendors in this environment prefer to release component libraries.

We can use a class library in several ways:

- As is. By leaving the code unmodified and instantiating objects from the existing definitions, we can use the classes exactly as they were produced by the designer. This has the advantage that they have usually already been extensively tested and production proven. Used in this way, a class library has many of the advantages of components. The down side is that
we will not have any tailoring, but will have to accept the limitations imposed by the designer

- By subclassing to derive our own specialised version. This leaves the original classes unmodified, but allows us to override certain characteristics or extend capabilities to suit our needs. It has the advantage that we can reuse everything that is as we would like, but allows us to tweak or extend where necessary. If we do this carefully, we could still replace the parent classes and preserve our extensions when the vendor releases a new version of the library. This gives us the reuse levels of a package, with the ability to tailor, but still take advantage of new releases from the vendor

- By taking the classes as a starting point for our own design. Here we have no intent to stay compatible with the original or to leverage future vendor releases, but rather to save ourselves some time and effort upfront. We freely modify the classes received without concern for their compatibility with what the vendor does in future. This gives us great flexibility, but we lose the advantage of the testing the vendor has invested and the potential to use their future work easily

**Control/Widget Libraries**

These can be provided either as class libraries or component libraries, or sometimes as combinations of the two. They contain designs for visual controls or widgets. The two most popular formats are Visual Basic Controls (VBX - 16 bit; OCX - 32 bit; ActiveX - 32 bit downloadable) and Java Beans (Java Bytecodes). There are a wide variety of these libraries available. An example is the Swing visual component library for Java which is implemented using the Java Beans component model. It allows users to write to standard Java Swing model, but have the components alter their look and feel to that of several native platforms, hence the name “swing”.

In the Visual Basic camp there are many products available. Examples include ActiveReports from Data Dynamics; JanusGridEx from Janus Systems; and TrueDBGrid from Component One. The first provides a library dealing with reporting, while the second two provide grid controls connecting to databases.

There are very rich libraries available. They allow us to extend our applications with very little effort into areas which would otherwise require a vast amount of expertise and effort. Some categories available are listed below:
• Database grid controls
• Reporting suites
• Graphics display, manipulation and editing
• Spreadsheet controls
• Calendar controls
• Geographical mapping controls
• Multimedia controls e.g for playing MP3 music tracks
• Video editing or display controls
• Text editing or layout controls
• Calculator controls
• System customization (sounds, colours, menus, icons etc.)
• Interfacing to various technologies (Communications Protocols; Other programming languages; Barcode readers; Analog to digital converters etc.)

Component Libraries

Component libraries are provided in compiled form. They are generally easier to use than class libraries, but do not offer as much flexibility. We can only alter components to the extent that the designer has provided for this by alterable properties. For example, we can probably change the size, display position and background colour of a button, but it is unlikely that we could alter its shape or the events to which it will respond. The most popular formats for the distribution of components are listed below:

• Microsoft OLE Controls (VBX - 16 bit, OCX - 32 bit, ActiveX - 32 bit downloadable) in the Microsoft world. These can be created and used in a variety of languages, including C++, C# (C sharp), Delphi and Visual Basic
• Java Beans which is a component standard from Sun. This is for components written in the Java language. Java beans come in two varieties - Visual Beans and Enterprise Java Beans. The former are controls designed to be used in designing visual interfaces. They can be downloaded from a web site and execute within a browser environment, provided there is a suitable Java virtual machine. Enterprise Java Beans are server side components which will execute in a container application
on the server. The container application might be a database engine, web server, operating system or transaction monitor. It will provide services to support the bean. The EJB’s occur in two varieties:

- Session beans which normally implement business logic
- Entity beans which are normally persistent and would hold business objects
- CORBA components (CORBA Beans) which are components written according to the CORBA standard from the OMG. These are similar to Java Beans in architecture, but can be written in any CORBA compliant language. Their interface must be declared in IDL.

### Business Object Libraries

Technical component libraries have been much more prevalent than business object libraries, mainly because components are created by technical people and technical domains are well defined and so easy to tackle. Business object libraries are emerging to support a variety of domains and industries. There has been some notable work done by the OMG with their Business Object Facility (BOF) groups. These forums have addressed creation of standard business object models and service interfaces for at least the following domains:

- Manufacturing
- Finance
- Health Care
- Transportation
- E-Commerce
- Telecommunications
- Life Sciences

IBM also did pioneering work with their San Francisco project, creating frameworks for business systems. These were initially developed using the IBM DSOM technology (an implementation of CORBA), but have more recently been converted to using Java technology.

Some vendors are releasing applications built using business objects - this is a trend that we suspect will accelerate, since it offers benefits to both vendors and clients. An example of these innovative products is the Bankframe application from EON technologies.
Frameworks

Frameworks is the name generally given to class libraries or component libraries which include a comprehensive set of classes to provide a solution in a particular technical context or business domain. A framework will normally not just have individual components, but rather a set of classes, with rich behaviours, which will interact to provide a (fairly) complete solution. Two good examples of frameworks are:

- IBM’s WebConnect feature for VisualAge, which provides features to support development of web interfaces, creation and management of connections and sessions for web based clients accessing a server application

- IBM’s Object Extender feature for VisualAge, which provides a mapping between an object business model and storage of the content of the business objects in a standard relational database, such as Oracle, DB2 or SQL server. The framework is comprehensive and provides support for the definition of the object model, generation of the relational schema, generation of all access code and runtime support for things such as database connections, object type collections, transactions and cacheing
This page intentionally left blank.
Chapter 20

Architectures

Client Server

Few terms can have been used as often or as widely as client server within the Information Technology community within the last five years. Unfortunately, it is still a very misunderstood concept and many practitioners embark on expensive and traumatic implementation projects without an understanding of the options available, what benefits are and are not achievable, and the strategic implications of their choices. This chapter sets out the core concepts underlying client server, discusses some of the technology and business drivers fuelling its adoption, looks at perceived and realisable benefits, identifies some of the players and options, and provides some guidance in terms of architecture and standards against which to make choices.

An architecture, not an implementation

Client server is fundamentally an architectural model in which the functionality of a software product is divided into several independent but collaborating components. Client components request services which they expect to be provided reliably by server components. A single server may process requests from many clients. Clients and servers can communicate in a variety of ways, including local procedure calls, remote procedure calls...
(RPC), message passing and queuing. Client and server components can reside in the same machine, or may reside in separate machines connected by a local or wide area network.

A model where one component (the client) requests services, data or operations of another component (the server) expecting a predictable result without regard for the location or implementation of the server.

Typical Clients and Servers

Typical client components are usually, but not necessarily, fairly close to the human system user. Examples include the component which will present a menu or toolbar of options to a user in a GUI environment, obtain a response, present a form and perform basic validation before communicating with a remote database. Typical servers are usually, but not necessarily, not in direct interaction with human users. An example would be a relational database engine providing control and management of a shared database on a LAN server. Other servers may provide specialised functions such as extensive high performance computation or printing.

Most client software will run in the user workstation, while most server software will run in network, application, file or database server machines which may be large PC-architecture machines, minicomputers or mainframes.
Joining the components

Where components reside in different machines, there must be a network to allow them to communicate. This may be a local area network (LAN) within one building or campus, or a wide area network (WAN) stretching between offices, possibly globally. Where a LAN is involved, it is most commonly a Novell, Windows NT, OS/2 or Unix based network. Typical protocols employed include Novell's IPX and the industry standard TCP/IP originally developed for the Internet. Wide area networks have traditionally used IBM SNA (in IBM host dominated networks) or the ubiquitous TCP/IP (in open, mixed or Unix based networks). Many IBM hosted networks are now switching to the more open TCP/IP protocol.
Client Server Drivers

Many technology and business pressures are fuelling the widespread and rapid adoption of client server. Business drivers include:

- Perception of more power for the money in commodity technology
- Desire to empower users
- Need to handle rich data types including multimedia, voice, video, image
- Desire for scalability, where applications can start out small and cheap, but be easily and rapidly "scaled up" should they prove successful, or business expansion warrant
- Global operations requiring more distributed data and more robust 24hrs 7days/week operation
- Pressure for more rapid application development
- Business Process Reengineering focus

Some technical drivers making client server possible and attractive include:
Excerpt from: Beyond UML: Advanced Systems Delivery with Objects, Components, Patterns and Middleware by Graham McLeod, Inspired Press 2001, with permission

- Productivity benefits of Graphical User Interfaces (GUIs)
- Object Orientation and Component Based Development
- New microprocessors with vast power on the desktop for low cost
- Cheap, high volume storage at the workstation and departmental server level
- Higher bandwidth local and wide area networks
- SQL and relational database engines

20.4 - Application Components

The Gartner Model

The respected Gartner consulting group published a model describing various styles of client server which has been widely adopted. Briefly, it includes the concept that an application can be partitioned into three main layers, viz:

- Presentation layer which looks after interaction with the user
- Business logic which implements the transaction logic, business rules and computational algorithms
• Data access which provides access to shared, persistent data stored in permanent files or databases

The model then describes how these layers may reside in or be split across a client and/or server machine. There are five permutations:

• Distributed Presentation where presentation only resides on the client workstation, all other functions are handled in the server. This is typical of early client server implementations using terminals or PC’s emulating terminals to access minicomputer or mainframe hosted applications. Problems include poor interactivity, particularly for GUI interfaces, since data must travel to the server for validation.

20.5 - Simple Gartner Client Server Model

• Remote Presentation is typical where we might have a GUI front end residing on the client, but all business logic and data access occurs on the remote server. An example would be a Windows application residing in the workstation, but essentially implementing form filling only, with all logic and data base access occurring on the server. This may be the case where the application is implemented using stored procedures in a relational database engine

• Distributed Database is a style where virtually all processing and user interaction is handled on the client platform, with only data access at the server. This is typical of early LAN based client server implementations. An example would be a Novell file server driving a LAN with
workstations which run Windows applications. The Windows applications would contain all GUI handling and processing logic. They would only ship database or file access requests to the server. Problems with this approach can include high data traffic on the network. For example, if a user develops a query which will select 10 records from a table containing 1,000,000, since there is no intelligence in the server, all 1,000,000 records will be moved to the client where the selection will take place, discarding all but 10 of them.

- **Remote Data Management** is somewhat more advanced in that business logic still resides in the client, but all of the data access logic can occur in the server. This is typical of clients running against a relational database engine, such as Sybase or Oracle. Where data requires selection, this can be performed in the server, with only the resultant set of records being returned to the client. Performance can be much better than the previous style, both because of reduced network traffic, as well as taking advantage of faster processing and larger capacity storage in the server.

- **Distributed Function** is one of the most advanced styles, and is growing in popularity. It allows distribution of the application logic between client and server, as appropriate. For example, I could have the GUI handling and initial validation done in the client workstation, while core business logic and database updating occurs in the server. This style allows considerable flexibility and can provide both good response time and relatively low network traffic. The downside is that the deployment and version management of these applications is potentially more complex than some of the other styles.

**Two and Three Tier**

Early client server implementations often consisted of workstations connected to a file or network server; or workstations connected to a mainframe. From a physical topology perspective, this is a two tier architecture. Advantages of this architecture included:

- Relative simplicity
- Reasonably wide choice of development tools available

But there were also several problems including:

- Potential for poor performance where incorrect choices were made in terms of splitting functionality between client and server
• A proliferation of interfaces, requiring high skills from developers and extensive rewriting if functionality was moved to a different platform
• Poor security and recoverability of data held at the workstation level
• Logistical problems of distributing new versions of software to large volumes of client workstations which might not
  • be switched on
  • be configured correctly
  • have sufficient space

20.6 - 2 Tier vs 3 Tier Client Server

To address these problems, three (or \( n \)) tier client server was introduced. This makes use of an intermediate server between the client and the main server. This often maps well to the structure of the organization itself:
• Client workstations are connected to a server using a local area network
  This configuration supports a department or work-group in a particular location
• Personal data resides on the workstation, or on a private storage area on the local server
• Work group or departmental data, or data peculiar to the geographical location is stored on the local server
Corporate data is stored on the central server, which would typically be a large Unix or Mainframe computer.

The three tier approach has numerous advantages:

- Security of data is better, since it can be controlled by a departmental level DBMS and administrator
- Data back up and recovery is facilitated
- Expensive resources such as printers, plotters and archival units can be shared at the departmental level
- Distribution of new software is much easier, since it can be distributed to and held on the server. This is better since:
  - There are fewer sites to update
  - Transmission times are reduced
  - All local workstations will run an image of the single copy on the server, ensuring consistency
  - The server is more likely to be up, correctly configured and have space available

But there are still some negatives:

- The approach is more complex and can be more expensive
- The number of tools supporting three tier architecture in a robust and comprehensive way is small, and these tools tend to be more proprietary

**Interfaces, Interfaces**

A developer in a client server environment is faced with a potential proliferation of interfaces to contend with. The typical interfaces are illustrated in figure 20-7.

- Relational databases are typically accessed either via SQL, or Open Database Connectivity (ODBC)
- The platform's GUI will normally have a unique and proprietary API e.g. Windows, Macintosh, OS/2
- Different operating systems will have unique Application Program Interfaces (APIs) to access their system facilities (e.g. printing, initiating a batch job)
Where business logic components want to communicate with peer components elsewhere, there are a variety of protocols which may be employed, including:

- IBM's LU6.2
- Remote Procedure Calls
- Messaging protocols or products
- Object oriented request brokers (ORBs)

Moving the functionality of a layer into another machine in the configuration, or replacing one component with another product, would frequently require extensive repackaging of the application with a change of interface code. This is both time consuming (hence expensive) and risky. It also requires an unreasonable plethora of skills from developers. Middleware attempts to address these problems.
Middleware

Middleware comes in many shapes and forms - from the very simple single focus products to the very large and ambitious multi-platform multi-standard open architecture products designed to support enterprise computing in heterogeneous environments, and even interaction with public networks and business partners. An example of a single purpose product would be a small piece of software that provides an ODBC call style interface to an application program while issuing SQL commands to a relational database management system on the other side. This has the essential characteristic of middleware: software which sits in the middle, typically between a client component and a server component, but also sometimes between application code and other facilities in the environment e.g. an operating system.

Without middleware, applications have to deal with myriad interfaces...

20.8 - Need for Middleware

Comprehensive middleware products have much larger objectives, including to insulate the application developer from:

- Specifics of underlying databases or files
- Specifics of communications protocols, making them transparent across platforms, LANS and WANS
- Mechanism used to communicate with other parts of a distributed application
Specifics of the supporting platform and operating system

- Needing to know which components are local or remote (and where they reside)

With middleware, applications have to deal with only one interface

Unfortunately, it's usually proprietary...

---

20.9 - Middleware Concept

In short, if the middleware is doing its job, the developer should have just one interface to worry about: the one to the middleware itself. While this is very attractive from the point of view of reducing complexity as experienced by the developer, and hence making deployment, scaling up and repackaging much easier, there is also a danger that the organization can become locked into the proprietary interface to the middleware product itself. One attempt to alleviate this problem has been to define industry standards for middleware, in the form of facilities which should be supported, and a standard protocol for invoking these. One such approach is Distributed Computing Environment (DCE) from the Open Systems Foundation (OSF).

Advanced Options, DCE

The OSF was founded as a collaborative venture by vendors including IBM, Novell and Hewlett Packard. While it produced a standard which addressed
many of the requirements needed to support distributed, open, heterogeneous computing, it had two fundamental flaws:

- The communication mechanism chosen between remote components was based upon Remote Procedure Calls (RPC). While RPC is a widely used and successful model for the interaction of code modules across networks, it assumes that the networks are generally available and reliable. Most implementations are synchronous, in that the caller will suspend and wait for a response from the addressee before resuming execution. This has proven problematic in client server and LAN based environments where nodes are frequently not ubiquitously present.

- DCE came from a Unix and mainframe perspective and largely ignored the PC in a world which is increasingly dominated by PC computing. Some of this was by accident and dint of the world view of its specifiers, and some by design, since the consortium were trying to oppose the domination of Microsoft by promoting Unix as a strong alternative

The acceptance of DCE has thus been slower than anticipated. It was dealt a further blow by the emergence of strong competition in the shape of newer object oriented options promoted by the Object Management Group and Microsoft. Microsoft’s option, DCOM, is based upon DCE protocols under the surface, though.

**Distributed Objects**

Object orientation offers many advantages which are discussed in notes for another module. One area of particular relevance here is the fact that the inherent communication of objects is the passing of messages. This means that objects can be distributed easily, provided that a mechanism for the communication of messages exists. This can easily take the form of network connections within client server architectures. Object environments and languages thus map well to client server. DCE and RPC mechanisms have been employed within some implementations to achieve the transport of messages, however, as noted earlier, due to the synchronous nature of many implementations, this has not been ideal, especially in networks containing many workstation nodes.

In the late 1980’s and early 1990’s, a standard was defined for object oriented middleware by the Object Management Group, a non-profit consortium dedicated to advancing the adoption of object orientation through standardisation efforts. In 1990, the Common Object Request Broker Architecture (CORBA) release 1 specification was made available. This specified the services and facilities which a vendor would need to make
available in an Object Request Broker (ORB) to provide object oriented messaging middleware.

CORBA I was moderately successful and fairly widely implemented by a wide variety of vendors, with the notable exception of Microsoft. CORBA enables multiple application components, residing in heterogeneous computing platforms, and potentially written in a variety of object languages, to seamlessly and fairly transparently communicate using a message passing paradigm.

20.10 - OMG Object Management Architecture

Language bindings currently exist for C, C++, Smalltalk and COBOL. Java bindings are available from several vendors, but are not yet standardised. While CORBA 1 specified the services and the application interface to these services successfully, it did not define how the distributed parts of the ORB itself should communicate - this was left up to the implementation. The result of this was that application components could communicate seamlessly, but only if the same vendor's ORB software was used on both ends. While this did not present a major problem for inhouse applications, it did inhibit the growth of interorganizational systems based upon ORB's and the expected growth of the component marketplace.

CORBA 2, released in 1994, addressed these limitations by specifying an inter-ORB protocol. What this means, is that provided an application
conforms to the CORBA 2 specification and uses a CORBA 2 compliant ORB locally, it can seamlessly communicate with other objects anywhere provided that they too interface to another CORBA 2 compliant ORB. This opens the door to very exciting possibilities for the development of components and the rapid growth of interorganizational systems across heterogeneous networks and platforms.

20.11 - CORBA 1 and CORBA 2 Compared

The OMG has also implemented a certification programme to assist vendors in achieving true compliance with the CORBA 2 specifications.

IBM has strong support for CORBA in their System Object Model (SOM) and Distributed SOM (DSOM) products. SOM provides messaging between components within the same machine, while DSOM provides distributed messaging across networks.
Microsoft meanwhile has seen the commercial potential for dominance of the inter-object communication API. With its dominance of the desktop, it was in a strong position to promote the adoption of its own much less capable, but more widely available, object messaging layer within Windows environments: Component Object Model or COM. One problem with this was that the API was not stable for some time, and had slight variances depending upon the flavour of Windows in use. These problems created considerable annoyance and expense for developers. Also, COM was only available in Windows and only facilitated communication within a single machine, which did not help if we wanted components to reside on Unix, AS400, or mainframe platforms, or to communicate in a distributed environment.

Microsoft next embarked on the expansion of COM to include support for communication across a network between Windows machines. This was also confusingly named COM, for Common Object Model.

Later initiatives by Microsoft include the bundling of COM within Object Linking and Embedding (OLE) version 2, which uses the COM facilities, and collaborative ventures with DEC and Software AG to develop Enterprise OLE, which should support the API's on nonWindows platforms. Their efforts have been helped by additional vendors wanting to tap into Microsoft's success and to attract Windows experienced developers. Several
vendors now provide ORBs which will support both a COM (Windows style) API and the CORBA API.

Microsoft's offerings are dominant on the desktop, simply because they come bundled with Windows. They are still less capable than the CORBA compliant products in open, heterogeneous networks. OLE is more mature than other competing products, but is now coming under competition from OpenDOC, a product of IBM, Novell and Apple. OpenDOC offers similar, and sometimes superior facilities for linking and embedding objects and does it across platforms and environments, as well as using CORBA facilities to achieve the linking across both LANs and WANs.

A compound document might have the basic structure held on a Unix box, an embedded graphic in a Windows format and a linked spreadsheet held on an OS/2 machine elsewhere in the network. All of this can be achieved transparently to the user. Implementations of OpenDOC are now emerging quickly. A key advantage of OpenDOC is that it also supports the OLE facilities, so it is a superset. With only a little extra effort, developers can write for OpenDOC rather than OLE, which makes facilities of both available, and will allow their applications to function in an open heterogeneous, rather than proprietary closed environment.

New Generation Tools

Early client server tools were relatively unsophisticated and left much of the interfacing and management as well as logistics of testing and deployment to the developer. Most early tools were either client or server oriented, but not both. Also, very few supported the three tier architecture. New tools have now emerged to address the full spectrum of client, application server and database server functionality within single or integrated products. High end tools do this across a wide range of platforms in a three or n-t tier architecture. Some tools also provide sophisticated CASE facilities for analysis and design modeling, and in some instances code generation from models. Some also provide distribution and version management in the production environment.

New tools include:

- Forte, from Forte systems
- Dynasty from Dynasty
- COOL Gen from Sterling Technologies
Some pure object tools have also embraced the open heterogeneous distributed environment. A good example is the Enterprise Object Server from Gemstone, which can integrate Smalltalk client tools such as Visual Works (ParcPlace Systems) or Visual Age (IBM) with object database and legacy relational environments.

**The Internet as a delivery platform**

One of the newest and most exciting platforms for the delivery of applications has arisen almost as a byproduct of other services. The Internet, which started life as a US Defence Force Advanced Research Projects Agency initiative, then spread to academia, and finally commercial users, has grown exponentially. Services available on the "net" include e-mail, file transfer, various search engines to help locate relevant information and the killer application, the world wide web (WWW) or simply, "the Web". First conceived at the European CERN labs with a text-based user interface, and later refined at NCSA in America to incorporate a graphical interface, the web provides a consistent hypertext interface to a vast resource of information globally.

A key standard on the Web is Hypertext Markup Language, or HTML which defines how information should be stored on host computers to allow it to be properly interpreted, displayed and navigated by a raft of "browsers" available on almost every computer platform. Examples include Netscape, Mosaic and Microsoft's Internet Explorer.

HTML is powerful in that it allows the simple embedding of links, or references, to other information anywhere in the world wide interconnected network of millions of computers. These links are in the form of Uniform Resource Locators (URLs) which specify the name of the host computer, and the directory path and file name to locate the desired information. A protocol known as Hypertext Transport Protocol (HTTP) allows HTML documents, including links and embedded graphics, to be moved across the Internet for presentation at the user's browser.

Later versions of HTML also support the definition of forms and tables, allowing more complex formatting at the user interface. Forms allow users to capture information which is sent back to the host site for processing. This can enable simple client server applications across the Internet. While very limited in functionality, due to the fact that the forms are essentially "dumb" - no local validation or processing occurs, this is still an attractive delivery option since it allows developers to write one application which can provide
a capture facility for millions of users worldwide, on an almost free global network, which will work across a wide range of client platforms. The limitation is that we are essentially restricted to the Remote Presentation style of client server. This can be problematic from the perspective of response times (data must travel to the host for validation) and network traffic.

**Enter Java**

Exciting new possibilities have come about as a result of the development of Java by Sun Microsystems. Java is an object programming language derived from C. It is different to C++ in that it is interpretive, supports more dynamic features, has automatic garbage collection and disallows direct memory referencing. It is a much more developer friendly and robust language than C++, but will generally execute substantially slower. Conceptually, it falls somewhere between C++ and Smalltalk. Java has been promoted as a way to make Web browsers application hosts. This is achieved by equipping the browser with a Java interpreter. A web host can provide the browser with HTML pages, but also embedded Java "applets" - little applications. HTML information is displayed in the normal way, but Java code would be fed to the interpreter for execution.

The interpreter allows protection against viruses and rogue application code since it can control what system resources a hosted program can see. This is one reason why an interpreter is used, rather than allowing "foreign" object code to execute on our client platform. A second reason for using an interpreter is to allow the same code to be executed in a variety of browsers across a wide range of platforms.

These facilities allow developers the full range of styles of client server using the Internet as a delivery vehicle. We can now choose how much of the application logic we wish to place on the client and on the server. For example, we could download an intelligent form, which will do validation before sending the transaction to the remote host. Alternatively, we could send data from the host to the browser client, together with an applet which allows its visual analysis and interpretation, possibly as a graph.
Further possibilities include support for multimedia and virtual reality, where the intelligence to work with data in formats which the browser has not been configured specially to cope with, can be transmitted with the data to the user workstation for execution there using local processing power. These possibilities will spawn an incredibly rapid move to the availability of components and the use of the Internet as a delivery platform. Vendors are rushing to support Java, and it has been licenced by most major web browser makers as well as development tool vendors, including Novell, Microsoft, IBM, Borland and Symantec.

Microsoft is again promoting its own competitor to Java, in the form of Active-X controls (derived from the concepts of Visual Basic Controls (VBXs) and Object Component Controls (OCXs) previously available in the Windows environment. The jury is still out on which will become the dominant approach, but it is likely that major browsers will be forced to support both...
20.14 - Java Applets Provide Internet Platform

Even with Java on the client inside the browser, difficulties remain to create transactional and secure applications in support of commercial applications as required for e-Commerce. Since HTTP is a stateless protocol, there is no “session” established between the client and server. The server does not keep track of the “state” of the client and tends to see each new message as a discrete occurrence. While this is fine for content delivery (the original intent of HTML and HTTP) it is inadequate for transactional systems. Let’s look at an example:

- Say we are wanting to allow clients to place orders over the Internet
- The client browses the catalog of products and prices as normal HTML - no problem
- Having identified an article to purchase, the customer selects a link which starts a Java applet to capture the necessary details. The applet will also be able to calculate the order price to allow the customer to confirm the order and corresponding debit authorisation.
- The Java applet captures and validates information for legal types and ranges. It sends this information to the server. At the server, we retrieve pricing info and return it to the browser. The server application is awaiting an order from the browser, but has no control - the user can
select to send the order, or could hit the "back" button to return to browsing the catalogue, could use a search engine to find other sites offering similar products, or could disconnect from the network entirely. Complex programming involving the use of cookies and keeping track of user requests and identities is required to support a sensible transacting environment.

To solve some of these problems, it would be wonderful if we can have ORBs at the server and client, providing the rich services defined by CORBA (e.g. Naming, security, transactions, etc. This has been possible for some time, but normally required the installation of special software by clients. This is tolerable within an intranet, but would limit appeal of an e-commerce site on the Internet. But, hold on. If Java is portable and can be downloaded into the browser, and an ORB can be written in Java, then maybe we do not need anything special installed at the client. We can create a downloadable ORB (or ORBlett) which can turn the client browser into a CORBA compliant platform within a very short time.

- If we can run Java applets in a browser
- AND if we can write a light ORB in Java
- THEN we can create a downloadable ORB
- WHICH turns any browser into a CORBA applications platform!
- Commercial Products:
  - Visigenic VisiBroker for Java (prev. Black Widow)
  - Sun's Joe
  - Iona's OrbixWeb

Vendors have been quick to see the potential of this, and three implementations are already available: Iona’s OrbixWeb, Visigenic’s ViviBroker, and Orbacus’s Orbacus. Even better, since late 1997, Netscape has bundled Visibroker with Netscape Communicator, so it is not even necessary to download the ORB!

With an ORB either side, the Internet becomes a terrific global client server platform, which can be secure, robust, transactional and extremely cost
effective. For around $US 50 per month, a business can host a site with a worldwide reach.

20.16 - Internet as a Client Server Platform with Java ORB

So, by the end of 1997, we can regard the Internet as a very cost effective, secure, transactional client server development environment. The potential for this, coupled with the massive interest in e-Commerce, has fuelled the rapid development of tools and applications written using them. It has also played a part in encouraging the OMG and Sun to put Java and CORBA technologies together in defining the forthcoming CORBA 3, which supports a component model very like the Java Enterprise Beans, and the Java 2 Enterprise Computing Architecture, which includes EJB and associated services, which are very close to their CORBA counterparts.

Component Models

Objects are great for experienced developers who have highly developed design skills and know how to work at a good level of abstraction. Using a mature and complex class library developed by someone else can be a daunting task for a typical commercial developer with less experience and working under intense pressure to get new solutions delivered into production. One of the advantages of objects, after all, is the encapsulation of complexity, hiding implementation details away behind a friendly interface. This means that if we are happy to use the provided objects unmodified (as we typically might for standard functions like graphical user
interfaces, talking to relational databases, communicating over a telecommunications link with a standard protocol and the like, then we can work with the objects at a component level instead.

With components, the source is maintained by the developer of the library and may not be released to the user, although it sometimes is. What the developer deals with is the available components, which present their attributes through editable properties (e.g. Size, colour, position, font, etc.) and their behaviours through a published interface. This is a list of the messages the object will respond to (or, in procedural environments, the calls we can make to it). Thus, a window component may respond to messages such as “Open”, “Close”, “Minimize”, “Scroll Left” etc. The developer thus follows a simple process when working with components:

- Choose a library or category of components which fits the current needs - these are often provide with the development environment
- Select a component of the desired type to include in the application e.g. Compose an interface of a window, menu items, text labels, entry fields and buttons, for example
- Customize the properties of the object to suit e.g. Change the label of the buttons, the title of the window, the size of the entry fields
- Link actions and events to the desired behaviour using the component’s provided interface e.g. When the Close button is clicked, send a message to the Window asking it to Close

Components allow rapid construction from pre-built and tested parts. This is a major revolution in the construction of applications and may have as profound an effect on our industry as the move to separate component suppliers and mass assembly was in the motor industry.

**Competing Component Models**

In the early days, components were very language dependent and frequently also tied to the operating environment and processor architecture. For example, you might get a library written in C++ for the Windows operating system on an Intel processor. Some vendors provided libraries with various versions of the components for a variety of platforms, allowing developers to write portable applications which could target a number of deployment platforms.

Ideally, though, we should be able to write code in the language of choice (which is well matched to the problem we are trying to solve and our skills), but invoke functions or share our capabilities with other components written by other developers in different languages. Going beyond this, into the world
of distributed systems, inter-organizational systems and global networks, such as the Internet, we may want to invoke components which reside elsewhere on a network, written by a third party, in a language unknown to us and executing on a variety of platforms. One of the earliest attempts to address this problem was CORBA, the Common Object Request Broker Architecture, from the Object Management Group (OMG). CORBA allows developers to write objects in a variety of languages and then to publish the interface to them via an Interface Definition Language (IDL).

Derived from C, IDL is a standard way of describing the interface to an object independent of the language in which the object itself is written and the platform on which the object executes. IDL defines the messages the object will respond to, the parameters which are expected, their types and what will be returned. When a developer uses a CORBA object with an IDL interface definition, the information about the interface is recorded in a repository, which is used by the ORB to invoke the object. Other parts of the architecture perform translation between the native types of the object programming language and the standard form used for communication between ORBs. I can thus pass a parameter from an object written in Smalltalk to a remote object written in C with the ORB and associated software providing the necessary translations and managing the communications. There are language bindings, which specify how the translations occur, for COBOL, C, Smalltalk, C++ and Java.

Containers

A container is a place where a component can run. Just like the component provides a published interface, so the component itself needs to know what services it may call upon in the environment where it is to execute. The idea of a component model is to specify protocols which components will adhere to, as well as what they can expect of their containers and how this is invoked. Sun exploited this concept by introducing Java Beans, which were visual components designed to run within a virtual machine within a browser.

Java Beans adhere to a protocol which allows them to be used in a variety of development tools. The tool can, for example, determine what properties the bean has, and open a properties editor to allow the developer to customise it. The attraction of Java Beans stems from the portability of Java, which allows developers of libraries to target an enormous audience - the beans can run anywhere there is a Java virtual machine. If they can also be used in a variety of development tools, that is even better! For bean users, it gives them a
source of components from a very wide variety of suppliers. This means that the market is competitive, keeping costs down and quality up.

Microsoft used modules in the form of dynamic link libraries as early components in the Visual Basic and C++ environment. In VB, these were known as VBXs or Visual Basic Controls. Initially, these were 16 bit modules for the Windows 3.1 API. With Windows NT and ‘95, and the move to 32 bits and increased use of COM for messaging, Microsoft extended this model to OCX’s or OLE Controls, which were 32 bit components for the Windows environment. When Sun Introduced Java Beans, which looked like staking a significant claim in the rapidly emerging Internet marketplace, and indeed, threatening the dominance of Windows itself by making it possible to create rich, portable GUI interfaces without reliance on the Windows API, Microsoft responded by repackaging the OCX technology into ActiveX, which added the capability to download the control over the Internet and execute it from within a browser.

From an end user perspective, Java Beans and ActiveX controls offer much the same capability. From an architectural perspective, the technologies are very different. A comparison is given below:

<table>
<thead>
<tr>
<th>Java Beans</th>
<th>ActiveX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move to browser as source (or tokenised) code</td>
<td>Move to browser as compiled object code</td>
</tr>
<tr>
<td>Single version for multiple target platforms</td>
<td>Multiple versions - one for each target platform (e.g. Intel, PowerPC)</td>
</tr>
<tr>
<td>Secure. Execute within a Java Virtual Machine provided by the client. VM can be configured to allow/disallow whatever user wants. Java has mechanisms to ensure that code has not been tampered with in transit. Applets can be verified as from trusted source.</td>
<td>Unsecure. Execute directly on the hardware/OS. Can easily do things not desired by client. Microsoft has tried to implement a certification scheme to ensure that controls are from trusted source, but this could be “spoofed”.</td>
</tr>
<tr>
<td>Interpreted on client. Performance dependent upon efficiency of VM. Slower than native code.</td>
<td>Execute as native code, therefor fast.</td>
</tr>
</tbody>
</table>
Published standard. Microsoft implementation supersedes any published specification. Often leaves non-Ms developers playing “catch up” as interfaces change.

Written in Java Written in MsLanguages, typically C++ or VB

Any platform supporting a Java VM Platforms supporting the Win32 API

Java “look and feel” - more similar to typical Unix GUI’s Windows look and feel

Following the above, we can draw some conclusions. If you are working on the Internet and require the ability to deploy a single code base to run on a diversity of clients on heterogeneous platforms with good security, then Java is your best bet. If you are running in-house on an intranet, are committed to the “Microsoft way” and are not worried about security (the components your clients will download are your own) then ActiveX may be the way to go. While we do not like ActiveX much architecturally, it has provided a quick way to migrate huge libraries of functionality from previous VBX and OCX libraries to the browser environment. We caution that those in-house applications can quickly become inter-organizational systems in today’s world of rightsizing, outsourcing and business partners. This may mean that it is wiser to invest in going the more portable route.

Server Components

While Java Beans and ActiveX controls target the visual client environment, we could potentially benefit greatly from components on the server, as well. The requirements are somewhat different, however. Server components typically:

- Have to be reentrant or multithreaded to allow multiple simultaneous users to access the functions they provide
- Have to be efficient and execute quickly, using minimal resources
- Must be robust, recovering themselves if they fail for some reason
- Should be replicatable to handle heavy workloads
- Require transaction management to allow the committing or rolling back of sets of updates to persistent data
- Do not have a visual interface

They may also may demand much more of their container, including:

- Starting the object in response to a client request (loading it from storage and making it active, then passing the initial message)
- Restarting it in event of an error having occurred
• Handling memory allocation, pooling, buffering and unique identities
• Dealing with priorities between various services offered
• Making the instance persistent between invocations
• Checking security
• Providing transaction management services

Sun saw an opportunity to create a standard for server components based upon Java Beans, but taking the foregoing into account. The new style beans are dubbed “Enterprise Java Beans” or EJBs. Their stated goals were to create a standard that would provide server components which were:

• Scalable (global web serving)
• Reliable (redundant, no single point of failure)
• Efficient (sharing of resources, threading etc)
• Transactional
• Secure
• Flexible
• Easy to maintain, manage

They also did something very smart in not constraining where EJBs might run, except to define characteristics and services which the container must provide. Containers are responsible for:

• Threading
• Creation, Destruction
• State Management
• Resource Sharing
• Identity Resolution
• Transaction Management
• Persistence
Surprisingly, many different types of system software can be extended to provide these facilities. Vendors were quick to perceive a major opportunity to become EJB hosts and quickly began adapting products to support the EJB standard. Some of the types of software now hosting EJBs include:

- **Transaction Processing Monitors (TPMs)**
  e.g. IBM CICS, BEA Tuxedo (Now M3), Encina, Jaguar, Microsoft Transaction Server

- **Database Management Systems (DBMS)**
  e.g. Oracle, Sybase, DB2, Gemstone

- **Object Request Brokers (ORBs)**
  e.g. Visibroker, Orbix

- **Application Servers**
  e.g. Gemstone, Bluestone

- **Web Servers**
  e.g. Java Web Server, Netscape Enterprise Server, IBM Websphere

- **Network Operating Systems**
  e.g. Novell Netware

Like Java Beans, EJBs can be authored in a variety of development tools, and deployed into many different environments on many different platforms. This creates a very attractive opportunity for component developers.
There are two types of EJBs:

- **Session beans**, whose intent is to hold data relevant to the duration of a user session. They will normally implement business process logic.
- **Entity beans**, which are persistent, normally stored in a database, and will typically represent a domain or business object, such as an order, customer or product

**Microsoft Alternative**

As usual, there is a Microsoft alternative. Microsoft promotes COM in-process DLLs as a server component model, in conjunction with COM (or more recently, DCOM or COM+) and the Microsoft Transaction Server (MTS) as a container. In an all Ms environment, this combination provides a fairly rich set of facilities at a good price point relative to the competition. In recent TPC benchmarks, this combination, on Windows 2000, has set new record levels for a single server throughput, even beating many Unix heavyweights. The latter still can scale beyond the MTS solution ultimately, since they support distribution of workloads across multiple servers. Nevertheless, the MTS results are food for thought and provide comfort to those shops already committed to the Microsoft route that they will be able to scale their applications without rewriting or re-architecting for a completely different environment.

**Wither CORBA**

Has all the activity with EJB and ActiveX spelt the death of CORBA? Well, it looked that way for a while, but let’s look a little deeper. When you examine the services expected of an EJB container, they look suspiciously like the services previously defined as part of the Object Management Architecture, of which CORBA is a part. It transpires that CORBA did a pretty good job of defining the fundamentals of how components and containers should behave, what services are required and how the elements communicate with each other. What CORBA lacked, was sexy new graphical development tools. CORBA worked, but it involved coding at source code level, compiling, linking and binding etc, things unpalatable to a new generation of developers brought up on visual tools. In effect, it was being beaten by the “X appeal” of the Microsoft environment.

When Sun defined the Java 2 standard, incorporating EJBs, they had on board many of the people who had created the CORBA standard. They used much of their knowledge and experience to build the new standard, which
draws heavily on the CORBA services. Also realising that the Java only components should be able to interoperate with other environments (e.g. Microsoft and legacy applications), they provided this by including the CORBA capabilities in Java 2. In fact, Java 2 implementations include classes that implement a CORBA 2 compatible ORB.

The plot thickens further, since the OMG liked the well defined EJB standard, and also wanted to capitalise on the marketing muscle and hype surrounding Java. OMG has thus adopted essentially the EJB specification, with slight modifications, to create a draft standard for..., you guessed it, CORBA beans! These will emerge with CORBA 3. A CORBA bean is essentially like a Java Bean (or EJB) except it can be written in any CORBA compliant language. With Java and CORBA camps growing so friendly, CORBA has also suddenly acquired a raft of really up to date visual development tools, including Inprise Delphi and J Builder, Symantec Visual Cafe and IBM VisualAge for Java. Some of the high end TP monitor vendors and ORB vendors have also brought out component assembly visual tools to do CORBA easily.

**Stateless or Statefull?**

The Microsoft model supports only a stateless architecture, where the server does not keep any session data between invocations. This is desirable from the perspective of efficiency, since it allows the server to deallocate resources quickly. It can complicate programming, requiring the developer to keep state on a persistent store between invocations, potentially having to remove it if the session is not completed. Microsoft argues that this is a necessary sacrifice to achieve efficiency and scalability.

By contrast, CORBA and EJB’s allow alternatives. Both statefull or stateless connections can be established and it is up to the developer to chose the most appropriate for the situation and application.

Time will tell how CORBA, Java and ActiveX/MTS models play out in the marketplace, but we can be sure CORBA/Java and ActiveX/COM+ will both be with us for the foreseeable future.
This page intentionally left blank.
Using Legacy Components

Our old systems contain valuable data and functionality which we can seldom afford to rewrite, and certainly not all at once. So, we need to access them from our object environment, but, we do not want to get back into a maintenance quagmire, with \( n \) different nasty technologies to use and maintain. What to do? The answer lies in making our old and useful stuff object friendly through a technique known as \textit{Wrappering}.

Identifying Useful Things

First, we need to know what we want to keep and reuse. This involves analysing our existing systems and their usage. High usage items are generally useful, even if they are odious. We would recommend creating a four box model to identify things worth wrapping. On the vertical axis, we look at things which are strategically important to the business. On the horizontal axis, we look at frequency of access. Items which fall in the upper right quadrant should obviously be our first candidates, especially those for which the effort to wrapper is low.
Harvesting

Harvesting is about taking the good candidates and making them object friendly then placing them in a library where they can be found and used. We make the previous code, modules and data friendly by providing it with a messaging interface. We write interface code that will invoke the desired functionality or retrieve the desired data on our behalf. On the friendly side of the interface code, we use pure standard object messaging. I.e. It exposes its features and services as object methods. On the legacy side, we use whatever means necessary to invoke the functionality we need. This can include:

- Calling the code
- Writing out interface files and triggering off batch jobs
- Pretending to be a mainframe terminal and submitting transactions
- Submitting card images
- Running batch jobs
- Calling library code

When we write the wrapper code, we should carefully document what features, functions, services and data are made available, as well as a bit about what is not available, but could be wrappered in future. This should go
into a library, together with the wrapper code. This provides a resource for all developers to draw upon in access the legacy assets.

**21.2 - Wrapper Turns a Legacy Application into a Component**

**IDL as a Lingua Franca**

If we plan to use a formal component model, such as that provided by CORBA or Java Beans, we should invest the effort to define the messaging interface in a standard interface language e.g. IDL. This will allow introspection (querying the objects about their available services) and will provide future proofing of the components.
Replacing the components over time

If we follow the above approaches, we will have clean access to our legacy domain components from our business logic. We can gradually unplug and replace legacy components with new ones as time, technology and money and resources permit. This can be achieved without disrupting the business logic, interface or user experience.
Visual Assembly

NOTE: This is a substantially revised version of a paper originally entitled: Visual Application Assembly from Components originally presented in 1994 to the Computer Society of South Africa.

Summary

Programming has been algorithm, language and code based for three decades. This reflected the needs of the machine for precise instructions and the skills base of early programmers who were mainly scientists and logicians. It was a paradigm which matched left-brain dominance and logical aptitude well. Unfortunately, these skills and aptitudes are in very short supply in the general population. This, coupled with the need for much more rapid application development and flexibility in the face of change, has necessitated a change in the way we develop applications.

Powerful machines and environments using object technology make it possible to develop applications visually and interactively in real-time using intelligent pre built components. This paradigm can lead to very rapid development, easier modification and higher quality and productivity through reuse. It can also make programming accessible to a much wider audience who think visually, predominantly with the right brain.

This chapter introduces the concepts of visual application assembly, identifies current commercial environments supporting the metaphor, and discusses some of the implications.
Introduction

Application development faces ever increasing demands:

- Business requires more sophisticated applications within shorter delivery times
- Technologies are changing very rapidly, creating a situation where developers are so busy learning new environments that they have little time to concentrate on the business requirements
- Applications are more critical to organizations and to safety, requiring that they be more robust and secure
- Competitive pressures and limited budgets mean that application development must become vastly more productive
- The dependence of business on system, and the rapidly changing nature of the business environment dictate that applications must be very easy to alter, and that they should allow businesses to quickly evolve systems to meet changing requirements
- A very limited proportion of the populace has the rigorous, logical aptitude for programming coupled with the high linguistic skills for programming language usage

Improvements in the Software Development Process

Humphrey, Gild and other authors show that improvements in software development productivity are usually realised by improvements in the process used rather than the tools.

This is borne out by the following table from the respected consultants Nolan-Norton:

<table>
<thead>
<tr>
<th>Decade</th>
<th>Technology and Toolset</th>
<th>Improvement Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1950’s</td>
<td>IO Control Systems/Machine Code</td>
<td>Base</td>
</tr>
<tr>
<td>1960’s</td>
<td>3GL Programming</td>
<td>+8% pa</td>
</tr>
<tr>
<td>1970’s</td>
<td>Modular Programming and Database</td>
<td>+3-6% pa</td>
</tr>
<tr>
<td>1980’s</td>
<td>4GL Programming &amp; CASE</td>
<td>+1-2% pa</td>
</tr>
<tr>
<td>1990’s</td>
<td>OO, Reuse, 5GL</td>
<td>??</td>
</tr>
</tbody>
</table>

22.1 - Improvements in software development due to tools

314
In contrast, organizations which have changed the process of system development to focus on quality and reuse of components have increased productivity as much as three fold (300%) [Card, Clarke, Berg, 1987].

Recent improvements in the process include:

- **Joint Application Development** (JAD) bringing users and sponsors intimately into the specification process, thereby improving specification quality
- **Prototyping** which has allowed clients and developers to visualize systems as they will appear in operation before major investment is made in detailed development
- **Rapid Application Development** (RAD) methods and lifecycles promoting small teams, prototyping, spiral/incremental delivery lifecycle and reuse. A good example of this is the eXtreme Programming (XP) approach
- **Continuous Process Improvement** models such as the Capability Maturity Model (CMM) and SPICE which use knowledge gained in each iteration to enhance the techniques used subsequently
- **Component integration technologies** which allow developers to integrate components or “parts” from a variety of sources to create rich applications quickly and reliably. Technologies supporting this include Java Beans (and EJBs) from Sun, CORBA from the OMG and COM+ from Microsoft
- **System Generation** from high level models and specification languages in I-CASE tools such as CoolGen and CoolJex
- **Visual Development** using tools which support the prototyping and production creation of user interfaces, including highly graphical ones (NextStep, Visual Basic, Delphi, PowerBuilder)
- **Object Orientation** which allows a better mapping of real world objects onto analysis, design and construction models; facilitates reuse; and allows creation of components encapsulating data definitions and behaviour - thus making them reusable without concern for their implementation

**Productivity and Quality**

Many studies have shown that productivity in systems work is inextricably linked to quality and reuse. There is a limit to the speed with which we can
create new code which is reliable, efficient and useful. To achieve higher levels of quality and productivity, we must reduce the amount of new code in applications, and increase the amount of pre-existing, tested, reliable code - i.e. We must write less and reuse more.

Reuse can potentially be achieved in a variety of ways:

- Reuse of code via libraries of functions (e.g. Fortran graphics library) or objects (e.g. Borland’s Object Windows Library (OWL)™)
- Reuse of entire applications by communicating with them via messaging (e.g. OLE) or embedding of objects (e.g. Embedding and Excel™ spreadsheet in a Word™ document)
- Reuse of definitions of objects, possibly with local extensions or alterations (e.g. Via use of a class library from which we inherit)
- Reuse of models or design frameworks (possibly via patterns) which simplify how we structure applications and provide a proven, working skeleton on which to hang our specific functionality
- “Cloning” and customizing objects (e.g. Window Controls, PowerBuilder’s Data Windows, Visual Basic Controls (VBXs or OCXs)

**Traditional Failings of CASE**

One of the great disappointments of recent years has been the failure of CASE technology to live up to the promises of faster application development and easy maintenance. Some of the failings identified include:

- **Lack of integration** between components of the tool, across the lifecycle, and with other tools. This has led to laborious manual intervention, causing many of the benefits to be lost, since we cannot alter designs at the model level in practice
- **Assumption of correct specifications.** Many tools assume that a correct specification can be captured, and that once this is complete, we can generate correct code for the whole system. As experience and the push towards iterative lifecycles proves, this is impractical in the real world. Large and complex systems grow through a succession of refinements and enhancements
- **Engineering Focus.** Tools have been primarily oriented to someone who is going to work from scratch to produce some new program. Very few tools recognise the need for or support the inclusion of existing code and components into new applications
• *Slow iteration*. Needing a correct, whole specification implies long delays between discussing requirements with a user and being able to interact with live code

Newer and more successful CASE tools are starting to overcome these limitations. Sterling’s COOLGen is demonstrably the most successful traditional I.E. CASE environment.

**Ways of Building a Car**

Let us use an analogy to contrast the different ways in which we could build a car (or a system):

• *From scratch, by hand*. When the likes of Daimler or Benz set out to build their first cars, they had to fashion every part from raw materials such as tell blocks, wood, canvas, copper wire, rubber etc. Painstakingly by the use of hand tools. This is clearly a very laborious and time consuming process. It also requires very high levels of skill. The finished article will probably not have a very high degree of sophistication and will be very expensive

This is how most people really build systems today. Given a specification, most developers will start coding! We build as much as 90% of applications from scratch. This means that we are doing things for the first time and can expect to make many mistakes. Fixing these mistakes will consume a great deal of effort and time, leading to poor productivity and unacceptable delivery times

• *Using Power Tools*. With the aid of steam, hydraulic or electrically driven tools, the process of transforming raw materials into parts could be speeded up considerably. This approach still requires high skill levels to use each of the power tools and a sequential repetitive production process to make it viable. It is capital intensive, since all the equipment must be obtained and integrated in a suitable plant and staff trained to operate it successfully. With mass production, cars can be produced quickly, but the creation of a new model, or customizing an old one is not easily accomplished

This is much like our current use of CASE technology, where automated tools support each step, but we are going through much the same process as the manual construction. If we were doing repetitive things, like making one car/program 10 000 times, it would be fine, but systems work is not like that. Reuse levels remain low, since the input to the process is still typically raw material (specification lines or model components)
• **Using component suppliers and assembling.** In this model, we rely upon specialist component suppliers to provide us with parts (e.g. Fuel injection from Bosch, tyres from Goodyear, spark plugs from Champion, etc.) And we only build those components that we cannot obtain elsewhere, or which provide the competitive uniqueness of our product (e.g. The three pointed star on the bonnet, or the BMW engine). We can rapidly evolve new designs, using many components already available. The finished product is highly sophisticated, since each supplier is focussed on creating the best range of components for a particular area (e.g. Instrumentation, tyres, etc.) And has built up a very high level of skill in this area, as well as being able to employ continuous improvement techniques on their components. We benefit in terms of cost from both economies of scale and competition.

This is where we are headed. The software component industry is maturing rapidly. It is now possible to source rich proven components from many suppliers and through conformance to standards, fairly easily plug these together into a sophisticated application. An example is integrating a database and a word processor via ODBC to create a marketing application. Reuse levels are very high: 80% plus

• **Power Tools for Assembly.** This extends the above approach by providing automation for the assembly of components (rather than their manufacture). In the car industry, this is exemplified by specialised automated assembly lines, positioning jigs and assembly robots

In software, we are seeing the growing acceptance of the early tools which provide this kind of facility: workbenches which allow the rapid assembly of software components into complete applications. In addition, they are capable of doing it in real time using a visual approach. They combine many of the process improvements we discussed earlier into a single integrated environment supporting a consistent process

• User involvement
• Prototyping
• Rapid iteration/feedback
• Object Orientation
• Reusable Components
• Integrated Assembly CASE
The Architecture of a Visual Assembly Tool

Most of these products are object oriented or object based. This is because the object approach encourages the building of components (or objects) which encapsulate both data (attributes) and behaviours (methods or functions) in such a way that the internal implementation is transparent to the user. The interface is explicit and that is all that we need to know to take advantage of the component. This is much like a physical component in the real world. The assembly line worker can happily fit a very complicated electronic engine management system by knowing which wires to connect where (or more likely, which single plug to push into which socket). He/she does not need to know anything about the internal workings of the box. Another good analogy is a calculator: we do not need to know anything about the internal circuitry to be able to push the right keys and get a correct answer.

Facilities which this type of tool should provide include:

- An underlying object oriented development environment or language which is general purpose (to tackle a wide range of applications) and open (to allow incorporation of components sourced outside this environment). The latter implies that the product adhere to some industry standards for component interoperability, such as Java Beans, COM+ or CORBA

- A rich set of components available to the developer for reuse (a “parts bin”). These should address inter-alia: GUI creation, Data Access, Communications, System Management and handling of Collections

- Facilities to visualize components and connect them together to create applications in real time. This requires tight integration to the underlying object environment. When we identify a kind of part to use and drag its image onto the construction canvas, we should be building real objects behind the scenes which can be run immediately
22.2 - Architecture of Visual Assembly Tool

- Interactive testing and debugging aids which help ensure that created applications are correct
- Prototyping tools to create the user interface interactively and visually
- The ability to customize created objects (e.g. Specify the colour, font and size of a control) easily and quickly
- The ability to create new components not provided by the environment and integrate these seamlessly into the “parts bin”

Additional desirable characteristics include:
- The ability to support multi-tier client server applications with layering to facilitate partitioning of application components
Multi-user and multi-project support to scale up to enterprise level development

Cross platform capabilities to allow single model creation of applications targeting multiple platforms, databases and GUIs

Large after market or third party components available, or open specifications published by the tool supplier to allow this to emerge

Repository to store design level information in an accessible way

**Commercial Products**

A number of commercial products have now emerged to support this kind of development. It is little surprise that most of these are built on the rich object oriented development environment of Smalltalk which already provides many of the features necessary to create this kind of tool, including:

- Pure object oriented language and runtime environment supporting dynamic binding, dynamic data typing, automated memory management, and a system dictionary containing details on all global objects
- Automated browsing and cross reference tools which allow management of components, classes and categories, facilitating reuse
- High level language accessible to commercial developers, upon which a unique application vocabulary can be built
- Rich class library (450-1000 classes) which provides most required application support, interfaces to data stores (OO and relational), interfaces to communications (e.g. TCP/IP and CORBA messaging), and interfaces to other products and standards (e.g. DDE, OLE, DLL's, SQL etc)

Smalltalk-based visual component assembly environments include:

- Visual Smalltalk from Digitalk (now merged into Parc Place Systems, itself renamed ObjectShare) which incorporates the "Parts Workbench"
- Visual Works from Parc Place Systems/ObjectShare (the original developers of Smalltalk) which runs on top of the Object Works environment
- Object Studio from VMark Software which uses the Smalltalk-based Enfin product underneath
- VisualAge from IBM, which uses IBM Smalltalk as the engine. Now also provided for development of Java applications
22.3 - VisualAge Visual Construction Editor

- ARGOS from Versant, also based on the Parc Place product, which adds many enterprise level development aids including high level modeling

Non-Smalltalk examples include:
- Object Management Workbench from Intellicorp, which is built on Pro-Kappa, an expert system and object development environment
- Novell's Appware based on an underlying C language product
- Next's system level application assembly from Portable Distributed Objects (PDO) using the Objective C language environment

The Smalltalk products have the distinct advantage of an industry standard language, with very rich and mature class libraries, but Java tools are catching up, particularly with the Java 2 Enterprise Edition Architecture.

**A typical construction scenario**

Let's take a look at how we go about building a small application in a visual construction tool. We are going to build a very basic telephone index system,
with the data stored in a database. The system should allow entry of new names and numbers, and retrieval of entries based on names or partial names. The interface will be graphical. Our screen could look something like the figure below.

To build the application we proceed as follows:

- Browse through the Parts Catalogue to find a Window. Click and drag with the mouse to place this on the construction workspace. Visually drag and size the window to suit our requirements.
- Double click on the window object to edit its properties, such as title, font, colour, initial state (minimized/maximized etc)

![Sample Application User Interface](image)

22.4 - Sample Application User Interface

- Browse the catalogue and drag two entry fields onto the window. Size them for the name and telephone number respectively. These could be simple text gadgets, or come with predefined validation (e.g. for a phone number)
- Drag titles for the entry fields onto the display and edit them to reflect the prompt text that we want for each field. Size, colour etc could also be customised
- Drag two buttons onto the window. Edit labels to make one a "Save" button and one a "Find" button
- Drag a database accessor part from the catalogue onto the workspace background (not the window). This is a non-visual part which allows
object-style access (by sending messages) to a relational table. It is an object which acts as a proxy for the table. We edit this to name the file for storage of the telephone list data, itemize the columns for the table (we could do this visually by clicking on the entry fields in the window) and select as key the "name" field.

At this point, we have all the objects we need to create the application. What we need to do now is to endow the application with behaviours by connecting the parts so they interact in a useful manner. Again, this is achieved visually. Each part is an object which has a predefined set of properties (some of which we edited above) as well as events to which it can respond (e.g. a button will recognise a "left click" or "right click" of the mouse); and messages which the object will generate to advise other interested parties of changes in its state, or to request services from other parts.

Clicking on one part with the right mouse and dragging a link to another part can bring up a requestor asking us to associate the messages generated by one part with the events of other parts. This is an intelligent facility which will only display legal messages and events for the two parts involved, and may even suggest defaults for how these two kinds of part are normally connected. Using these facilities, we perform the following actions:

- Link the fields in the database accessor with those in the window so that whenever the value in one changes, the corresponding one in the database or window is informed (i.e. connect the "valueChanged" event in each case to the "setValue" message of the corresponding field, in both directions)

- Connect the "clicked" event of the "save" button to the "write" message of the database accessor. Connect the "clicked" event of the "find" button to the "getGreaterOrEqual" message, of the database accessor

Once we have done this, we should have a working application. We can launch and test it immediately. Iterating to add new items to the window or application, or to add additional behaviours should be accomplished easily and quickly.

Notice that the entire application was created without writing a single line of code. This is possible provided standard parts are available for the things we want to do. If this is not the case, or if our application has particularly complex or non-standard behaviour, we could:

- Customize predefined parts using a scripting language (very similar to the tool host language, but with different variable naming conventions), or
• Create entirely new parts in the base language. These could then be given icons and included in the parts catalogue to extend the items available

Even in the latter case, a very small amount of new code needs to be written, typically less than 10% compared with the amount of code in the reused parts. Other facilities might allow:

• Building of subassemblies which combine parts into larger parts preserving encapsulation, allowing these new, more complex parts to be included in applications in their entirety as single objects. For example, we could embed our whole telephone number application within a customer contact application

• Dynamic invocation of parts at run time to reduce memory usage, or allow flexibility dependent upon circumstances (polymorphism)

• Packaging other objects built in other environments (e.g. a compiled language such as C++) so they can be presented and used as parts

• Partitioning the application by drawing boundaries between parts and indicating target platforms or environments for these. This can greatly facilitate the ease of designing for multi-tier client server environments

• Support for communication with legacy data or systems via interface parts, which make the complexities and differences of the other environments transparent to the visual developer (an example is Digitalk's interface parts for the CICS environment)

• Incorporation of distributed objects conforming to the CORBA specification from the OMG via an ORB (transparent to the visual developer)

From the above, it should be apparent that the visual application assembly approach can be applied widely, that it can scale up to handle large and complex multi-developer systems, and that it can seamlessly address some of the current challenges in technological environments, such as client server, messaging and legacy integration.

**Early Productivity Indicators**

Combining data from the Capers Jones/Software Productivity Research language productivity database with metrics from early visual assembly projects produces some interesting results. These should be regarded with caution, since the sample sizes were relatively low. Nonetheless, the results look promising.
Implications

This author believes that visual development environments based on object oriented technology will have a rich and productive future. They should grow in popularity very quickly as organizations come to appreciate their ease of use, insulation from underlying technological turbulence, productivity through reuse and support for heterogeneous distributed environments. Smalltalk based products should dominate because of their maturity, standardised, rich fully object-oriented language and an extensive component industry which is emerging. This is supported by the IDC which found that Smalltalk was the most rapidly growing development language in 1995 (60% per annum). The Smalltalk Industry Council has been founded to promote the development and use of Smalltalk. This was before the hype and excitement of Java and the Internet which has now grabbed centre stage. It looks like Java tools (and possibly some on the very similar CORBA component model) will now dominate.

Challenges for the visual assembly tool makers will be to take on the established CASE vendors in producing tools which scale up to support enterprise level multiple team development. Visual development may prove to be as fundamental a change to building business applications for systems people as the move to graphical user interfaces was for users. Remember though, that improvements in productivity come from changing the process of building systems, not purchasing new tools. No major new benefits will
be achieved by using the new tools without a corresponding change in the way that we build systems.
References and Further Reading

Baum, David, 1995, Welcome to the Visual Age, Datamation, vol 40 no 17
Booch, G 1991, Object-Oriented Design with Applications, Benjamin Cummings Publishing
Card, Clark and Berg, 1987, Improving Software Quality and Productivity, Systems, vol 29 no 5 June
Coffee, Peter, 1995 Easel tames visual application development, PC Week, vol 11 no 23
Coffee, Peter, 1995 VisualAge: Industrial-Strength Smalltalk, PC Week vol 11 no 17
Digitalk Inc, 1995, Smalltalk/V and PARTS Workbench Product Literature
International Business Machines, 1995-2000 Visual Age Product Literature
Intelicorp, 1994 Kappa and OMW Product Literature
International Data Corp., 1994 The Smalltalk Report, commissioned by the Smalltalk Industry Council
Object Technology International, 1995 Envy Product Literature
O'Brien, Timothy, 1995 Hewlett-Packard's client-server strategy, Distributed Computing Monitor, vol 9 no 4
Parc Place Systems, 1995 VisualWorks Product Literature
Roti, Steve, 1994 Gemstone DBMS, DBMS, vol 7 no 3
Servio Corporation, 1994 Gemstone and GEODE Product Literature
Taylor, D A 1990 Object Oriented Technology: A manager's guide Addison-Wesley Publishing 1990
Watterson, Karen, 1995 PARTS Workbench 2.0 for Windows, Windows Sources, vol 2 no 4
Internet/WWW
Smalltalk FAQ by David N Smith (IBM) on comp.lang.smalltalk
Patterns

History

Patterns in one form or another have been with us for many centuries. Clothes makers know how to use a pattern to produce a garment of a certain style for a certain purpose, say a long evening dress. A cabinet maker may use a similar concept to produce a certain piece of furniture. In each case the article produced will have its own unique characteristics e.g.

- For the dress: size appropriate to the wearer; fabric according to occasion and budget; fastenings appropriate to fabric etc.
- For the item of furniture: the type of wood and finish; the fittings; the detail of the joints and so on

But each will also conform to the overall style, set of parts and criteria and relationship between its elements that distinguishes it from other clothing articles or items of furniture. A pragmatic definition of a pattern might be:

An accepted solution to articulated requirements within a context

The pattern expresses the general characteristics of a good solution, which has proven effective on many occasions. It does not specify every detail of the physical solution delivered, but rather the overall shape, concept, relationships and key attributes of a suitable design or solution.
The use of patterns in software specification and design is generally attributed to work beginning with the application of pattern concepts expressed by the eminent (building) architect Alexander to software designs. This involved Kent Beck and Ward Cunningham, who wrote up five patterns using the Alexander style to assist novice Smalltalk programmers and presented a paper at OOPSLA’87. Other early software pattern pioneers include Jim “Cope” Coplien, who wrote up idioms and style guidelines for C++ and Erich Gamma, Richard Helm, John Vlissides and Ralph Johnson (the so called “Gang of Four” or GOF) who published a landmark book on Software Design Patterns in 1994 following much activity in the OOPSLA conferences and other forums in the preceding few years. The first dedicated Patterns conference (PLOP) was held in 1994.

Definition

The best definition for software related patterns is hotly debated, but there are some useful ones:

- **From the preface of the 1994 “GoF” book:** “simple and elegant solutions to specific problems in object-oriented software design. Design patterns capture solutions that have developed and evolved over time. Hence they aren't the designs people tend to generate initially. They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software. Design patterns capture these solutions in a succinct and easily applied form”

- **Dirk Riehle and Heinz Zullighoven** offer the following: “A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts”

- **Brad Appleton** suggests a definition used in the patterns community: “A pattern is a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces” or more succinctly: “A pattern is a named nugget of insight that conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns”

So patterns are the essential abstraction of a type of solution which balances competing forces in a given context and provides a guide to a suitable solution. Some essential elements:
• The pattern expresses a general solution, which will be tailored to the specifics when applied
• The pattern is suitable to a certain context, and will probably not work in other contexts (at least not in the same form)
• The pattern provider will normally guide the user by specifying what the rationale was in arriving at the solution i.e. What forces caused the choices that were made and what priorities were used in decision making - i.e. What does the pattern optimise and what does it sacrifice?

A good quote from Jim Coplien illustrates the nature of a good pattern in illustrating the solution so that it is truly grasped by the user:

“I like to relate this definition to dress patterns. I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself.”

Like a dress pattern, patterns will often describe the relationship of various parts to each other and the essential characteristics of the organization.

**Types of Patterns**

Related to information systems, there are various types of pattern:

• **Architecture patterns**, which describe the structure of a solution. This will normally provide overall organizing guidance. An example of this is the Model-View-Controller pattern

• **Conceptual or domain patterns** are usually expressed in the language of the problem or application domain and may be independent of any software artifacts such as classes, data structures etc. An example would be the system of double entry bookkeeping

• **Design patterns**, which may describe the structure of a solution in terms of classes, data structures, algorithms and other computer related but environment independent artifacts. An example would be the way in which we organize an extensible set of attributes for an item with measurement values, units of measure, legal values etc. Another might
be the process pattern for performing maintenance on an item (add, change, delete, enquire) with appropriate validation

- Idioms or programming patterns express design patterns or finer grained solutions in terms of a specific programming language and technical environment. An example might be the collection pattern in Smalltalk

**Pattern Languages**

Pattern languages are structured ways of expressing the definition, suitability, intended context, rationale and other characteristics of a pattern or set of related patterns. They allow for comparison, selection, cataloguing and more accurate communication. Generally accepted elements which should be included for a pattern include:

**Abstract**

Brief summary of the information below. A few paragraphs which allow the reader to determine the usefulness of continuing to examine the pattern in detail.

**Name**

A short, descriptive and unique name that quickly allows us to identify the pattern, or communicate to someone which pattern we are using. If there are several names, we may detail these as aliases.

**Problem**

A statement of the problem which describes its intent: the goals and objectives it aims to satisfy within the given context and forces. Often the forces oppose these objectives as well as each other.

**Context**

The preconditions under which the problem and its solution seem to recur, and for which the solution is desirable. This tells us the pattern's applicability. It may document the situation to which the pattern is a solution.
Forces

What are the factors that influence the situation? Which items will the pattern help you address? What tradeoffs are made in which dimensions? What priorities does the pattern use in choosing a solution?

Solution

The abstracted design that provides the general solution. It will often take the form of a model (e.g. Class Diagram, Process Model, Event Diagram), or for Language Patterns, code templates. Specializations or variants may be documented.

Examples

One or more sample applications of the pattern which illustrate: a specific initial context; how the pattern is applied to, and transforms, that context; and the resulting context after application. Examples help the reader understand the pattern's use and applicability. Visual examples and analogies can often be especially useful. An example may be supplemented by a sample implementation to show one way the solution might be realized. Easy-to-comprehend examples from known systems are usually preferred.

Resulting Context

The state or configuration of the system after the pattern has been applied, including the consequences (both good and bad) of applying the pattern, and other problems and patterns that may arise from the new context.

Rationale

The decision making process used by the pattern provider in reaching the solution offered. Why were the choices made? How does the pattern work? What advantages does it offer?

Related Patterns

Patterns that may precede, follow, depend upon, use or otherwise be related to this pattern. Patterns which may be considered as alternatives.

Known Uses

Where has it been used to good effect already?
Beyond documenting individual patterns, a pattern language may also describe how patterns can be used together, which ones would conflict or represent opposing choices or optimisations. It should assist the user in selecting and combining suitable patterns that will lead to solutions that are durable, functional and aesthetically pleasing.

**Pattern Sources**

There is a rich and growing literature on patterns. A starting point for exploring this would be the References and Further Readings at the end of this chapter. Patterns are widely discussed at software architecture, OO and patterns conferences, including OOPSLA and PLOP. There are several pattern libraries on the web, including the Portland Pattern Library.

**Anti-Patterns**

Anti-patterns serve two main purposes:

- Documenting bad patterns and situations that you can land in if you make the wrong choices (basically what to avoid)
- Providing a way of extricating yourself from a problematic situation. An example of this would be refactoring code to achieve better reuse and reduced maintenance
References and Further Reading

Doug Lea's "Patterns-Discussion FAQ"
http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html


Doug Schmidt and Ralph Johnson's introduction to the October 1996, CACM Special Issue on patterns
http://www.cs.wustl.edu/~schmidt/CACM-editorial.html

Jim Coplien's SIGS management briefing Software Patterns
http://www.sigs.com/books/wp_patterns_5pp.html

Jim Coplien's paper "Software Design Patterns: Common Questions and Answers" ftp://st.cs.uiuc.edu/pub/patterns/papers/PatQandA.ps

John Vlissides' article "Patterns: The Top 10 Misconceptions" in the March 1997 Object Magazine Online

The "History of Patterns" on Ward Cunningham's WikiWiki Web
http://c2.com/cgi-bin/wiki?HistoryOfPatterns

"Pattern Definitions" from the Patterns Home page
http://hillside.net/patterns/definition.htm

Steve Berczuk's "Finding solutions through pattern languages"

"Some Notes on Christopher Alexander", by Nikos A. Salingaros
http://www.math.utsa.edu/sphere/salingar/Chris.text.html

"Design Patterns: Elements of Reusable Architectures", by Linda Rising
http://www.agecs.com/techpapers/patterns.htm

Brian Kurotsuchi's Design Patterns Tutorial
http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns/

Ravi Palepu's "Modelling the Real World: Application of Patterns to Reduce Complexity in the Software Development Process"
http://www.scs.carleton.ca/~palepu/pat.html

Doug Schmidt's "Pattern Writer's Workshop Tutorial"
http://www.cs.wustl.edu/~schmidt/writersworkshop.html
Ward Cunningham's "Tips for Writing Pattern Languages" on the WikiWiki Web http://c2.com/cgi/wiki?TipsForWritingPatternLanguages

"A Pattern Language for Pattern Writing" by Gerard Meszaros and Jim Doble
http://hillside.net/patterns/Writing/pattern_index.html

Richard Gabriel's article "Developing Patterns Studies" from InfoWorld on-line
http://www.infoworld.com/cgi-bin/displayArchives.pl?dt_iwe05-97_72.htm

A "Patterns BookList" on the WikiWiki Web

Other more general patterns resources on the web are:

The Patterns Home Page
http://hillside.net/patterns/patterns.html

The Portland Pattern Repository
http://www.c2.com/ppr

Ward Cunningham's wonderful WikiWiki Web
http://c2.com/cgi/wiki?WelcomeVisitors

Cetus Links: Patterns, hundreds of links to pattern-related pages
http://www.objenv.com/cetus/oo_patterns.html

Brad Appleton's "Software Patterns Links"

Excellent!

AG Communications Systems Patterns Pages
http://www.agcs.com/patterns/

The OrganizationPatterns FrontPage
http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns

The "Organization-patterns Mail Archive by thread"
http://www.bell-labs.com/~cope/Patterns/organization-patterns-archive/
Mapping the Models to a Layered Architecture

Introduction

Domain Object modeling, business process modeling, prototyping and event modeling, as we have discussed in preceding chapters, allow us to build an extremely rich and subtle picture of how the system might simulate the real world and provide the required behaviours. The last thing that we want is to lose these benefits through an inadequate design mapping. Fortunately, there are ways that allow us to achieve a seamless mapping, while providing other desirable benefits such as scalability, flexibility and robustness in the face of future change. This chapter is about achieving such a mapping.

At this point we assume that we have the following deliverables, at least for the components of the system we want to build at the implementation level:

- A business domain model expressed as a class diagram with associations and attributes, but no behaviours yet
- A set of event models expressing the required operation of the system. These should be decomposed to the level where each activity affects the state of instances of at most one domain class
- Prototype input and output artifacts, including screens, reports and others, such as batch file layouts
It is also useful if we know the target implementation technology and technical architecture, although the logical mapping we will describe in this chapter can be performed without this.

In the sections that follow, we will show how these models are used to map responsibilities to domain classes; how the logic of our event models determines the responsibilities of business logic elements and how the interface prototypes are translated to technical components. We will do this by means of a small example, but first, let us introduce the architectural pattern on which we base our translation.

**The Model-View-Controller Pattern**

This is the underlying detailed design and programming model in many object systems, including the prototypical OO environment, Smalltalk (Goldberg, 1983; Bourne, 1992; Voss, 1991). It was developed at Xerox Palo Alto Research Centre (PARC) by Trygve Reenskaug in the late 70’s. It is easily implemented in both pure OO environments (e.g. Smalltalk, Actor, Eiffel) as well as hybrid ones (C++, Object Pascal). The functions of the components are as follows (Figure 24.1):

- **View**
  - Windows, Display Objects, Visual Components
- **Controller**
  - Business Process Logic, Rules
  - Transaction Logic, Current State
- **Model**
  - Business Domain Objects
  - Persistent Objects
  - Enduring Business (integrity constraints and rules)

24.1 - Model, View, Controller Design Pattern

- The *model* is that set of classes, methods and objects which pertain directly to the problem being addressed or simulated by the system. I.e. the objects are domain or application objects. These classes are normally created by the application developers, but could be derived from industry architectures (e.g. IBM’s Insurance Application Architecture or the OMG’s Business Object Frameworks)
- The **view** describes the interaction of the system with the user and will include definitions and behaviours for all visual objects including windows, menus, lists, entry fields, buttons, scroll bars etc. Normally these classes are provided by the environment, and instantiated by the developer. If special behaviour is required that the standard classes do not provide, developers might specialise the provided classes. If this is done, it should be performed at a corporate level to preserve a common look and feel to the interface across applications. We extend the view to deal with all external interaction of the system, including batch commands from the environment and interfacing with other devices such as scanners and printers.

- Creates the user interface (view) layer and activates it
- Waits for events
- Processes business events
- Implements transaction control
- Coordinates effects by sending messages to the model, and results back to the view
- Can be at two levels:
  - Technical (normally provided by the framework or class library)
    - Technology specific
    - Could be GUI, text interactive or batch interface
  - Business (normally user written)
    - Not technology specific

24.2 - Controller or Business Logic Layer  Marshalls Services of Business Objects

- The **controller** is the component which links the view(s) to the model(s) in the application. It may be a single instance of a controller class (in a pure OO environment) or a main control loop and sub-components in a hybrid language. In this latter case, we may make use of what is termed an application framework (Voss, 1991). It is responsible for intercepting events which occur in the user's interaction with the view, and determining what these mean in terms of the underlying model. It is sent messages by the view, and passes them on to the model. In turn, the model objects will return messages or results, which the controller must route to the appropriate components of the view. In a complex environment, the controller may contain a variety of transaction classes,
each implementing a business process. This has the advantage of modularity and allows easy implementation of parallelism and instrumentation of process performance.

The MVC paradigm is powerful, as it allows the model to exist independently of the view. We could, for example, create several completely different user interfaces to the same underlying model: A graphical user interface (GUI), a text interface and an Internet browser interface. The same classes and metaphor can be used for all view definitions across all applications, thus achieving a consistent "look and feel" across disparate applications. This is illustrated by Windows applications from a variety of vendors who have made use of the Microsoft™ Foundation Classes for Windows objects.

We can also have different types of objects in the model layer. The simplest option, but one which is frequently not available to the designer, is to directly implement the domain classes as persistent objects in an object database. This could be done, for example, with a product like Gemstone™ tightly integrated with the Smalltalk and/or Java development language. In this case, the programming language class definition is also the Gemstone data definition language. Where we have less flexibility, are constrained by prior commitment to relational database technology, or have to integrate with legacy applications and data, we need to extend the concept of the model layer. We do this by having a gateway to a relational product, which provides an object/message interface to the controller layer and a relational (typically SQL) interface to the relational database environment. This can be achieved by home grown wrapper code or commercial products such as ObjectLens (ObjectShare, Formerly: ParcPlace Digitalk, 1995) or TopLink (Object-People, 1998).

To access the legacy applications or non-relational data, we make use of customised or purchased wrappers. These provide an object/message interface to the controller layer, and interface to the legacy applications in whatever way is necessary on the other side. This could include simulating terminal transactions, calling an API, generating an interface file, executing an application or a variety of other mechanisms. A key benefit of this approach is that the legacy components can be gradually phased out over time to be replaced by new domain objects in the object database environment without any impact on the business logic and user interface.

The amended MVC model catering for multiple user interfaces and legacy/relational integration is represented below (Figure 24.3).
24.3 - Extended MVC Architecture

Integrating the Models

To map the models to the modified MVC architecture, we proceed as follows:

- Each of the prototype user interfaces will be implemented using vendor provided classes or components as part of the view layer. This can be done iteratively during prototyping, if the tools employed support this. The final iteration represents the view layer of the application.

- Map each lowest level activity from the event models to the underlying objects as defined on the domain model, thus adding behaviour to the business domain classes. If the target environment is Smalltalk, care should be taken here to name methods so as to create a usable high level language tailored to the application knowledge. For example, we may create a method called `debitWith:` which takes a parameter for the amount on the `Customer` class. This would increase the customer’s balance by the amount passed. We could also create a method called `amount` on the
The Domain Model with the added behaviours is implemented as user defined classes and methods. Since these methods are derived from those on the event models which affect only one object type, they should have effects which are limited to only the attributes of the containing class. This ensures that domain classes have high internal cohesion and very low coupling. These are desirable attributes for modularity, reliability and maintainability. These classes constitutes the Model component.

The logic (sequence and selection) of the event model is translated into transactional methods (business processes) within the controller component. The highest level controller component identifies which of these methods to invoke for external events occurring in the environment passed on by the view layer. This connection may be established by the way the user interface is defined or informing the system in some way of which events should be trapped and which method they should trigger. This may be accomplished via an Open method in the controller, or graphically during application construction. An example of the latter is the way in which we can associate an event on a control (e.g. A mouse click on a button) to a script in languages and tools such as Visual Basic or Delphi.

Effectively, each significant user interface event (i.e. One that will cause a change to, or requires information from, domain objects) is translated to a controller layer method.

Where an object or other database management system is used, the high level scripts can easily be bracketed by begin and end transaction logic.

The Object Oriented Design is thus derived directly from the high-level models. The class definitions, including inheritance and attributes (instance variables) come from the Domain Model Class Diagram. The lowest level Event Model activities (which affect only one object type) translate to methods associated with a specific class in the model. Higher level operations, which affect multiple object types, are implemented as transaction methods in the controller. The flows from the event models typically translate to message sends between various objects. The mapping is illustrated in Figure 24.4.
24.4 - Mapping Models to the Architecture

An Example

The provided domain model (Figure 24.5) describes the classes for Person, Customer, Employee, Sale and Product in a retail undertaking. Our system is concerned with the processing of a sale, including a single product, to a customer.

Business Domain Objects (Class Diagram)

A simple class diagram showing that Employee and Customer are subclasses of Person. Customers are related to zero or more Sales. Products are related to zero or more Sales. A Sale is related to one Customer and one Product only. The latter is a simplification to keep the example small.

24.5 - Domain Objects in Class Diagram
The event model provided (Figure 24.6) shows the processing of a sales request initiated by a customer.

The Process Sale event model proceeds as follows:

We check to see if there is sufficient stock. If there is, we proceed with the sale, if not, then we backorder the goods for the customer and advise the operator.

- If there is sufficient stock, we proceed to check if we have an existing customer. If we do, we can go on to conclude the sale by debiting the customer, issuing the stock and recording the sale (creating a new Sale instance). Finally, we advise the customer.

- If we do not have a valid customer, we trigger the taking on of customer details (another event model) and expect this process (the sale) to be triggered again when the customer take on has been completed.

The activities have been decomposed to a level where they affect only one class each. We can map these operations to the classes on the Domain Model. The result of this is shown overleaf as an expanded UML class diagram (Figure 24.7). You will recall that this has the class name in the upper box, the attributes in the centre box and the method patterns (services
or interfaces) listed in the lower box. For the variable and method names we have followed Smalltalk conventions:

- Lowercase first letters indicate local variables, and user method names
- Uppercase first letters are reserved for global variables and class names
- A colon indicates that a parameter is expected to follow. Its absence indicates that no parameter is expected

The translation from the event model has been performed as follows:

- the Customer class must have the intelligence to debit itself with an amount. This is necessary to support Sales, and other transactions
- since we want to be able to quickly see the list of purchases made by a Customer, the Customer has a collection of Sale references, where each new purchase will be added by the method recordPurchase
- Products must be able to tell us how many are in stock (quantityOnHand); decrease the available stock when issued (issue); return the price for calculation of debits (salePrice); and record the list of Customers waiting for stock to come in (orderFor)
- Sales will be recorded anew and must be able to update the various data items (customer:, product:, date:, and amount: )
- the controller object will receive from the view values for the items: currentCustomer, currentProduct and saleQuantity. The high-level transaction logic has been implemented as a single method, sale, since it is so simple. We could also have broken this into multiple levels if things became more complex. We would do this where there were multi-level event models

What we have shown here for the sale method is actual Smalltalk code, not pseudocode. Notice how designing classes and method patterns carefully has resulted in a near-English language in which to solve our problem. Consider the line:

```plaintext
    currentCustomer debitWith: ((currentProduct salePrice) * saleQuantity)
```

This is certainly something a user can relate to.
24.7 - Domain Class Responsibilities
24.8 - Controller Logic

**Reusability**

The approach discussed can achieve a high level of reusability and flexibility, since:

- vendor supplied view classes are used for the interface
- application classes created within the model reflect their natural characteristics in the real world and can thus be reused or extended for use in future applications
- transactional operations on the underlying model can be easily implemented at a very high level in the controller. All that this requires is a very short script in the form of a customized syntax reflecting the underlying object behaviours
- The application is fairly independent of the user interface and can be retargeted for new deployment environments, e.g. Internet or mobile technology without changing business logic or domain classes

**Maintenance and Flexibility**

Maintenance or enhancement of the functionality provided by the application should be easily accomplished.

---

**Controller**

<table>
<thead>
<tr>
<th>currentCustomer</th>
<th>aCustomer</th>
</tr>
</thead>
<tbody>
<tr>
<td>currentProduct</td>
<td>aProduct</td>
</tr>
<tr>
<td>saleQuantity</td>
<td>anInteger</td>
</tr>
</tbody>
</table>

```
sale
"check product quantity on hand, if sufficient, debit customer, issue stock and record sale
if not, order more stock and advise customer"
sale
System: beginTransaction.
(currentProduct quantityOnHand < saleQuantity)
  ifTrue: [currentProduct orderFor: currentCustomer.
    SalesView alert: "Out of Stock, backordered"]
  ifFalse: [(currentCustomer isMemberOf: Customer)
    ifTrue: [currentCustomer debitWith: ((currentProduct salePrice) * saleQuantity).
      currentProduct issue: saleQuantity.
      sale:=Sale new.
      sale customer: currentCustomer; product: currentProduct; date: Date today;
      amount: (saleQuantity * (currentProduct salePrice))
      currentCustomer recordPurchase: sale.]
    ifFalse: [Customer getNew]].
System: endTransaction.
```
• If there is no fundamentally new data to manage or behaviour of application objects in the model, changes can be accomplished at the high level in the transaction scripts

• If new data or behaviour is required, it can be added to the appropriate classes in the model, where it becomes available for reuse

• If errors are found, they can be traced very quickly to a method in a model object, or a transaction script in the controller

Summary

The proposed approach can ease the adaptation of commercial developers to the object oriented approach. It provides an extension to the familiar paradigm of entity modeling as a starting point. Data flow diagramming can give way fairly easily to event modeling, which is much better suited to interactive, GUI applications. The seamless integration to the design model and the mapping of this to popular programming languages means that the extra richness afforded by object analysis techniques is not lost in the translation to code structures. Procedure normalisation provides benefits of reduced redundancy of code, increased reuse and easier maintenance. The implementation of high level transaction scripts will facilitate proper transaction control in systems using object database technology. The approach creates a very responsive environment which can track new business requirements and business reengineering initiatives in real time. Applications can be easily translated to multi-tier graphical interface client-server environments deployed on GUI workstations or via Inter/intranet browser technology. Relational and legacy application components are easily accommodated, and migration to pure object technology made easier.
References and Further Reading


Date, Chris, 1990, **An Introduction to Database Systems**, vol 1, 5th Ed., Addison-Wesley, Reading, MA

Goldberg, Adele, 1983, **Smalltalk-80: The Interactive Programming Environment**. Addison-Wesley, Menlo Pk, CA.


Khoshafian, Setrag, 1990, **Object Orientation: Concepts, Languages, Databases, User Interfaces**, John Wiley & Sons, NY


Martin, James, 1987, **Recommended Diagramming Standards for Analysts and Programmers**, Prentice Hall, Englewood Cliffs, NJ


ParcPlace Digitalk, 1995, **VisualWorks User Guide**, ParcPlace Digitalk, Sunnyvale, CA

Index

A
Abstract Data Types, 245
Abstraction Level, 206
Activity diagrams, 127
Agents, 128
Aggregation, 102
Anti-Patterns, 336
Application Programming Interfaces, 286
Architecture, 12
Development Tools, 211
Architecture Frameworks
   Inspired, 123
Architectures, 277
Arrayed Collections, 248
Association, 98
Association Class, 101
Associations, 77
Asynchronous, 253

B
Bag, 247
Banking Domain Model, 109
Basic, 199
Benefits, 7
Big Picture, 41
boundaries
   managing, 12
Bounding Boxes, 134
Browser
   as client, 234
Building a Car, 317
Business Domain Modeling, 28
Business Domain Object Modeling, 93
Business Engineering, 141, 147
   principles, 154
   respect, 158
   steps, 156
Business Engineers
   caveats for, 163
Business Object Libraries, 274
Business Objects, 56, 94
Business Problem, 5
Business Process
   cuts across functional boundaries, 117
   Identifying, 119
Business Process Model, 132
   Simple Example, 132
Business Process Modeling, 29, 121, 123
Business Reengineering, 29
Business Rules, 36, 189

C
C#, 203
C++, 198
CASE, 36, 216
   advanced features, 220
   aims, 216
   Critical Success Factors, 224
   Failings, 316
   functions, 217
   realities, 223
   success and failure, 219
   types, 218
Change, 14
change management, 16
Class Diagram
   Sample, 79
Class Libraries, 72
Classes, 58
Client Server, 277
   Two and three tier, 283
CLOS, 201
COBOL, 201
Code Generation, 220
Collections, 65, 246
   Conversion, 251
Commercial Products, 321
Commercial Tools, 223
Communication, 15
Complex Objects, 63
Complexity, 11
Component Libraries, 273
Component Models, 299
Components, 14, 40, 69
Composite Objects, 64
Composition, 78, 102
Computerised Components selecting, 178
Concepts
  Object Orientation, 53
Constraints, 152
Containers, 301
Context, 45
context diagram, 26, 49
Control, 56
Control/Widget Libraries, 272
Controller or Business Logic Layer, 341
CORBA, 291
Cost to Fix Errors, 83
CREWS project, 165
Critical Success Factors, 113

D
DCE, 288
Decomposition, 11
  of event models, 180
Definition
  Patterns, 332
  Project, 26
Delivery, 2
Delphi, 200
Design Level Models, 181
design mapping, 25, 33
Development Environment Architecture
  Example, 216
Development Tools
  Architecture, 211
  Dictionaries, 250
  Discipline, 16
  Disintermediation, 153
  Distributed Object Standards, 292
  Distributed Objects, 289
  Distribution, 106, 185
  Domain Modeling
    Notation, 94
    Purpose, 93
  DTD, 269
  Dynamic link library, 252

E
Effort to Correct Errors, 82
Eiffel, 206
EJB Container Options, 305
electronic strategy, 153
Encapsulation, 57
Enterprise Java Beans, 264
Enterprise Modeling, 113
Entity beans, 264
Error Handling, 239
Errors, 240
Errors in Lifecycle, 82
Event Driven Application, 231
Event Handler, 231
Event Modeling, 31, 177
Event Processing, 233
Evolutionary
  Prototype, 172
Existing System and Data Assets, 185
Existing Systems, 39
Expectations
  Managing, 9

F
Facilitation, 38, 81
  skills, 86
Flexibility, 350
Frameworks, 14, 40, 271, 275
Frequent Delivery, 10
G
Generalisation, 95
Gesellschaft für Matematik und Datenverarbeitung, 123
GMG, 123
Graphical User Interfaces, 229
Group dynamics, 90
GUIs, 229

H
Harvesting, 310
Help, 238
History, 67
Hybrid Languages, 198
Hyperlinks, 167
Hypermedia, 38

I
Identity, 55
IDL, 311
Image files, 258
Impedance Mismatch, 265
Improvement, 159
In memory call, 251
Increasing Fidelity, 24
Infrastructure Components, 183
Inheritance, 59
Innovation, 159
Input specifications, 181
Inputs, 129
Inspired Method, 21
Overview, 22
Integrating the Models, 343
Intelligible deliverables, 11
Interfaces, 57
Interfacing, 251
Internet as delivery platform, 294
Internet Deployment, 40
intrananet, 40
Invocation, 251
Iterative lifecycles, 17
Iterators, 65, 247

J
JAD, 81
and data management, 85
Caveat, 92
leader attributes, 86
participants, 86
Post-session Work, 91
process, 88
Results, 92
Rules, 89
Session Preparation, 88
Success Factors, 91
use in lifecycle, 85
Venue, 89
Java, 202, 295
Class Definition, 62
Java Environment, 296
Joint Application Development, 81
Joint Requirements Planning, 81
JRP, 81

K
Kaizen, 160
KISS (Keep It Simple, Stupid!), 13

L
Languages Patterns, 334
Layered Architecture, 33, 339
Legacy Components, 309
Libraries, 14, 271
LifeStreams, 237
Logical Transactions, 184

M
Maintenance, 350
Messages, 57
Messaging, 252
Methods, 57
Microsoft Alternative, 306
Middleware, 287
Miller, 11
Modal Processing, 230
Models
   Integrating, 343
Model-View-Controller Pattern, 340
Multimedia
   Notation, 168
Multimedia Specifications, 38, 165
Multiple Inheritance, 62
   Resolving, 104
Multiplicities, 99

N
Navigability, 103
New technology, 16
Nonfunctional Requirements, 2
non-functional requirements, 2
notation, 77
   Multimedia, 168
   Rules, 192

O
Object COBOL, 201
Object Data Management Group, 75
Object Database
   characteristics, 265
Object Databases, 264
Object Management Architecture, 290
Object Orientation
   Concepts, 53
   Reasons to Use, 54
Object Oriented Databases, 72
object oriented languages, 68
Object Pascal, 200
Object Technology, 34
Object to Relational Mapping, 260
Objectives

P
Packages, 14
PARC, 67
Patterns, 14, 106, 331
   Definition, 332
   Design, 1
   History, 331
   Languages, 334
   Sources, 336
   Types of, 333
Persistence, 66, 257
   options, 257
   persistence framework, 263
Persistent Conversations, 167

Project, 46
Objectives of Systems Delivery, 165
Objects, 14
OCL, 191
ODBMS
   Strengths and Weaknesses, 268
   Vendors, 268
Odell notation, 125
ODMG, 75
ODMS Standards, 266
OLE, 166
OODBMS, 74
Open CASE, 224
OpenDoc, 166
Orbletts, 298
Organizations
   fractal, 150
   in transition, 148
   leaner, 149
   virtual, 149
OS files, 259
Outcomes, 131, 179
   typical, 179
Output
   specifications, 181
Outputs, 129
Polymorphism, 63
Porter's Value Chain Model, 116
Post-Conditions, 181
Power Tools, 317
Power Types, 107
Preconditions, 181
Process Architecture, 124
Process Improvement, 147
Process Model
  Instrumenting, 135
Process Orientation, 121
Productivity and Quality, 316
Project
  Definition, 26
  Objectives, 46
  Scoping, 26
Project Definition, 46
Prototype
  Dangers of, 172
  Evolutionary, 172
  Requirements, 171
  Technical, 171
  Throw Away, 172
  Uses of, 172
Prototyping, 17, 24, 31, 171
  definition, 171
  types, 171
Q
Quality Assurance, 41
Queuing, 253
R
Reintermediation, 153
Relational Databases, 260
Relational Technology, 35
Remote procedure call, 252
Repository, 15, 36
Requirements
  Prototype, 171
Resources, 133
Reusability, 186, 349
Reuse, 316
Reverse Engineering, 221
Rich Data Types, 244
Rich media, 244
Role, 105
Rules, 133
  Definition, 189
  linking with models, 190
  Notation, 192
  Types, 190
S
Scenarios, 165
Scope management, 18
Scoping, 45
  Project, 26
Selective Invocation, 132
Server Components, 303
Set, 247
Simula, 204
Size, 18
Smalltalk, 61, 204
Software Development Process, 314
  Improvements, 315
Sorted Collections, 249
Sources
  Patterns, 336
  Specialization, 95
  specifications
    Input, 181
    Output, 181
SQL3, 262
Squeak, 204
Stakeholder Model, 115
Stakeholders, 114
State, 55
Statefull, 307
Stateless, 307
Stereotypes, 104
Stored Procedures, 263