Object-Oriented Programming for Artificial Intelligence

A Guide to Tools and System Design

Ernest R. Tello

Library of Congress Cataloging-in-Publication Data

Tello, Ernest R.
Object-oriented programming for artificial intelligence : a guide to tools and system design / Ernest R. Tello.
p. cm.
Bibliography: p.
Includes index.
ISBN 0-201-09228-X
1. Artificial intelligence. 2. Object-oriented programming (Computer science) I. Title.
Q336.T44 1989
006.3 -- dc19 89-147
CIP

Sponsoring Editor: Carole Alden
Cover design by Liz Alpert
Text design by Ken Wilson
Set in 10-point Janson by Compset, Inc.

ABCDEFGHIJ-AL-89
First printing, July 1989

Acknowledgments

First of all I would like to thank the people at Addison-Wesley who worked on this book with me, starting with Carole Alden, Aquisitions Editor; Rachel Guichard, Assistant Editor; Joanne Clapp, Developmental Editor; the Production Supervisor, Perry McIntosh; and numerous other people in various departments and roles too numerous to mention.

I would also especially like to thank Alan Kay for his visionary role in the development of object-oriented programming and the thousands of programmers worldwide who have helped develop this important new software technology.

The following people, whose names are listed alphabetically, also contributed to the writing of this book: Jim Anderson, Jerry Barber, Dan Bobrow, Alan Borning, George Bosworth, Ron Clark, Will Clinger, Brad Cox, Linda DeMichiel, Chuck Duff, Richard Gabriel, Les Gasser, Adele Goldberg, Carl Hewitt, Danny Hillis, Dan Ingalls, Chris James, Ken Kahn, Sonya Keene, Gregor Kiczales, Doug Lenat, Marvin Minsky, David Moon, Amos Oshrin, Kamran Parsaye, Patrick Perez, Alain Rappaport, Kurt Schmucker, Doug Shaker, Randy Smith, Mark Stefik, Bjarne Stroustrup, Michael Teng, Terry Winograd.

Preface

It is a very exciting time to be involved in advanced software systems for machines that are rapidly becoming such a familiar presence in businesses, schools, and homes. It's not every day that irreversible advances are made in a field like computing. New languages are added to the list of those in widespread use relatively rarely. New models for programming are rarer still. But with object-oriented programming we are faced with one of those advances that changes the way computers are programmed in such a significant way that in the near future most people will not resort to previous methods, except for special reasons.

The evolution of software technology can be compared to the growth of a tree. Athough it is an organic process that takes place continually, after all is said and done, there are definite "rings" or layers that can be seen that represent both its current structure and the distinct phases through which it has passed. The place where the analogy breaks down is that in software systems, in some cases, the layers continue to be functional, whereas in the case of trees, it is apparently only the uppermost layer at the surface that is alive at any given time.

The various layers of today's computers' software were built up one by one over the last forty years or so. On the very first layer we have the ubiquitous ones and zeros, the binary code on which all else is built. Then, on the next layer, we have the bit patterns that are encoded as primary instructions for the current generation of processing hardware. As everyone in computing knows, at one time these were the only layers that existed. Programmers had the thankless task of writing codes composed of just these ones and zeros as elements. But, before long, a new layer had to be developed—that of assemblies of binary instructions according to mnemonic symbols. Never again will programmers be required to program in the impoverished vocabulary of binary digits. As software technology advanced, new layers were added to the original ones when it became clear that the next tier of the programmer's interface to machines had been discovered.

Alan Kay, one of the developers of the object-oriented paradigm, has said that "hardware is crystallized software." We have experienced the truth of this in a number of different ways. If it is true as applied to the object-oriented paradigm itself, we can see that it is here that our metaphor based on the rings of a tree trunk breaks down. Not only is the object-oriented paradigm the latest external software layer surrounding a hardware core. This paradigm is pregnant with implications for the architecture of the core itself. The message-passing metaphor of object-oriented languages is of the utmost relevance because it mirrors one of the most influential parallel hardware designs. But here we

arrive the other part of our theme in this book because the human brain is clearly a parallel processing system on a vast scale.

Artificial intelligence is involved with attempting to fulfill the dream probably first envisioned by a marketing executive who described the computer as an "electronic brain." A discernible degree of success in this endeavor has resulted in increasing commercial use of AI techniques and tools, particularly in the field of expert systems. However, taking the field of AI beyond this rather modest level of accomplishment and on a road that can bring it closer to its ambitious goal has proven far more difficult than many had suspected. This has led to larger and larger AI programs, and the trend continues. The object-oriented paradigm fits in well with this trend because all of its practitioners agree that this programming paradigm is particularly well suited to the development of very large applications.

This book has been designed to serve readers in both academic and commercial settings. It is especially well suited for use with one of the various object-oriented programming tools that are currently available for low-cost computers. The emphasis is on choosing the right tool for the right job and on cultivating the new skill called object-oriented design and applying it to some of the current problems of AI, in both of its main spheres of activity, that is, commercial AI and AI research.

Chapter 1 is intended as a simple introduction to object-oriented programming concepts for those who are entirely new to them. Chapter 2 briefly introduces some of the main problems addressed in the field of artificial intelligence and gives reasons for regarding the object-oriented paradigm as particularly suitable for well-crafted attempts at their solution. This is hardly a new discovery, but rather the stating of a conclusion categorically that many people have already arrived at through practical experience.

Chapter 3 surveys of some of the most interesting AI applications to date that have implications for how object-oriented systems may most effectively be used in AI. Chapter 4 takes the reader on a tour of all the major object-oriented programming languages and tools that are commercially available. Chapter 5 focuses on object-oriented extensions to the LISP language. Here, the high point is a thorough overview of CLOS, the new object-oriented extension to CommonLISP that is being proposed as a temporary pragmatic standard until a more permanent one can be devised with the virtue of some invaluable hindsight.

Chapter 6 moves to a vital topic in commercial AI, that of evaluating object-oriented expert systems tools. Chapter 7 demonstrates some state-of-the-art techniques in expert systems technology, using a tool called Nexpert Object, and demonstrates using object-oriented programming to build inference engines. Chapter 8 is devoted to exploring the importance of object-oriented techniques for representing and modeling the world. The topics of discrete and continuous simulation, rule-based simulation, content-addressable memory, and neural modeling are introduced. Finally, Chapter 9 explores the topic of concurrent object systems. It is here that the message-passing paradigm is truly fulfilled.

One of the more important ideas presented in this book is that of active data. At this point in time, this is as much a prescriptive concept as it is a descriptive one. That is, it

is something that is just beginning to appear in practice, but it is a most important goal for those involved in practical applications development today as well as those who are hard at work researching those techniques that will become the staples of tomorrow's programming experts.

Although the era of active data machines is just beginning, it is still important to ask what it will mean for the future of computing and AI. So many of the algorithms that have provided the backbone of our current era of computing assume a duality of active processes and passive data. We need to ask how many of these solutions can still be applied in the new architectures that are now emerging. To what extent can the active data model make a difference on current hardware still based on the Von Neumann model? These are some of the important questions that will be raised in this book, though they are of such magnitude that the full answer awaits us only in the decisive years of the next decade.

Table of Contents

Acknowledgments xi

Preface xiii

Chapter 1 Object-Oriented Programming Defined

1.1 Programming Paradigms 3 / 1.2 The Object-Oriented Paradigm 3 / 1.3 Object-Oriented Programming Metaphors 5 / 1.4 Active Data 6 / 1.5 Message Passing 6 / 1.6 Classes, Instantiation, and Inheritance 7 / 1.7 Types of Object-Oriented Systems 8 / 1.8 How Object-Oriented Systems Work 9 / 1.9 Using an Object-Oriented System 10 / 1.10 Why Object-Oriented Programming? 10 / Conclusion 11 / True or False? 13

Chapter 2 Advantages of Object-Oriented Programming for Artificial Intelligence

2.1 Artificial Intelligence: An Overview 15 / 2.2 Fields of AI Research 15 / 2.3 AI Techniques 17 / 2.4 Inference Engines 17 / 2.5 AI Paradigms 19 / 2.6 The State of the Art 19 / 2.7 Sequential Closure in Procedural Programming 21 / 2.8 Conventional Libraries Versus Classes and Methods 22 / 2.9 Extensible Templates 22 / 2.10 Key Features of Object-Oriented Systems 24 / 2.11 Explicit Versus Implicit Knowledge 25 / 2.12 Frame-Based Systems 26 / 2.13 Frames and Recognition 26 / 2.14 Frames and Scripts 27 / 2.15 Linking Rules and Objects 28 / 2.16 Automatic Investigation 29 / 2.17 Object-Oriented Approaches to Learning Systems 29 / 2.18 Automatic Programming 29 / 2.19 The Concept of the BIOLOG Language 30 / 2.20 Generic Rules 31 / Conclusion 32 / True or False? 33

Chapter 3 Examples of Object-Oriented Applications in AI

3.1 FORMES 35 / 3.2 Starplan II 37 / 3.3 PRIDE 38 / 3.4 EURISKO 39 / 3.5 CYC 45 / 3.6 BACAS 45 / 3.7 The Future Construct in Butterfly LISP 46 / 3.8 Intelligent User Interfaces 47 / 3.9 ROOMS: User-Defined Office Suites 49 / 3.10 Visual and Iconic Programming 52 / 3.11 Advanced Interfaces for Programming 52 / 3.12 Iconic Programming 53 / 3.13 From Icons to Gesture Recognition 54 / 3.14 Visual Languages and Visual Grammars 54 / 3.15 Programming by Rehearsal 55 / 3.16 The Virtual Workstation 56 / 3.17 A Look Ahead: Custom 3-D Virtual Work Environments 56 / Conclusion 57 / True or False? 58

Chapter 4 Object-Oriented Programming Tools

4.1 Smalltalk History 61 / 4.2 Smalltalk-80 63 / 4.3 Metaobject Protocol 64 / 4.4 The System Browser 66 / 4.5 Smalltalk Classes 68 / 4.6 Sample Programs 70 / 4.7 Smalltalk/V/286 71 / 4.8 Smalltalk/V/Macintosh 78 / 4.9 C++ 83 / 4.10 Friends 84 / 4.11 Operator Overloading 84 / 4.12 Constructors 85 / 4.13 The PFORCE++ Library 88 / 4.14 Objective-C 90 / 4.15 The VICI Interpreter 97 / 4.16 Ctalk 102 / 4.17 The ACTOR Language 108 / 4.18 Expert System Demo 114 / 4.19 Sample Code 116 / Conclusion 120 / True or False? 120

Chapter 5 Object-Oriented LISP

5.1 Background 123 / 5.2 The Scheme Dialect 126 / 5.3 Scheme Control Structures 127 / 5.4 SCOOPS 128 / 5.5 Composite Objects: An Example of Object-Oriented Programming in SCOOPS 129 / 5.6 Future LISP 132 / 5.7 ObjectLISP 142 / 5.8 Old and New Flavors 144 / 5.9 Method Combination 146 / 5.10 Portable CommonLOOPS 147 / 5.11 Multiple Inheritance 148 / 5.12 Future Directions in Object-Oriented LISP 150 / 5.13 The CommonLISP Object System 150 / 5.14 Generic Functions 152 / 5.15 Class Redefinition 155 / 5.16 Encapsulation by Convention in CLOS 157 / 5.17 CLOS Functions, Macros, and Special Forms 157 / Conclusion 160 / True or False? 161

Chapter 6 Object-Oriented Expert System Tools

6.1 GoldWorks II 163 / 6.2 Attempts 170 / 6.3 Sponsors and Agendas 170 / 6.4 ART 176 / 6.5 Viewpoints in ART 177 / 6.6 Hypothetical Worlds 180 / 6.7 LOOPS 183 / 6.8 The Lattice Browser 185 / 6.9 Active Values 190 / 6.10 Virtual Copies 194 / 6.11 LOOPS Applications 194 / 6.12 KEE 196 / 6.13 HUMBLE: An Expert System Shell in Smalltalk-80 215 / 6.14 Note on Blackboards 218 / Conclusion 218 / True or False? 219

Chapter 7 Object-Oriented Expert System Applications

7.1 Augmented Rule Format 222 / 7.2 Weak Links and Knowledge Islands 223 / 7.3 SPACEMED 225 / 7.4 An Object-Oriented Inference Engine 235 / 7.5 Inference Engine Design 239 / Conclusion 241 / Projects 259 / True or False? 259

Chapter 8 Conceptual Models and Hierarchies

8.1 Classification and Reclassification 261 / 8.2 Full Access to Objects 263 / 8.3 Object-Oriented Simulation 264 / 8.4 Combining AI and Discrete Simulation 266 / 8.5 ROSS 266 / 8.6 Continuous Simulation 268 / 8.7 Rule-

Based Simulation 268 / 8.8 DOORS 276 / 8.9 A Neural Modeling Example 278 / Conclusion 281 / True or False? 282

Chapter 9 Concurrent Object-Oriented Systems

9.1 Actors 285 / 9.2 Concurrent Message-Passing Modes 290 / 9.3 ABCL 291 / 9.4 Project Teams and Project Leaders 292 / 9.5 POOL/T 293 / 9.6 MACE 294 / 9.7 Object-Oriented PROLOG 295 / 9.8 PROLOG/V 296 / 9.9 Using PROLOG/V 297 / 9.10 Concurrent PROLOG 302 / 9.11 SPOOL 304 / 9.12 Vulcan 305 / 9.13 Message Management: The Key to Concurrent Message Passing 307 / 9.14 Cooperating Systems 308 / 9.15 Hybrid AI Architectures for Real-Time Processing 312 / 9.16 Progressive Deepening and Progressive Reasoning 313 / 9.17 Censored Rules and Variable Precision Logic 314 / 9.18 DIPOLE 315 / 9.19 The Resonance Machine 316 / Conclusion 321 / True or False? 322 /

Appendix 325

Bibliography 327

Index 331

Object-Oriented Programming for Artificial Intelligence

A Guide to Tools and System Design

Ernest R. Tello

Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn
Sydney • Singapore • Tokyo • Madrid • San Juan

Chapter 1

Object-Oriented Programming Defined

Although this book is about artificial intelligence from the point of view of object-oriented programming, it is important to look at object-oriented programming in its own right first, before considering its relation to AI. The result should be a more balanced perspective—one that will provide a somewhat more objective background for what follows. This chapter assumes that the reader has had no previous exposure to object-oriented systems, but does have some general programming background.

Object-oriented programming is the most recent stage in the development of programming. It represents a significant technological advance, just as the structured programming of Algol and Pascal was an improvement over the earlier block-structured programming of FORTRAN. As with most technological advances, it is not difficult to see that there is something very "correct" about object-oriented programming, which is why it is here to stay.

1.1 Programming Paradigms

A *programming paradigm* is a basic model for approaching the way programs are written. To some, it may come as a surprise that different programming paradigms even exist. Most programmers are familiar with just one paradigm, that of procedural programming. But there are now many others. The object-oriented paradigm, which is what we are concerned with here, is one, but there are also the rule-based programming, logic programming, parallel programming, visual programming, constraint-based, data flow, and rehearsal paradigms.

Why are there so many paradigms? The answer is partly because programming computers is still a relatively new discipline and partly because people still want to be able to do many different things with computers. Also, the current type of digital computer is just one type of machine architecture among many. Developers are currently experimenting with alternative machine architectures that are considered to be improvements in one respect or another. Many of them either require or expect an alternative programming paradigm.

1.2 The Object-Oriented Paradigm

If you ask someone what object-oriented programming is, you should be prepared for somewhat different answers, depending on the person you ask. People are apt to use the latest "buzz words" to describe their projects, but they often do this with surprisingly little justification. When it is a question of object-oriented programming methodology rather than products and tools, the emphasis tends to be on what the programmer finds most interesting or useful about an object-oriented system. Object-oriented programming, then, can be different things to different people.

Before proceeding any further, we need to eliminate any confusion in terminology, for there is at least one other use of the term "object-oriented" besides the one that will be used in this book. Certain graphics application programs are referred to as object-oriented

programs. Generally speaking, this is a way of contrasting such applications with pixel-based or screen-oriented graphics programs. The main idea is that the object-oriented type of graphics program stores images in such a way that the objects that make up a complete picture retain their individual identity. This means that we can always perform various changes on an object-by-object basis with such a program. More often, computer graphics images are stored on a pixel-by-pixel basis, simply as screen information.

As it is used in this book, the term "object-oriented" has an entirely different meaning. In the sense that we are using it, "object-oriented" does not refer merely to specific graphics application programs, but rather to a style of programming or programming tool. It is true that the type of graphics program in question is particularly easy to write with object-oriented programming techniques. However, it is also true that such applications can be written without the type of programming we will be discussing, and also that pixel- or screen-oriented graphics programs can be written with object-oriented tools and methods. In this book, the only sense in which we will refer to application programs as being "object-oriented" is when they have been written in the object-oriented manner.

The preliminary definition of object-oriented programming that we adopt here should allow the reader to follow the current introduction, but it will be expanded upon at a later point. The initial definition we will adopt for now is as follows:

> Object-oriented programming is a type of programming that provides a way of modularizing programs by establishing partitioned memory areas for both data and procedures that can be used as templates for spawning copies of such modules on demand.

According to this definition, an object is considered to be a partitioned area of computer memory. Many people take the phrase "object-oriented" literally to mean that this type of programming is concerned with objects in the world, as opposed to being the state of the computer being programmed. There is some truth to this, but it is not the essence of object-oriented programming.

Essentially, object-oriented programming's method of partitioning a computer's memory allows each module or object to function relatively independently of one another. There are various consequences of this, most of which are advantageous to the programmer. However, most programmers tend to emphasize primarily those advantages that they find most interesting, or with which they are most impressed.

What exactly do we mean by partitioning the memory of the computer? This means to provide a system of dividing the computer's memory into areas that have a certain functional independence. The memory partitions are independent in the sense that they may be used in a variety of different programs without modification and with complete assurance that none of the memory locations will be overwritten or in any unanticipated way function differently because of their presence in a different environment.

The definition states that the partitioned areas of memory are not just those that store data, but also those that store code in the form of executable procedures. This is essential

to keeping the object protected. Any procedure that was allowed access to the memory locations of an object could alter or even corrupt the data stored in the object. This would clearly interfere with its functionality. For this reason, the active processes of an object-oriented system are defined as local functions or procedures that alone may have access to the object's data. In this way, the object protects itself from having its data and functionality corrupted by unknown events external to it. Thus, a functioning element of a program that has been written correctly and completely debugged cannot be changed by subsequent additions or modifications to any of the programs in which it is used.

Finally, the definition states that the partitioned memory structure can be used as a template for spawning further copies of the object. For example, once a window object has been defined, as many of these objects can be created as memory allows, without writing additional code to ensure against interference between different instances. In stating that additional copies are made, we are referring only to the behavior of the objects. Everything behaves as though complete multiple copies are in operation. Note, however, that we are not suggesting how the objects should actually be implemented.

1.3 Object-Oriented Programming Metaphors

Each programming paradigm has its own metaphors that contribute to the ways in which programmers think about program design. Computer science itself is full of metaphors that have become the jargon of the field. Two excellent examples of this are the terms "memory" and "windows." If you really think about it, these metaphors are actually rather remote. Memory and windows in computing have little resemblance to those we encounter in the everyday world. Nevertheless the terms have stuck and we use them indiscriminately, apparently without any great difficulty.

The metaphors used in describing programming models present a somewhat different situation in that to be of service, they must reflect a genuine model whose features actually have practical implications. The more accurate these metaphorical models, therefore, the better. What are some of the metaphors of object-oriented programming? As soon as we ask the question, we can see that many of the basic concepts such as message passing and inheritance have a metaphorical significance. However, at least in the case of these two metaphors, the analogy can be taken quite far with good results.

Some of the metaphors adopted to explain object-oriented systems, however, can be more remote and, therefore, less useful. In his book *Object-Oriented Programming*, Brad Cox uses the metaphors of software ICs and factories to try to explain the concepts of object-oriented systems. As we will argue later in more detail, although these metaphors have pedagogical value, they are not adequate to serve as more lasting metaphorical models of the components of object-oriented systems. Object-oriented systems have features that actually make them significantly more powerful technologically than ICs or factories. This is largely due to their extreme modularity. We cannot use off-the-shelf ICs to build other ICs, and the end products of factories generally do not resemble the factories that make them. However, it is an everyday matter in object-oriented

programming to use existing objects to make still newer ones that inherit all the properties of their ancestors.

1.4 Active Data

One very useful way to understand objects is to think of them as an attempt at providing a form of *active data*. It would be true to say that an object represents active data if the paradigm for performing operations in an object-oriented system were to send messages to objects telling them what we wish them to do. So, for example, if we wish to print a string on the screen, we send the string a message telling it to print itself. An object that is a string is not just a passive piece of textual data. It is an active entity that knows how to perform various operations on itself correctly.

One of the most basic distinctions in programming is that between data and procedures. To some degree, this reflects the hardware difference between memory chips and processors. The basic model of a program according to this design of computers is that data is stored in memory chips and the instructions of processors are used to manipulate the data. In this model, therefore, data is essentially passive and instructions are the active elements that operate on the passive element. Object-oriented programming is part of a movement to define the programming model somewhat differently. An object is both code and data. It is a form of active data.

1.5 Message Passing

One of the main models in object-oriented programming for allowing object modules to interact is the *message-passing* model. Objects are seen as communicating by passing messages that they can either accept or reject. Generally an object accepts messages it recognizes and rejects those it does not. In this way, the types of interactions between objects are carefully controlled. Because all interactions follow this protocol, code external to an object has no opportunity to interfere in any way with the object's functioning in unpredictable and undesirable ways.

At the very minimum, sending a message involves specifying the name of an object and the name of the message to be sent to that object. Often, arguments will also have to be specified. The arguments may be the names of variables known only to the type of object that recognizes the message, or they may be global variables known to all objects. They may not be variables known privately by another unrelated type of object.

As mentioned earlier, message passing produces the appearance of active data. You can send a message to an object to write itself to disk, display itself on the monitor, delete itself, and so on. Is this only an appearance, or is there more substance to the connection between active data and message passing? This is a difficult question that requires more background than we have provided so far to answer adequately. Let us just say for the moment that message passing has the effect of making objects behave as active data from

outside them, but not from inside. From within an object we have the same dichotomy between passive data and active procedures that is found in conventional programming.

What differences does it make whether or not a system uses message passing? Again, a final answer to this question is not possible here. However, the general answer is that message passing allows for advanced programming techniques that can exploit the partitioned functionality of objects that respond to messages. Although there are ways to achieve many of the same results without message passing, it is generally more difficult to do and requires greater knowledge and skill on the part of the programmer.

On the whole, objects in message-passing systems have a very special kind of autonomy that is unique in programming. In a certain sense, each object behaves like a little specialized computer communicating with other specialized computers. Although this is, of course, another metaphor, it is a fruitful metaphor that is worth pursuing. As we will see later, using the message-passing model can suggest many interesting and powerful algorithms that might not otherwise have occurred to programmers and software designers.

1.6 Classes, Instantiation, and Inheritance

One of the most basic concepts in object-oriented programming is that of *class*. As it is understood in this context, a class is a template for creating actual functioning objects of a given type. In many object-oriented systems, a class is an object in its own right, but one with very limited capabilities. We might say that whereas a class provides the blueprint or genetic code for creating cows, it is the actual cow instance, not the cow class, that gives milk.

We say that a class is *instantiated* when it has actually been used to stamp out one or more objects of its type. Appropriately, the objects that have been created from a given class are called *instances* of that class. What really gives practical justification to the class structure is the facility for *inheritance*, which provides the ability to create classes that will automatically model themselves on other classes. When a class B has been defined so as to model itself on class A, B is called a subclass of A. Reciprocally, we can also say that A is a superclass of B, or, in other words, B inherits all of A's behavior.

Having one class inherit the behavior of another would be pointless if that were all that was involved. The point of creating a subclass is to add some additional behavior besides whatever is inherited. Usually the subclass is outfitted with behavior that allows it to act as a more specialized version of its superclass. Some object-oriented systems also have *multiple inheritance*, which means that they may have more than one superclass. In such systems, sometimes we create a new class so that it can inherit from two or more superclasses. Other times we provide the additional behavior by hand coding. In either case, the result is a class that can stamp out objects with a somewhat different behavior.

Through this process of defining subclasses, an object-oriented system comes to have a *class hierarchy*. This is a tree or network of classes that starts with the most general as the

uppermost branches and descends to the bottom leaves which are the most specific. The power of an object-oriented environment is principally defined by the class hierarchy and its capabilities. Programmers extend an object-oriented language by expanding its class hierarchy and the vocabulary of messages that can be exchanged by various objects.

The active element of objects, the messages, are also known as *methods* and are similar to functions that are local to a particular object. The names of these methods are often called *selectors*. They have this designation because, when they are called by name, they allow the system to select which code is to be executed.

One of the great advantages of object-oriented systems is that the protocol for handling various objects stays essentially the same as the language becomes extended. Thus, large systems that are developed by more than one programmer can be managed efficiently because minimal communication and documentation of new features are required between programmers.

One practice that facilitates this is often called *overloading*. Overloading is the convention of giving the exact same selector name to methods that do the same thing for different classes of objects. Most strictly typed languages require that functions that perform the same operation but for different datatypes be given different names. So, for example, we might have two distinct functions named divideInteger and divideReal that perform the operation of division for integers and real numbers, respectively. Systems with many classes could become very difficult to use because the programmer must continually look up function names. With overloading, the method is simply called "divide" for all classes of numbers, even though the code that implements the divide method may vary from class to class. This way, the programmer or user has a uniform protocol for classes in a large range of operations, which makes the system much easier to use. The programmer does not have to look at the code written by other programmers to know how to use that code in writing new routines.

1.7 Types of Object-Oriented Systems

The various types of object-oriented systems can be distinguished in several different ways. The most basic distinction is between pure and hybrid systems.

A *pure* object-oriented system is one in which everything is an object. An example of this is Smalltalk. In this type of system, even classes are objects that are instances of a class. At first this may appear confusing, but there is nothing at all paradoxical about it. It is in fact a direct result of the desire for total consistency in the system. Just as objects are created from a class that serves as their model or template, classes themselves are objects that are created according to the blueprint laid out in a specific class. Usually this blueprint is called the *metaclass*.

The more inquisitive and logically minded reader will undoubtedly have sensed a familiar conceptual pattern here. The question may have come to mind that if classes always have to be instances of another class, how does the original class come into being. The answer is that we are not dealing just with concepts here. An object-oriented system