1

Introduction

Object-oriented modeling and design is a way of thinking about problems using models organized around real-world concepts. The fundamental construct is the object, which combines both data structure and behavior. Object-oriented models are useful for understanding problems, communicating with application experts, modeling enterprises, preparing documentation, and designing programs and databases. This book presents an object-oriented notation and process that extends from analysis through design to implementation. The same notation applies at all stages of the process as development proceeds.

1.1 What Is Object-Orientation?

Superficially the term ***object-oriented (OO)*** means that we organize software as a collection of discrete objects that incorporate both data structure and behavior. This contrasts with previous programming approaches in which data structure and behavior are only loosely connected. There is some dispute about exactly what characteristics are required by an OO approach, but they generally include four aspects: identity, classification, inheritance, and polymorphism.

Identity means that data is quantized into discrete, distinguishable entities called ***objects***. The *first paragraph in this chapter*, *my workstation*, and the *white queen in a chess game* are examples of objects. Figure 1.1 shows some additional objects. Objects can be concrete, such as a *file* in a file system, or conceptual, such as a *scheduling policy* in a multiprocessing operating system. Each object has its own inherent identity. In other words, two objects are distinct even if all their attribute values (such as name and size) are identical.

In the real world an object simply exists, but within a programming language each object has a unique handle by which it can be referenced. Languages implement the handle in various ways, such as an address, array index, or artificial number. Such object references are uniform and independent of the contents of the objects, permitting mixed collections of objects to be created, such as a file system directory that contains both files and subdirectories.

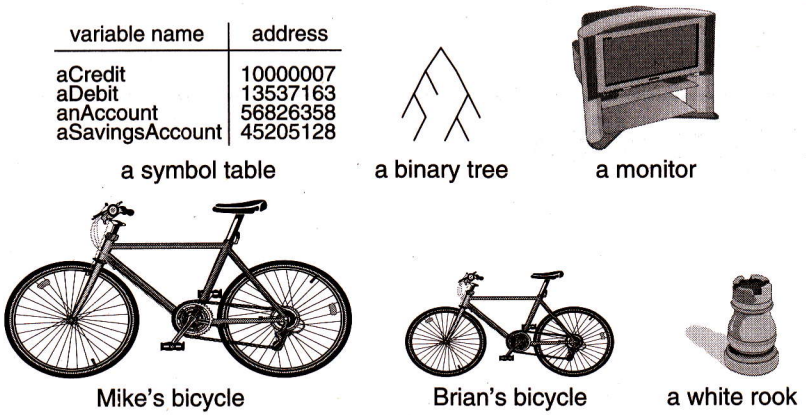

Figure 1.1 Objects. Objects lie at the heart of object-oriented technology.

Classification means that objects with the same data structure (***attributes***) and behavior (***operations***) are grouped into a class. *Paragraph*, *Monitor*, and *ChessPiece* are examples of classes. A ***class*** is an abstraction that describes properties important to an application and ignores the rest. Any choice of classes is arbitrary and depends on the application.

Each class describes a possibly infinite set of individual objects. Each object is said to be an ***instance*** of its class. An object has its own value for each attribute but shares the attribute names and operations with other instances of the class. Figure 1.2 shows two classes and some of their respective instances. An object contains an implicit reference to its own class; it "knows what kind of thing it is."

Inheritance is the sharing of attributes and operations (***features***) among classes based on a hierarchical relationship. A ***superclass*** has general information that ***subclasses*** refine and elaborate. Each subclass incorporates, or inherits, all the features of its superclass and adds its own unique features. Subclasses need not repeat the features of the superclass. For example, *ScrollingWindow* and *FixedWindow* are subclasses of *Window*. Both subclasses inherit the features of *Window*, such as a visible region on the screen. *ScrollingWindow* adds a scroll bar and an offset. The ability to factor out common features of several classes into a superclass can greatly reduce repetition within designs and programs and is one of the main advantages of OO technology.

Polymorphism means that the same operation may behave differently for different classes. The *move* operation, for example, behaves differently for a pawn than for the queen in a chess game. An ***operation*** is a procedure or transformation that an object performs or is subject to. *RightJustify*, *display*, and *move* are examples of operations. An implementation of an operation by a specific class is called a ***method***. Because an OO operator is polymorphic, it may have more than one method implementing it, each for a different class of object.

In the real world, an operation is simply an abstraction of analogous behavior across different kinds of objects. Each object "knows how" to perform its own operations. In an OO programming language, however, the language automatically selects the correct method to

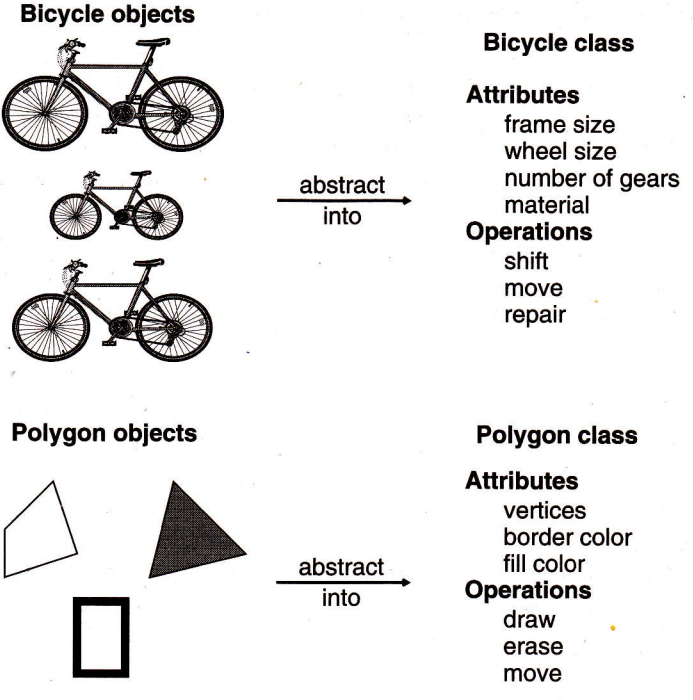

Figure 1.2 Objects and classes. Each class describes a possibly infinite set of individual objects.

implement an operation based on the name of the operation and the class of the object being operated on. The user of an operation need not be aware of how many methods exist to implement a given polymorphic operation. Developers can add new classes without changing existing code, as long as they provide methods for each applicable operation.

1.2 What Is OO Development?

This book is about OO development as a way of thinking about software based on abstractions that exist in the real world as well as in the program. In this context ***development*** refers to the software life cycle: analysis, design, and implementation. The essence of OO development is the identification and organization of application concepts, rather than their final representation in a programming language. Brooks observes that the hard part of software development is the manipulation of its *essence,* owing to the inherent complexity of the problem, rather than the *accidents* of its mapping into a particular language [Brooks-95].

This book does not explicitly address integration, maintenance, and enhancement, but a clean design in a precise notation facilitates the entire software life cycle. The OO concepts and notation used to express a design also provide useful documentation.

1.2.1 Modeling Concepts, Not Implementation

In the past, much of the OO community focused on programming languages, with the literature emphasizing implementation rather than analysis and design. OO programming languages were first useful in alleviating the inflexibility of traditional programming languages. In a sense, however, this emphasis was a step backward for software engineering—it focuses excessively on implementation mechanisms, rather than the underlying thought process that they support.

The real payoff comes from addressing front-end conceptual issues, rather than back-end implementation details. Design flaws that surface during implementation are more costly to fix than those that are found earlier. A premature focus on implementation restricts design choices and often leads to an inferior product. An OO development approach encourages software developers to work and think in terms of the application throughout the software life cycle. It is only when the inherent concepts of the application are identified, organized, and understood that the details of data structures and functions can be addressed effectively.

OO development is a conceptual process independent of a programming language until the final stages. OO development is fundamentally a way of thinking and not a programming technique. Its greatest benefits come from helping specifiers, developers, and customers express abstract concepts clearly and communicate them to each other. It can serve as a medium for specification, analysis, documentation, and interfacing, as well as for programming.

1.2.2 OO Methodology

We present a process for OO development and a graphical notation for representing OO concepts. The process consists of building a model of an application and then adding details to it during design. The same seamless notation is used from analysis to design to implementation, so that information added in one stage of development need not be lost or translated for the next stage.The methodology has the following stages.

- **System conception**. Software development begins with business analysts or users conceiving an application and formulating tentative requirements.
- **Analysis**. The analyst scrutinizes and rigorously restates the requirements from system conception by constructing models. The analyst must work with the requestor to understand the problem, because problem statements are rarely complete or correct. The analysis model is a concise, precise abstraction of *what* the desired system must do, not *how* it will be done. The analysis model should not contain implementation decisions. For example, a *Window* class in a workstation windowing system would be described in terms of its visible attributes and operations.

  The analysis model has two parts: the ***domain model***, a description of the real-world objects reflected within the system; and the ***application model***, a description of the parts of the application system itself that are visible to the user. For example, domain objects for a stockbroker application might include stock, bond, trade, and commission. Application objects might control the execution of trades and present the results. Application experts who are not programmers can understand and criticize a good model.

- **System design**. The development team devise a high-level strategy—the ***system architecture***—for solving the application problem. They also establish policies that will serve as a default for the subsequent, more detailed portions of design. The system designer must decide what performance characteristics to optimize, choose a strategy of attacking the problem, and make tentative resource allocations. For example, the system designer might decide that changes to the workstation screen must be fast and smooth, even when windows are moved or erased, and choose an appropriate communications protocol and memory buffering strategy.
- **Class design**. The class designer adds details to the analysis model in accordance with the system design strategy. The class designer elaborates both domain and application objects using the same OO concepts and notation, although they exist on different conceptual planes. The focus of class design is the data structures and algorithms needed to implement each class. For example, the class designer now determines data structures and algorithms for each of the operations of the *Window* class.
- **Implementation**. Implementers translate the classes and relationships developed during class design into a particular programming language, database, or hardware. Programming should be straightforward, because all of the hard decisions should have already been made. During implementation, it is important to follow good software engineering practice so that traceability to the design is apparent and so that the system remains flexible and extensible. For example, implementers would code the *Window* class in a programming language, using calls to the underlying graphics system on the workstation.

OO concepts apply throughout the system development life cycle, from analysis through design to implementation. You can carry the same classes from stage to stage without a change of notation, although they gain additional details in the later stages. The analysis and implementation models of *Window* are both correct, but they serve different purposes and represent a different level of abstraction. The same OO concepts of identity, classification, polymorphism, and inheritance apply throughout development.

Note that we are not suggesting a waterfall development process—first capturing requirements, then analyzing, then designing, and finally implementing. For any particular part of a system, developers must perform each stage in order, but they need not develop each part of the system in tandem. We advocate an iterative process—developing part of the system through several stages and then adding capability.

Some classes are not part of analysis but are introduced during design or implementation. For example, data structures such as *trees, hash tables,* and *linked lists* are rarely present in the real world and are not visible to users. Designers introduce them to support particular algorithms. Such data structure objects exist within a computer and are not directly observable.

We do not consider testing as a distinct step. Testing is important, but it must be part of an overall philosophy of quality control that occurs throughout the life cycle. Developers must check analysis models against reality. They must verify design models against various kinds of errors, in addition to testing implementations for correctness. Confining quality control to a separate step is more expensive and less effective.

1.2.3 Three Models

We use three kinds of models to describe a system from different viewpoints: the class model for the objects in the system and their relationships; the state model for the life history of objects; and the interaction model for the interactions among objects. Each model applies during all stages of development and acquires detail as development progresses. A complete description of a system requires models from all three viewpoints.

The ***class model*** describes the static structure of the objects in a system and their relationships. The class model defines the context for software development—the universe of discourse. The class model contains class diagrams. A ***class diagram*** is a graph whose nodes are classes and whose arcs are relationships among classes.

The ***state model*** describes the aspects of an object that change over time. The state model specifies and implements control with state diagrams. A ***state diagram*** is a graph whose nodes are states and whose arcs are transitions between states caused by events.

The ***interaction model*** describes how the objects in a system cooperate to achieve broader results. The interaction model starts with use cases that are then elaborated with sequence and activity diagrams. A ***use case*** focuses on the functionality of a system—that is, what a system does for users. A ***sequence diagram*** shows the objects that interact and the time sequence of their interactions. An ***activity diagram*** elaborates important processing steps.

The three models are separate parts of the description of a complete system but are cross-linked. The class model is most fundamental, because it is necessary to describe *what* is changing or transforming before describing *when* or *how* it changes.

1.3 OO Themes

Several themes pervade OO technology. Although these themes are not unique to OO systems, they are particularly well supported.

1.3.1 Abstraction

Abstraction lets you focus on essential aspects of an application while ignoring details. This means focusing on what an object is and does, before deciding how to implement it. Use of abstraction preserves the freedom to make decisions as long as possible by avoiding premature commitments to details. Most modern languages provide data abstraction, but inheritance and polymorphism add power. The ability to abstract is probably the most important skill required for OO development.

1.3.2 Encapsulation

Encapsulation (also ***information hiding***) separates the external aspects of an object, that are accessible to other objects, from the internal implementation details, that are hidden from other objects. Encapsulation prevents portions of a program from becoming so interdependent that a small change has massive ripple effects. You can change an object's implementa-

tion without affecting the applications that use it. You may want to change the implementation of an object to improve performance, fix a bug, consolidate code, or support porting. Encapsulation is not unique to OO languages, but the ability to combine data structure and behavior in a single entity makes encapsulation cleaner and more powerful than in prior languages, such as Fortran, Cobol, and C.

1.3.3 Combining Data and Behavior

The caller of an operation need not consider how many implementations exist. Operator polymorphism shifts the burden of deciding what implementation to use from the calling code to the class hierarchy. For example, non-OO code to display the contents of a window must distinguish the type of each figure, such as polygon, circle, or text, and call the appropriate procedure to display it. An OO program would simply invoke the *draw* operation on each figure; each object implicitly decides which procedure to use, based on its class. Maintenance is easier, because the calling code need not be modified when a new class is added. In an OO system, the data structure hierarchy matches the operation inheritance hierarchy (Figure 1.3).

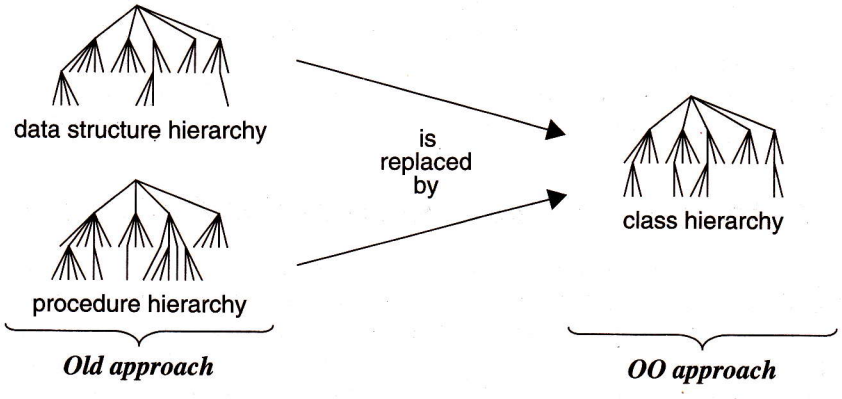

Figure 1.3 OO vs. prior approach. An OO approach has one unified hierarchy for both data and behavior.

1.3.4 Sharing

OO techniques promote sharing at different levels. Inheritance of both data structure and behavior lets subclasses share common code. This sharing via inheritance is one of the main advantages of OO languages. More important than the savings in code is the conceptual clarity from recognizing that different operations are all really the same thing. This reduces the number of distinct cases that you must understand and analyze.

OO development not only lets you share information within an application, but also offers the prospect of reusing designs and code on future projects. OO development provides the tools, such as abstraction, encapsulation, and inheritance, to build libraries of reusable

components. Unfortunately, reuse has been overemphasized as a justification for OO technology. Reuse does not just happen; developers must plan by thinking beyond the immediate application and investing extra effort in a more general design.

1.3.5 Emphasis on the Essence of an Object

OO technology stresses what an object *is,* rather than how it is *used.* The uses of an object depend on the details of the application and often change during development. As requirements evolve, the features supplied by an object are much more stable than the ways it is used, hence software systems built on object structure are more stable in the long run. OO development places a greater emphasis on data structure and a lesser emphasis on procedure structure than functional-decomposition methodologies. In this respect, OO development is similar to information modeling techniques used in database design, although OO development adds the concept of class-dependent behavior.

1.3.6 Synergy

Identity, classification, polymorphism, and inheritance characterize OO languages. Each of these concepts can be used in isolation, but together they complement each other synergistically. The benefits of an OO approach are greater than they might seem at first. The emphasis on the essential properties of an object forces the developer to think more carefully and deeply about what an object is and does. The resulting system tends to be cleaner, more general, and more robust than it would be if the emphasis were only on the use of data and operations.

1.4 Evidence for Usefulness of OO Development

Our work on OO development began with internal applications at the General Electric Research and Development Center. We used OO techniques for developing compilers, graphics, user interfaces, databases, an OO language, CAD systems, simulations, metamodels, control systems, and other applications. We used OO models to document programs that are ill-structured and difficult to understand. Our implementation targets ranged from OO languages to non-OO languages to databases. We successfully taught this approach to others and used it to communicate with application experts.

Since the mid 1990s we have expanded our practice of OO technology beyond General Electric to companies throughout the world. When we wrote the first edition of this book, object orientation and OO modeling were relatively new approaches without much large-scale experience. OO technology can no longer be considered a fad or a speculative approach. It is now part of the computer science and software engineering mainstream.

The annual OOPSLA (Object-Oriented Programming Systems, Languages, and Applications), ECOOP (European Conference on Object-Oriented Programming), and TOOLS (Technology of Object-Oriented Languages and Systems) conferences are important forums for disseminating new OO ideas and application results. The conference proceedings describe many applications that have benefited from an OO approach. Articles on OO systems have also appeared in major publications, such as *IEEE Computer* and *Communications of the ACM.*