CHAPTER 1

Object-Oriented Design

INTRODUCTION

Distributed computing, virtual reality, multimedia, and other new technologies have increased the complexity of software design. It is common for software development projects in these areas to involve different development platforms and programs consisting of more than 100,000 lines of code. With these types of systems, ad hoc design and development is impractical, if not impossible. The successful delivery of these types of systems relies on a structured and formalized analysis, design, and implementation process.

The object-oriented methodology has gained popularity because the analysis, design, and implementation process revolves around a well-defined framework with focus on minimizing risks and maximizing software *maintainability*, *reliability*, and *reusability*. An object-oriented system undergoes analysis, design, and implementation through an iterative process that is depicted in Figure 1.1. Even though the iterative process is not unique to object-oriented methodology, they work well together:

1. **Object-Oriented Analysis (OOA)**: The software requirements are analyzed and a real-world *object* model is established. The *object* model provides an abstract representation of the actual entities to be modeled by the software application. For example, the object model for a flight control system views the plane's actuators, sensors, engine, and flight control stick as *objects* that interact with each other and keep the plane in flight. In an object-oriented model, the software algorithms such as flight control algorithms become secondary issues.
2. **Object-Oriented Design (OOD)**: The *object* model is further refined. In the design phase, the details of the architecture, and the relationships and interactions between the objects, are specified. The algorithms and processes are mapped to the applicable objects.
3. **Object-Oriented Programming (OOP)**: Using a programming language that supports OOD, the completed design is implemented and tested.

This chapter describes and focuses on Object-Oriented Design (OOD). It presents a formal definition for an object, and identifies the framework for an object model. The chapter also describes how OOD techniques help localize software changes, encourage reusability, and enhance reliability.

Similar to other design methodologies, graphical presentations are important in OOD for capturing and conveying information. Since this book relies on Booch-93 notation for the presentation of the designs, this chapter provides an overview of the Booch notation.

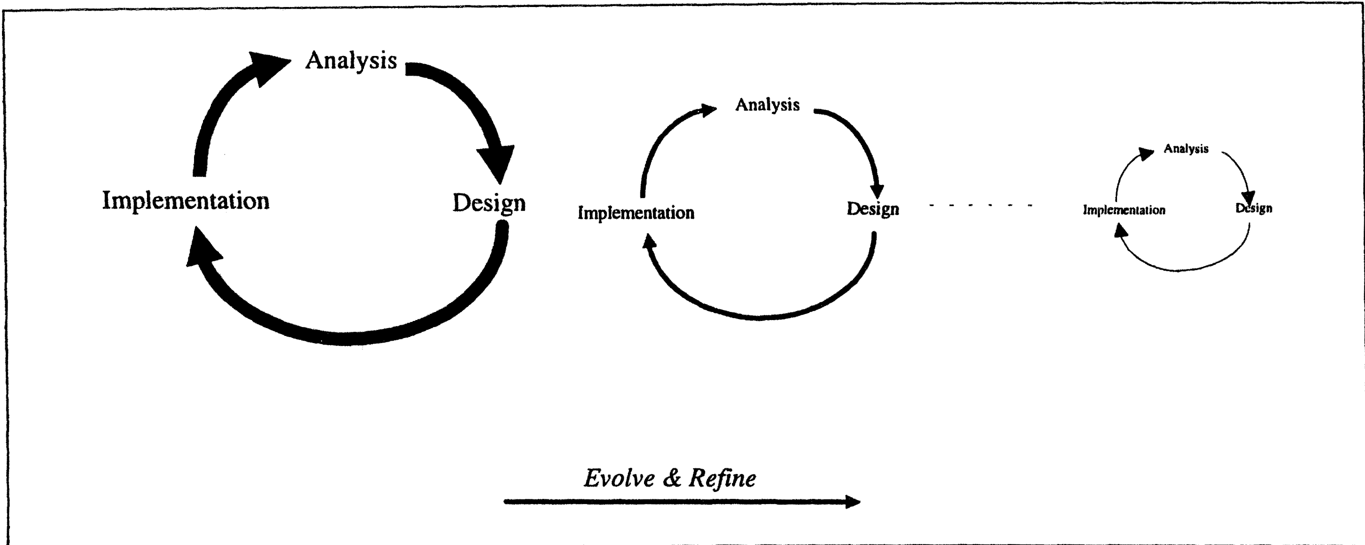

FIGURE 1.1 Iterative design process.

1.1 DEFINITION OF AN OBJECT

In object-oriented designs, software requirements are analyzed and mapped to *objects* instead of processes and functions. Software can be viewed as a collection of *objects* that collaborate with each other to perform the tasks defined by the software requirements. For example, an object-oriented design will translate and map the following system requirement to a *modem* object (Figure 1.2):

*"The system shall transmit and receive data via an internal **modem** using a minimum transmission rate of 9,600 baud"*

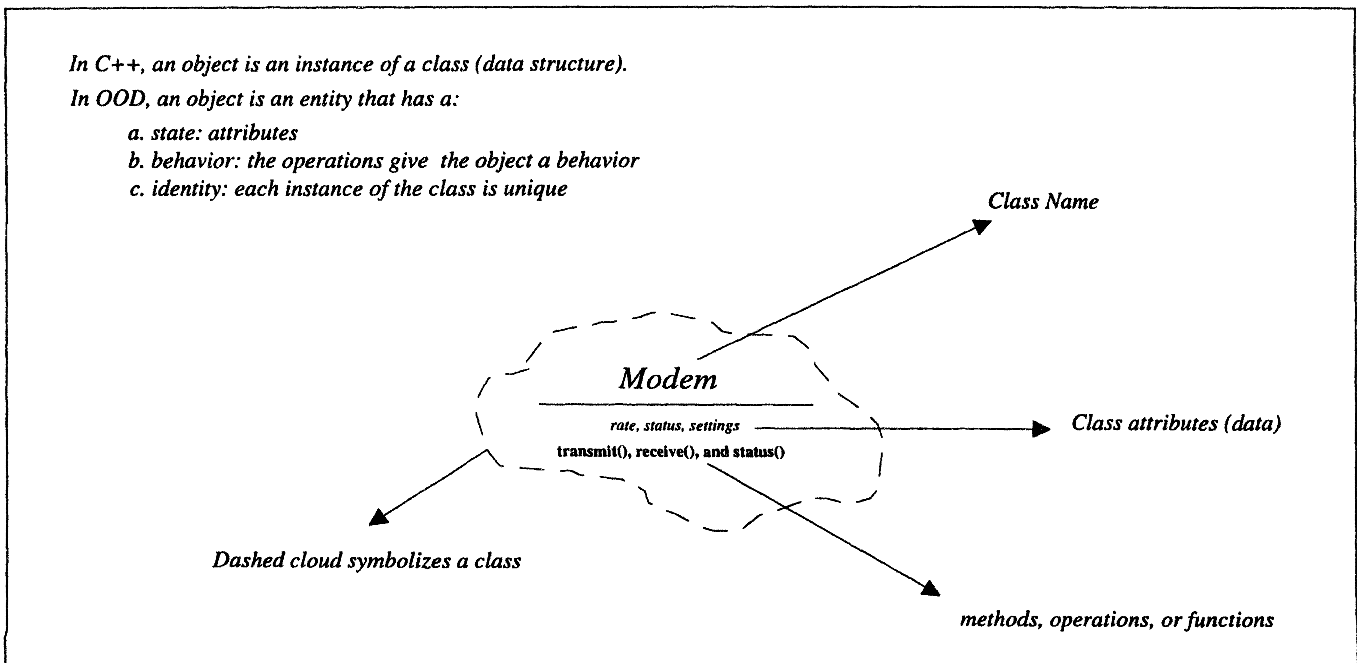

FIGURE 1.2 Modem class.

The **transmit( )** and **receive( )** functions are the *operations* (*methods*) on the *modem* object. The transmission rate is one of the object's *attributes* and the minimum transmission rate of 9,600 baud identifies the lower threshold for this attribute. Figure 1.2 depicts an abstract view of a *modem* using the Booch-93 notation.

The cloud shape represents an object and highlights the abstraction produced by the design. *Attributes* such as the **transmission rate** and **status** specify the modem's state and characteristics. On the basis of the values and settings, the attributes identify the state of the object. For example, **mode** and **status** specify the object is idle, transmitting, receiving, or in a degraded state. The **transmit( )**, **receive( )**, and **status( )** methods operate on the *attributes* and alter the state of the object. These functions cause the *modem* to transition state, such as going from idle to transmitting.

Other objects use these methods to transmit and receive data via the *modem* object. These *operations* (methods) hide and protect the attribute's name, type, and data architecture from these objects. Since these operations define the external interface of the *modem* object, they also protect the data attributes from being corrupted by other objects. The operations provide a consistent user interface even when their internals change.

An *object* provides an abstract representation of a real entity that is being modeled and mimicks the real entity's behavior. The definition of an *object* has been formalized in OOD as an "*entity that has a **state**, **behavior**, and an **identity***" [Booch 1994]:

1. **State**: The values of the *attributes* identify the state of the object. Figure 1.3 depicts the possible states of a *modem* object using Booch-93 notation. On the basis of the communication mode and status, the *modem* object may be in idle, setup, transmit, receive, error, or termination states. An event or an operation may cause an object to transition to another state. For example, an incoming phone call will change the state of *modem* from idle to receive. The transition diagram identifies the known states for an object. At any point within its life span, an object must remain in a known and stable state.
2. **Behavior**: *Operations* such as **transmit( )** give the *modem* object a **behavior**. During file transmission, the modem would need to dial the phone number, establish communication with the remote system, transmit data, verify that the transmission has been successful, and then terminate communication. Some of the operations cause changes in the **states** of the object and the object then displays a different behavior.
3. **Identity**: A system may contain several different *modem* objects. Even though each *modem* object has the same attributes and operations, each object has its own unique state. On the basis of the state, each object exhibits its own behavior. For example, one modem may be transmitting a file and the other receiving a fax.

In C++, an *object* does not exist at the source file level. An *object* comes into existence when the software is executed and the memory space for it is reserved and initialized. At the source file level, only the architecture for the object can be defined and is referred to as a *class* definition. A *class* is a type definition (for example, a **struct** definition) and it logically groups data in a set, and functions become the operations on the set (Figure 1.4). The *class* definition identifies the types, names, and layout of the data members. This information defines the size and content of memory space used by an *object*. In addition, it identifies the member functions that are allowed to operate on the data members. An *object* is an instance of a *class*. For example, a *modem* object would be an instance of the *modem* class. Any reference to an *object* is implicitly considered as an instance of a *class*.

1.2 OBJECT MODEL

An object model consists of a collection of objects that interact with each other and represent the design being modeled. For instance, a computer and its peripherals are illustrated as *classes* in Figure 1.5. Through coordinated interaction, these *classes* represent a computer.

Figure 1.5 has loosely coupled the objects and does not depict the relationship and interaction between the objects. In representing a system, an *object model* must not only identify the objects but must also identify states, relationships, and interactions between the objects. The relationships and interactions specify how the objects use each other's services and how they communicate. The object model must reflect and address both static and dynamic information. For instance, a *computer* object may use a *printer* object to print a *document*. The relationship between the *computer* and the *printer* becomes a "*use-a*" relationship. Booch-93 notation provides graphical symbols that represent common relationships such as "***has-a***," "***is-a***," and "***use-a***" between objects in a class diagram:

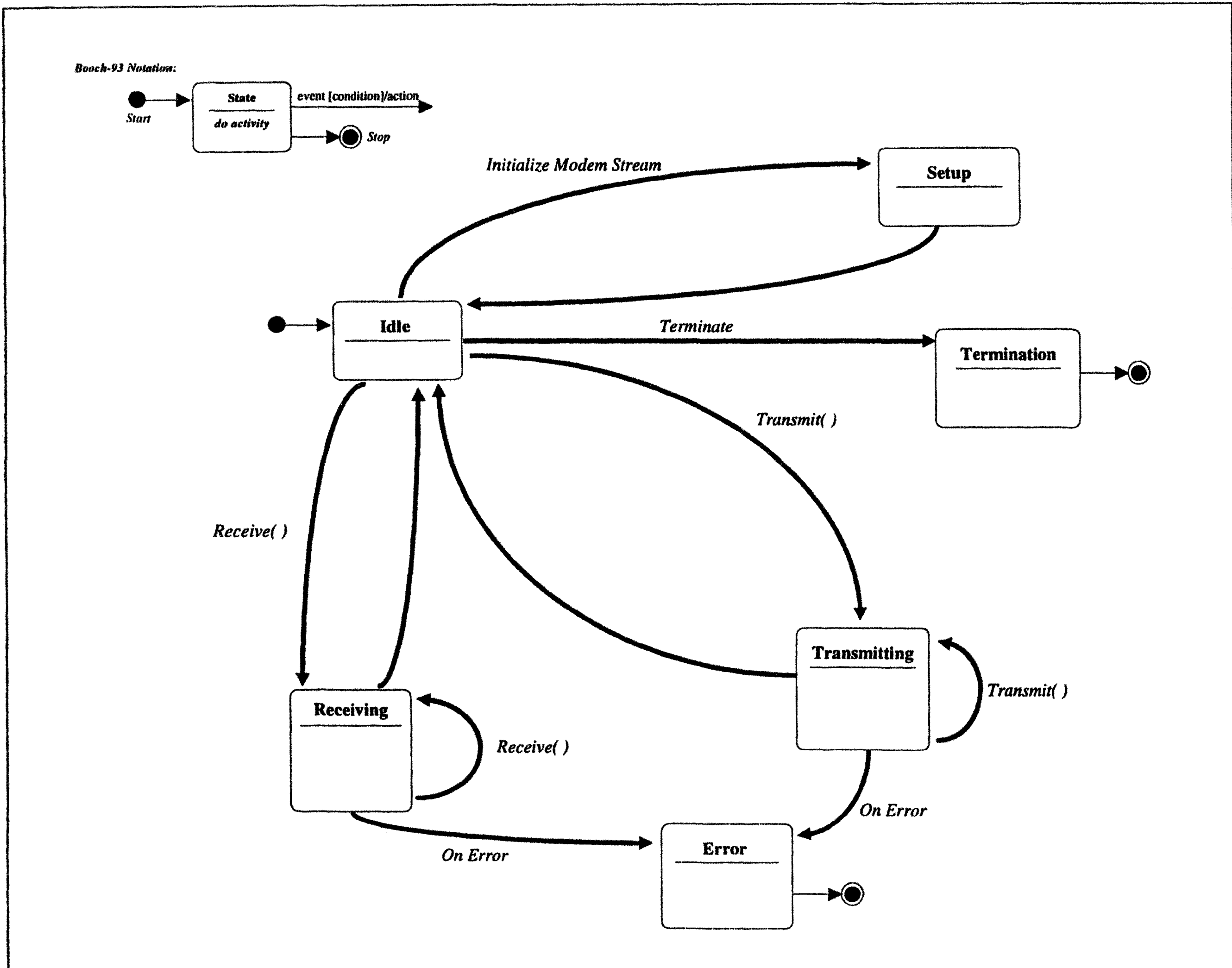

FIGURE 1.3 Modem state transition diagram.

1. **Composition**: When a class uses another class as its data member, the classes form a "***has-a***" relationship. In Figure 1.6, a *computer* contains a *floppy disk*. The *floppy disk* is perceived as a subsystem of the *computer*, and the relationship characterized as the *computer* "***has-a***" *floppy disk*.
2. **User**: When a class uses another class in the implementation of its member functions or for the function interfaces, it is ***using*** the services of the other class. For instance, the computer class may provide a **print()** function. The body of the **print()** function would "***use-a***" *printer* object for printing a document. In Figure 1.6, the *monitor* and *printer* classes are not components of a *computer*, but are used by a *computer* to display and print information, respectively.
3. **Inheritance**: A class may inherit the properties and attributes of another class, which forms a parent-child relationship. The child class becomes a subtype of the parent class. The child class ***is-a*** more specialized form of its parent class. For example, a *floppy disk* inherits the data members and member functions of the *disk* class, and adds features that are specific to a *floppy disk*. Thus, the *floppy disk* "***is-a***" subtype of a *disk* class.

Figure 1.6 illustrates a representation of the object model for a personal computer (PC). Figure 1.6 highlights the hierarchical architecture formed by the objects and their relationships, such as the *memory* and *port* classes. For example, a *computer* ***has*** multiple ports. The *serial* and *parallel* ports ***are*** different types of ports ("***is-a***" relationship). These relationships are described in greater detail in the subsequent sections.

The framework of an *object model* is partitioned into the following components and the design of an object-oriented system must reflect and encompass them:

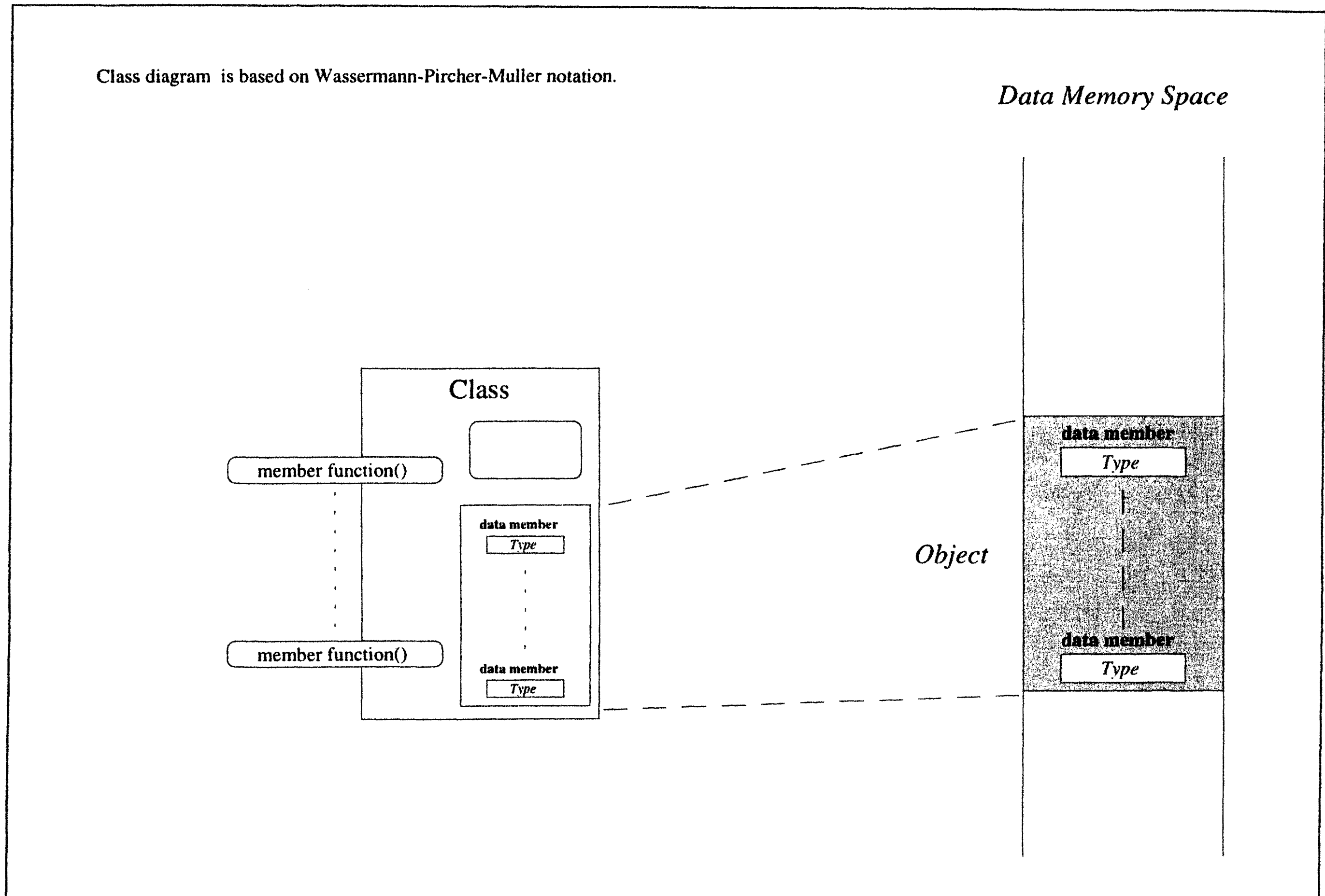

FIGURE 1.4 Class and object.

1. **Modularity:** The internal architectures and designs of the classes must be decoupled from each other. A modular architecture decouples the designs of the classes and results in formation of libraries and independent software modules.
2. **Encapsulation:** The internal architecture and data representation of a class must be kept hidden from other classes. The methods provided by a class specify the design interface for a class, and other objects operate on an instance of the class via these operations.
3. **Abstraction**: The methods must hide the processes and algorithms associated with the functions of a class. For example, the **transmit( )** function of the *modem* abstracts the data transmission process from its clients.
4. **Design Hierarchy**: The classes in the design form hierarchical relationships. As depicted in Figure 1.6, the *floppy* and *hard disk* classes inherit the properties and attributes of the *disk* class. The design of the *disk* classes is hierarchical. This is a crucial criterion in object-oriented designs and will be discussed in greater detail in subsequent sections.
5. **Typing**: Identifies a programming language type conversion and characteristics. For instance, a strong typing means that conversions between different data types in expressions require explicit conversions.
6. **Concurrency**: In *distributed architectures*, objects can be distributed across multiple processors. Since they are no longer controlled by a single process (program), their operations, states, and interactions must be controlled and coordinated in order to avoid resource contention and processing deadlocks.
7. **Persistence**: In *distributed architectures* or database applications, as a process terminates, the state of critical objects associated with the terminated process must be stored in a database or in a file. This provides *persistence*, and objects from other processes can reactivate and interact with each other without loss of information.

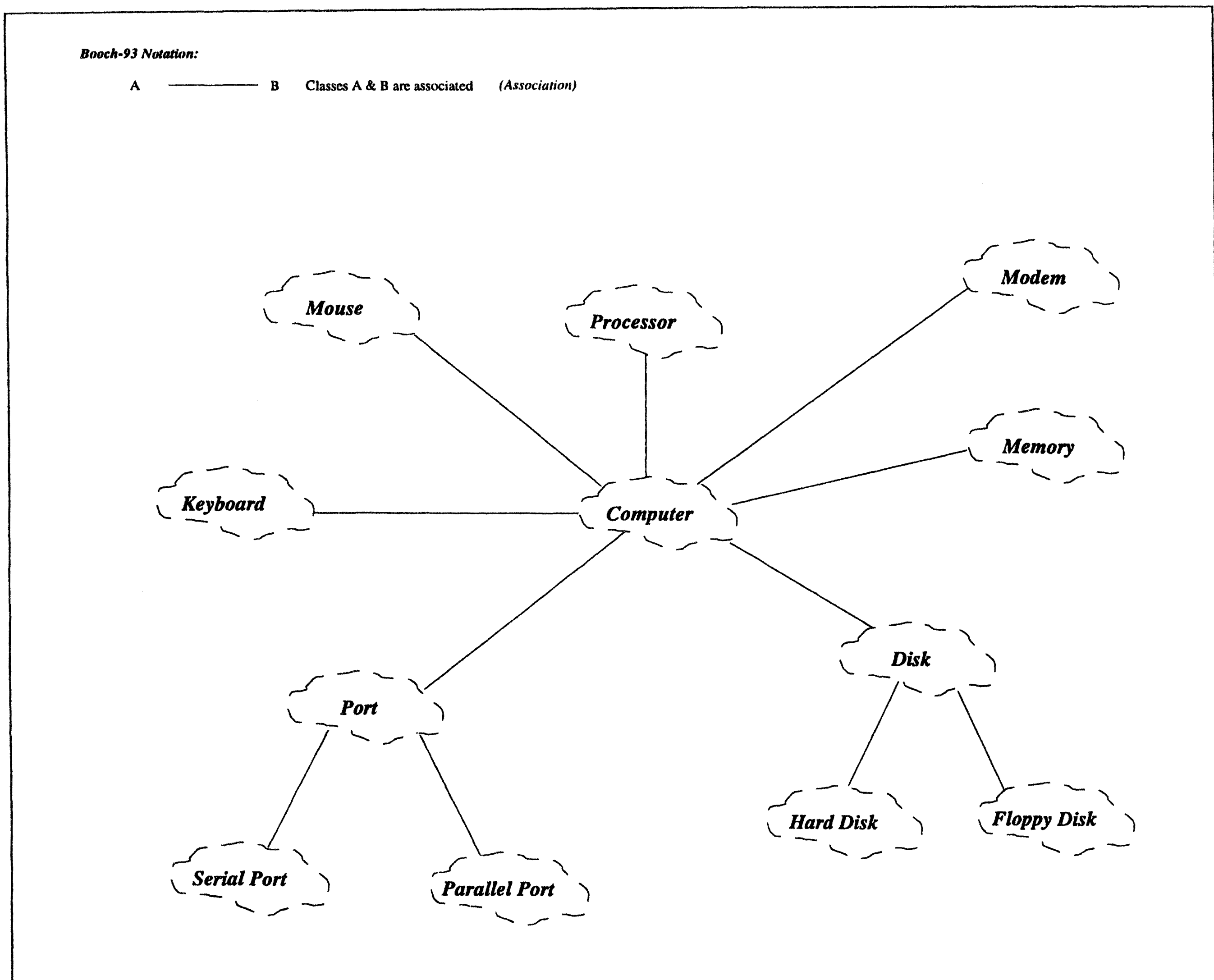

FIGURE 1.5 PC class diagram.

A true object-oriented design supports *modularity*, *encapsulation*, *abstraction*, and *design hierarchy* elements [Booch 1994]. If a design lacks any of these four characteristics, it is not considered an *object-oriented* design. For instance, a design that is based on *modularity*, *encapsulation*, and *abstraction* is considered an *object-based* design rather than an *object-oriented* design because it does not utilize a *design hierarchy*.

Concurrency, *persistence*, and *typing* are **optional** components of OOD. *Concurrency* and *persistence* usually apply to applications whose objects are distributed across different systems and platforms (*distributed architecture*). *Typing* is controlled by the programming language.

1.2.1 Modularity

During a complete software product life cycle, a product will undergo modifications and enhancements. A modular design makes product maintenance and modification easier since changes can be localized to specific modules. Modular design promotes software reuse across projects and cuts the cost of developing software. Use of common libraries also simplifies integration and evolution of software packages by centralizing functionality. Decoupling the class design improves the maintainability of the software. *Modularity* is achieved by minimizing *coupling* (interdependence between objects), and increasing *cohesion* (functional correlation between data and process).

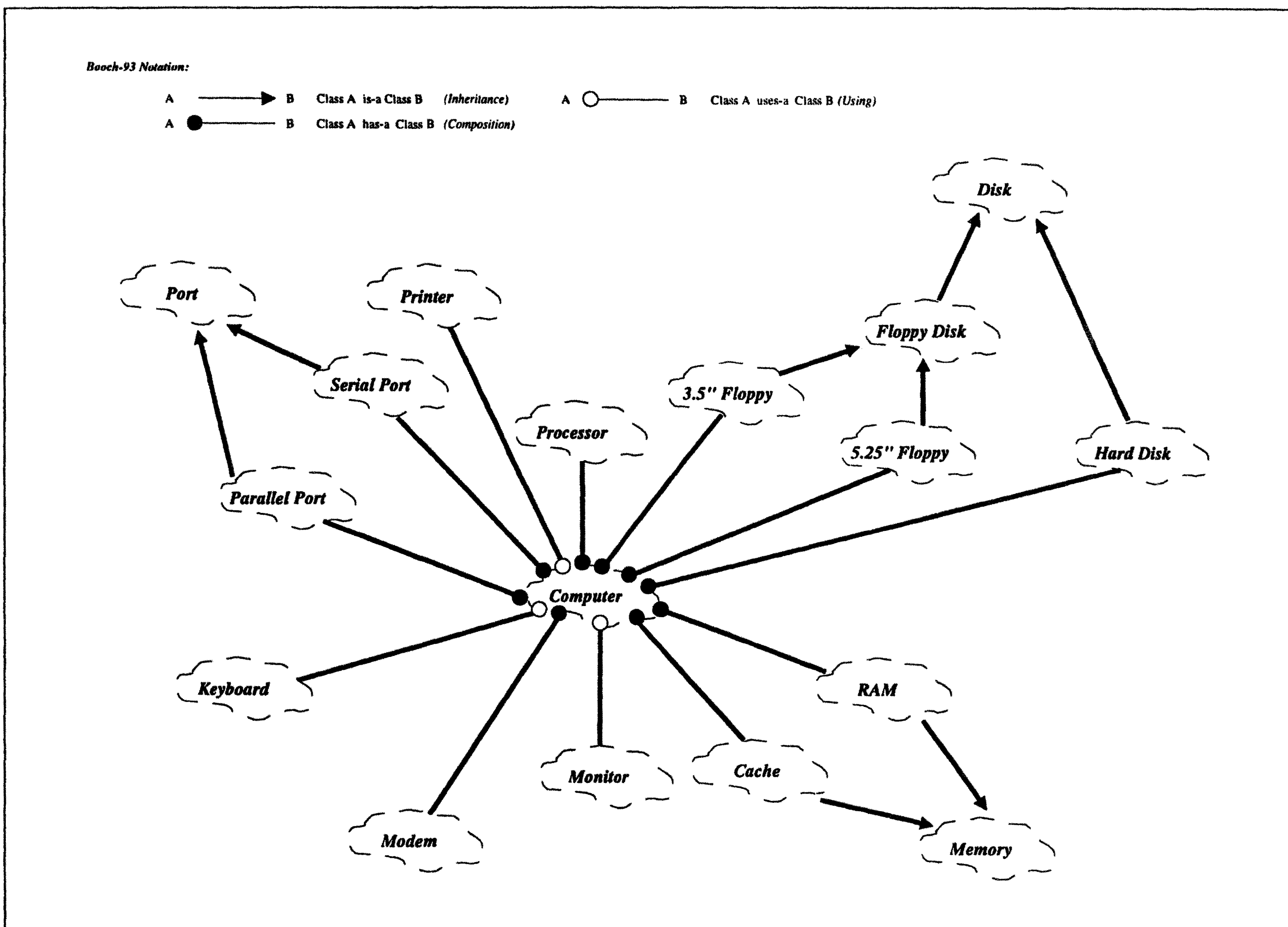

FIGURE 1.6 PC class diagram.

A modular design is created by logically partitioning a system into a group of objects and constraining these objects to interact through well-defined interfaces. In Figure 1.6, a computer system is partitioned into a set of classes by viewing its tangible components and peripherals as classes. The task of storing and retrieving information to/from a floppy disk is mapped to the *floppy disk* class, while the graphical displaying of information is allocated to the *monitor* class. The logical partitioning of classes has created a modular view of the computer.

1.2.2 Encapsulation

In object-oriented design (OOD), the class encapsulates its internal architecture and data representation from other classes. In Figure 1.6, the *floppy disk* class stores and retrieves information to a floppy disk and the *monitor* class displays information on the screen. Each class hides its internal operations and only the methods provided can be used to operate on an object (refer back to Figure 1.2). This approach makes the *computer* class design independent of *monitor* and *floppy disk* designs. For example, changes to *3.5" floppy* are localized to a single point and the *computer* remains unchanged. Furthermore, as long as the interfaces between the computer and these classes remain unchanged, changes to the internal architecture and operations are localized to the modified class.

In addition to maintainability, *encapsulation* enhances the reliability of the design. The likelihood of data corruption and of the object being placed into an *unpredictable* state is minimized. For example, if the *computer* object wrote information directly onto a floppy disk, there is a danger it would accidently corrupt the file allocation table. In such an event, the *floppy disk* object would be placed into an *unpredictable* state and the

reliability of the *floppy disk* is adversely affected. By using the **store( )** function provided by the *floppy disk* class (Figure 1.7), the *floppy disk* object would ensure the information is written properly to the disk while keeping the *floppy disk* object in a valid state.

1.2.3 Abstraction

The *operations* provided by a class abstract the internal processes from other classes. For instance, the *computer* class would use the *floppy disk* **store( )** function to store a file to a floppy disk (Figure 1.7). The **store( )** function needs to reserve tracks/sectors on the disk. Then the file content must be partitioned into blocks and the data blocks stored on the medium. As blocks are stored to specific sectors on the disk, the **store( )** function also needs to update the file allocation table to identify the locations of the blocks. The *floppy disk* **store( )** function abstracts the details of the data storage process and hardware interaction from the *computer* object or other users of the *floppy disk*. In addition, the implementation for the storage operation varies among computer systems and operating systems; the abstraction provided by the method localizes and hides these dependencies. *Abstraction* hides the implementation's complexity from other classes.

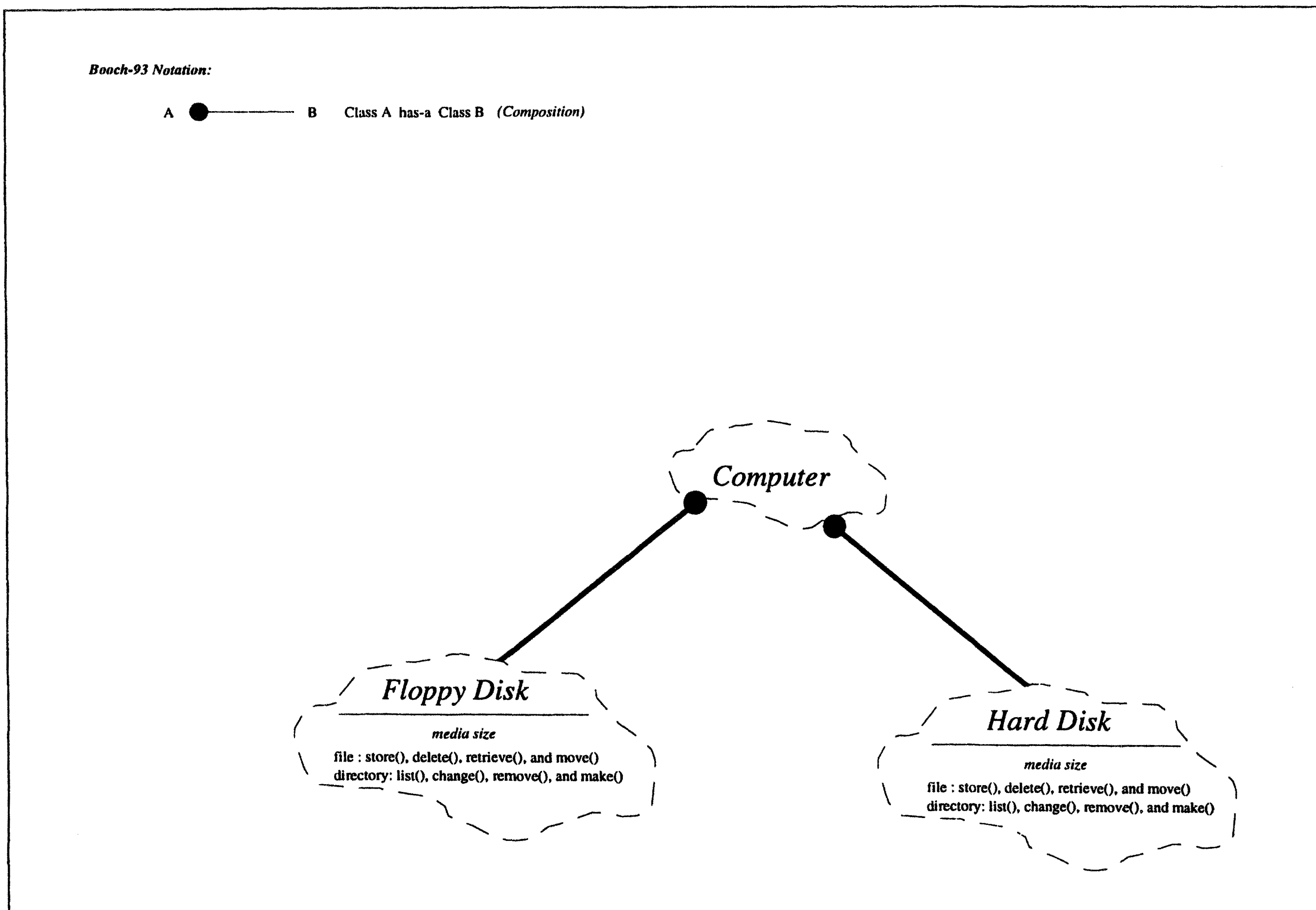

FIGURE 1.7 Disk class diagram.

1.2.4 Design Hierarchy

Design hierarchies are formed by allowing new designs to *inherit* the properties and attributes of the existing designs. Through *inheritance*, a new class inherits the attributes and services of an existing class. In a *design hierarchy*, classes form an "*is-a*" relationship. By inheriting the properties of an existing class, a new class builds on tested and mature classes, while existing designs are reused and development productivity is improved. In

Figure 1.7, both the *hard disk* and *floppy disk* objects provide directory and file services such as the **store( )**, **create( )**, **delete( )**, and **move( )** operations. The implementations for the above operations vary on the basis of system, hardware, and storage medium characteristics. The two objects can be viewed as different types of disks. From the perspective of the *computer*, the disk operations on both *floppy disk* and *hard disk* objects are identical. The design presented in Figure 1.7 does not convey any relationship between the two, which forces the *computer* object to support unique and custom code for using each of them. In addition, the *floppy disk* and *hard disk* objects are duplicating the common attributes and features in the implementation.

The design presented in Figure 1.7 can be modularized and simplified by using a *design hierarchy*. In Figure 1.8, the design for *hard disk* and *floppy disk* classes has been rearchitected. The hierarchy as shown improves the design by providing the following benefits:

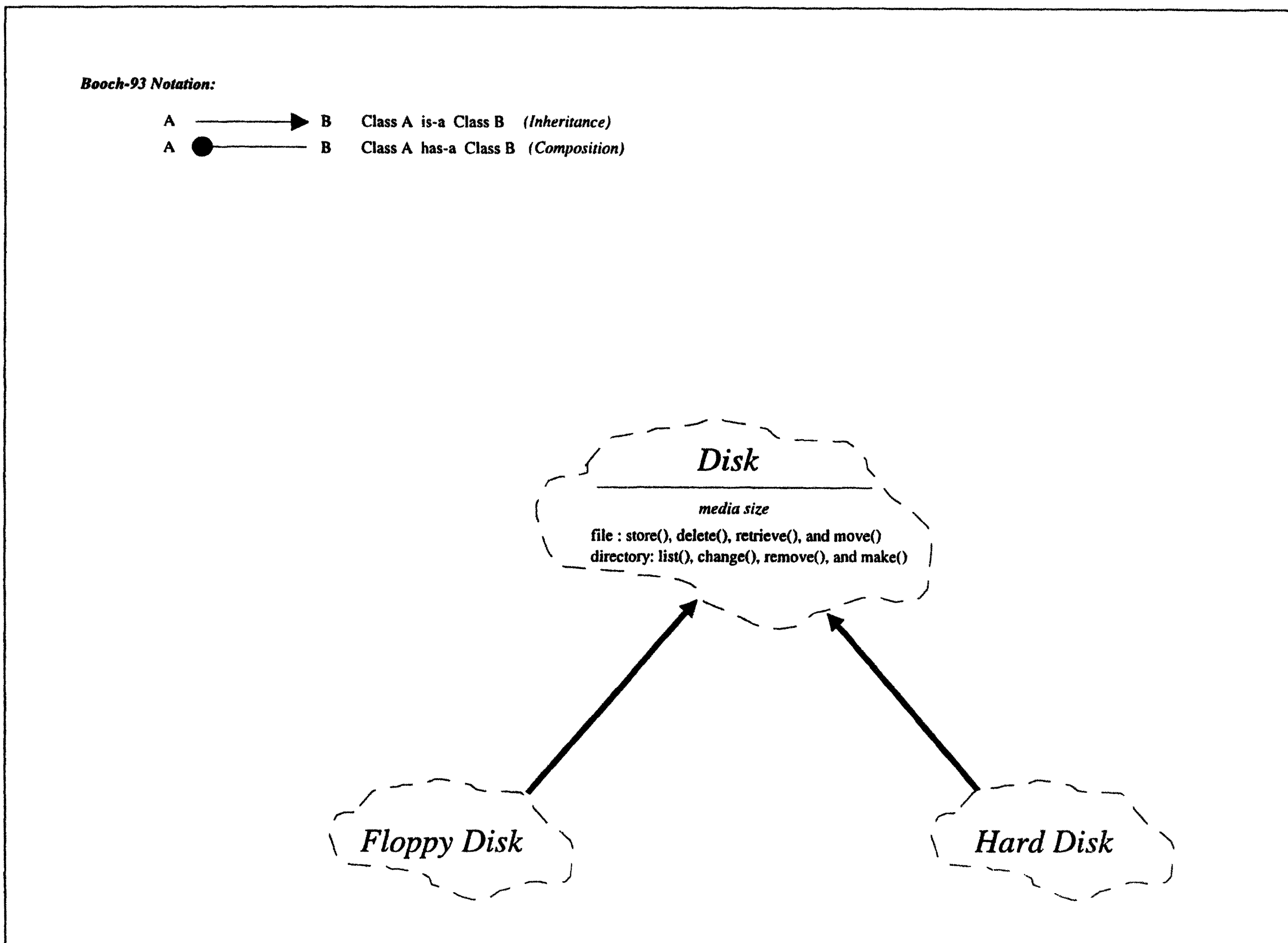

FIGURE 1.8 Disk class diagram

1. **Common Characteristics**: The common attributes and operations between the disk classes are maintained in the common *disk* class, and the *floppy* and *hard disk* classes do not need to provide unique implementations for these common features. Since common properties are inherited through *disk*, changes to the common attributes become localized to *disk*. This localization of changes will reduce redevelopment, test, validation, and integration time. As an added benefit, both the *floppy* and *hard disk* classes are built on a tested *disk* class.
2. **Data Dependencies**: Even though the services such as the **store( )** operation provided by the *floppy* and *hard disk* objects are implemented differently for each distinct object, the client perceives them as the same. By defining a common interface (parameter list) for this function, a greater level of abstraction can be

achieved because the actual **store( )** operation on different media is transparent to the client. This approach allows the client to develop applications that are independent of a specific type of *disk*:

```
disk.store( arguments )
```

The client is not dependent on any specific disk type in the hierarchy. Depending on the type of *disk* object, the applicable **store( )** operation is invoked at run time. This gives the software the ability to operate on different but related data types, and is referred to as *polymorphic* behavior. When a new type of *disk* is later added to the hierarchy, the client's program will require minimal or no modifications. The decoupling of the client's code to specific data types minimizes the data dependencies and limits the scope of redevelopment and retesting.

3. **Customization**: New objects can be added to the design hierarchy simply by creating customized versions of the existing ones. In Figure 1.9, the *floppy disk* object has been customized and two new customized versions are introduced: *3.5"* and *5.25"* floppies. These new objects are built on existing designs that are tested and validated. This characteristic can help reduce development time for a new class in the hierarchy.

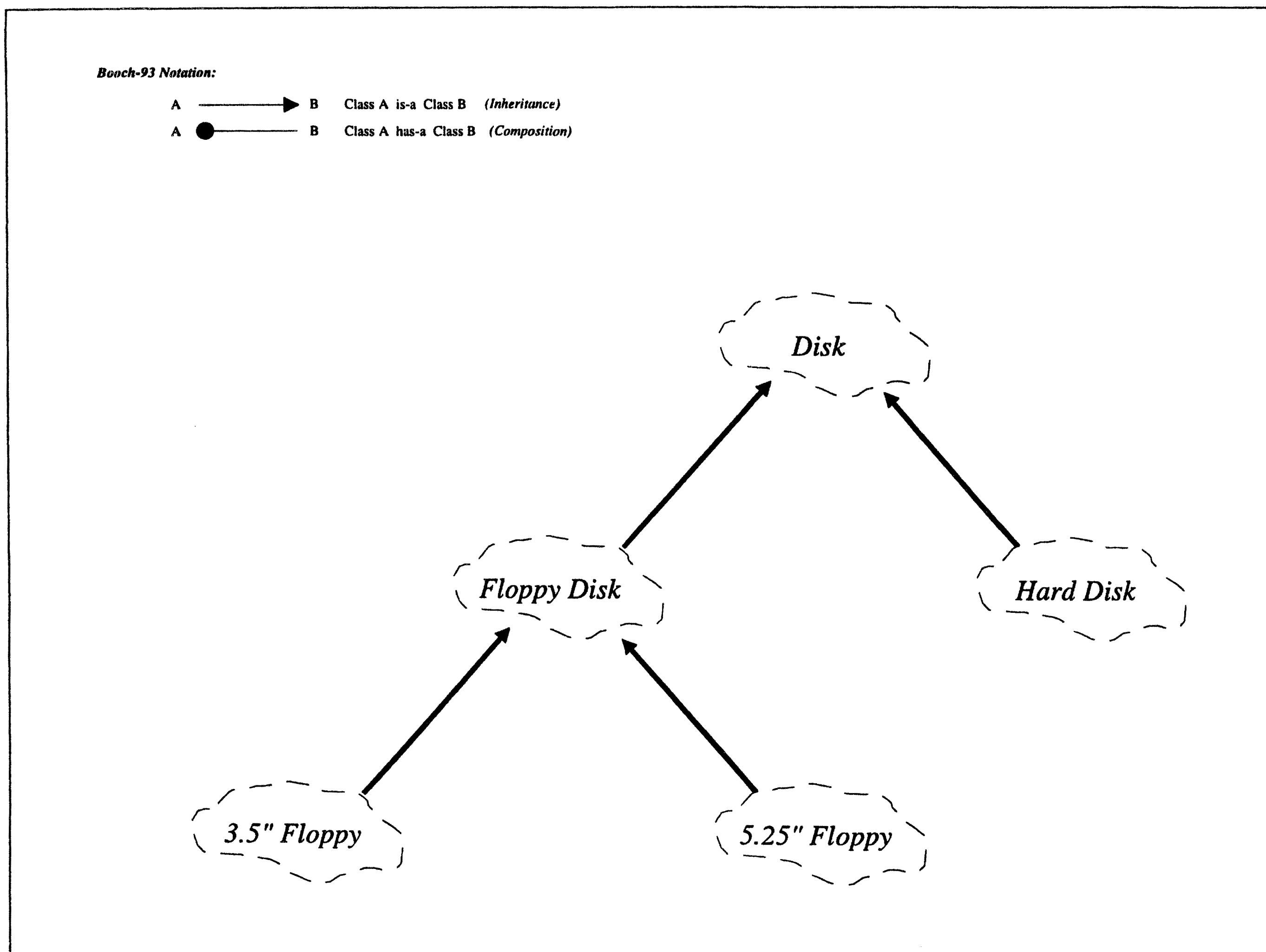

FIGURE 1.9 Disk class diagram.

A design hierarchy forms a *parent-child* relationship between the classes. The *parent* class is referred to as the *base* or *superclass* and the *child* is called the *derived* or *subclass*. In Figure 1.9, *disk* is the base class for the

floppy and *hard disk* derived classes. Similarly, the *floppy disk* is the base class for *3.5"* and *5.25" floppy disk* classes. The *computer* class diagram depicted in Figure 1.6 illustrates design hierarchies for several different types of classes, such as *memory*.

1.2.5 Concurrency

With distributed processing and client/server applications, a software application requires the cooperation of several programs running asynchronously on different platforms/systems. The objects become scattered across hardware boundaries and are no longer contained within the address space of one computer. Figure 1.10 depicts a distributed Graphical On-Line Documentation (GOLD) software application that would maintain aircraft electronic maintenance data across heterogeneous systems [PARTA 1993].

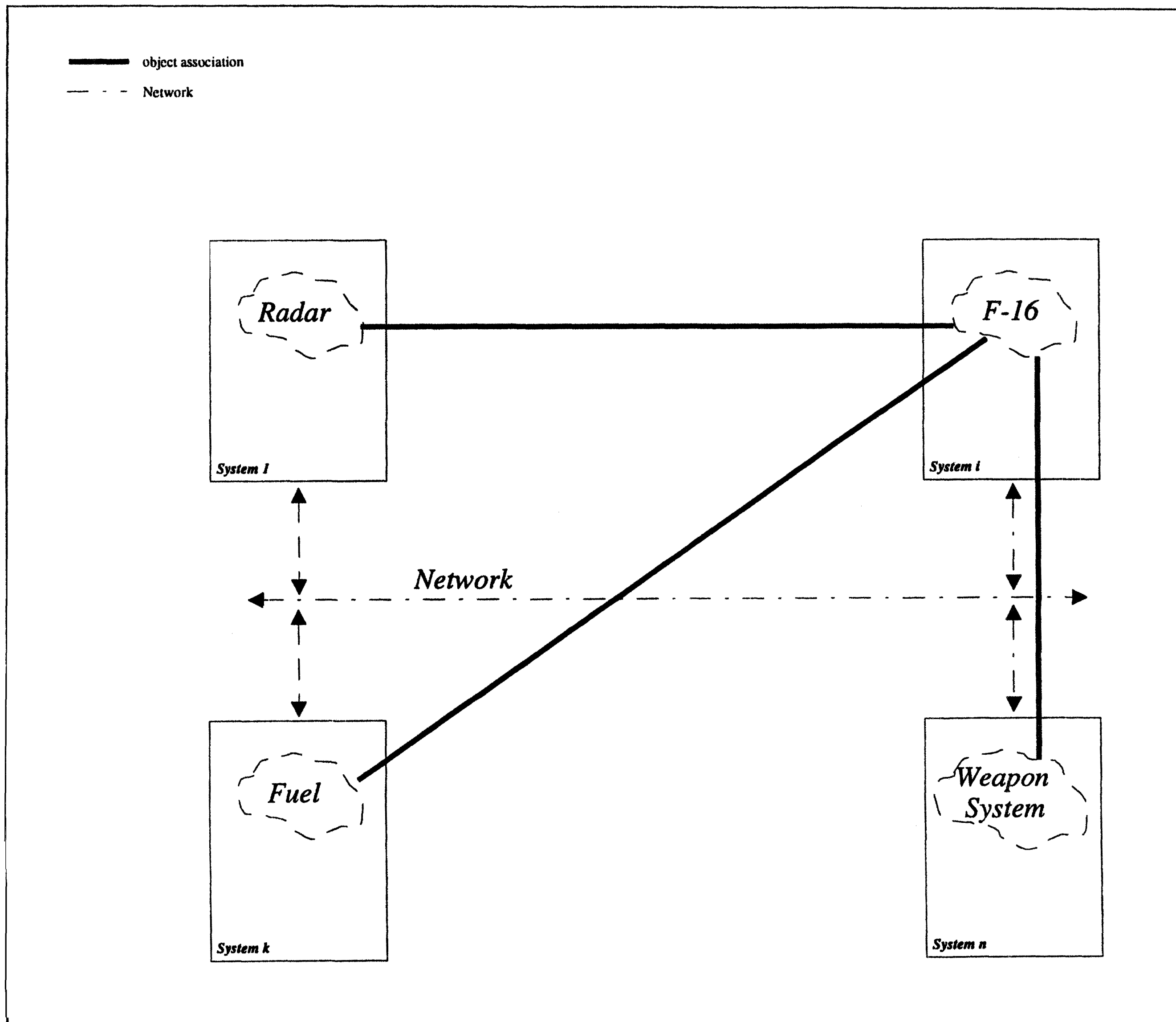

FIGURE 1.10 Aircraft repair maintenance data system.

Like a single process application, the distributed objects must interact with each other in some logical and structured manner. The distributed architecture requires the design to address *concurrency* issues; otherwise the objects will contend for resources, which could result in *deadlocks*. For instance, several objects may invoke

operations on another object simultaneously. The requests may conflict with each other and may cause the object to go into an unpredictable state. Unlike a single process application, the requests must be coordinated so that the objects will remain in a predictable state.

Regulation of concurrent processes through mechanisms such as *semaphores* assure an object's state and information are protected. A semaphore is a locking mechanism that regulates access to data by allowing only a single process to access the locked entity within a given time domain. When executing a critical area, the active process activates the lock, thus preventing other processes from accessing the protected entity. Requests by other processes will wait until the lock is released. The object will not be placed into an unpredictable state since access to protected code and data is synchronized.

Object-oriented design views *concurrency* as an optional aspect of the object model [Booch 1994]. However, with distributed architectures and objects, *concurrency* is an integral part of an object-oriented design and must be addressed. Chapter 12 discusses general concurrency issues.

1.2.6 Persistence

In a single platform application, objects are created during the program's execution and, depending on the scope of the object, they are later destroyed. The life span of the objects is directly tied to the lifetime of the program. When the program execution ceases, the objects and their states are permanently lost unless the information is stored in a database. For instance, a *human resource* software application can have *persistence* by using a database to store the employee record objects. The state of the objects will then persist beyond the end of the software execution.

In distributed applications, when a process terminates, certain objects associated with the process cannot be destroyed because other processes running on different platforms may depend on these objects and their information. Thus, these objects must persist independent of the process which they are associated. The *state* of these objects is normally stored in a database or file. Figure 1.11 depicts the graphical user interface (GUI) of the GOLD system [PARTA 1993]. In a distributed architecture, the *F-16 fighter plane* object model may be allocated across several computer systems. For instance, the *radar* object may be maintained on a UNIX-based platform and the *weapon system* object may reside on a Windows-NT-based system. When a user accesses and updates the *F-16 radar* object, the updated radar information must persist. If the updated *radar* object is destroyed at the termination of the user process, the current information would be lost. The reliability of the software application can also be compromised if the state of a distributed object is not maintained. If a second user on another system happened to access the *F-16 radar* object simultaneously, then the destruction of the *radar* object will cause unexpected results. In distributed architectures, the state of objects must transcend through time and space [Booch 1994].

The *persistence* of information for distributed objects is achieved using databases such as object-oriented database management systems (ODBMs). These systems make information storage and retrieval transparent to the client.

1.2.7 Typing

Typing is an optional criterion in OOD and is determined by the programming language. A programming language can provide *weak* or *strong* typing. In a *weakly* typed language, the designer has complete freedom to mix data types in expressions whether it makes sense or not. The following example illustrates an expression that properly mixes built-in types with an instance of a class:

```
Complex z1 = z2 + 3.0 ; // add a complex number to a real number
```

A *strongly* typed language will not allow the above operation to take place and will result in compilation error. By preventing a design from using mixed data types in expressions, a *strongly* typed language helps to avoid the logical errors caused by implicit conversions. A strongly typed language requires a software developer to specify conversions explicitly. The above operation can then take place through explicit conversions:

```
Complex z1 = z2 + Complex (3.0) ;
      // convert the real number to a complex number and
      // then add the two complex values
```

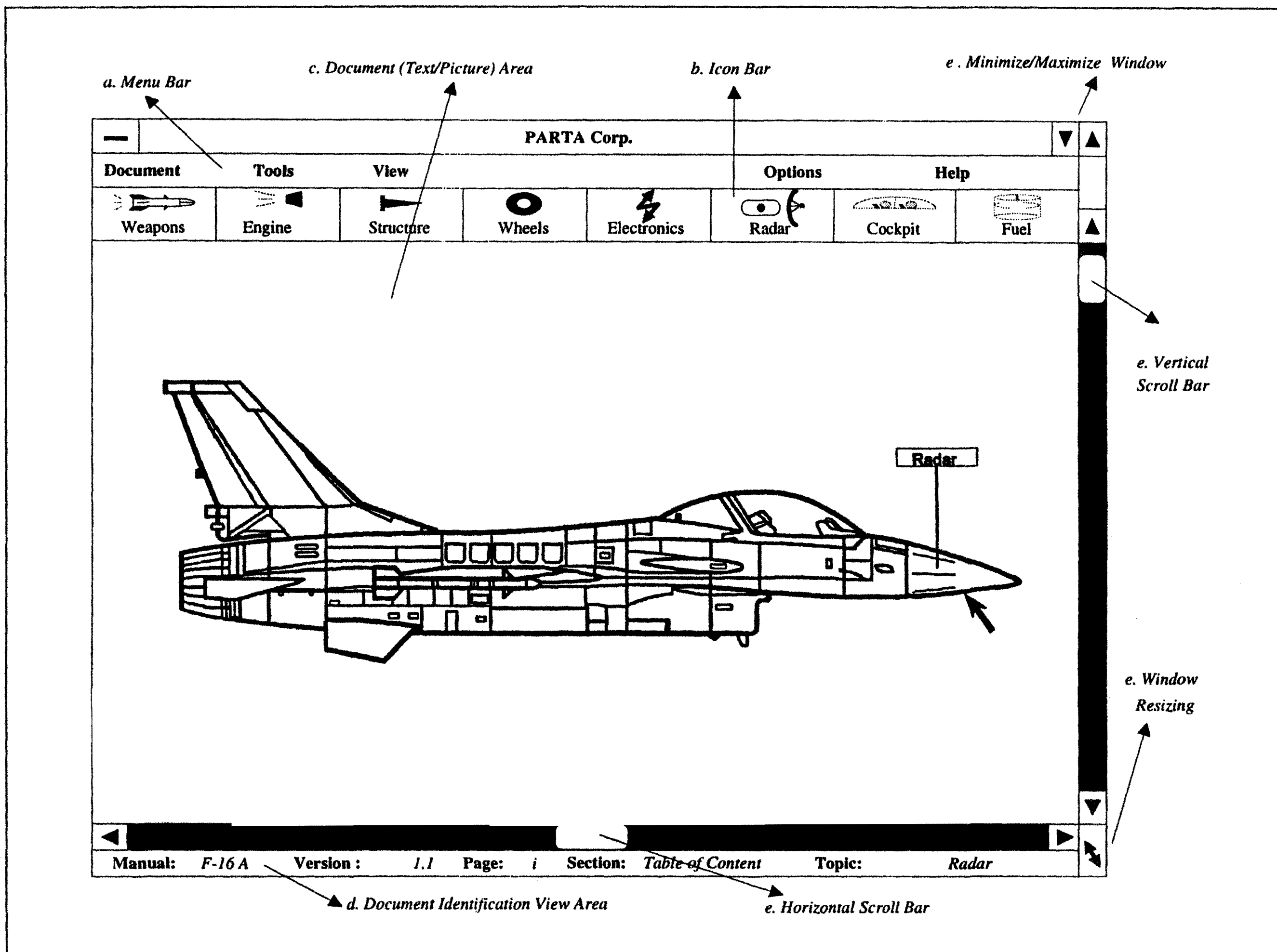

FIGURE 1.11 GOLD software user interface.

A strongly typed language makes the implementation more tedious because it forces the developer to address each type conversion. In addition, changes to the data types in the interfaces would require explicit changes to the client's code due to lack of implicit data conversion.

C++ is neither a *strong* nor a *weakly* typed language and is considered to be between the two. Through implicit conversion, C++ resolves expressions consisting of built-in data types. However, there are restrictions on implicit conversions among objects and built-in data types. Since C++ is not *strongly* typed, a designer is responsible for making sure allowable implicit conversions make sense and provide correct results.

The customer's requirements on typing determine the choice of the object-oriented language that will produce the necessary implementation, such as using Ada 95 for military projects.

1.3 RELATIONSHIPS AMONG OBJECTS

When objects interact with each other, they form relationships. One object becomes a client of another and uses the other object in some capacity. The relationship between the objects can be categorized as the following [Booch 1994]:

1. **Using**: An object uses the resources of another for accomplishing a task. In Figure 1.12, a person uses a car to go to the market. The classes form the "***use-a***" relationship:

```
person uses-a car
```

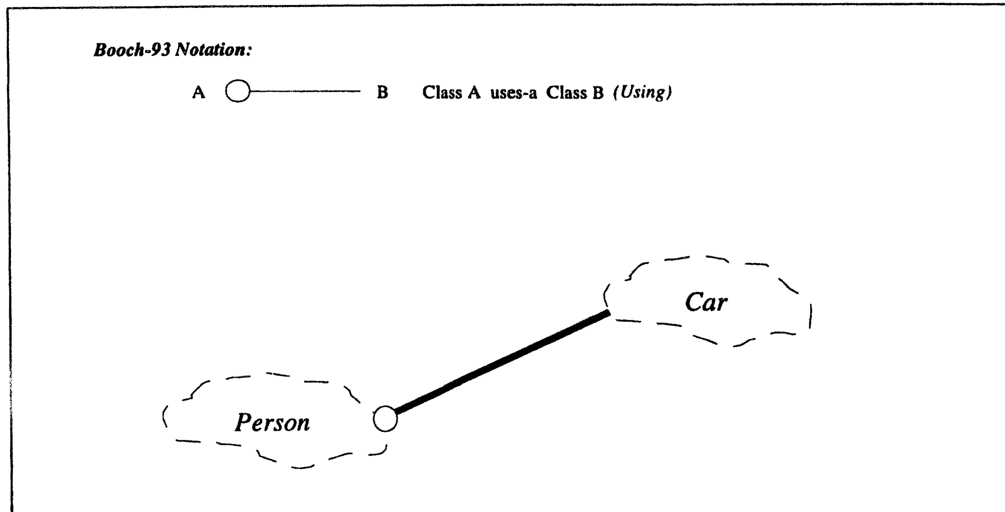

FIGURE 1.12 Car/person class diagram

2. **Composition**: In *composition*, the objects form a *system-subsystem* architecture and the design is broken into aggregated components. In Figure 1.13, the *F-16 radar* class is composed of the *transmitter, receiver,* and *antenna* classes. The decomposition simplifies the modeling by breaking a class into several components. In composition, objects form the "***has-a***" relationship:

```
radar has-a receiver, transmitter, and antenna
car has-an engine, body, and wheels
```

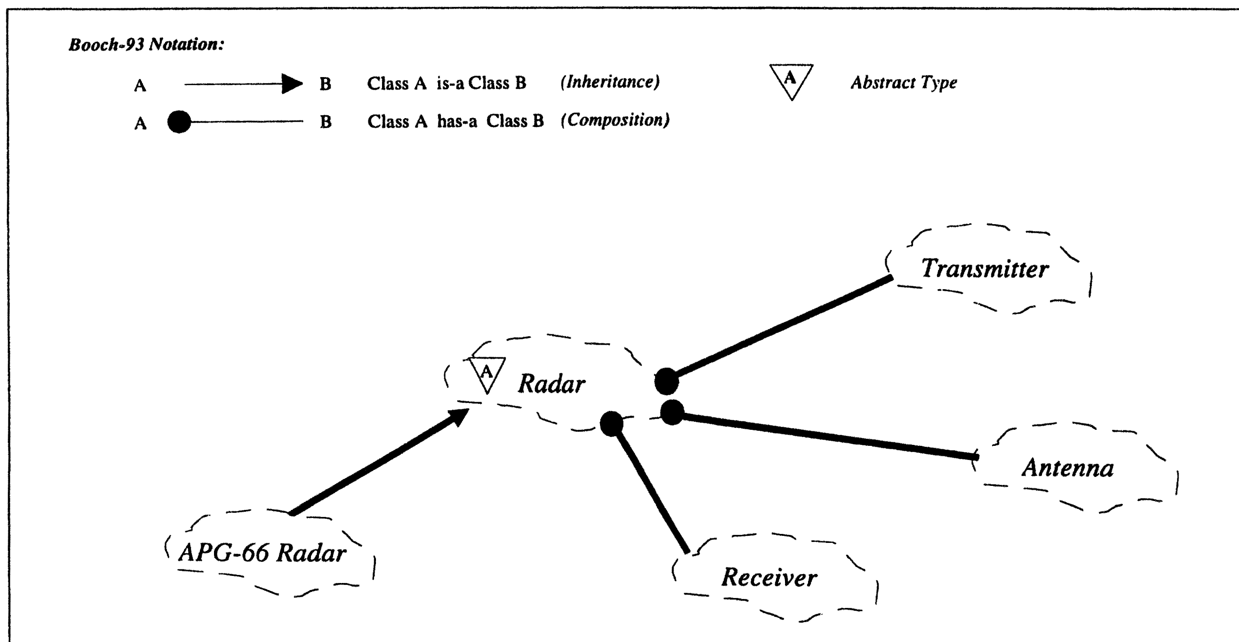

FIGURE 1.13 Radar class diagram.

3. **Inheritance**: A new class is created from an existing class by inheriting its properties. For example, a *laser printer* and *ink jet printer* are created by inheriting the properties of a general *printer* (Figure 1.14). The architecture forms an "***is-a***" relationship:

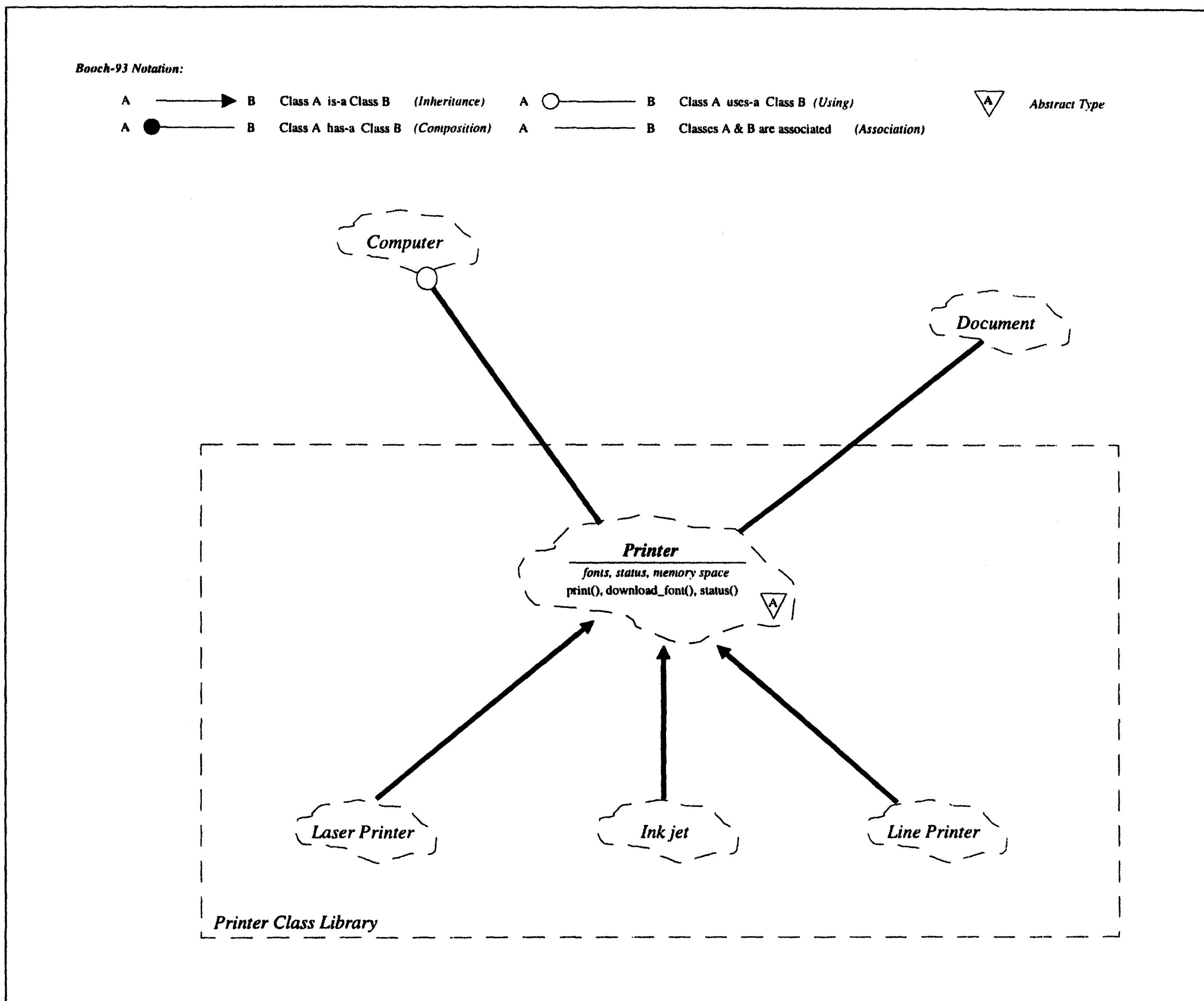

FIGURE 1.14 Printer class diagram

```
laser printer is-a printer
ink jet printer is-a printer
```

Figure 1.15 depicts a design hierarchy for military aircraft. The *airplane* class creates a design umbrella for the different types of military aircraft by identifying them as some type of an airplane. This class is considered *abstract* because in reality there is no generic airplane. In diagrams, the abstract property is highlighted by using the triangular symbol with the letter "A" enclosed in it. This symbol specifies that the *airplane* is an *abstract* class, whereas *F-16* is a real airplane and is referred to as a *concrete* class:

```
F-16 is-a fighter plane
```

An *abstract* class defines the design interface for its *derived* classes, and captures the common operations and attributes between the *derived* classes. An *abstract* class allows a client to operate on any of the *derived concrete* classes in a generic way (*polymorphism*). Chapter 10 discusses this property in detail.

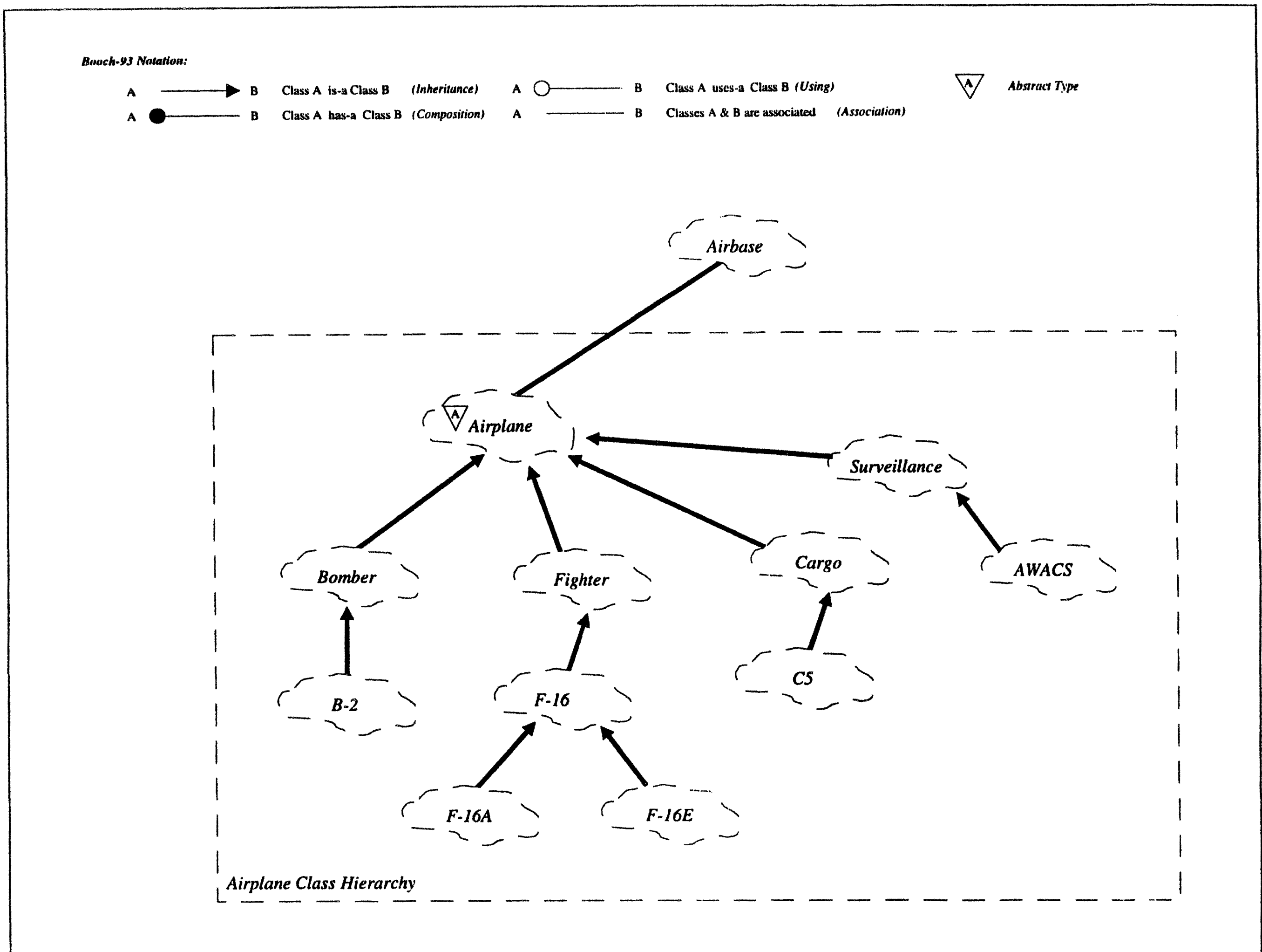

FIGURE 1.15 Aircraft hierarchy class diagram.

4. **Association**: When there is a loose relationship between two objects, they become associated with each other. It is typically used in the early stages of analysis before a more concrete relationship such as "***is-a***" has been defined. For example, a *patent attorney* and a *legal case* are associated with each other. In Figure 1.14, *printer* and *document* form an "*associated*" relationship and in Figure 1.15, *airplanes* are associated with an *airbase*:

```
a document is-associated with a printer
an airplane is-associated with an airbase
```

During the design phase of an object model, relationships such as "***is-a***" are used to describe the association between the classes. The classes in an object-oriented (OO) design must form proper relationships, and the relationships must make sense. Figure 1.16a depicts an incorrect model for a car because *a car **is-not** an engine or a wheel.* The proper design approach would have been to use ***composition*** and make ***engine***, ***wheel***, and *body* aggregate components of a *car* (Figure 1.16b). Improper use of relationships leads to OO designs that are hard to maintain.

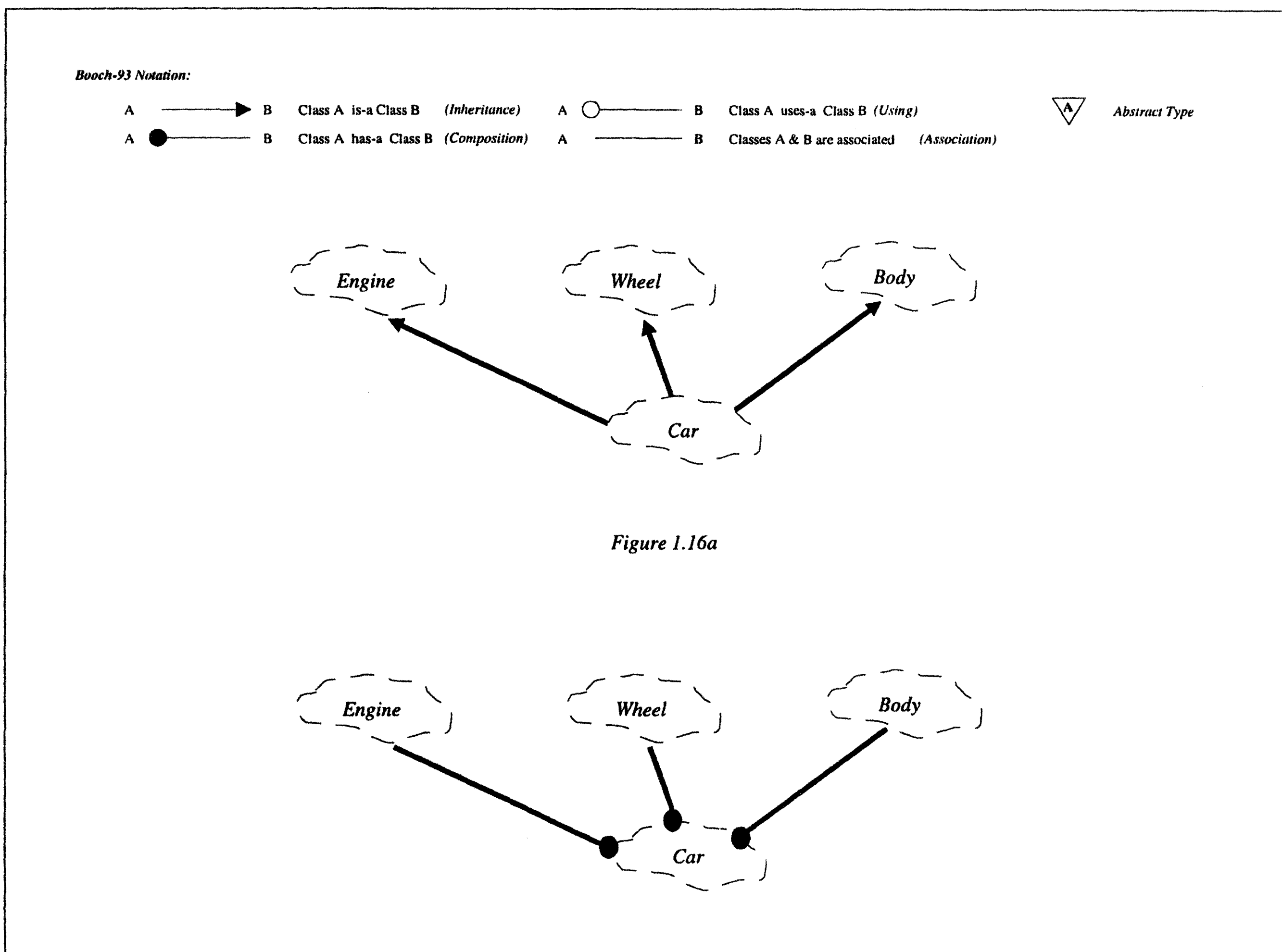

FIGURE 1.16 Improper car class diagram.

1.4 NOTATION

1.4.1 Booch-93 Notation

Graphical presentations help convey design information. For example, the architectural plans for a house presents the architect's vision to the customer before the house is built. The floor plan shows the architecture, room layouts, and sizes. The three-dimensional plan shows the facade and shape of the house. By using different types of diagrams, different information is conveyed to the customer.

Similarly, Booch-93 diagrams depicted in this book convey and describe the design. Booch-93 notation uses different types of diagrams to present different aspects of the object-oriented design, and captures both *static* and *dynamic* aspects of the object model. The system architecture, classes, and their relationships represent the *static* aspects. The *dynamic* behavior describes the interactions between the objects and their states. Figure 1.17 categorizes the following diagrams using the static and dynamic models:

a) Process diagram
b) Category diagram
c) Module diagram
d) Class diagram
e) Class specification

f) Object diagram
g) State transition diagram
h) Interaction diagram

Except for the *process* diagram and *class specification*, this book utilizes all of the Booch diagrams for its presentations. For a complete description of these diagrams, the reader should refer to *Object-Oriented Analysis and Design with Applications* by Grady Booch [Booch 1994].

In typical object-oriented designs, the object models may consist of hundreds of classes. To present these classes using a single class, object, and interaction diagram is not only impractical but counterproductive. The audience will become overwhelmed with the vast amount of information. The purpose of these diagrams is to convey design ideas in a concise and standard manner, and are not to overwhelm the reader. Therefore, different aspects of the design are illustrated using different and multiple diagrams.

Static Model	*Dynamic Model*
Process Diagram Module Diagram Category Diagram Class Diagram Class Specification	State Transition Diagram Interaction Diagram Object Diagram

FIGURE 1.17 Booch-93 Diagrams

1.4.1.1 Module Diagram

The *module* diagram presents a high-level view of the system and partitions a system in terms of *subsystems* and *modules*. A *subsystem* is a collection of *modules* logically grouped together. Similarly, a *module* is a collection of classes logically grouped together in order to perform a specific task in a system. For example, the classes associated to a fax system's *communication* module are responsible for coordinating the transmission of data via the phone line.

In Figure 1.18, a high-level module diagram for the GOLD system is presented [PARTA 1993]. The dependencies between *modules* and *subsystems* are shown, using arrows. The system is partitioned into GUI, multimedia, maintenance log (persistence), airplane repair manuals, and airbase spare parts. The module *specification* and *body* icons are based on Ada's package specification and package body. These icons can be loosely compared to a C/C++ header and source file. These icons are superimposed in the module diagram to denote the association between the declarations and definitions.

The use of a *module* diagram partitions the design along well-defined boundaries by breaking down the system architecture into *subsystems* and *modules*. The logical partitioning of a system into *subsystems* permits the targeted audience to focus on a high-level view of the system.

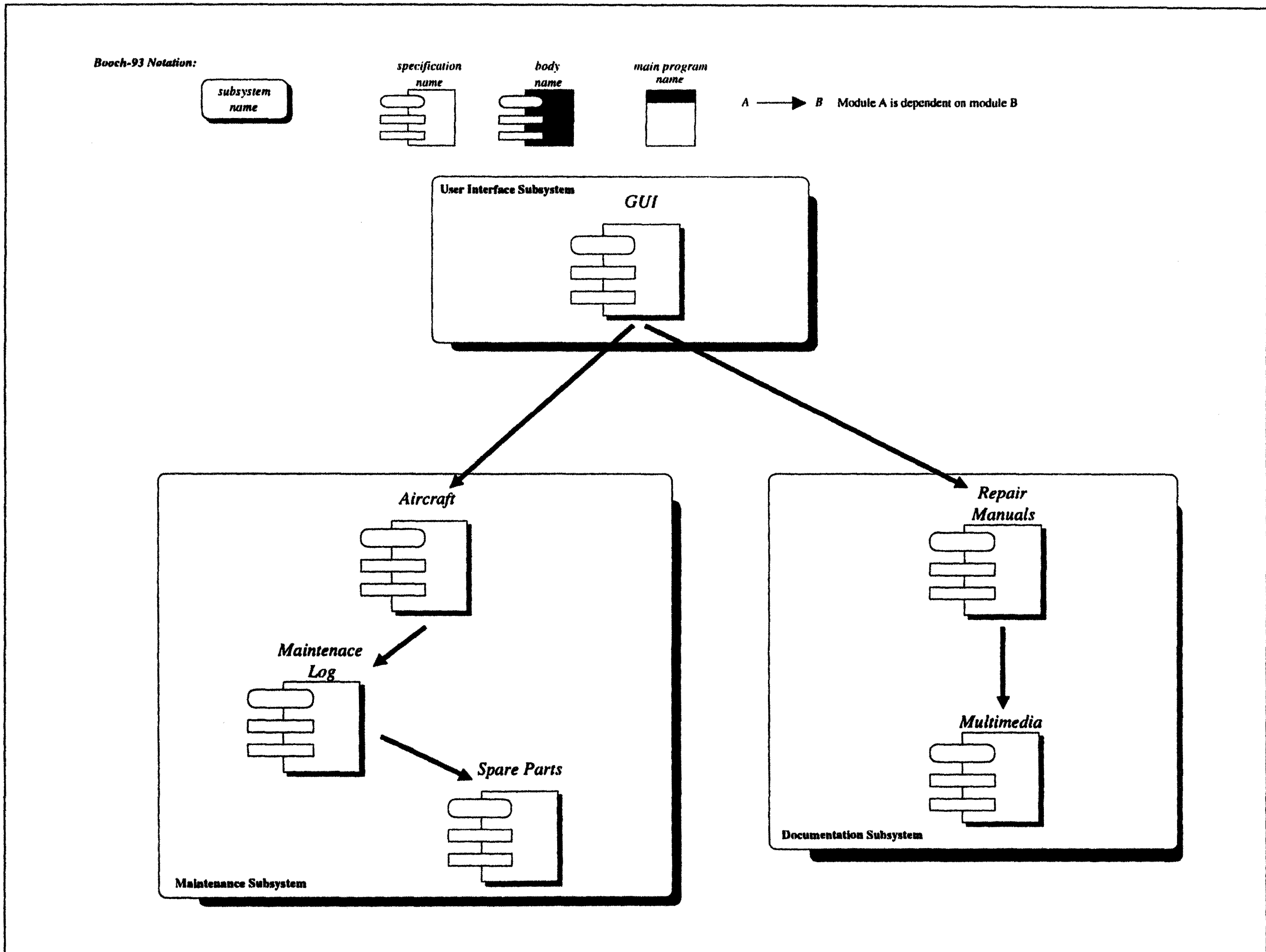

FIGURE 1.18 GOLD system module diagram.

1.4.1.2 Category Diagram

The *category* diagram facilitates the presentation and partitioning of a *subsystem* and *module* into logical and cohesive categories. Each category will consist of its own set of classes. A category organizes a group of classes in a set. In the design of complex modules, a category diagram provides a high-level view of the module's detailed design. The classes associated to category may be documented in the category diagram by listing the classes in the applicable category box. The categories interact with each other through well-defined interfaces.

Figure 1.19 depicts the categories used in the design of the *spare parts* module for the GOLD system [PARTA 1993]. This diagram views the system as a hierarchical (multilayered) architecture by having the higher level categories appear on top.

In the diagram, the categories form a "***using***" relationship. The common and core categories that are used by all or most categories in the diagram are grouped at the bottom of Figure 1.19 and are denoted using the "**global**" keyword. This notation prevents the diagram from becoming too cluttered with the "***using***" relationship lines.

1.4.1.3 Class Diagram

For a detailed view of a module or class category, a *class* diagram is used to show the exact relationships between the classes. A *class* diagram captures detailed information for the static model by identifying the classes and their relationships. Examples of this diagram have been depicted throughout the chapter, such as in Figure 1.15. In addition to relationships, a class diagram may specify other details such as *containment, cardinality, properties,* and *export controls*:

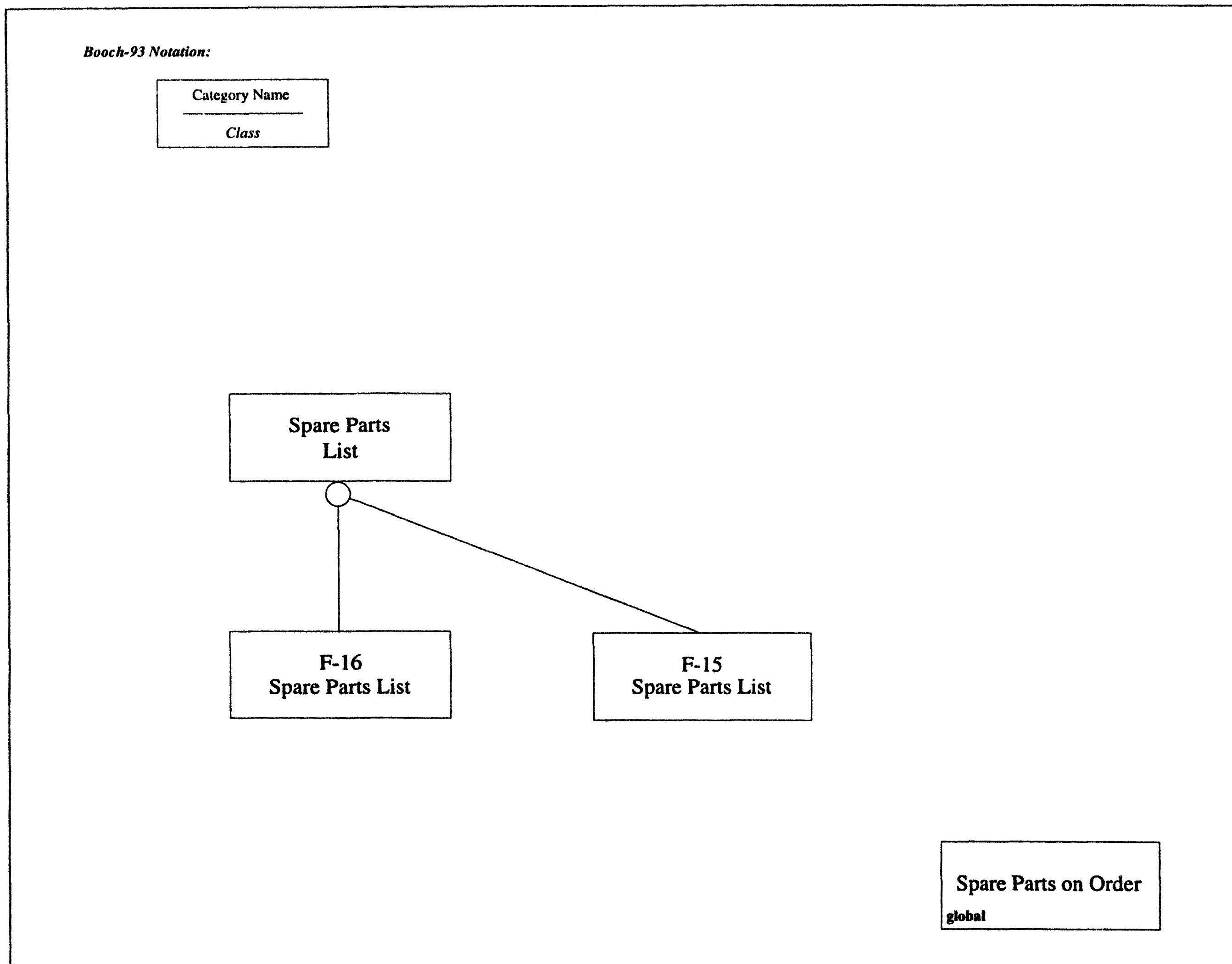

FIGURE 1.19 Spare part list category diagram.

1. **Containment:** Identifies how a system class maintains its subsystem class: *externally* or *internally*. For example, an *F-14 fighter plane* may be modeled on the basis of its structural architecture. Figure 1.20 decomposes this plane and shows the relationship and containment type for some of its parts. For every instance of an *F-14 fighter plane* class, there are two instances of the *engine* and three instances of the *wheel* classes.

   Containment is denoted by the square shape symbols. A darkened square denotes internal containment (value) and a white square denotes external (reference) containment. With *internal* containment, an instance of *F-14* contains three instances of *wheel* objects. The lifetime of *F-14* and its *wheels* are closely coupled. Since an *F-14* contains three instances of *wheel*, when an *F-14* object comes into existence the *wheel* objects are also created, and when the *F-14* object is destroyed, the *wheel* objects are also destroyed. In the case of *external* containment, the above is not necessarily true. *F-14* may create the *engine* objects or they may have been created prior to construction of *F-14*. The lifetime of the two objects becomes an implementation issue. However, *F-14* is basically referencing two *engine* objects, which reside outside of it.
2. **Cardinality**: Specifies the number of instances associated between two classes. In Figure 1.20, for every *F-14* object there are two instances of engines. Therefore, there is a 1-2 cardinality. Booch notation supports one-to-many, many-to-one, or many-to-many cardinality [Booch 1994]:

.Exactly one relationship:	1
.Unlimited number (0 or more):	n
. Zero or more:	0 .. n
. One or more:	1 .. n
. Range:	10 .. 30
.Range and number:	2 .. 4 , 8

Cardinality applies to *composition* and *association* relationships.

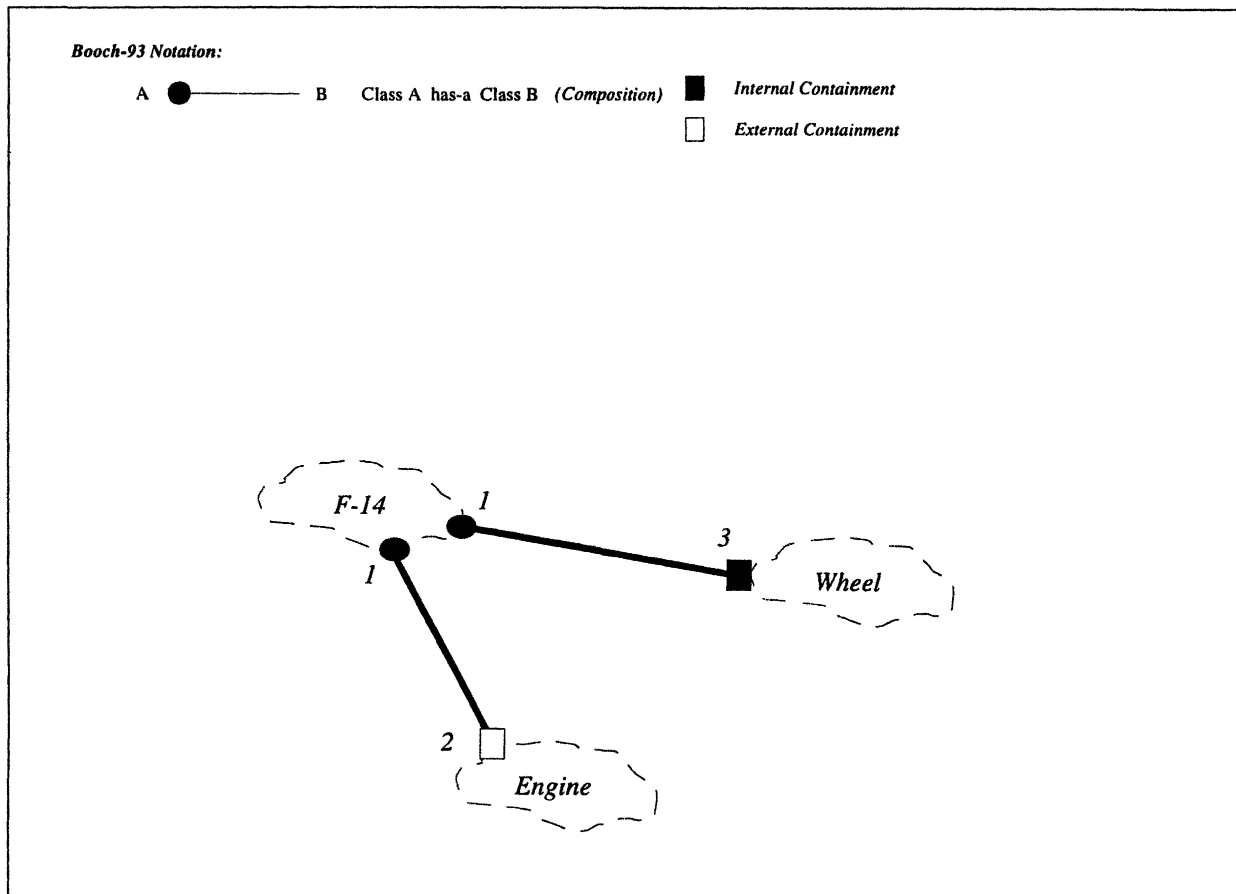

FIGURE 1.20 F-14 fighter plane class diagram.

3. **Property**: A property is enclosed in an upside-down triangle. The following identifies the available properties:

   a. *abstract (A)*: An abstract class is used in the formation of a design hierarchy. In Figure 1.15, a generic aircraft was marked as an abstract class. These types of classes establish a common design interface for their derived classes and cannot have an instance.

   b. *friend (F)*: This property denotes that a class is a friend of another and has access to the internal data members of another class. This property tightly couples design components and eliminates modularity and encapsulation between the two classes. This property should be used sparingly in a design.

   c. *virtual (v)*: The virtual property is used when a class may be inherited indirectly twice or more. Examples of abstract and virtual properties are presented in Chapters 9 and 10 of this book.

d. *static (s)*: This property identifies that the class is a static member of another class and one instance of the class will be shared by all instances of the other class. Static members are described in detail in Chapter 5.

4. **Export Controls**: Identifies the access levels. There are four types of export control: public, protected, private, and implementation. The export controls are depicted using vertical bars on the relationship lines. Examples of this feature are provided in subsequent chapters.
Except for the abstract property, the other properties appear on the line identifying the relationship between two classes. Examples of these properties are provided in later chapters.

1.4.1.4 State Transition Diagram

A *state transition* diagram defines the dynamic behavior of an object by identifying the possible states for an object. This diagram identifies the *events* and *operations* that cause the object to transition from one state to another. In Figure 1.21, the state transition diagram identifies the possible states for the *modem* object: **idle, setup, transmitting, receiving, error,** and **termination**. The condition enclosed in brackets identifies the criterion that has caused the state transition.

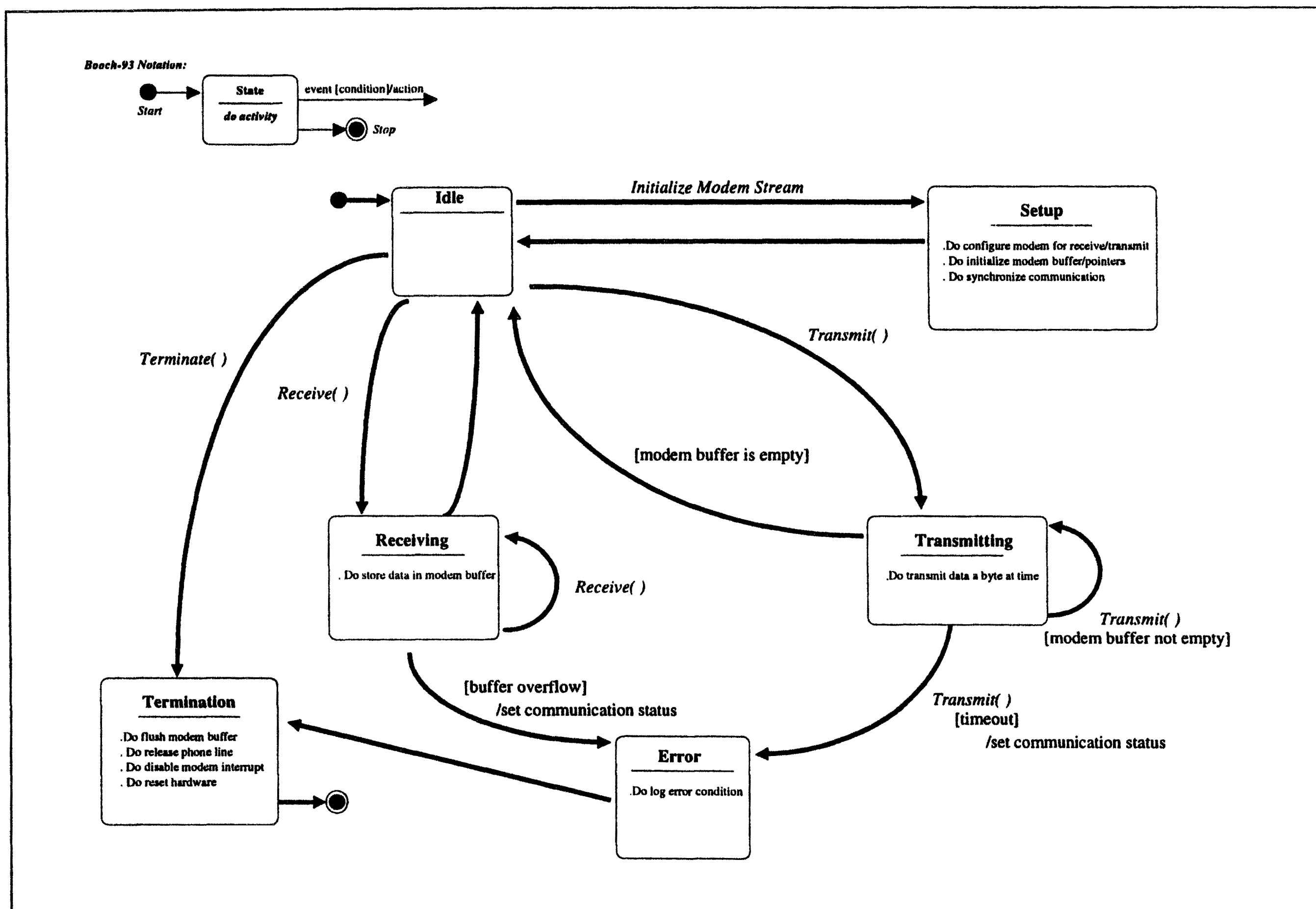

FIGURE 1.21 Modem state transition diagram.

Some of the *events* and *operations* cannot cause the object to transition unless a certain *condition* is met, also known as a *guarded condition*. For instance, a transmit request event causes the **transmitting** state to transition to the **error** state when the modem times out. Since an *event* may cause a state to transition to several other states, the *guarded condition* is used to identify the criterion for the transitioning and makes the state diagram deterministic. In the **transmitting** state, the **transmit()** operation either maintains the object in the same state or transitions it to the *error* state. Without the *guarded condition*, the behavior is not clear. Associated with an *event*, there also may be an *action*. For instance, the communication failure status is set before transitioning

from the **receiving** state to the **error** state. An *action* is considered to take zero processing time and cannot be interrupted. The following identifies the sequence for transitioning from an event:

1. An event occurs
2. Guarded condition is evaluated
3. Action is performed
4. State transition takes place

Within a *state*, several activities can take place and are documented by "*do activity*." In the error state, an activity is to log the error condition. An activity may be interrupted by an event.

A transition diagram may identify entry and exit points denoted by darkened and semidarkened circles, respectively. The entry point identifies the initial state when an object comes into existence. There can be only one entry point in a transition diagram but there can be many exit points (Figure 1.21).

1.4.1.5 Interaction Diagram

The *interaction* diagram traces events within a design and defines the *messages* (events and operations) between the objects. Figure 1.22 illustrates the events and operations associated with the interaction among the personnel

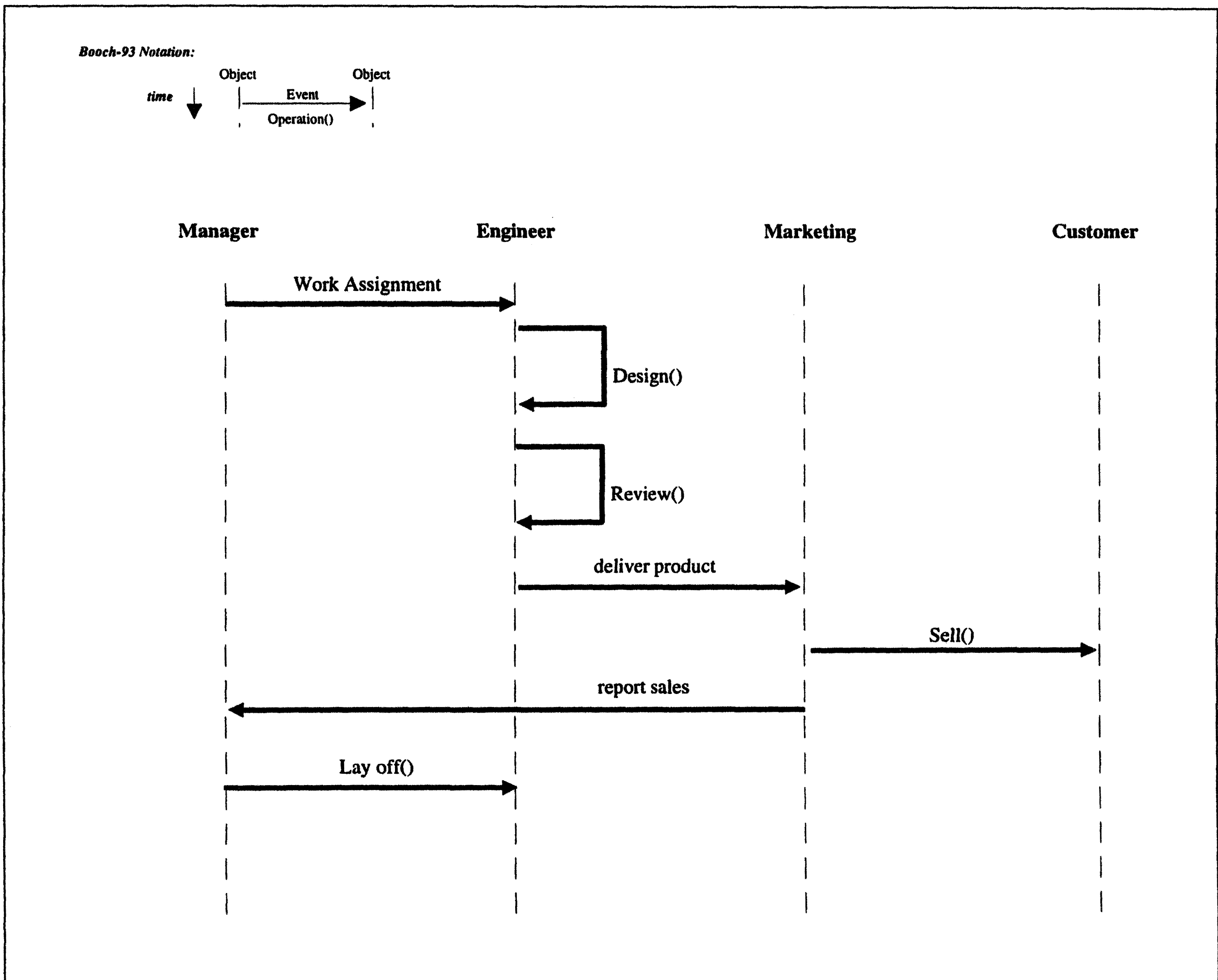

FIGURE 1.22 Software development group interaction diagram.

of a software development company in the 1990s. The *operations* are differentiated from *events* by using parentheses. In this diagram, the company's personnel are viewed as objects. This diagram provides a chronology of events: the company executive provides the Research/Development (R/D) engineer with a statement of work. The engineer designs the product and then reviews the design. Upon product acceptance, the engineer delivers the product to the marketing department, which later sells the product to the customer. Marketing reports the product sales to the management. At the end of the project, the management consolidates the company by laying off the engineer. The order of events is from top to bottom with the time axis pointing downward.

The interaction diagram does not provide many details, making it useful for high- level design. The diagram permits algorithm notes and pseudocode to appear on the left-hand side, explaining the events in greater detail in addition to identifying conditional statements, decisions, and loops.

1.4.1.6 Object Diagram

The *object* diagram presents the same information as the *interaction* diagram except that it shows greater detail. This type of diagram defines the object interactions, synchronization, roles, visibility, data flow, and data direction.

Figure 1.23 uses a new notation to show the event interactions for the software development company depicted in Figure 1.22. Unlike an *event interaction* diagram, the location of interaction lines in the *object* diagram are unimportant. Thus, the events and operations must be numbered in order to identify the sequence of events.

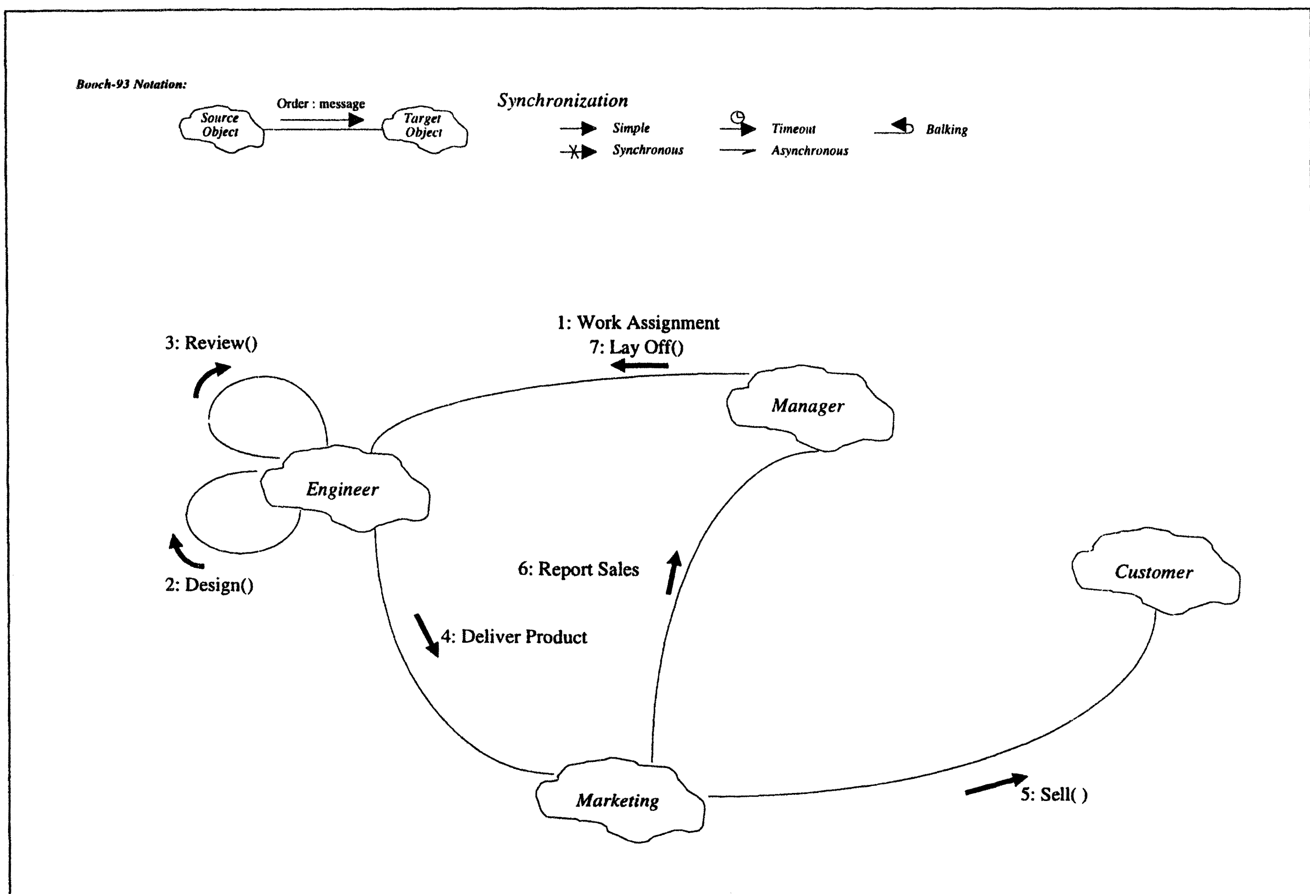

FIGURE 1.23 Software development group object diagram.

The direction of the message is shown by the arrows. The object diagram supports different notation for the messages in order to support real-time applications and concurrency:

1. **Simple**: The standard arrow is used to depict a single thread control from a source to a target object.
2. **Synchronous**: A synchronous message is shown by an arrow with an X mark. This message indicates the operation that takes place after the target object accepts the request. The source object waits until the message is accepted.
3. **Timeout**: The timeout synchronization is shown by the clock above an arrow to indicate that the event or operation must be completed within a specified amount of time. The source object abandons the operation if it is not completed within the allowed time.
4. **Asynchronous**: An asynchronous message is shown by a *modified arrow* (¬). This synchronization scheme is used for multithreaded applications when the source object sends a message to another object and continues its execution without waiting for a response. The *modified arrow* (¬) denotes that the operation is executed asynchronously.
5. **Balking**: The balking synchronization is shown by an arrow that points back to the source object. The message is passed to the target object only if the target object is ready to receive it. The operation is abandoned if the target is not ready.

The above synchronization scenarios are useful for describing the dynamic behavior of distributed objects. Subsequent chapters utilize the above symbols in their object diagram examples.

1.4.2 Function Hierarchy

For classes whose operations form a hierarchy, the function hierarchy diagram is used to illustrate the relationships between the functions. This diagram is useful for low-level design and is not part of the Booch methodology. This book uses this type of diagram to show the relationships between the functions. For example, Figure 1.24 depicts the data transmission function hierarchy for a *Serial Stream* class library. The external operations are denoted by **bold face** characters and are part of the design interface. The operations denoted by nonbold face characters are internal functions to the design of the class. This diagram creates a pictorial representation of the functions and their relationships in a complicated design. In Figure 1.24, the **transmit_packet()** function calls several private and public member functions such as the **transmit_string()** and **receive()** functions.

SUMMARY

The object-oriented design (OOD) methodology focuses on *objects* rather than on processes and algorithms. Objects become abstract representations of the entities in the problem domain. For example, the design of a facsimile (fax) machine will focus on identifying the objects required to represent a fax machine, such as a modem, scanner, keypad, and printer. These objects collaborate with each other and model the operation of a fax machine. Object-oriented design formalizes the definition of an object as *"an entity that has a state, identity, and behavior."*

In C++, only the memory layout and content (static behavior) of the object can be defined at the source file level, and is known as the *class* definition. A class definition logically groups data and functions. On the basis of the values and settings of the data members, the instance of the class would exhibit a state. The member functions operate on the data elements and perform operations such as initialization and cleanup. These functions give the instance of the class a behavior. Since a class may have many instances, each instance is considered to have a unique identity. *An object is an instance of a class.*

In object-oriented systems, an *object* model is created by analyzing the software requirements. The granularity of the *object* model is enhanced and completed iteratively. Finally, the design is implemented using an object-oriented programming language. An object-oriented design must incorporate and address the *major* elements of the object model's framework:

1. Modularity
2. Encapsulation
3. Abstraction
4. Design hierarchy

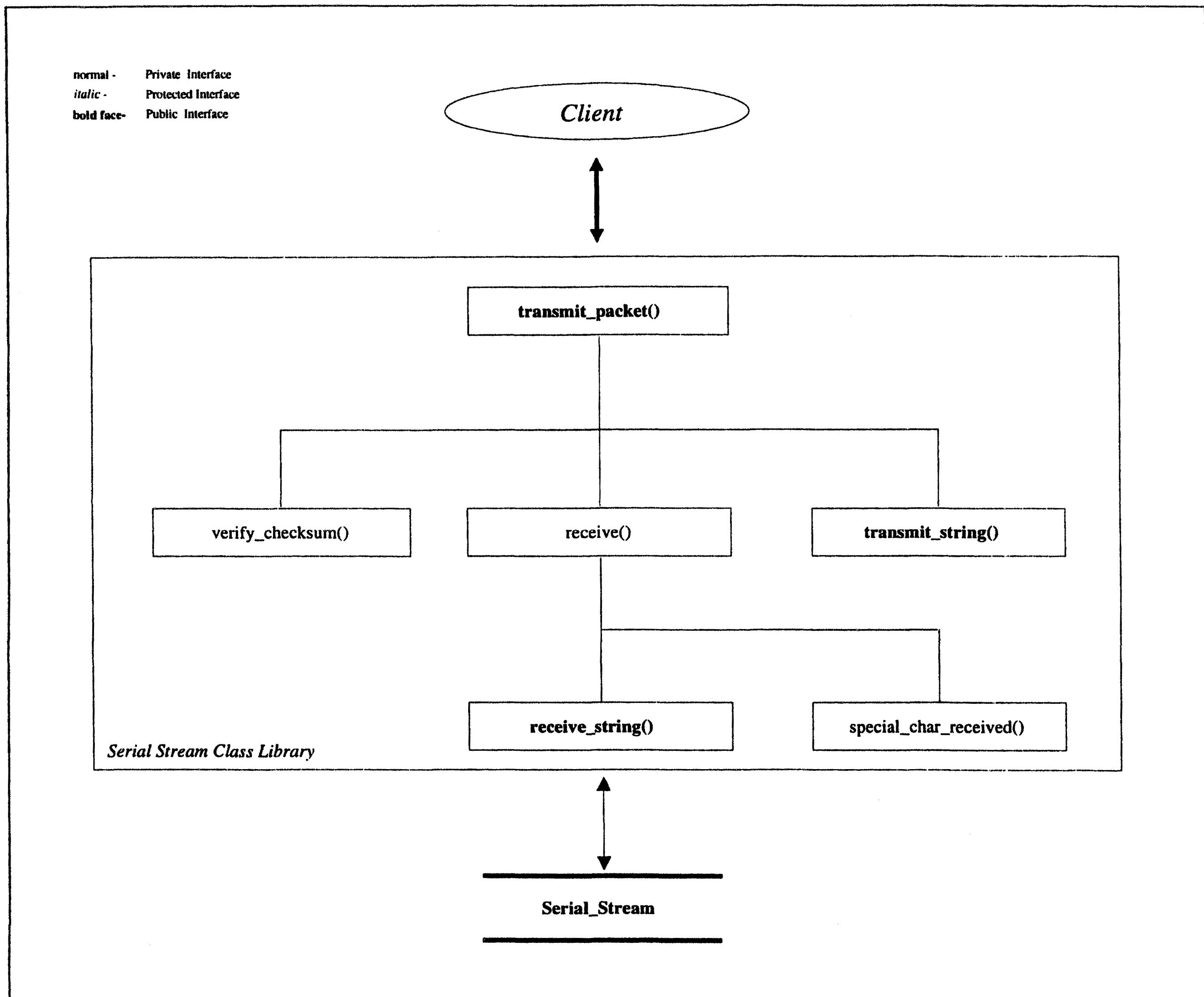

FIGURE 1.24 Serial stream class function hierarchy (data transmission)

The framework is formalized to enhance software maintainability, reliability, and reusability. These elements are required components of the *object* model. Designs lacking the design hierarchy elements are called *object-based* designs instead of *object-oriented* designs.

The object model also has three other *minor* elements that are considered optional:

1. Concurrency
2. Persistence
3. Typing

Concurrency and persistence play an important role in distributed objects where the objects reside across multithreaded and multiprocess environments.

At the time of implementation, the *object model* must identify both dynamic and *static* behaviors:

1. **Static Behavior**: The system's architecture, data representation, interfaces, and relationships among the classes are identified.
2. **Dynamic Behavior**: The state and the interaction among the objects are specified.

To help capture the above information, industry standard notations such as Booch are used. Owing to the graphical nature of the Booch notation, this book uses the Booch-93 notation. The diagrams represent the details and the architecture of design and convey ideas visually, similar to other notations such as the *Unified Modeling Language* [Rational 1996].

GLOSSARY

Abstraction
Hiding the complexity of the design and the underlying processes involved in carrying out the operations on an object

Base class
In inheritance, the *base* class is used to create new classes (*derived*). The *derived* classes inherit the properties and attributes of the existing class (*base*)

Cardinality
Number of instances of a class

Class
A class is a structured set consisting of the declarations of member data with the member functions being the operation on the set. A class identifies the memory layout for an object and specifies the functions that operate on the object

Client
A class becomes a client of another class by using its features in its design. Depending on the relationships and syntax in C++, the clients can be grouped into *is-a*, *associate*, *has-a*, and *use-a* relationships

Data encapsulation
The process of hiding the internal architecture of a data structure and the underlying data representation by building a series of functions around the design. The functions act as the gateway by regulating the access of *clients* to the data members

Derived class
By inheriting properties and attributes of an existing class (*base*) and adding additional features, a new class (*derived*) emerges from the design

Inheritance
The ability to inherit the architecture, attributes, and properties of an existing (*base*) class in the design of a new (*derived*) class. Inheritance creates an "***is-a***" relationship between the *base* class and its *derived* class

Modularity
A design that exhibits high cohesion and low coupling

Member function
A function that can operate freely on the data members of a class and is part of the class design. A member function hides the internal architecture and specifies the operations on the class

Method
An Ada term for function. This book uses method, function, and operation interchangeably

Object
An object is an entity that has a **state, behavior**, and **identity**

Object-based
A design that focuses on collaborative effort of objects that are instances of different classes. However, the objects do not form hierarchical architecture

Object-oriented
A design must not only satisfy the *object-based* criteria, but also the classes must form design hierarchies through *inheritance*

OOA
Object-oriented analysis (OOA)

OOD
Object-oriented design (OOD)

Subclass
Another term for the *derived* class

Superclass
Another term for the *base* class