JSD is ill-suited for some other applications:

- High level analysis. JSD does not foster broad understanding of a problem. JSD is ineffective at abstraction and simplification. JSD meticulously handles details but does not help a developer grasp the essence of a problem.
- Databases. Database design is a more complex topic than Jackson implies. JSD modeling is biased towards actions and away from entities and attributes. As a natural consequence, it is a poor technique for database design.
- Conventional software running under an operating system. JSD's abstraction of hundreds or thousands of processes is confusing and unnecessary.

12.3 INFORMATION MODELING NOTATIONS

OMT object modeling combines object-oriented concepts (class and inheritance) with information modeling concepts (entities and associations). Information modeling originated within the database community and is concerned with modeling the structure of data so that it can be properly managed with a database.

The Entity-Relationship (ER) approach [Chen-76] is the most common approach to information modeling. ER is a graphical technique that is popular because the notation is easy to understand, yet powerful enough to model real problems. ER diagrams are readily translated into a database implementation.

There really is no "standard ER"; all practical implementations, and there are many of them, extend ER in various ways. For instance, [Teorey-86] discusses the LRDM approach. LRDM is a useful extension to ER and supports basic concepts such as inheritance, association, and entities that are described by attributes. [Shlaer-88a] describes another enhancement to ER.

OMT object modeling is also an enhanced form of ER. We add several new concepts, such as qualification, and a methodology for programming and database design. Figure 12.1 and Figure 12.2 compare the readability and expressability of ER and OMT notations for the same problem. Our presentation of ER syntax was taken from [Ullman-88] and [Chen-76].

In these figures, a person has a name, address, and social security number. A person may charge time to projects and earn a salary. A company has a name, address, phone number, and primary product. A company hires and fires persons. *Person* and *Company* have a many-to-many relationship. Job title depends on both person and company.

There are two types of persons: workers and managers. Each worker works on many projects; each manager is responsible for many projects. A project is staffed by many workers and exactly one manager. Each project has a name, budget, and internal priority for securing resources.

A company is composed of multiple departments; each department within a company is uniquely identified by its name. A department usually, but not always, has a manager. Most managers manage a department; a few managers are not assigned to any department. Each department manufactures many products; while each product is made by exactly one department. A product has a name, cost, and weight.

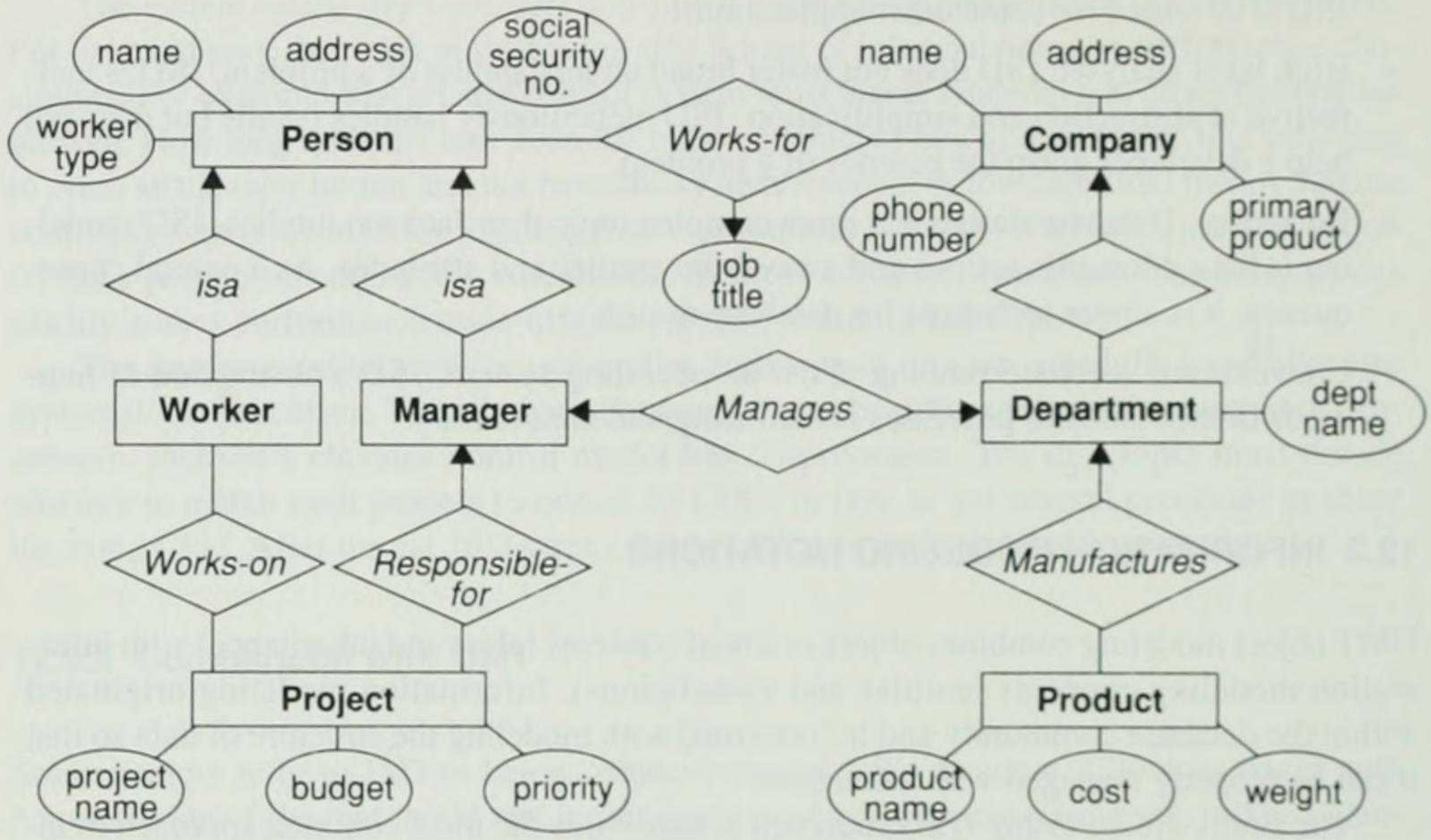


Figure 12.1 ER model

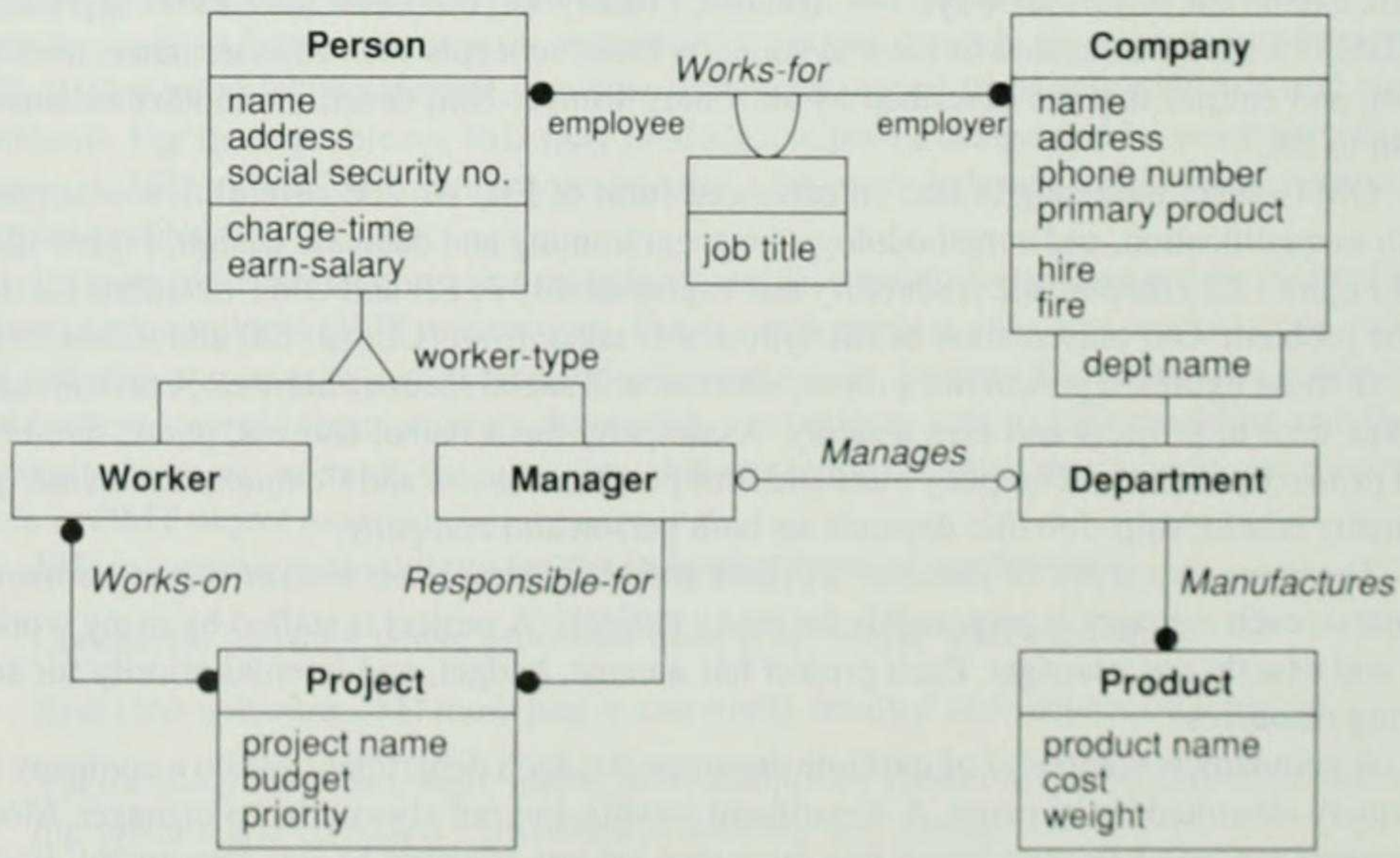


Figure 12.2 OMT object model

12.4 OBJECT-ORIENTED WORK

Now we compare the OMT with other object-oriented approaches. Our comparison is necessarily limited because little has been published on object-oriented methodologies for software engineering. Our approach is compatible with the few publications we have encountered to date. To a large extent, our work synthesizes several different camps of thought: databases, object-oriented concepts, and software engineering. We consider our work to be a consolidation of past efforts with some incremental improvements.

[Booch-86] describes the rudiments of object-oriented software development. He explains that object-oriented development is fundamentally different from traditional functional approaches to design, such as those based on data flow. Object-oriented software decomposition more closely models a person's perception of reality, while functional decomposition is only achieved through a transformation of the problem space. Thus it is not surprising that software developed with an object-oriented approach is more understandable, extensible, and maintainable.

[Booch-91] extends previous Ada-oriented work to the entire object-oriented design area. His excellent discussion of inheritance and classification are particularly worth reading. Booch's methodology includes a variety of models that address the object, dynamic, and functional aspects of a software system. We would say that his book places less emphasis on analysis and more emphasis on design than we do. A major distinction between Booch's approach and the OMT approach is the emphasis we place on associations. Booch mentions associations in referring to our past work but has not truly incorporated them into his methodology. The similarities between the approaches are more striking than the differences, and both approaches complement one another.

[Meyer-88] is really not a methodology (and does not claim to be) but has many tips on good design. Meyer's book is heavily oriented towards using language as a vehicle for expressing a design; we use graphics. Meyer does not deal with conceptual modeling or analysis.

[Shlaer-88b] describes a complete methodology for object-oriented analysis which is similar to ours. Shlaer and Mellor's methodology, like our OMT methodology, breaks analysis down into three phases: static modeling of objects, dynamic modeling of states and events, and functional modeling. All in all, we think that their methodology is quite good. A flaw with Shlaer and Mellor's treatment is their excessive preoccupation with relational database tables and database keys.

Shlaer and Mellor only claim their methodology as an approach to analysis, and caution that the final design might be different. We have tried to show how the object-oriented paradigm can permeate the entire software development process—from analysis to design to implementation.

Coad and Yourdon [Coad-90] present an approach to object-oriented analysis that is similar to the original OMT approach taken by us [Loomis-87] and by Shlaer and Mellor. They touch briefly on design.

Jacobsen [Jacobsen-87] claims to have a full object-oriented development methodology, but only limited details of his methodology have been published; the rest is available through

a commercial course. He analyzes a system in terms of entities (an object model) and *use cases* (prototypical scenarios covering dynamic behavior). For implementation, functionality is grouped into *services*, groups of related functional requirements. Design consists of constructing a system architecture in terms of modular *blocks*.

All of the object-oriented methodologies, including ours, have much in common, and should be contrasted more with non-object-oriented methodologies than with each other.

12.5 CHAPTER SUMMARY

Several popular software engineering approaches are based on the notion of data flow. The Structured Analysis/Structured Design (SA/SD) methodology is representative of the data flow approach. SA/SD begins with a single process or function that represents the overall purpose of the desired software. SA/SD recursively divides complex processes, until one is left with many small functions that are easy to implement.

SA/SD and OMT modeling have much in common. Both methodologies support the three orthogonal views of a system—the object, dynamic, and functional models. The difference is that SA/SD emphasizes the functional model while OMT emphasizes the object model. We believe that for most problems an object-oriented approach is superior to a data flow approach. An object-oriented design is more extensible, provides better traceability, and better integrates database and programming code.

Michael Jackson advocates a different approach to system development, called JSD. A JSD model begins with consideration of the real world. One culls out the most important entities and actions in the real world, from the perspective of the application. The remaining JSD steps develop detailed pseudocode that precisely specify desired software behavior and their correspondence to real-world actions.

We regard JSD as a valuable approach, as are SA/SD and OMT. Each methodology has its niche where it clearly excels. JSD is an excellent methodology for real-time and microcode applications. We consider JSD a poor approach for high-level analysis and database design.

We have compared OMT object modeling to the entity-relationship (ER) information modeling notation. In essence, OMT object modeling is an enhanced form of ER. OMT object modeling improves on ER in the areas of expressiveness and readability.

The OMT methodology builds on earlier object-oriented work and benefits from insights that have come with experience.

entity-relationship (ER) diagram information modeling notations Jackson Structured Development (JSD) Object Modeling Technique (OMT) other approaches to object-oriented development Structured Analysis/Structured Design (SA/SD)

Figure 12.3 Key concepts for Chapter 12

REFERENCES

[Booch-86] Grady Booch. Object-oriented development. *IEEE Transactions on Software Engineering 12*, 2 (Feb.1986), 211-221.

[Booch-91] Grady Booch. *Object-Oriented Design*. Redwood City, Calif.: Benjamin/Cummings, 1991.

[Cameron-89] John Cameron. *JSP & JSD: The Jackson Approach to Software Development*. Washington, DC: IEEE Computer Society Press, 1989.

[Chen-76] P.P.S. Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database System 1*, 1 (March 1976).

[Coad-90] Peter Coad, Edward Yourdon. *Object-Oriented Analysis*. Englewood Cliffs, New Jersey: Yourdon Press, 1990.

[Jackson-83] Michael A. Jackson. *System Development*. Englewood Cliffs, New Jersey: Prentice Hall International, 1983.

[Jacobsen-87] Ivar Jacobsen. Object oriented development in an industrial environment. *OOPSLA'87* as *ACM SIGPLAN 22*, 12 (December 1987), 183-191.

[Loomis-87] Mary E.S. Loomis, Ashwin V. Shah, James E. Rumbaugh. An object modeling technique for conceptual design. *European Conference on Object-Oriented Programming*, Paris, France, June 15-17, 1987, published as *Lecture Notes in Computer Science, 276*, Springer-Verlag.

[Meyer-88] Bertrand Meyer. *Object-Oriented Software Construction*. Hertfordshire, England: Prentice Hall International, 1988.

[Shlaer-88a] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, New Jersey: Yourdon Press, 1988.

[Shlaer-88b] Sally Shlaer, Stephen J. Mellor, Deborah Ohlsen, and Wayne Hywari. The object-oriented method for analysis. *Proceedings of the Tenth Structured Development Forum*, 1988.

[Teorey-86] Toby J. Teorey, Dongqing Yang, James P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *Computing Surveys 18*, 2 (June 1986), 197-222.

[Ullman-88] Jeffrey Ullman. *Principles of Database and Knowledge-Base Systems, Volumes 1 and 2*. Rockville, Maryland: Computer Science Press, 1988.

[Yourdon-79] Edward Yourdon, Larry L. Constantine. *Structured Design*. Englewood Cliffs, New Jersey: Yourdon Press, 1979.

[Yourdon-89] Edward Yourdon. *Modern Structured Analysis*. Englewood Cliffs, New Jersey: Yourdon Press, 1989.

EXERCISES

12.1 (6) You are designing a portable tester for integrated circuits. The tester will have several different types of sockets. An integrated circuit will be tested by placing it in the socket that matches its pin configuration and identifying the type of circuit. The tester will then run through a series of tests, applying power and signals to the appropriate pins and measuring the response of the circuit. A portion of the object diagram for the tester is shown in Figure E12.1. There is a doubly qualified association between test case and socket. Each test case applies several named signals to pins of a socket. The same signal may be applied to more than one pin. Each pin may receive signals from several test cases. Prepare a corresponding ER diagram; preserve as much semantic content as you can.

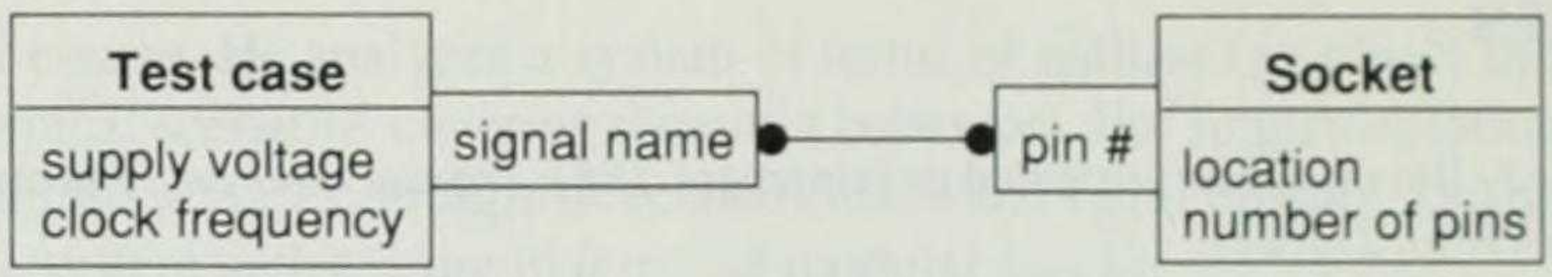


Figure E12.1 Portion of an object diagram for an integrated circuit tester

12.2 (3) Figure E12.2 is an object diagram for a portion of a simplified simulator for training glider pilots. The simulator is for one glider with wings and rudder only. Effects of the wind and forces generated by the body of the glider are neglected. Details of the user interface to the simulator are outside the scope of this exercise. A glider has several associated lifting surfaces, in this case two wings and a rudder. The wings provide lift and the rudder is used to steer. Methods would be provided to perform simulation. For instance, the force on each surface would be calculated from its attribute values, and the orientation, velocity, and rotational rate of the glider. The force on the rudder would also depend on its deflection. The translational acceleration would be computed by retrieving the results of force calculations and masses of associated surfaces. The accelerations would be numerically integrated to update position, orientation, velocity, and rotational rate.

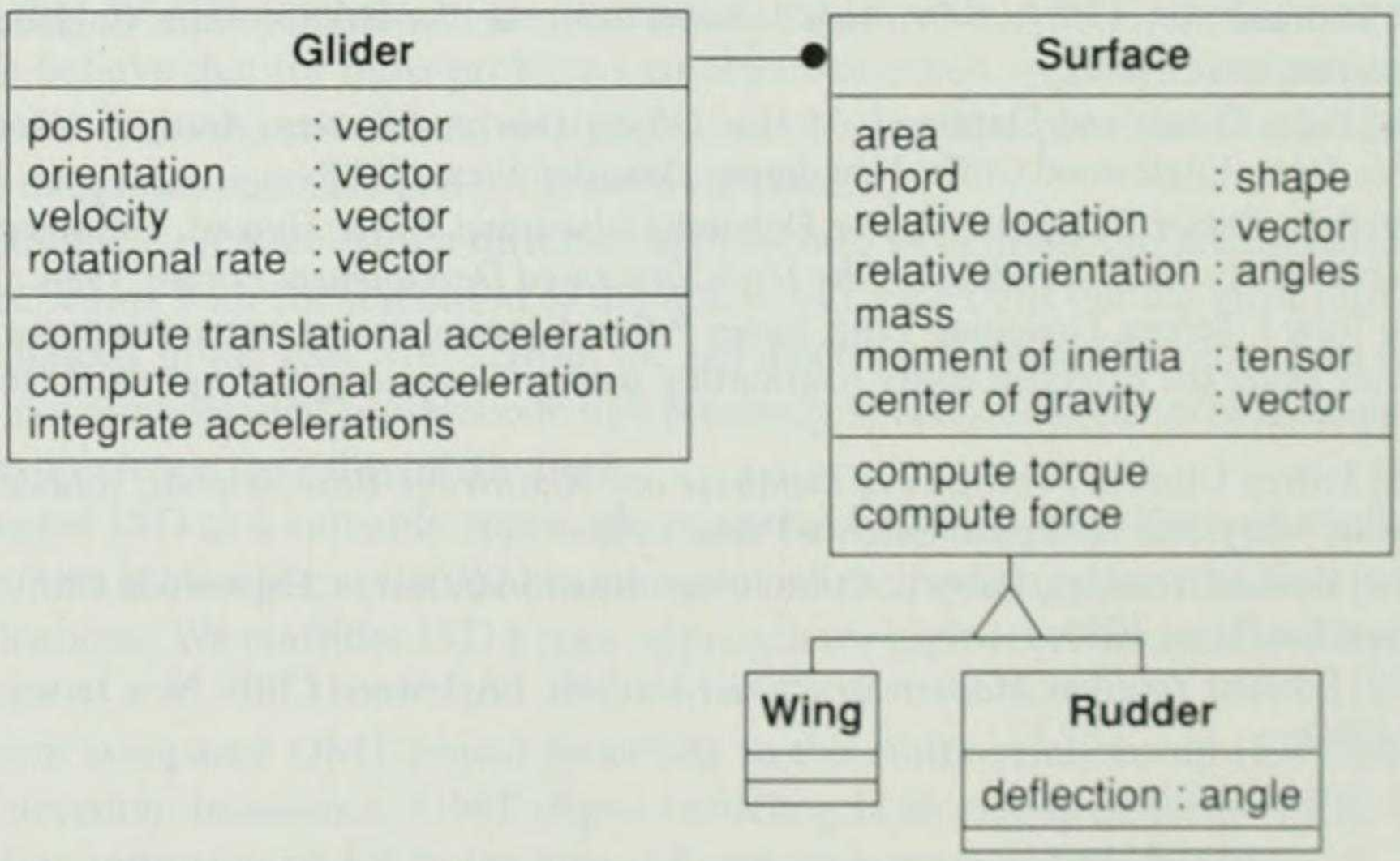


Figure E12.2 Object diagram of a portion of a glider simulator

Figure E12.3 is the corresponding data flow diagram for the glider. Position, orientation, velocity, and rotation vectors are data stores which serve as state variables for the simulator.

a. Modify both diagrams to add an elevator to the tail of the glider to provide additional control and to add another small wing on the nose of the glider to provide additional lift and stability.
b. How would each diagram be modified to simulate the behavior of the glider after any combination of surfaces falls off, including elevator, rudder, left main wing, right main wing, left nose wing, and right nose wing. (There are 63 combinations.)

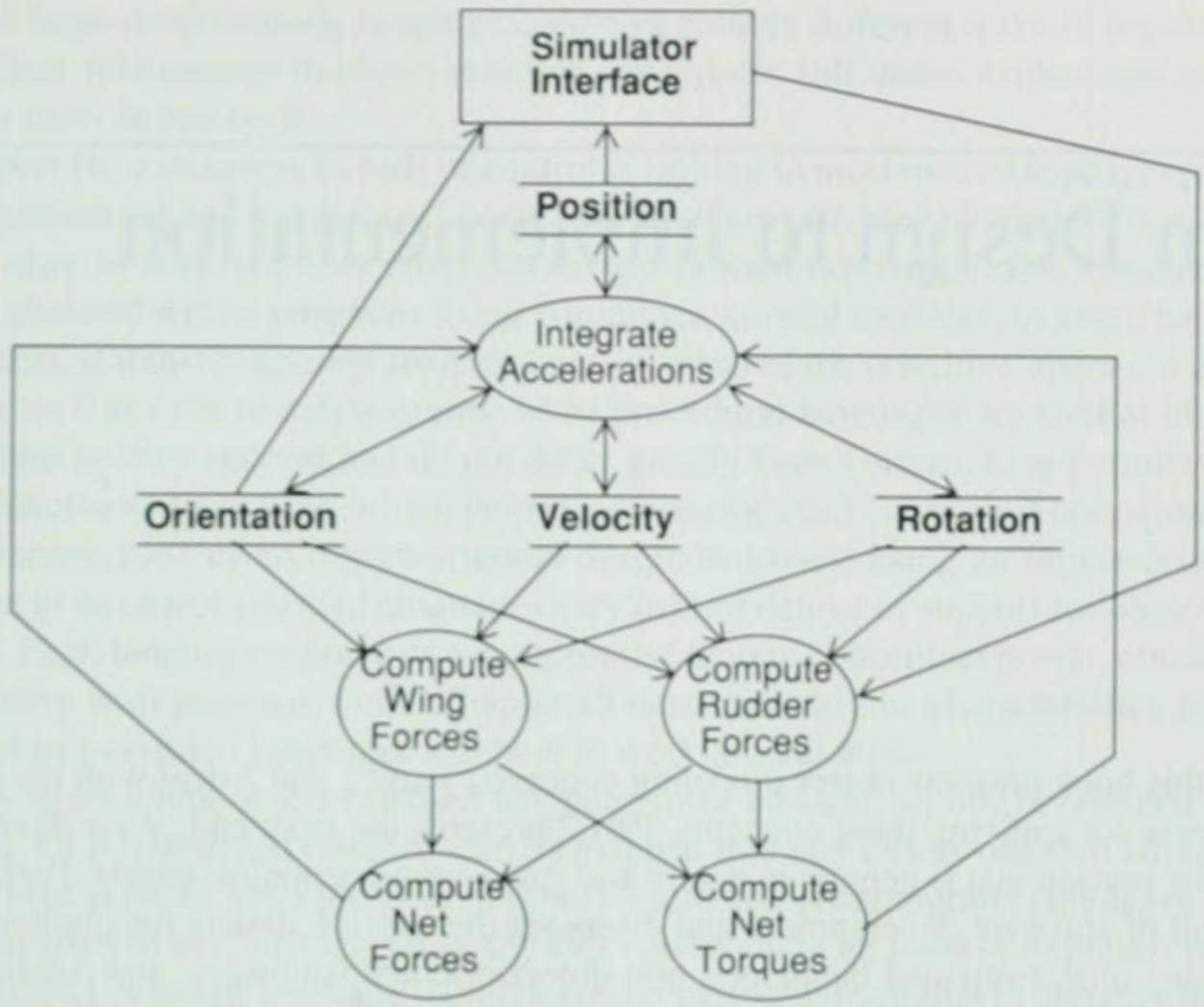


Figure E12.3 Data flow diagram for a portion of a glider simulator

12.3 (Project) Consult a few of the references at the end of the chapter and compare and contrast the way inheritance, methods, and ternary relationships are handled by Shlaer and Mellor, Booch, Teorey, and OMT.

13

From Design to Implementation

Part 1 of this book presents object modeling concepts; Parts 2 and 3 deal with the development process for applying these concepts. Part 2 presents the front end of the development process, the portion that is generic in nature and spans implementation targets. Part 3 covers the tail end of software development and discusses the specific details for implementing a system using object-oriented languages, non-object-oriented languages, and database management systems.

Writing code is an extension of the design process. Writing code should be straightforward, almost mechanical, because all the difficult decisions should already have been made during design. The code should be a simple translation of the design decisions into the peculiarities of a particular language. Decisions do have to be made while writing code, but each one should affect only a small part of the program so they can be changed easily. Nevertheless, the program code is the ultimate embodiment of the solution to the problem, so the way in which it is written is important for maintainability and extensibility.

13.1 IMPLEMENTATION USING A PROGRAMMING LANGUAGE

Most executable languages are capable of expressing the three aspects of software specification: data structure, dynamic flow of control, and functional transformation.

Data structure is expressed in a declarative (nonprocedural) subset of a language. The statements that are used to declare data structures are sometimes mixed in with the procedural statements but are not executable. In some languages, such as Ada, a sharp distinction is made between external specification, which is purely declarative, and internal specification, which is often combined with procedural statements.

Flow of control may be expressed either procedurally (conditionals, loops, and calls) or nonprocedurally (rules, constraints, tables, and state machines). Traditional languages are purely procedural, although the programmer can implement nonprocedural constructs as data. Nonprocedural languages, such as rule-based systems, constraint maintenance sys-

tems, and logic programming languages, support entirely different ways of organizing programs. Their relationship to object-oriented concepts is still under exploration, and we do not cover them in this book.

Support for concurrent threads of control is lacking in most major languages (except for Ada). Multitasking and interprocess communication are provided by modern operating systems but must be accessed from programs using awkward subroutine calls. Concurrency can also be simulated within programs using coroutines, control modules, or event handlers.

Functional transformations are expressed in terms of the primitive operators of the language, as well as calls to subprograms. Most procedural languages are similar in the kinds of functionality they support and do not differ greatly from Fortran. Lisp permits constructions of functions at run time, which permits interesting (and confusing) operations.

Implementation of an object-oriented design is easiest using an object-oriented language, but even object-oriented languages vary in their degree of support for object-oriented concepts. Each language represents a compromise among conceptual power, efficiency, and compatibility with previous work. Chapter 15 describes various characteristics and limitations of object-oriented languages and how to work around them.

Even when a non-object-oriented language must be used, an object-oriented design is beneficial. Object-oriented concepts can be mapped into non-object-oriented language constructs. There is really not a problem of power—after all, programming languages are eventually converted to machine language anyway—as much as an issue of expressiveness. Use of a non-object-oriented language requires greater care and discipline to preserve the object-oriented structure of the program, and the programmer cannot obtain help from the language in finding violations. Chapter 16 describes implementation of object-oriented designs using C, Ada, and Fortran.

Any language, object-oriented or non-object-oriented, is a tool that can be used well or poorly. Object-oriented programming can greatly improve the expressiveness of programs, but it can also increase the opportunity for obfuscation by careless programmers. As with any craft, good programming requires discipline and adherence to principles of good style. Chapter 14 describes style guidelines that promote the effective use of object-oriented techniques.

13.2 IMPLEMENTATION USING A DATABASE SYSTEM

When the main concern is access to persistent data, rather than the operations on the data, a database is often the appropriate form of implementation. The main focus of a database is the structure and constraints on the data. Database commands typically operate on sets of data from the database, incorporating a high degree of parallelism, while most conventional languages are highly serial. Database operations are much less procedural than conventional programming language statements, although they are more procedural than a rule-based system. Databases provide concurrent operations on data by different users as part of their fundamental structure.

We discuss implementation of database applications using existing relational database management systems (RDBMS) in Chapter 17. Most RDBMS provide separate languages for data declarations and operations, analogous to the distinction in programming languages.

The data definition language is used to declare the structure of the data. A query language may be provided which allows the expression of functional transformations (record locking and update) and flow of control. The query language may be limited to simple record access based on fields that act as keys, or it might offer more of the features of a general purpose programming language.

Some recent object-oriented database systems attempt to integrate an object-oriented language with a database in a single seamless package. Operations may be defined for each class of object, but the programmer need not explicitly read and write information to persistent storage. Although object-oriented database systems promise better performance and easier use in the long run, they are not yet as mature as conventional relational database systems and may also pose problems integrating with existing conventional applications. Integrated object-oriented languages and databases are discussed as part of Chapter 15.

13.3 IMPLEMENTATION OUTSIDE A COMPUTER

The specification and design techniques described in this book are useful for implementation targets besides programming languages and databases. Often the design of a computer application involves an implementation that will consist of a mixture of software and some other structure. Typical domains include design of hardware, design of a knowledge base, and modeling business enterprises. Object modeling techniques can be useful in capturing structural, dynamic, and functional relationships in such domains in a way that can be communicated to domain experts. It is beyond the scope of this book to explore implementation in these other domains.

13.4 OVERVIEW OF PART 3

Part 3 of the book discusses implementation of object-oriented designs in various target languages:

- Chapter 14 contains style guidelines for writing programs in object-oriented and non-object-oriented languages in a proper object-oriented spirit, maximizing the possibility of readability, reusability, and extensibility.
- Chapter 15 describes implementation using object-oriented languages having varying degrees of support for object-oriented concepts. It includes a survey of several commercially available languages. It also discusses integrated object-oriented languages and databases.
- Chapter 16 shows how to implement object-oriented designs in a non-object-oriented language and addresses the limitations of such an implementation. Various object-oriented concepts are mapped into C, Ada, and Fortran code.
- Chapter 17 describes implementation of object-oriented designs using existing relational database management systems, including alternate ways of mapping various object-oriented constructs and ways to optimize performance.

14

Programming Style

As any chess player, cook, or skier can attest, there is a great difference between knowing something and doing it well. Writing object-oriented programs is no different. It is not enough to know the basic constructs and to be able to assemble them together into programs. The experienced programmer follows principles to make readable programs that live beyond the immediate need. These principles include general design principles, programming idioms ("how to do common tasks with the tools at hand"), rules-of-thumb, tricks of the trade, and cautionary advice. Good style is important in all programming, but it is even more important in object-oriented design and programming because much of the benefit of the object-oriented approach is predicated on producing reusable, extensible, understandable programs.

14.1 OBJECT-ORIENTED STYLE

Good programs do more than simply satisfy their functional requirements. Programs that follow proper design guidelines are more likely to be correct, reusable, extensible, and quickly debugged. Most style guidelines that are intended for conventional programs also apply to object-oriented programs. In addition, facilities such as inheritance are peculiar to object-oriented languages and require new guidelines. We present object-oriented style guidelines under the following categories, although many guidelines contribute to more than one category:

- Reusability
- Extensibility
- Robustness
- Programming-in-the-large

14.2 REUSABILITY

Reusable software reduces design, coding, and testing cost by amortizing effort over several designs. Reducing the amount of code also simplifies understanding, which increases the likelihood that the code is correct. Reuse is possible in conventional languages, but object-oriented languages greatly enhance the possibility of code reuse.

14.2.1 Kinds of Reusability

There are two kinds of reuse: sharing of newly-written code within a project and reuse of previously-written code on new projects. Similar guidelines apply to both kinds of reuse. Sharing of code within a project is a matter of discovering redundant code sequences in the design and using programming language facilities, such as procedures or methods, to share their implementation. This kind of code sharing almost always pays off immediately by producing smaller programs, faster debugging, and faster iteration of the design.

Planning for future reuse takes more foresight and represents an investment. It is unlikely that a class in isolation will be used for multiple projects. Programmers are more likely to reuse carefully thought out subsystems, such as abstract data types, graphics packages, and numerical analysis libraries.

14.2.2 Style Rules for Reusability

Keep methods coherent. A method is coherent if it performs a single function or a group of closely related functions. If it does two or more unrelated things, break it apart into smaller methods.

Keep methods small. If a method is large, break it into smaller methods. A method that exceeds one or two pages is probably too large. By breaking a method into smaller parts, you may be able to reuse some parts even when the entire method is not reusable.

Keep methods consistent. Similar methods should use the same names, conditions, argument order, data types, return value, and error conditions. Maintain parallel structure when possible. The Unix operating system offers many examples of inconsistent functions. For example, in the C library, there are two inconsistent functions to output strings, *puts* and *fputs*. The *puts* function writes a string to the standard output, followed by a newline character; *fputs* writes a string to a specified file, without a newline character. Avoid this kind of inconsistency.

Separate policy and implementation. Policy methods make decisions, shuffle arguments, and gather global context. Policy methods switch control among implementation methods. Policy methods should check for status and errors; they should not directly perform calculations or implement complex algorithms. Policy methods are often highly application-dependent, but they are simple to write and easy to understand. Policy methods are the "mortar."

Implementation methods perform specific detailed operations, without deciding whether or why to do them. If implementation methods can encounter errors, they should only return status, not take action. Implementation methods perform specific computations on fully-

specified arguments and often contain complicated algorithms. Implementation methods do not access global context, make decisions, contain defaults, or switch flow of control. Because implementation methods are self-contained algorithms, they are likely to be meaningful and reusable in other contexts. Implementation methods are the "bricks."

Do not combine policy and implementation in a single method. Isolate the core of the algorithm into a distinct, fully-specified implementation method. This requires abstracting out the particular parameters of the policy method as arguments in a call to the implementation method.

For example, a method to scale a window by a factor of 2 is a policy method. It should set the target scale factor for the window and call on an implementation method that scales the window by an arbitrary scale factor. Later if you decide to change the default scale factor to another value, such as 1.5, you just have to modify the parameter in the policy method, without changing the implementation method which actually does the work.

Provide uniform coverage. If input conditions can occur in various combinations, write methods for all combinations, not just the ones that you currently need. For example, if you write a method to get the last element of a list, also write one to get the first element.

Broaden the method as much as possible. Try to generalize argument types, preconditions and constraints, assumptions about how the method works, and the context in which the method operates. Take meaningful actions on empty values, extreme values, and out-of-bounds values. Often a method can be made more general with a slight increase in code.

Avoid global information. Minimize external references. Referring to a global object imposes required context on the use of a method. Often the information can be passed in as an argument. Otherwise store global information as part of the target object so that other methods can access it uniformly.

Avoid modes. Functions that drastically change behavior depending on current context are hard to reuse. Try to replace them with modeless functions. For example, a text processing application requires insert and replace operations. One approach is to set a mode to *insert* or *replace*, then use a *write* operation to insert or replace text depending on the current mode. A modeless approach uses two operations, *insert* and *replace*, that do the same operations without a mode setting. The danger of modes is that an object left in a mode in one part of an application can affect an operation applied later in the application.

14.2.3 Using Inheritance

The preceding guidelines improve the chance of inheriting shared code. Sometimes, however, methods on different classes are similar but not similar enough to represent with a single inherited method. There are several techniques of breaking up methods to inherit some code.

Subroutines. The simplest approach is to factor out the *common* code into a single method that is called by each method. The common method can be assigned to an ancestor class. This is effectively a subroutine call and is shown in Figure 14.1.

Factoring. In some cases the best way to increase code reuse between similar classes is to factor out the differences between the methods of different classes, leaving the remainder of

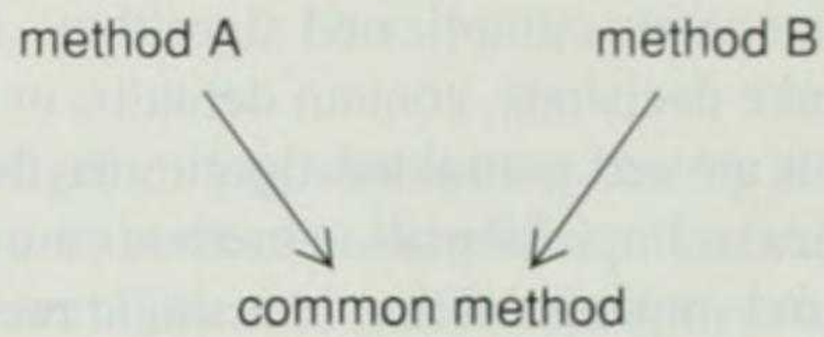


Figure 14.1 Code reuse via subroutines

the code as a shared method. This approach is effective when the differences between methods are small and the similarities are great. As shown in Figure 14.2, the common portion of two methods is made into a new method. The new method calls an operation that is implemented by a different method containing the code differences in each subclass. Sometimes an abstract class must be added to hold the top-level method. This approach makes it easier to add new subclasses because only the difference code need be written.

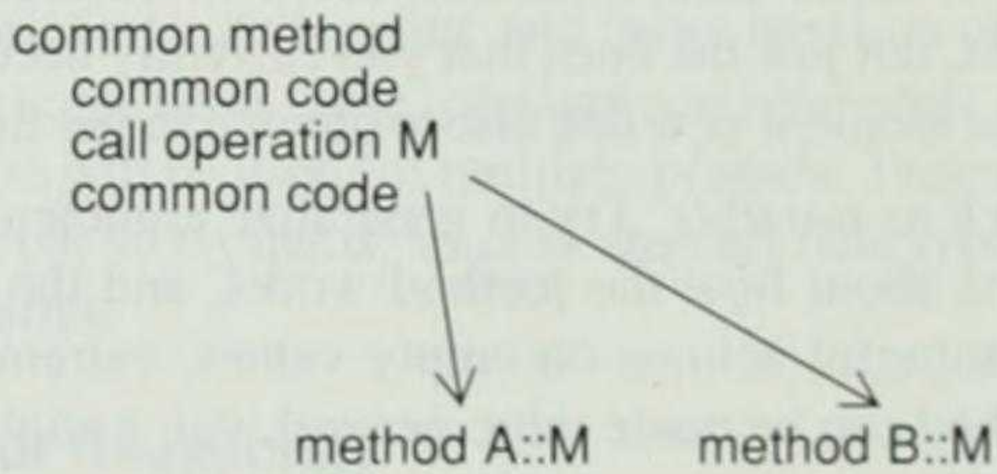


Figure 14.2 Code reuse via factoring

A package for plotting numerical data provides a good illustration of factoring. *DataGraph* is an abstract class that organizes common data and operations for its subclasses. One of *DataGraph*'s methods is *draw*, consisting of the following steps:

1) Draw border
2) Scale data
3) Draw axes
4) Plot data
5) Draw title
6) Draw legend

Subclasses of *DataGraph*, such as *LineGraph*, *BarGraph*, and *ScatterGraph*, draw borders, titles, and legends the same way but differ in the way they scale data, draw axes, and plot data. Each subclass inherits the methods *drawBorder*, *drawTitle*, and *drawLegend* from abstract class *DataGraph*, but each subclass defines its own methods for *scaleData*, *drawAxes*, and *plotData*. The method *draw* need only be defined once, on class *DataGraph*, and inherited by each subclass. Each time the draw method is invoked, it applies *drawBorder*, *drawTitle*, and *drawLegend* inherited from the superclass and *scaleData*, *drawAxis*, and *plotData* supplied by the subclass for the object. To add a new subclass, only the three specialized methods need be written.

Delegation. Sometimes it appears that use of inheritance would increase code reuse within a program, when a true superclass/subclass relationship does not exist. Do not give in to the

temptation to use this *implementation inheritance*; use delegation instead. Inheritance should only be used when the generalization relationship is semantically valid. Inheritance means that each instance of a subclass truly is an instance of the superclass; thus all operations and attributes of the superclass must uniformly apply to the subclass. Improper use of inheritance leads to programs that are hard to maintain and extend. Object-oriented languages are permissive in their use of inheritance and will not enforce the good programming practice that we recommend.

Delegation provides a proper mechanism to achieve the desired code reuse. The operation is caught in the desired class and forwarded to another class for actual execution. Since each operation must be explicitly forwarded, unexpected side effects are less likely to occur. The names of operations in the catching class may differ from those in the class supplying the operations. Each class should choose names which are most appropriate for its purposes.

Encapsulate external code. Often you will want to reuse code that may have been developed for an application with different interfacing conventions. Rather than inserting a direct call to the external code, it is safer to encapsulate its behavior within an operation or a class. This way the external routine or package can be changed or replaced, and you will only have to change your code in one place.

For example, you may have a numerical analysis application but, knowing that reliable matrix inversion software already exists, you do not want to reimplement the algorithm in your object-oriented language. A matrix class might be written to encapsulate the functionality provided by the external subroutine package. The matrix class would have, for example, an inverse operation that takes the tolerance-for-singularity as an argument and returns a new matrix that is the inverse of the operation's target.

14.3 EXTENSIBILITY

Most software is extended in ways that its original developers may not expect. The guidelines for reusability enhance extensibility as well. In addition, the following object-oriented principles enhance extensibility.

Encapsulate classes. A class is encapsulated if its internal structure is hidden from other classes. Only methods on the class should access its implementation. Many compilers are smart enough to optimize operations into direct access to the implementation, but the programmer should not. Respect the information in other classes by never reaching inside the class for data.

Hide data structures. Do not export data structures from a method. Internal data structures are specific to a method's algorithm. If you export them, you limit flexibility to change the algorithm later.

Avoid traversing multiple links or methods. A method should have limited knowledge of an object model. A method must be able to traverse links to obtain its neighbors and must be able to call operations on them, but it should not traverse a second link from the neighbor to a third class because the second link is not directly visible to it. Instead, call an operation on the neighbor object to traverse the operation; if the association network changes, the opera-

tion method can be rewritten without changing the call. Similarly, avoid applying a second operation to the result of an operation call unless the class of the result is already known as an attribute, argument, or neighbor, or the result class is from a lower-level library. Instead, write a new operation on the original target class to perform the combined operation itself. The principles in this paragraph were proposed in [Lieberherr-89] as the "Law of Demeter."

Avoid case statements on object type. Use methods instead. Case statements can be used to test internal attributes of an object but should not be used to select behavior based on object type. Dispatching operations based on object type is the whole point of methods, so don't circumvent them.

Distinguish public and private operations. Public operations are visible outside a class and have published interfaces. Once a public operation is used by other classes, it is costly to change its interface, so public operations should be carefully defined. Private operations are internal to a class and are used to help implement the public operations. Private operations can be deleted or their interfaces can be changed to modify the implementation of the class with impact limited to other methods on the class.

Why classify operations as public and private?

- There is no need to bother the user of a class with internal details. Private methods just confuse the external user of the class.
- Since private methods depend on internal implementation decisions, the method designer may change the number and types of arguments if the implementation changes.
- Private methods may rely on preconditions or state information created by other methods in the class. Applied out of context, a private operation may calculate incorrect results or cause the object to fail.
- Private methods add modularity. Internal details of the method only affect methods on the class, not other methods.

Similarly, attributes and associations should be classified as public or private. In addition, public attributes and associations may be classified as read-only or writable outside the owner class.

14.4 ROBUSTNESS

You should strive for efficiency in writing methods but not at the expense of robustness. A method is robust if it does not fail even if it receives improper parameters. Robustness against internal bugs may be traded off against efficiency. Robustness against user errors should never be sacrificed.

Protect against errors. Software should protect itself against incorrect user input. Incorrect user input should never cause a crash. Any method that accepts user input must validate input that could cause trouble.

The method designer must consider two kinds of error conditions. Application (user) errors are identified during analysis and report on conditions that exist in the problem domain. For example, an automatic teller machine application should report or process errors about the ATM card scanner and communications lines. The response to these errors is part of the analysis. On the other hand, low-level system errors concern programming aspects of a method. These low-level errors include operating system errors, such as memory allocation errors or file input/output errors, and hardware faults. Your program should check for these errors and at least try to die gracefully if nothing else is possible.

Try to guard against programming bugs as well as possible, and give good diagnostic information even if fatal bugs occur. During development, it is often worthwhile to insert internal assertions into the code to uncover bugs, even though the checks will be removed for efficiency in the production version. A strongly-typed object-oriented language provides greater protection against type mismatches, but assertions can be manually inserted in any language.

Optimize after the program runs. Don't optimize a program until you get it working. Often programmers spend too much effort trying to improve portions of code that are infrequently executed. Measure the performance within the program before optimizing it; you may be surprised to find that most parts consume little of the total time. Study your application to learn what measures are important, such as worst case times and operation frequencies. If an operation may be implemented in more than one way, assess the trade-offs of the alternatives as they relate to memory, speed, and simplicity of implementation. In general, avoid optimizing more of the program than you have to, as optimization compromises extensibility, reusability, and understandability. If methods are properly encapsulated, they can be replaced with optimized versions without affecting the rest of the program.

Validate arguments. External operations, those available to users of the class, must rigorously check their arguments to prevent failure. But internal methods may assume their arguments are valid for efficiency reasons. Public methods must take more care to check the validity of arguments because external users are more likely to violate restrictions on arguments. Internal, or private, methods can often assume preconditions since the implementor has tighter control and can rely on the public methods that call them for error checking.

Don't include arguments that cannot be validated. For example, the infamous *scanf* function in Unix reads a line of input into an internal buffer without checking the size of the buffer. This loophole has been exploited to write virus programs that force a buffer overflow in system software that did not validate its arguments. Don't write or use operations whose arguments can't be validated.

Avoid predefined limits. When possible use dynamic memory allocation to create data structures that do not have predefined limits. During design, it is difficult to predict the maximum capacity expected of data structures in an application, so don't set any limits. The day of fixed limits on symbol table entries, user names, file names, compiler entries, and other things should be long over. Most object-oriented languages have excellent dynamic memory allocation facilities.

Instrument the program for debugging and performance monitoring. Just as a hardware circuit designer instruments an IC board with test points, you should instrument your code for debugging, statistics, and performance. The level of debugging that you must build into your code depends on the programming environment presented by the language. In Smalltalk, for example, a class browser allows a developer to explore the class hierarchy, displaying attributes and the code for methods. An inspector lets the user interrupt execution and print the internal state of an instance. If your implementation language does not have similar capabilities, then you can provide print methods for each class. These methods can be accessed from your system debugger. Also, you can add debug statements to the methods. These debug statements are conditionally executed depending on an instance variable that contains the debug level. You can print a message on entry or exit and selectively print input or output values.

Adding code to gather statistics will help you understand the behavior of your classes. Some operating systems, such as Unix and VMS, offer tools to create execution profiles of an application. Typically, these tools report the number of times each method was called and the amount of processor time spent in each method. If your system does not have comparable tools, you can instrument your code for gathering statistics much like for debugging.

14.5 PROGRAMMING-IN-THE-LARGE

Programming-in-the-large refers to writing large, complex programs with teams of programmers. Human communication becomes paramount on such projects and requires proper software engineering practices. The following guidelines should be observed:

Do not prematurely begin programming. It is important to first complete the generic thought process before confronting the quirks of an implementation target. All software development methodologies emphasize the importance of first designing, then coding.

Keep methods understandable. A method is understandable if someone other than the creator of the method can understand the code (as well as the creator after a time lapse). Keeping methods small and coherent helps to accomplish this.

Make methods readable. Meaningful variable names increase readability. Typing a few extra characters is cheaper than the misunderstanding that can come later when another programmer tries to decipher your variable names. Check the readability of your methods by running them through a spelling checker. Avoid abbreviations that may confuse other programmers. Use temporary variables instead of deeply nested expressions. Do not use the same temporary variable for two different purposes within a method, even if their usage does not overlap; stack space is cheap anyway.

Use exactly the same names as in the object model. The choice of names used within a program should exactly match those found in the object model. A program may need to introduce additional names for implementation reasons, and this is fine, but the uniformity of

names where they carry forward should be preserved. This practice improves traceability, documentation, and understandability for the software as a whole.

Choose names carefully. Make sure your names accurately describe the operations, classes, and attributes they label. Follow a uniform pattern in making up names. For example, you might use the pattern *"verb_object"* in making up operation names, such as *add_element* or *draw_highlight*. Be sure to define the verbs being used frequently (for example, you may want to distinguish between *copy* and *deepcopy* or between *new* and *create*). Many object-oriented languages automatically build method names from the class name and operation.

Do not use the same operation name for semantically different operations. All classes that use the same name should have the same origin class and the same signature (number and types of arguments).

Good	Bad
Circle::Area	Matrix::Invert (performs matrix inversion)
Rectangle::Area	Figure::Invert (turns figure upside down)

Use programming guidelines. Project teams should use programming guidelines available in their organizations. If guidelines do not exist, the software team should create guidelines that address issues such as the form of variable names, indentation style for control structures, method documentation headers, and in-line documentation.

Package into modules. Group classes with similar functions into a module.

Module	Classes
plotting	line plot, bar chart, pie chart
geometry	polygon, circle
windows	menus, buttons, toggles, panels
animation	scenes, cues, key frames

Document classes and methods. The documentation of a method describes its purpose, function, context, inputs, and outputs as well as any assumptions and preconditions about the state of the object. You should describe the algorithm, including why it was chosen. Internal comments within the method should describe major steps.

Publish the specification. The specification is a contract between the producer and the consumer of the class. Once a specification is written, the producer cannot break the contract, for doing so would affect the consumer. The specification only contains declarations. The user of a method should be able to use the method just by looking at the specification. Some languages, such as Ada and C++, support separation of specification and implementation. On-line descriptions of the class and its features help promote the correct use of the class. Each operation should also be documented by itself (apart from its method on each class), giving its origin, generic meaning, telling what each subclass must do to implement the operation, and what related methods are needed by the subclass. A partial sample specification follows.

Class Description
Class Name: Circle
Version: 1.0
Description: Ellipse whose major and minor axes are equal
Super Classes: Ellipse
Features:
Public Attributes:
center: Point — location of its center
radius: Real — its radius
Public Methods:
draw (Window) — draws a circle in the window
intersectLine (Line): Set of Points — finds the intersection of a line and a circle, returns set of 0-2 points
area (): Real — calculates area of circle
perimeter (): Real — calculates circumference of circle
Private Methods: none
Method Description
Method Circle::intersectLine (line: Line) : Set of Points
Description: Given a circle and a line, finds the intersection, returns a set of 0-2 intersection points. If the line is tangent to the circle, the set contains a single point.
Inputs:
self:Circle — circle to be intersected with line
line:Line — line to be intersected with circle
Returns:
A set of intersection points. Set may contain 0, 1, or 2 points
Side Effects: none
Errors: If the figures do not intersect, returns an empty set.
If the line is tangent to the circle, returns the tangent point.
If the circle's radius is 0, returns a single point if the point is on the line.
Operation Description
Operation intersectLine (line: Line) : Set of Points
Origin Class: GeometricFigure
Description: Returns a set of intersection points between the geometric object and the line. The set may contain 0, 1, or more points. Each tangent point only appears once. If the line is colinear with a line segment in the figure, only the two end points of the segment are included.
Status: Abstract operation in the origin class, must be overridden.
Inputs:
self: GeometricFigure — figure to be intersected with line
line:Line — line to be intersected with circle
Returns:
A set of intersection points. Set may contain 0 or more points.
Side Effects: none

Errors: If the figures do not intersect, returns an empty set.
If the line is colinear with a line segment in the figure, the set includes only the end points of the segment.
If the figure is an area, then its boundary is used.

14.6 CHAPTER SUMMARY

Good style is important to maximize the benefits of object-oriented design and programming; most benefits come from greatly reduced maintenance and enhancement costs and in reuse of the new code on future projects. Object-oriented programming style guidelines include conventional programming style guidelines as well as principles uniquely applicable to object-oriented concepts such as inheritance.

A major goal of object-oriented design is maximizing reusability of classes and methods. Reuse within a program or project is a matter of looking for similarities and consolidating them using inheritance. Planning for reuse by future projects takes more foresight because designing reusable software takes more time and effort up front. Reusability is enhanced by keeping methods small, coherent, and local. Separation of policy and implementation is important. One way to use inheritance is by factoring a generic method into suboperations, some of which are inherited from the origin class and some of which are overridden by each subclass. Delegation should be used when methods must be shared but classes are not in a true generalization relationship.

Most software is eventually extended. Extensibility is enhanced by encapsulation of classes and methods, minimizing dependencies between classes and methods, using methods to access attributes of other classes, and distinguishing public from private operations on a class.

Robustness should not be sacrificed for efficiency. Because objects contain references to their own classes, they are less vulnerable to mismatched typing than conventional programming variables and can be checked dynamically to see that they match the assumptions within a method. Programs should always protect against user and system errors. Testing assertions to catch programming bugs takes time but can be used during debugging and removed during production.

Writing large programs with teams of programmers requires more discipline, better documentation, and better communication than one-person or small projects. Writing readable, well-documented methods is essential.

delegation	public versus private
encapsulation	reusability
extensibility	robustness
factoring	specification of public interface
programming-in-the-large	

Figure 14.3 Key concepts for Chapter 14

BIBLIOGRAPHIC NOTES

In object-oriented programming, application concepts must be rendered into programming language constructs without obscuring either application meaning or program structure, so good programming style is important. Most conventional programming principles are also valid for object-oriented programming. [Kernighan-78] is a classic style guide for programming, a bit low-level perhaps for current languages, but useful nevertheless. [Dijkstra-76] stresses elegance in the design of demonstrably-correct programs.

REFERENCES

[Dijkstra-76] Edsger W. Dijkstra. *A Discipline of Programming*. Englewood Cliffs, New Jersey: Prentice Hall, 1976.

[Kernighan-78] Brian W. Kernighan, P.J. Plauger. *The Elements of Programming Style*. New York: McGraw-Hill, 1978.

[Lieberherr-89] Karl J. Lieberherr, Arthur J. Riel. Contributions to teaching object-oriented design and programming. *OOPSLA'89* as *ACM SIGPLAN 24*, 11 (Nov. 1989) 11-22.

EXERCISES

14.1 (4) A technique that can occasionally be used to reuse code is to use a method itself as an argument for another method. For example, one operation that can be performed on a binary tree is ordered printing. The subroutine *Print(node)* could print the values in a tree rooted at *node* by a recursive call to *Print(node.left_subtree)*, if there is a left subtree, followed by printing *node.value*, followed by a recursive call for the right subtree. This approach could be generalized to apply other operations. List at least three operations that could be performed on the nodes of a binary tree. Prepare pseudocode for a subroutine *Ordered_visit(node, method)* that applies *method* to the nodes of the tree rooted at *node*, in order.

14.2 (3) Combining similar operations into a single operation can improve code reuse. Revise, extend, or generalize the following two operations into a single operation. Also list the attributes needed to track both types of accounts.

a. *Cash_check(normal_account, check)* If the amount of the *check* is less than the balance in *normal_account*, cash the check and debit the account. Otherwise, bounce the check.

b. *Cash_check(reserve_account, check)* If the amount of the *check* is less than the balance in *reserve_account*, cash the check and debit the account. Otherwise, examine the reserve balance. If the check can be covered by transferring funds from the reserve without going over the reserve limit, cash the check and update the balances. Otherwise, bounce the check.

14.3 (4) Figure E14.1 is a function coded in C to create a new sheet for a computer-aided design application. A sheet is a named, displayable, two-dimensional region containing text and graphics. Several sheets may be required to completely represent a system being designed. The function given in the figure creates a new vertical or horizontal sheet and constructs a name from a root

and a suffix. The C functions it calls are *strlen* to compute the length of a string, *strcpy* to copy a string, *strcat* to concatenate two strings, and *malloc* to allocate memory. The data types *Sheet_type* and *Sheet* are defined outside of the function in the same module. The functions *strlen*, *strcpy*, and *strcat* will cause a crash if they are called with 0 for any argument. As it stands, the subroutine is exposed to several types of errors. The arguments *root_name* and *suffix* could be zero and *sheet_type* could be an illegal enumerated value. The call to *malloc* could fail to allocate memory.

a. Prepare a list of all the ways the function could fail. For each way, describe the consequences.

b. Revise the function so that it does not crash as a result of any of the errors you listed in part *a* and so that it prints out a descriptive error message for each kind of error as an aid in debugging programs in which it is called.

```
Sheet create_sheet (sheet_type, root_name, suffix)
Sheet_type sheet_type;
char *root_name, *suffix;
{ char *malloc(), *strcpy(), *strcat(), *sheet_name;
  int strlen(), root_length, suffix_length;
  Sheet sheet, vert_sheet_new(), horiz_sheet_new();
  root_length = strlen(root_name);
  suffix_length = strlen(suffix);
  sheet_name = malloc(root_length + suffix_length + 1);
  sheet_name = strcpy(sheet_name, root_name);
  sheet_name = strcat(sheet_name, suffix);
  switch(sheet_type)
  { case VERTICAL:
      sheet = vert_sheet_new();
      break;
    case HORIZONTAL:
      sheet = horiz_sheet_new();
      break;
  }
  sheet->name = sheet_name;
  return sheet;
}
```

Figure E14.1 Function to create a new named sheet

14.4 (3) Rewrite the C function in Figure E14.2 by using better names. The corresponding object diagram is in Figure E14.3. The function determines the value of a term by recursively expanding expressions. The function *compute* applies a binary operation to two values and *var_get* returns the value of a variable. Rename these functions as well.

```
Value ptree_get(top_nde)
Term top_nde;
{ Value val, val1, val2, compute(), var_get();
  switch(top_nde->node_type)
  { case CNSTNT:
      val = top_nde->value;
      return val;
      break;
    case VAR:
      val = var_get(top_nde->name);
      return val;
      break;
    case EXP:
      val1 = ptree_get(top_nde->left_nde);
      val2 = ptree_get(top_nde->right_nde);
      val = compute(val1, val2, top_nde->binary_operator);
      return val;
      break;
  }
}
```

Figure E14.2 Function to evaluate a *Term*

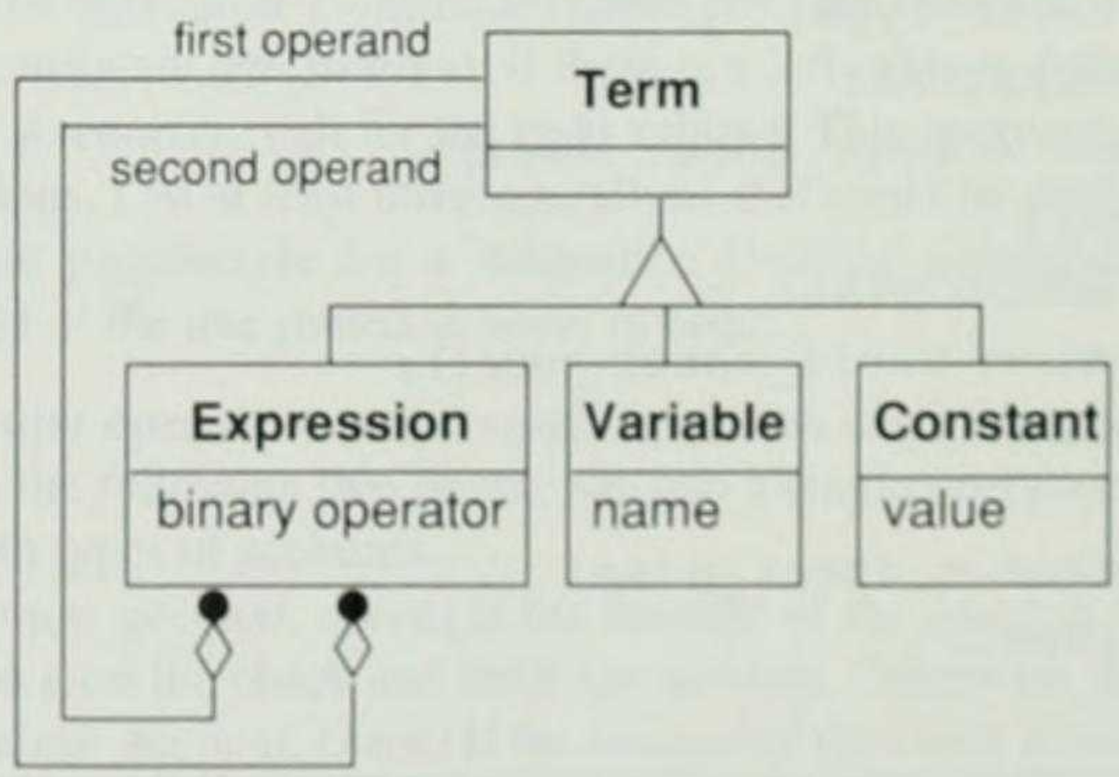


Figure E14.3 Object diagram of a *Term*

14.5 (7) Add methods to the object diagram in Figure E14.4 to satisfy the following queries. You may wish to review exercise 8.19. Write pseudocode for each method that you add and indicate whether the method is public or private. Separate policy from implementation. Avoid traversing multiple links.

a. Find all the members of a given team.

b. Find which figures were held more than once in a given season.

c. Find the net score of a competitor for a given figure at a given meet.
d. Find the team average score over all figures in a given season.
e. Find the average score of a competitor over all figures in a given meet.
f. Find the team average score in a given figure at a given meet.
g. Find the set of all individuals who competed in any events in a given season.
h. Find the set of all individuals who competed in all of the events held in a given season.
i. Find all the judges who judged a given event in a given season.
j. Find the judge who awarded the lowest score during a given event.
k. Find the judge who awarded the lowest score for a given figure.

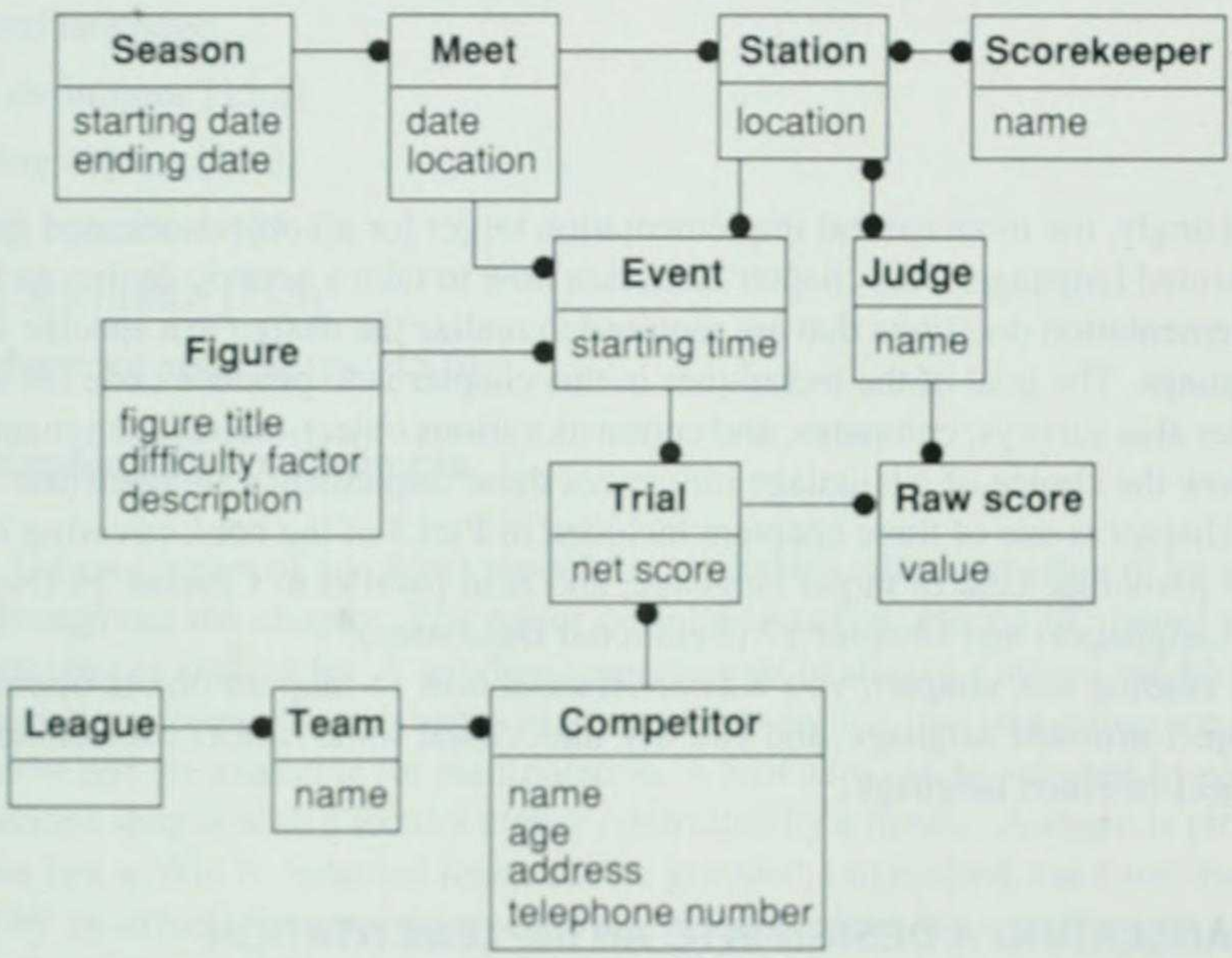


Figure E14.4 Portion of an object diagram of a scoring system

15

Object-Oriented Languages

Not surprisingly, the most natural implementation target for an object-oriented design is an object-oriented language. This chapter discusses how to take a generic design and make the final implementation decisions that are required to realize the design in a specific object-oriented language. The goal of the techniques in this chapter is to produce code for a program. The chapter also surveys, compares, and contrasts various object-oriented languages in order to show how the choice of a language influences these implementation decisions.

This chapter is one of three chapters included in Part 3 of the book covering implementation in a particular kind of target language, and is in parallel to Chapter 16 (Non-Object-Oriented Languages) and Chapter 17 (Relational Databases).

After reading this chapter, you will understand how to map an object-oriented design into an object-oriented language, and you will understand some factors to consider in choosing an object-oriented language.

15.1 TRANSLATING A DESIGN INTO AN IMPLEMENTATION

It is relatively easy to implement an object-oriented design with an object-oriented language since language constructs are similar to design constructs. A precise definition of an OO language is not crucial for selecting one. In general, an OO language supports *objects* (combining data and operations), *polymorphism* at run-time, and *inheritance*. A more formal discussion of the essential concepts of OO languages and the diverse ways in which these concepts are supported is found in [Stefik-86] and [Wegner-87].

We use C++, Eiffel, and Smalltalk to explain how to implement an OMT design with OO languages. We discuss implementation of the basic object-oriented concepts common to all object-oriented languages, describing the various ways in which different languages support the concepts. Section 15.7 discusses more advanced features of object-oriented lan-

guages, including some features that are not well supported by current languages. Section 15.8 discusses distinctive aspects of several commercially available OO languages: Smalltalk, C++, Eiffel, CLOS, and object-oriented database languages.

All three OMT models contribute to code development. The object model contains most of the declarative structure: the specification of classes, attributes, inheritance hierarchy, and associations. The dynamic model specifies the high-level control strategy for the system: procedure-driven, event-driven, or multi-tasking. The functional model captures functionality of objects that must be incorporated into methods.

The following considerations apply to implementing an object-oriented design in an object-oriented language:

- Class definitions [15.2]
- Creating objects [15.3]
- Calling operations [15.4]
- Using inheritance [15.5]
- Implementing associations [15.6]

15.1.1 Graphics Editor Example

Figure 15.1 shows a part of an object model for a graphics editor, which will be used as an example throughout the chapter. The editor permits recursive groups of shapes to be constructed from boxes and circles. A window contains a set of shapes. Groups can be built from shapes or smaller groups. Items (shapes or groups) that are not part of a group are root items in the window and are available for manipulation. A root item can be selected by picking one of its embedded shapes with a locator cursor controlled by a mouse. A shape is picked if the cursor point lies within it. Selected items can be grouped, ungrouped, cut from the window, or moved by an offset. Commands are also provided to clear the selections or create new shapes. Shapes are erased by writing over them in the background color.

The diagram shows the attributes and operations of each class. Abstract operations are indicated by the word *abstract* in the origin class

15.2 CLASS DEFINITIONS

The first step in implementing an object-oriented design is to declare object classes. Each attribute and operation in an object diagram must be declared as part of its corresponding class. It is good practice to carry forward the names from the design diagram. If you have not already assigned data types to attributes, you must now do so. Declare attributes and operations as either public or private, if the language supports the distinction. Public features can be accessed by any method, while private features are only accessible by methods of the same class.

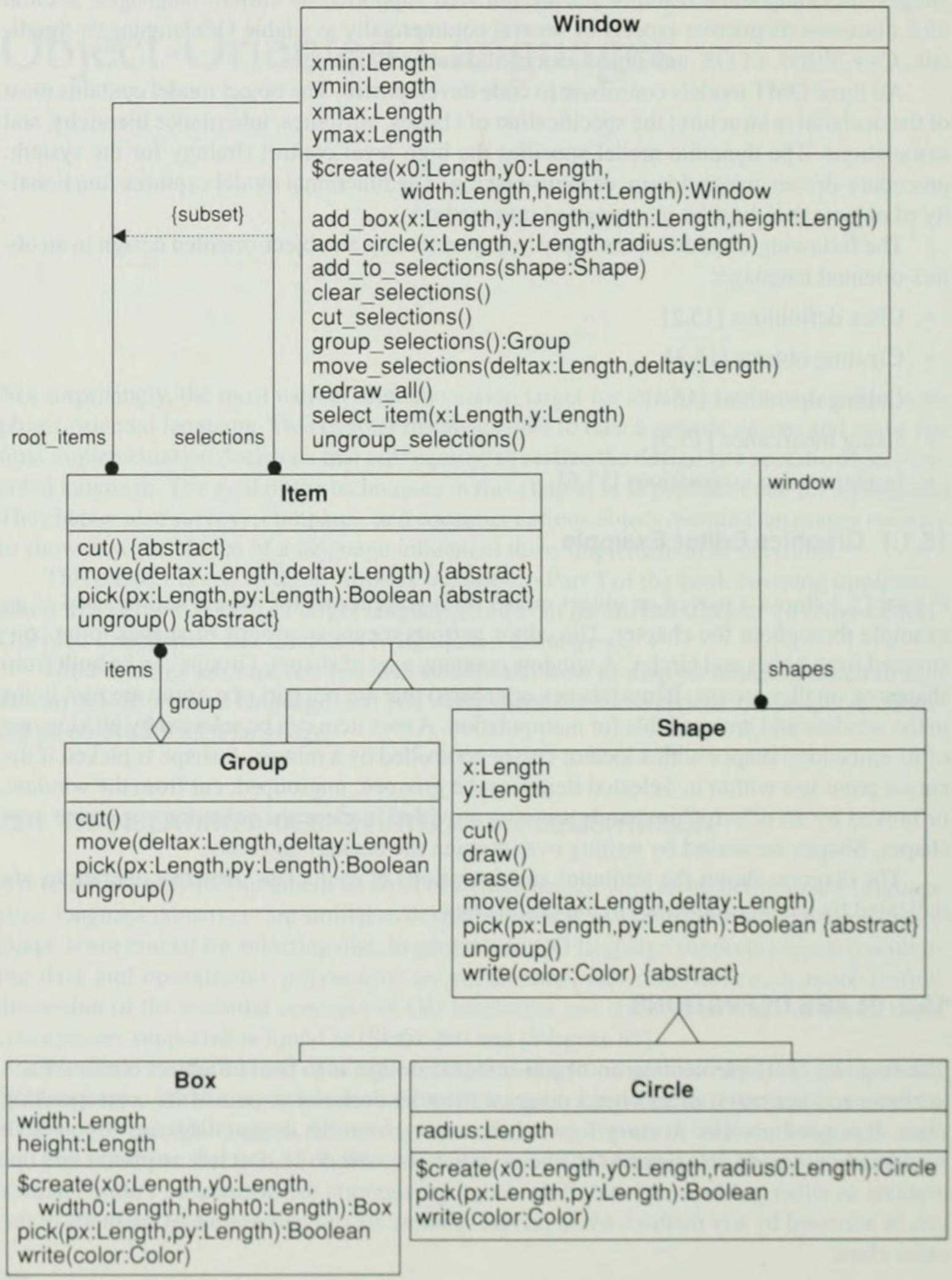


Figure 15.1 Simple graphics editor

15.2.1 Class Definitions in C++

In C++, the declaration for the class *Window* is:

```
class Window
{
public:
    // constructor method must have same name as the class
    Window (Length x0,Length y0,Length width, Length height);
    // destructor method must have same name as the class
    ~Window ();
    // instance methods
    void add_box (Length x, Length y,
                  Length width, Length height);
    void add_circle (Length x, Length y, Length radius);
    void clear_selections();
    void cut_selections();
    Group * group_selections();
    void move_selections (Length deltax, Length deltay);
    void redraw_all();
    void select_item (Length x, Length y);
    void ungroup_selections();
private:
    Length ymin;
    Length xmax;
    Length ymax;
    void add_to_selections(Shape* shape);
};
```

In C++, both attributes and methods are declared together as *members* of a class. A method may not have the same name as an attribute. The object creation routine, called a *constructor*, must have the same name as the class (Section 15.3.1).

In C++, you must declare members as either public or private (the default is private). Public members can be accessed by any function; private members can only be accessed by methods on the same class (not necessarily the same object, however). All attributes in this example are private because arbitrary changes to a window are not permitted. In general, it is good practice to keep attributes private and provide access to them only through methods, to prevent client code from being dependent on the exact implementation of a class. All operations are public, except *add_to_selections*, which is used internally by the *Window* class. We have omitted the associations from this declaration. Later, we explain how to represent associations in an object-oriented language.

Length is a user-defined type (not a class) that hides the actual implementation of the length (so that it could be either integer or real, for example). It could be declared as a *typedef* in C++:

```
typedef float Length;
```

or

```
typedef int Length;
```

15.2.2 Class Definitions in Eiffel

In Eiffel, the class declaration for the class *Window* is:

```
class WINDOW
export
    add_box, add_circle, clear_selections, cut_selections,
    group_selections, move_selections,
    redraw_all, select_item, ungroup_selections
feature
    xmin, ymin, xmax, ymax: REAL;
    Create (x0, y0, width, height: REAL) is body end;
    add_box (x, y, width, height: REAL) is body end;
    add_circle (x, y, radius: REAL) is body end;
    add_to_selections (ashape: SHAPE) is body end;
    clear_selections is body end;
    cut_selections is body end;
    group_selections: Group is body end;
    move_selections (deltax, deltay: REAL) is body end;
    redraw_all is body end;
    select_item (x, y: REAL) is body end;
    ungroup_selections is body end
end -- class WINDOW
```

In Eiffel, features (attributes and operations) are private unless they are explicitly made public with the *export* keyword. Create is automatically public. *REAL* is a simple, or predefined, type. Although Eiffel is not case-sensitive, by convention simple types and class names are capitalized.

15.2.3 Class Definitions in Smalltalk

Smalltalk programs are normally entered using the Smalltalk browser. The printed code we show here would normally be added interactively. All Smalltalk classes are ultimately descended from class *object*.

```
class name              Window
superclass              Object
instance variables      xmin ymin xmax ymax
class methods
    instantiating
    createAt: aPoint ofWidth: width ofHeight: height
instance methods
    adding shapes
    addBoxAt: aPoint ofWidth: width ofHeight: height
    addCircleAt: aPoint ofRadius: radius

    refreshing window
    redrawAll
```

```
    manipulating selections
    clearSelections
    cutSelections
    groupSelections
    moveSelectionsBy: deltaPoint
    selectItemAt: aPoint
    ungroupSelections

    private
    addToSelections: aShape
```

To make attributes (*instance variables* in Smalltalk) private, we omit methods to query and set the attributes. The groupings in italics are called *categories*, used to organize methods of a class. They have no other semantic meaning. By convention, methods that are for internal use only are put in the category *private*, but Smalltalk does not enforce the privacy. The programmer must be careful to honor the convention.

We have replaced separate x and y coordinates by class *Point* in several places. Smalltalk has a convenient method '@' to create a point from a pair of numbers. For example:

```
aPoint ← 3 @ 4
```

15.3 CREATING OBJECTS

Object-oriented languages create new objects in one of two ways. Some languages, such as Smalltalk and DSM, have classes that are full objects in their own right. In these languages, an operation applied to a class object (called a *class operation*) creates a new object of the class. Other languages, such as C++ and Eiffel, do not have class objects. These languages have special operations that create new objects.

When a new object is created, the language must allocate storage for its attribute values and must assign it a unique object ID, either the address of the storage block or an index in a table. Object-oriented languages free the programmer of the necessity to explicitly allocate memory for objects.

Different languages use one of two styles of destroying objects that are no longer needed. In some languages, objects are destroyed by an explicit operation (such as *destroy*). The programmer must take care that no references to a destroyed object remain, or memory access errors may result. Explicit memory management is error-prone, so some languages, such as Smalltalk, include automatic garbage collectors that destroy objects that are inaccessible, without requiring (or permitting) any explicit deallocation.

15.3.1 Creating Objects in C++

C++ does not have class objects or class operations to create instances of a class. Instead it has a special *constructor* operation to initialize new instances. The name of the method for the class constructor has the same name as the class, such as:

```
Window::Window (Length x0,Length y0,
                Length width, Length height)
{
    xmin = x0; ymin = y0;
    xmax = x0 + width; ymax = y0 + height;
}
```

In C++, multiple constructors for a single class can be defined, distinguished by the number and types of their arguments (this is an example of *overloading*). For example, we might have constructors with the following arguments:

```
Window (); // default position and size
Window (Length x0, Length y0); // default size
Window (Length x0, Length y0, Length width, Length height);
```

A constructor is executed whenever a new object instance is allocated. At the time of allocation, the programmer can specify the arguments for the constructor; the constructor with matching argument types is executed. If no arguments are specified, then the default constructor with no arguments is executed.

C++ provides three kinds of memory allocation for objects: preallocated by the compiler in fixed global memory (*static*), allocated on the stack (*automatic*), and allocated from a heap (*dynamic*).

Static storage is obtained by declaring a variable outside of any function or by using the keyword *static* on an attribute. A static attribute is a class attribute common to all instances of the class. Static storage is preallocated by the compiler and does not change during execution. A constructor can be specified, which is executed during program initialization. The following statement declares a static global variable to hold an initialized *Window* object:

```
Window main_window = Window (0.0,0.0,8.5,11.0);
```

Local variables within functions normally use automatic storage. When a function is entered, enough space on the hardware stack is reserved to hold all the local variables of the function. The storage is deallocated when the function exits, so references to automatic variables must not be stored in other objects whose lifetimes might exceed the automatic variables. The declaration of local variables has the same form as the declaration of global variables shown above, except they are declared within a function body.

Each class can have one *destructor*. A destructor performs any necessary cleanup before an object is destroyed. Destructors do not take any arguments. Destructor methods have the name of the class prefixed by a tilde, '~':

```
Window :: ~Window ()
{
    // erase the window and repaint the underlying region
}
```

When a function exits, the *destructor* for each automatic variable is called. After the destructors have been run, the storage for automatic variables is implicitly deallocated when the function return instruction adjusts the stack to its pre-call value.

Dynamic storage is allocated from a heap on an explicit request from the programmer. The *new* operator allocates storage for a new object and returns a pointer to it, which can be stored in a pointer variable. The *new* operator can include arguments for the constructor:

```
Window * window = new Window (0.0,0.0,8.5,11.0);
```

A dynamic object may outlive the function in which it is created. Dynamic objects can only be deallocated by applying the *delete* operator to the object pointer. The programmer must make sure that no references to the object remain. The *delete* operator first invokes the destructor for the class and then deallocates the storage for the object:

```
delete window;
```

Standard implementations of C++ do not have a garbage collector and heap-based objects must be explicitly deallocated by the *delete* operator, so there is danger of dangling object references to incorrectly deleted objects, as well as the loss of memory to inaccessible objects which have not been deleted. A dangling reference is a reference by one object to another object that no longer exists.

Note that constructors and destructors simply initialize and clean up objects. Memory allocation and deallocation are invoked explicitly by the *new* and *delete* operators or implicitly by declaration of local variables.

C++ permits considerable flexibility in memory allocation. The programmer can override the built-in memory allocator for a given class and supply a new one. Because operators such as assignment and type conversion can also be overridden on a class-by-class basis, it is possible to completely control allocation and creation of objects, although most programmers will not need to use such techniques.

15.3.2 Creating Objects in Eiffel

Eiffel separates the declaration of a variable (called an *entity* in Eiffel) from the creation of an object. A declaration of an entity makes a name that can hold a reference to an object of a given type, but the reference is initially void, that is, it does not refer to any object:

```
w: WINDOW
```

This declaration makes a variable that can hold a reference to a window, but does not store an actual object in the variable. It is similar to a C++ declaration for a pointer or reference variable.

All Eiffel objects are dynamic; there is no equivalent of the C++ static or automatic objects. To create an object of the given type and store its reference in the entity, the *Create* operation is applied to the entity:

```
w.Create (0.0, 0.0, 8.5, 11.0);
```

There is a default *Create* operation for each class that allocates a new instance of a class and initializes its attributes to zero values. The programmer can override the default *Create* operation to perform additional actions or accept arguments:

```
class WINDOW
...
feature
   Create (x0, y0, width, height: REAL) is
      do
         xmin := x0; ymin := y0;
         xmax := x0 + width; ymax := y0 + height
      end; -- Create
   ...
end -- class WINDOW
```

A second special operation, *Clone*, copies an existing object and creates a new object whose attributes are identical.

There is no way to explicitly destroy an Eiffel object (such as the C++ *delete* operator). The *Forget* operation removes the object reference from an entity, but it does not destroy the object itself. The Eiffel garbage collector is responsible for destroying objects that are no longer accessible because they cannot be reached from the root object or from any program variables. This simplifies the programmer's job considerably and prevents many insidious bugs, but imposes a run-time cost on the system.

Eiffel's garbage collector runs as a coroutine that checks for unused objects when memory usage reaches predefined thresholds. The garbage collector can be explicitly turned on and off.

15.3.3 Creating Objects in Smalltalk

All Smalltalk objects are dynamic and are allocated from a heap. Deallocation is performed by a built-in garbage collector. All variables are untyped and can hold objects of any class. New objects are created using the same *message passing* (operation calling) mechanism used for operations on objects. Smalltalk does not require any special creation operations, as do C++ and Eiffel.

To allocate a new object in Smalltalk, a message is sent to a *class object*, that is, an object that describes the class itself. Instance creation is a *class operation*, that is, it is an operation on a class, rather than an operation on an instance object. By convention, the *new* operation creates a new instance of a class:

```
w ← Window new
```

Unlike in C++ and Eiffel, there are no restrictions on the names of instance creation operations. Because a creation operation is explicitly applied to a class object, it cannot be confused with an instance operation. The developer can write custom instance creation methods that accept arguments:

```
w ← Window createAt: 0 @ 0 ofWidth: 8.5 ofHeight: 11.0
```

The method would be defined as follows:

```
class name Window
class methods
```

```
    createAt: aPoint ofWidth: width ofHeight: height
        | w |
        w ← self new.
        w initialize: aPoint width: width height: height.
        ↑w
instance methods
    initialize: aPoint width: width height: height
        xmin ← aPoint x.
        ymin ← aPoint y.
        xmax ← xmin + width.
        ymax ← ymin + height
```

The class method cannot directly access the attributes of the new instance, so an instance method *initialize:width:height:* must be written to initialize the attributes.

Objects cannot be explicitly deallocated. Storage is reclaimed by an automatic built-in garbage collector.

15.4 CALLING OPERATIONS

In most OO languages, each operation has at least one implicit argument, the target object, indicated with a special syntax. Operations may or may not have additional arguments. Some languages permit a choice between passing arguments as read-only values or as references to values that can be updated by a procedure.

15.4.1 Calling Operations in C++

A C++ operation is declared as a member of a class along with attributes. An operation is invoked using a similar notation to attribute access: the member selection operator "->" is applied to an object pointer:

```
Shape* shape;
shape->move (dx, dy);
```

Additional arguments may be objects, built-in types such as *int*, *float*, and *char*, and user-defined types, such as *typedef* types.

An attribute name or operation name used as an identifier within a C++ method implicitly refers to the target object of the method. In the following example, *x* and *y* are attributes of the target object of class *Shape*:

```
void Shape::move (Length deltax, Length deltay)
{
    x = x + deltax;
    y = y + deltay;
}
```

The implicit argument *this* contains a reference to the target object (usually to pass it as an argument to another operation). The previous code could be written as:

```
void Shape::move (Length deltax, Length deltay)
{
   this->x = this->x + deltax;
   this->y = this->y + deltay;
}
```

Reference to an attribute of an object that is not the target requires a qualified reference:

```
window->xmin = x1;
```

Unless the qualified object is of the same class as the method, such direct attribute access should be avoided because it violates encapsulation of classes. Use an access method on the class instead, which permits the internal representation of the class to be changed.

Unlike Smalltalk and Eiffel, C++ makes a distinction between the object record and a reference to an object. Use of the object record directly permits the attributes of the object to be updated, but does not permit references to the object to be inserted into associations. Most of the time it is best to consider the object reference as "the object." To use object records directly, omit the C++ indirection operator (* or &):

```
Box box(10.0,13.4,5.12,3.14);
box.move (dx, dy);
```

15.4.2 Calling Operations in Eiffel

Methods in Eiffel are called *routines*. Arguments to routines can be simple types (*REAL*, *INTEGER*, *BOOLEAN*, and *CHARACTER*) or user-defined classes. Although all objects are kept as pointers, Eiffel does not permit a routine to modify a formal parameter through assignment or by applying an operation (such as *Create*, *Clone*, or *Forget*) that can modify the reference. However, other operations can be applied to an object that is a formal parameter, and these operations may result in changes to the state of the object.

The Eiffel syntax for calling an operation is similar to C++. Note that the Eiffel attribute selection operator '.' is equivalent to the C++ operator '->':

```
local
   aShape: SHAPE;
   dx, dy: REAL
do
   ...
   aShape.move (dx, dy);
end
```

Eiffel permits implicit access to the features of the target object by writing the name of the feature. Identifiers *x* and *y* are attributes of the target *SHAPE*:

```
move (deltax, deltay: REAL) is
   -- move a shape by delta
do
   x := x + deltax;
   y := y + deltay
end
```

Eiffel supplies a predefined identifier, *Current*, that names the target object of an operation. This identifier is equivalent to *this* in C++ and *self* in Smalltalk. The code above is equivalent to:

```
move (deltax, deltay: REAL) is
    -- move a shape by delta
    do
        Current.x := Current.x + deltax;
        Current.y := Current.y + deltay
    end
```

Reference to a feature of an object that is not the target requires a qualified reference:

```
window.xmin := xl;
```

15.4.3 Calling Operations in Smalltalk

All arguments and variables in Smalltalk are objects. All operations are methods associated with objects. The programmer *sends messages* (applies operations) to objects. A *message* is the name of an operation with a list of argument values. Smalltalk binds the message to a method at run-time by examining the class of the object and searching the methods on the class and its ancestors. Formal parameters to methods cannot be changed by assignment within a method. The Smalltalk syntax for message passing does not use punctuation marks but does use keywords to separate the arguments:

```
aShape moveDelta: aPoint
```

This method would be implemented as follows:

```
class name Shape
instance variables
    x  y
instance methods
    moveDelta: aPoint
        x ← x + aPoint x
        y ← y + aPoint y
```

Within a method, attributes of the target object (*instance variables* in Smalltalk) may be accessed directly by writing the name of the attribute, as in C++ and Eiffel.

Smalltalk provides a pseudovariable, *self*, that refers to the receiver of the message. There is no Smalltalk equivalent of the C++ and Eiffel operations to access an attribute from an object. Only the attributes of the target object can be accessed directly. Attributes of other objects must be accessed by (user-provided) access operations:

```
aPoint ← aWindow getLocation
```

All attributes are therefore private to the class. Unfortunately, there is no way to restrict access to operations of a class; all operations are public.

15.5 USING INHERITANCE

Object-oriented languages vary in the mechanisms provided to implement inheritance. [Kim-88a, Chapter 3] discusses three independent dimensions for classifying inheritance mechanisms: static or dynamic, implicit or explicit, and per object or per group. Many of the popular languages are static (inheritance is bound at compile time), implicit (behavior of an object depends on its class, which cannot be changed), and per group (inheritance characteristics are specified for a class, not for specific objects). In most languages, the declaration of each class includes a list of superclasses from which it inherits attributes and methods.

15.5.1 Using Inheritance in C++

The superclass or superclasses of a class are specified as part of the class declaration. A subclass is called a *derived* class. The C++ code that follows declares *Shape* to be a subclass of *Item*. *Box* and *Circle* are subclasses of *Shape*.

```
class Item
{
public:
    virtual void cut() = 0;
    virtual void move (Length deltax, Length deltay) = 0;
    virtual Boolean pick (Length px, Length py) = 0;
    virtual void ungroup() = 0;
};
class Shape: public Item
{
protected:
    Length x;
    Length y;
public:
    void cut();
    void draw() {write (COLOR_FOREGROUND);}
    void erase () {write (COLOR_BACKGROUND);}
    void move (Length deltax, Length deltay);
    virtual Boolean pick (Length px, Length py) = 0;
    void ungroup() {}
    virtual void write (Color color) = 0;
};
class Box: public Shape
{
protected:
    Length width;
    Length height;
public:
    Box (Length x0, Length y0, Length width0, Length height0);
    Boolean pick (Length px, Length py);
    void write (Color color);
};
```

```
class Circle: public Shape
{
protected:
   Length radius;
public:
   Circle (Length x0, Length y0, Length radius0);
   Boolean pick (Length px, Length py);
   void write (Color color);
};
```

The attributes declared in a superclass are inherited by its subclasses and need not be repeated. They can be accessed from any subclass unless they are declared *private*. Only methods of a class can access its private attributes. Attributes declared *protected* are accessible to subclasses but not to client classes.

```
Boolean Box::pick (Length px, Length py)
{
   return x<=px && px<=x+width && y<=py && py<=y+height;
}
```

Methods declared in a superclass are also inherited. If a method can be overridden by a subclass, then it must be declared *virtual* in its first appearance in a superclass. For example, method *write* in class *Shape* can be overridden in classes *Box* and *Circle* and is therefore virtual. Methods *draw* and *erase* on class *Shape* are not overridden by the subclasses so they need not be declared virtual (they are implemented by calling virtual function *write* with a color of black or white). Method *Shape::write* is declared virtual and "initialized" to 0; this identifies it as a *pure virtual function*, that is, an abstract operation. Any class with a pure virtual function is an abstract class and cannot be instantiated directly. The compiler verifies that any concrete subclass defines or inherits implementations for all pure virtual functions.

Methods that override inherited methods must be redeclared in the subclass, but those that are inherited (and not overridden) need not be repeated. Likewise, inherited attributes need not be repeated. Virtual operations are called using the same syntax as nonvirtual operations:

```
Shape * shape;
Length x,y;
Boolean status;
status = shape->pick (x,y);
```

The knowledge of a class's ancestry can be declared *public* or *private*. If a superclass derivation is *private*, then clients of the class cannot call inherited operations directly nor access attributes of the ancestors. Masking inherited operations permits the safe use of inheritance for implementation purposes where a true generalization relationship does not exist, although we would discourage this kind of usage.

C++ supports multiple inheritance, by specifying a list of superclasses in the derivation statement. The use of conflicting attribute or operation names from different superclasses is not allowed. C++ supports several complicated variations on multiple inheritance, including a chain of constructors that are automatically invoked when a new instance is created.

15.5.2 Using Inheritance in Eiffel

In Eiffel, inheritance is indicated with the *inherit* clause as shown in the following example:

```
class ITEM
export
   cut, move, pick, ungroup
feature
   cut is deferred end;
   move (deltax, deltay: REAL) is deferred end;
   pick (x,y: REAL):BOOLEAN is deferred end;
   ungroup is deferred end
end
class SHAPE
export cut, draw, erase, move, ungroup, write
inherit ITEM
feature
   x,y: REAL;
   cut is body end;
   draw is body end;
   erase is body end;
   move (deltax, deltay: REAL) is body end;
   ungroup is body end;
   write (acolor: COLOR) is deferred end
end
class BOX
export pick, write
inherit SHAPE redefine pick, write
feature
   width, height: REAL;
   Create (x0, y0, width0, height0: REAL) is body end;
   pick (x,y: REAL): BOOLEAN is body end;
   write (acolor: COLOR) is body end
end
class CIRCLE
export pick, write
inherit SHAPE redefine pick, write
feature
   radius: REAL;
   Create (x0, y0, radius0: REAL) is body end;
   pick (x,y: REAL): BOOLEAN is body end;
   write (acolor: COLOR) is body end
end
```

Abstract operations are indicated with the *deferred* declaration. These operations must be implemented by all subclasses. Features in the subclass that override features in the superclass must be indicated with the *redefine* clause (not shown in this example). Features that are inherited and not overridden, such as operation *move* within class *CIRCLE*, need not be listed in either the *export* clause nor the *redefine* clause.

External visibility of features is specified by the *export* list, which replaces the *public* and *private* declarations of C++. All the features of a superclass are available to its descendents (there is no distinction corresponding to C++ *private* and *protected*). Features can be selectively exported to a specified class (and all its descendents), similar to the C++ *friend* declaration. Eiffel features can be renamed in a subclass.

Eiffel supports multiple inheritance. Renaming of inherited features can help to reduce name conflicts from different superclasses.

15.5.3 Using Inheritance in Smalltalk

In Smalltalk, inheritance is indicated by supplying the name of the superclass for an object. As in C++ and Eiffel, only those features that are unique to a subclass are specified. Any methods to be overridden are defined in the subclass.

```
class name          Item
superclass          Object

class name          Shape
superclass          Item
instance variables
   x
   y
instance methods
   cut
   draw
   erase
   move: aPoint
   ungroup

class name          Box
superclass          Shape
instance variables
   width
   height

instance methods
   pick: aPoint
   write: aColor
class methods
   createAt: aPoint width: widthSize length: lengthSize

class name          Circle
superclass          Shape
instance variables
   radius
instance methods
   pick: aPoint
   write: aColor
class methods
   createAt: aPoint radius: radiusSize
```

All the attributes of a superclass are available to all of its descendents. There is no need to indicate that a method can be overridden; all methods can be overridden. In C++ and Eiffel, an operation must be defined in its *origin* class, that is, the least common ancestor of all the classes that implement methods on the operation. In Smalltalk, by contrast, there is no need to declare abstract methods, but the programmer must ensure that an undefined operation is not applied to an object. A normal convention and good practice is to define abstract methods as "*self subclassResponsibility*" in case a method on a subclass is forgotten. The method *subclassResponsibility* then prints an error message if an undefined abstract operation is invoked. Errors in Smalltalk programs are more likely to show up at run time, because of its weak typing. There is no way to block the use of an operation by any class, so complete encapsulation is not possible in Smalltalk.

The standard implementations of Smalltalk do not support multiple inheritance, although some experimental implementations have supported it.

15.6 IMPLEMENTING ASSOCIATIONS

Chapter 10 discussed how to implement associations and aggregations. There are two general approaches: buried pointers and distinct association objects.

If your language does not explicitly support association objects (and most currently do not), then buried pointers are easiest to implement. Buried pointers may be added during design or deferred until now. In either case, the attributes needed to implement the buried pointers must be added to the class definitions. A binary association is usually implemented as an attribute in each associated object, containing a pointer to the related object or to a set of related objects. In many cases, however, the association is only traversed in one direction, so a pointer need only be added to one of the classes. Pointers in the "one" direction are simple to implement; they simply contain object references. Pointers in the "many" direction require a set of objects, or an array if the association is ordered, implemented most easily with a collection class object from a class library.

An association can also be implemented as a distinct container object. An explicit association object is conceptually a set of tuples, each tuple containing one value from each associated class. A binary association object can be implemented as two *Dictionary* objects, each dictionary mapping in one direction across the association.

There is an important consequence of buried pointers that is overlooked by many users and designers of object-oriented languages. Associations cannot be simulated by attributes on classes without violating encapsulation of classes because the paired attributes composing an association are not independent. Updating one pointer in the implementation of an association implies that the other pointer must be updated as well to keep the implementation consistent. The individual attributes should not be made freely available externally because they must not be updated separately. On the other hand, an externally-available method to update the attributes cannot be attached to either class of the association without access to the internal implementation of the other class because the attributes are mutually constrained. C++ allows limited relaxation of encapsulation using the *friend* construct, and

Eiffel provides export of features to selected classes, but there is no clean way to encapsulate associations as attributes in Smalltalk or most other languages.

Associations are intrinsically important since they arise when capturing and describing requirements. (See Section 3.2.3.) The real-world need for associations does not go away merely because most existing OO languages are not rich enough to directly capture them. The best solution to this problem would be for language designers to provide support for association objects as a basic concept, as is done in the DSM language [Shah-89].

15.6.1 Implementing Associations in C++

The many-to-one association between *Item* and *Group* could be implemented in C++ using pointers:

```
class Item
{
   // other declarations as before
private:
   Group * group;
   friend Group::add_item (Item *);
   friend Group::remove_item (Item *);
public:
   Group * get_group () {return group;}
};
class Group: public Item
{
   // other declarations as before
private:
   ItemSet * items;
public:
   void add_item (Item *);
   void remove_item (Item *);
   ItemSet * get_items () {return items;}
};
```

Whenever a new link is added to the association, both pointers must be updated. When a link is removed, both pointers must also be updated. The code might look as follows:

```
void Group::add_item (Item *item)
{
   item->group = this;
   items->add (item);
}
void Group::remove_item (Item *item)
{
   item->group = 0;
   items->remove (item);
}
```

The *Group* methods can update the *group* attribute in an *Item* object because they are declared as *friend*s of class *Item*. The *friend* construct provides a clean mechanism for allowing access to internal details without totally destroying encapsulation.

In this example we have omitted code to check whether an item is already in a group. Before putting an item in a group, we should first check to see if it is already in an group, and remove it if necessary. This check is not absolutely necessary if we can guarantee that we will never put an item in a group without first removing it from a previous group, but an explicit check adds protection against bugs.

Class *ItemSet* is a collection class that holds sets of items. It has the following protocol:

```
class ItemSet
{
public:
    ItemSet (); // create an empty set
    ~ItemSet (); // destroy a set
    void add (Item *); // add an item to the set
    void remove (Item *); // delete an item from the set
    Boolean includes (Item *); // test for an item in the set
    int size (); // report the number of items in the set
};
```

Collection classes that are more general (but weakly-typed) are found in available class libraries, such as the NIH class library [Gorlen-90] produced at the USA National Institutes of Health. This library also includes *iterator* objects to scan collections in *for*-loops without knowing the internal structure of the collection. There are operations to create an iterator, to advance it while testing for the end of the collection, and to get the current object from the collection:

```
Boolean Group::pick (Length x, Length y)
{
    ItemIterator it(items);
    while (it++)
        if (it()->pick (x,y)) return TRUE;
    return FALSE;
}
```

Variable-length array classes can be used to implement ordered associations. Some applications may require other data structures, such as circular lists or trees. (We have omitted the details of adapting a generic class, such as *Set*, to hold a specific kind of object, such as *Item*. This will eventually be done using parameterized types in C++. Until those are part of the standard C++ language, class libraries implement various tricks to permit storing specific classes in generic containers.)

Distinct association objects are not difficult to implement if the class library does not already contain them. *Association* can be implemented as a new container class and placed in the class library. The simplest approach is to implement an association object as two dictionary objects. (A dictionary is a look-up table that maps one value into another. It is provided by most class libraries.) One dictionary maps the association in the forward direction

and the other in the reverse direction. Both dictionaries must be updated when the association is updated, but traversal of an association is efficient.

Each association in the object model is implemented by an association object. The links in the association are the elements of the association object. Using association objects preserves encapsulation of the constraint between the classes implicit in an association. For example, the previous routine could be written as:

```
Association* item_group_asn = new Association (many_to_one);

void Group::add_item (Item *item)
{
   item_group_asn->add (item, this);
}

void Group::remove_item (Item *item)
{
   item_group_asn->remove (item, this);
}

Group * Item::get_group ()
{
   return (Group*) item_group_asn->index_forward (this);
}

ItemSet * Group::get_items ()
{
   return (ItemSet*)item_group_asn->index_reverse (this);
}
```

We have glossed over the details of storing specific object classes in generic container objects, which can be done either by casting the contents (as we have shown) or by using parameterized classes (when they are added to C++).

If an association is implemented as a separate object, then no attributes need be added to either class in the association. If the association is sparse (that is, a small fraction of objects participate in it), then a separate association object uses less storage than a pointer in each object. Access of values is somewhat slower than with pointers in objects, but if associations are implemented with dictionaries that contain hash tables, the access time should be a small constant, independent of the number of links in the association.

15.6.2 Implementing Associations in Eiffel

The options for implementing associations in Eiffel are basically the same as in C++. Eiffel supports parameterized (*generic* in Eiffel) container objects, which are parameterized according to the type of objects they hold. The Eiffel *LINKED_LIST*, supplied in the basic class library, may be used to implement the many-to-one association between *ITEM* and *GROUP*:

```
class ITEM
export
    get_group
     -- selective export to class GROUP
    set_group {GROUP}, forget_group {GROUP}
feature
    mygroup: GROUP;
    get_group: GROUP is
        do
            Result := mygroup
        end;
    set_group(g:GROUP) is
        do
            mygroup := g
        end;
    forget_group is
        do
            forget(mygroup)
        end;
end -- ITEM
class GROUP
export add_item, remove_item, get_items
inherit
    ITEM
feature
    items:LINKED_LIST[ITEM]; -- parameterized type
    Create is
        do
            items.Create
        end;
    add_item(value:ITEM) is
        do
            items.finish;
            items.insert_right(value);
            value.set_group(Current)
        end;
    remove_item(value:ITEM) is
        do
            items.search (value, 1);
            items.delete;
            value.forget_group
        end;
    get_items(number:INTEGER):ITEM is
        do
            Result := items
        end;
end -- GROUP
```

Eiffel permits selective export of any feature. Here, we have exported *ITEM*'s *set_group* and *forget_group* features to the *GROUP* class. This maintains encapsulation by limiting write access to participants in the *ITEM/GROUP* association.

The Eiffel *forget* operator is for storage deallocation. It frees the object denoted by the object-id it operates on, and also sets the object-id to *void*.

15.6.3 Implementing Associations in Smalltalk

Smalltalk's rich class library helps developers to implement a variety of associations. For the many-to-one association (in Figure 15.1) between a Group and the Items it contains, we use a *Set*, found in the Smalltalk class library:

```
class name          Item
super class         Object
instance variables
   group
instance methods
   other instance methods as before
   getGroup
      ↑group
   private methods
   putGroup: aGroup
      group ← aGroup

class name          Group
super class         Item
instance variables
   items
class methods
   new
      ↑((super new) putItems: (Set new))
instance methods
   other instance methods as before
   addItem: anItem
      items add: anItem.
      anItem putGroup: self
   removeItem: anItem
      items remove: anItem.
      anItem putGroup: nil
   getItems
      ↑items copy
   private methods
   putItems: aSet
      items ← aSet
```

Since Smalltalk does no type checking, there are no restrictions on the class of items that can be added to a *Group*. Any object that responds to the *putGroup:* message is allowed. Note

that proper encapsulation of the association is impossible in Smalltalk. The method *put-Group:* on class *Item* is marked as "private," but it is up to the programmer to respect the privacy of the method, because all methods are actually public.

15.7 OBJECT-ORIENTED LANGUAGE FEATURES

Object-oriented languages vary in their support of advanced object-oriented concepts. There is no single language that matches all styles and meets all needs. In this section we discuss various features of object-oriented languages that vary among languages. Select a language appropriate to your requirements. For example, if your design requires multiple inheritance, you should consider languages that support it well. Interpreted languages are useful for rapid prototyping but may not be adequate if you have stringent timing constraints. There is no point in building your own class library if the language provides one with suitable classes for your design. If you want extra compile-time error-checking, use a strongly-typed language. If memory usage is convoluted in your application, consider languages which provide automatic garbage collection. If you require persistent data accessible by multiple users, you may want to consider an OO database. Unless a language is exceptionally clumsy to use, syntax should only be a minor consideration.

This chapter refers to a variety of OO languages. C++, Eiffel, Objective-C, Smalltalk-80 and CLOS are commercially available languages. Trellis-Owl and DSM are research languages. GemStone and ORION are database languages. Figure 15.2 at the end of the chapter summarizes the features supported by certain commercially-available languages.

15.7.1 Multiple Inheritance

Some languages, such as C++ (Version 2), CLOS, Eiffel, and DSM, support multiple inheritance. Many others do not. If the language you select supports multiple inheritance, there is a direct translation from your design to implementation. Otherwise, you must use some of the workarounds suggested in Chapter 4.

Multiple inheritance introduces the possibility of conflict among attribute and operation names. A class could have more than one ancestor with the same attribute. There are several approaches that languages take to solving the problem. For example, the Eiffel compiler rejects programs with such clashes. CLOS, on the other hand, has a protocol for resolving them. In any case, we strongly recommend that you avoid such conflicts. Relying on the compiler to resolve conflicts is a matter of poor style because it damages extensibility and can lead to confusion.

15.7.2 Class Library

Most OO languages include a library of useful generic classes, which may be used as is or customized to meet a specific need through the creation of subclasses. The availability of a class library means that many components need not be reimplemented by the programmer.

The most useful kinds of classes implement general purpose data structures, such as sets, dynamic arrays, lists, queues, stacks, dictionaries, trees, and so forth. These classes, often referred to as *container classes*, serve as a framework for organizing collections of other objects. Classes implementing various kinds of associations should also be available in the class library.

More complete class libraries might include device-independent abstractions of interface classes (such as streams, input devices, display devices, file systems) or concurrent processes. A separate library of user interface classes for use with one or more windowing systems is useful, as is a graphics library. Support for string manipulation is needed by most programs.

15.7.3 Efficiency

Object-oriented languages have an undeserved reputation for inefficiency because some early languages (Smalltalk and Lisp-based languages) were interpreted rather than compiled. The availability of compiled languages and more efficient interpreters has given developers more latitude in choosing an appropriate language. Use of an OO language with a mature class library often results in code that runs *faster* than that written with a non-OO language. This is because the overhead of an OO language is often more than compensated by better implementations of data structures and algorithms available in a mature class library. For example, many programmers would not bother to implement hash tables or balanced trees for small programs, but such data structures are available in a good class library.

One aspect of object-oriented languages that seems inefficient is the use of *method resolution* at run-time (also known as dynamic binding) to implement polymorphic operations. Method resolution is the process of matching an operation on an object to a specific method. This would seem to require a search up the inheritance tree at run time to find the class that implements the operation for a given object. Most languages, however, optimize the look-up mechanism to make it more efficient. As long as the class structure remains unchanged during program execution, the correct method for every operation can be stored locally in the subclass. With this technique, known as *method caching*, dynamic binding can be reduced to a single hash table look-up and performed in constant time regardless of the depth of the inheritance tree or the number of methods in the class. Clever compiler tricks can reduce even the size of the constant cost.

In a strongly-typed language, such as C++, the cost of run-time method resolution can be reduced to a single structure reference, almost insignificant compared to the cost of calling a procedure. Each class has a structure containing methods accessible by objects of the class. A pointer to each method is stored at a known location with the structure. A pointer to the method structure for the class is stored in each object. Run-time method resolution is performed by retrieving the method structure and indexing it by the known offset of the method. This technique cannot be used if new methods can be created at run time (because the structure cannot be precomputed) or if the name of a method can be constructed at run time (because the offset cannot be precomputed).

A language system can further optimize method resolution to use dynamic look-up only where it is needed. A large percentage of the method calls in a typical application can be stat-

ically bound by the compiler if it is provided with enough information. There are two ways the compiler can acquire this information. One way is for the programmer to declare which operations may be overridden, thus limiting dynamic binding to only those functions. The virtual functions of C++ and the generic functions of CLOS reflect this approach. The other approach is to provide a final optimization pass that analyzes the application as a whole, decides what methods are not overridden, and then recompiles the application, taking advantage of this information. This approach is adopted by the Eiffel language system. Note that the declaration of types for variables allows the compiler to be much more precise in its determination of which method calls can be optimized.

15.7.4 Strong Typing Versus Weak Typing

Object-oriented languages vary greatly in their approach to typing. The term *typing* refers to whether each variable and attribute value is merely known to be an object (*weak typing*) or may be declared more precisely as belonging to a particular class or one of its descendents (*strong typing*). The sample design in Figure 15.1 assumes strong typing. For example, the attribute *radius* is declared to be of type *Length*. Smalltalk is a weakly typed language—all variables in Smalltalk are objects of unspecified class. Eiffel, DSM, Objective-C, and C++ are strongly typed languages. CLOS and Objective-C support strong typing but do not require its use—a variable that references an object may optionally by declared, and the compiler will check usage of that variable. Hybrid languages, such as DSM, C++, and Objective-C, even permit an attribute to assume a value which is not an object but rather is an underlying C language type. Permitting nonobject values increases efficiency of common operations, such as arithmetic, at some cost in uniformity of reference.

Strong typing in a language serves two purposes. Strong typing provides active support to the programmer in detecting mismatched method arguments and assignment statements. Strong typing also increases opportunities for optimization. The compiler can detect when a general purpose operation can be replaced by a specific function call (see Section 15.7.3). No power is lost by the use of strong typing; the programmer can always declare everything to be of class *Object* to get the same effect as weak typing.

Although weak typing is flexible and powerful, it permits dangerous coding practices. Modern programming language theory has been evolving in the direction of strong type checking. Much of the advances in programming language theory have derived from restricting the programmer's ability to perform dangerous operations. Most new languages support strong typing. The premise behind strong typing is that it is easier to detect and correct software bugs at compile-time rather than at run-time. Strong typing improves the reliability of delivered software.

15.7.5 Memory Management

The freedom with which objects can be created and accessed causes a problem when memory space must be reclaimed. Most OO languages allocate memory from a heap rather than using fixed blocks of global memory or a stack. A dynamic memory allocation system can run out of memory unless objects that are no longer needed are deallocated. The principal

problem lies in determining when an object is not needed or can no longer be accessed. The severity of the problem depends on the type of application and the memory architecture. Many programs that have access to a large amount of virtual memory can simply ignore the problem. On the other hand, a highly interactive application expected to run indefinitely must consider memory management no matter how large the available memory space.

There are two approaches to memory management: either it is done automatically by the run-time system of the language or it is done explicitly by deallocation statements written by the programmer. The preferred approach, automatic memory management, relieves the programmer of the responsibility of deciding when to deallocate memory and avoids the risk of dangling object references that explicit deallocation is vulnerable to. However, users are understandably dissatisfied with some garbage collection schemes that result in long pauses at unpredictable points during execution. Incremental garbage collection schemes avoid long delays but impose a higher average cost. Improved garbage collection algorithms offer some hope in this area.

CLOS and Smalltalk offer fully automatic garbage collection. C++ requires the programmer to deallocate unneeded objects, but the C++ language allows the programmer to define destructor functions, which are called automatically when a variable goes out of scope. This gives the programmer a convenient way of organizing and invoking explicit storage deallocation but is not a garbage collector. Dynamically-allocated objects must be explicitly deallocated by the *delete* operator. Explicit deallocation runs the dual risk of deallocating objects that are still referenced (leading to program crashes) and failing to deallocate objects that are inaccessible (leading to memory waste), but it does not impose any additional time or memory costs on the program.

15.7.6 Encapsulation

Encapsulation (also *information hiding*) consists of separating the *protocol*, the external aspects of an object that are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects. Encapsulation prevents a program from becoming so interlaced that a small change has massive ripple effects. You can change the implementation of a class without affecting its clients. You may want to change the implementation of an object in order to improve performance, fix a bug, consolidate code, or port it to a different system. Encapsulation is not unique to object-oriented languages, but the ability to combine data structure and behavior in a single entity makes encapsulation cleaner and more powerful than in conventional languages that separate data structure and behavior. Languages differ greatly in the degree to which the class boundary is enforced.

One way that encapsulation can be violated occurs when code associated with one class directly accesses the attributes of another class. Direct access makes assumptions about the storage format and location of the data. These implementation details should be hidden within a class. For example, an attribute value may actually be computed from other information. The proper way to access an attribute of another object is to "ask for it" by invoking an operation of the object, rather than simply "taking it." Many languages (such as Smalltalk) forbid direct access to the attributes of another object or (as in C++ and Trellis-Owl) permit attributes to be declared either public or private. Eiffel provides perhaps the finest control of

encapsulation through its *export* statement, which lists attributes that may be read and operations that may be executed. The C++ *friend* declaration allows limited loosening of encapsulation for access by specified classes or functions.

Encapsulation can be violated through careless use of inheritance. A subclass inherits the attributes of its superclass, but if the methods of the subclass directly access superclass attributes, the encapsulation of the superclass is defeated. CLOS, Owl, Eiffel, and C++ permit a class to restrict its visibility from its subclasses. There are arguments in favor of allowing a class to access its superclasses, but the cost is tight binding between the classes.

Often it is useful to write some "private" operations that are for internal use only by other methods of the same class. It is desirable to restrict the visibility of these operations so that other classes cannot use them. It is often necessary to allow subclasses to access these internal operations, so a simple distinction between public and private is not possible. Some languages permit a third level of access to subclasses (such as the C++ *protected* declaration).

A more complete discussion of encapsulation is found in [Micallef-88].

15.7.7 Packaging

The class is not an adequate structuring construct for large systems. Most OO languages lack partitioning mechanisms for controlling visibility between classes. We have suggested the use of modules containing classes as a structuring mechanism during analysis.

A related problem is that object-oriented languages typically require that the names of classes must be unique. Often this is not the case when two or more applications that were developed independently are combined. For example, two designers working independently might define different versions of a class called *Symbol* with different operations and semantics. The problem is similar to name clashes in block-structured languages.

In the Ada language and in CLOS, the *package* construct provides the means to structure a system into separate components with their own name-spaces. In Ada, names declared in the specification of a package are not visible outside the package unless explicitly requested, allowing control of both the name-space and inter-package dependencies. Entities that are known by one name outside the package can be renamed within. Packages may be nested, allowing an entire subsystem to be encapsulated within a well-defined interface.

Object-oriented languages would benefit by the addition of packaging capability. In particular, the ability to nest the packaging components of a system allows better control of visibility.

15.7.8 Development Environment

The tools that are available for browsing and editing source code, compilation, debugging, system integration, and testing have a big effect on your productivity. Support tools are especially important in object-oriented languages because of the difficulty of managing inherited and dynamically-bound features in a large system.

A *browser* allows you to explore the source code in a structured way, asking what classes are present and what operations are defined for each class. The Smalltalk-80 browser, described in [Goldberg-84], is the most prominent example.

The *compiler* or *interpreter* is the most important implementation tool. Whether the language is interpreted, compiled, or translated affects the speed with which you can make small changes and test their effect. An interpreter allows more flexible debugging. Interpreted systems such as Smalltalk and Lisp allow an executing program to be stopped, edited to correct a bug, and resumed. Objective-C offers an optional interpreter in addition to the compiler.

Some compilers translate an object-oriented language to an intermediate language (such as C) which is then compiled. The C++ language was first implemented by translation into C, but true compilers are now available. Translation may pose problems for *symbolic debuggers* because they do not understand the original object-oriented code. In evaluating the debugger, find out whether it shows you the original object-oriented source code or the intermediate code (which may be hard to read). The debugger should be able to inspect attribute values and evaluate object-oriented expressions.

Tools for *system-building* and *change control* are essential for large projects but are also valuable for individual programmers. Consider whether the proposed language provides them or integrates smoothly with existing tools. System rebuilding can consume much time if a simple change in one module requires that client modules must be recompiled as well. Try to determine what kinds of change trigger recompilation of clients. Experience has shown that the use of conventional building tools, such as UNIX *make*, are too crude for many purposes. Finer-grained development environments are needed but are only recently available.

15.7.9 Metadata

METADATA (discussed in Chapter 4) is data about data. METADATA that is present at run-time allows an application to reason about, and possibly even change, its own structure and capabilities, including the operations an object supports, the attributes it possesses, or the types of attributes. There are also more conventional uses of metadata, such as writing a generic procedure that prints or saves any object using the attribute names and types.

Languages containing explicit class descriptor objects (such as Smalltalk and DSM) contain run-time metadata about classes. A class object acts as a template for the creation of new instances. The class may also contain descriptions of the attributes and operations, such as name, type, or arguments. For example, a *Complex number* class might define *real part* and *imaginary part* attributes each of which is represented as a 32-bit floating point number.

Of course, the designer of the system knows the metadata, so in principle this data could be embedded in the code. But if the design changes (say, by adding a new attribute to a class) then all the code that depends on embedded data will become invalid. The use of metadata at run-time rather than at compile-time allows the construction of extensible systems with generic procedures and abstract classes that are reusable in unforeseen applications. METADATA also can be used to support data persistence or by programming support tools such as debuggers, browsers, and inspectors.

15.7.10 Parameterized Classes

Parameterized classes, or generics [Meyer-86], allow a parameterized template to be written that can then be applied in several cases that differ only in the types of the parameters. For example, a generic *List* class can be instantiated as a *list of points* or a *list of integers*. The generic *add-element* method on the list implicitly depends on the type of element. Local variables within the method have generic type that depends on the instance. Parameterized classes are present in the Ada language and in Eiffel. Parameterized templates have been proposed for C++ [Stroustrup-88].

15.7.11 Assertions and Constraints

Assertions and constraints improve the odds that the behavior of a class matches the expectations of its clients. We can do this informally by writing a natural language description of the desired behavior, but how can we be sure that the description will be understood in the same way (or even read) by another programmer who wants to use the class?

Usually there are a few critical assumptions about the behavior of a class, operation, or method that can be expressed mathematically. These assumptions take the form of *assertions* that must be true at particular points during execution (such as preconditions and postconditions), and *constraints* that must be maintained, or *invariants* that must always be true.

Chapter 8 showed how to use assertions and constraints as part of the functional model. Assertions should also be written into the software in a form that can be optionally compiled and automatically checked at run-time. While this technique can be used in many languages (such as the *assert* macro available in many C implementations), it is particularly well-supported in the Eiffel language [Meyer-88].

Constraints can be more than simply conditions to be tested. They can be viewed as a way of expressing declaratively what might otherwise have to be written as procedural code. A language system could understand constraints that are declared and insure that they remain true by taking action at run-time to maintain the constraints. The Prolog logic programming language has such capabilities. It would not be surprising to see such constraint maintenance systems become widely available under object-oriented languages.

15.7.12 Data Persistence

All computer programs operate on data. If you require data that persists beyond the lifetime of a single program execution, then you will need to use a permanent data store. There are various reasons for wanting persistent data:

- Persistent data may provide the easiest mechanism for passing data from one program to another.
- Persistent data allows the same program to resume processing at a later date.
- Data stores are often useful for historical or archival purposes.

There are several approaches to providing persistent data services: files, special purpose hardware devices, databases, and intrinsic language support. Some new object-oriented databases are especially convenient in their data storage services. To a large extent, such an advanced object-oriented database looks like an object-oriented language, except that the orthogonal property of persistence may be specified for data structures. The advantages and disadvantages of object-oriented databases, relative to conventional databases, are discussed in Section 15.8.5.

15.8 SURVEY OF OBJECT-ORIENTED LANGUAGES

Having considered various language characteristics, we now examine how they have been combined in several commercially available object-oriented languages: Smalltalk, C++, Eiffel, CLOS, and database programming languages.

15.8.1 Smalltalk

Smalltalk was the first popular object-oriented language, developed at Xerox PARC, and its success engendered many other object-oriented languages. Smalltalk is not only a language but also a development environment incorporating some functions of an operating system. For single-user development, it offers arguably the best features of both language and environment. Smalltalk falls short in areas where it was not intended to be used—in multiple person projects and in its weak or unspecified ability to interface with external software or hardware devices. Smalltalk elegantly articulates the goals of extensibility and reusability.

All aspects of the Smalltalk language system are available through an on-line interpreter and class browser. The language syntax is simple. Variables and attributes are untyped. Everything is an object, including classes. Classes can be added, extended, tested, and debugged interactively. A garbage collector frees the programmer of the burden of memory management.

What does Smalltalk provide the implementor? Perhaps the most important contribution is the highly interactive development environment, which avoids the edit-compile-link cycle delays of the traditional compiler-based language. The Smalltalk environment permits rapid development of programs. Another strength is the class library, which was designed to be extended and adapted by adding subclasses to meet the needs of the application. Because Smalltalk is an untyped language, library components can be combined to rapidly prototype an application.

The Model/View/Controller (MVC) architecture for user interface design is another important contribution of Smalltalk. A user interface is divided into an underlying application-defined model, any number of different views of the model, and controllers that synchronize changes to the model and the views. MVC makes it possible to concentrate on the essentials of an application (the Model) and add the user-interface (the Views and Controllers) independently. The class library provides standard versions of each of these components, which can be subclassed and extended incrementally. There can be many different view/controller

pairs for each model, and the views and controllers can be modified extensively with little or no change in the model. However, the MVC is a complex system that is not easy to learn.

Smalltalk is a pure object-oriented system with extensive metadata available and modifiable at run-time. Implementation of the language as an interpreter, tightly integrated with other parts of the its self-contained environment, gives ideal support for rapid incremental development and debugging.

15.8.2 C++

C++ is a hybrid language, in which some entities are objects and some are not. C++ is an extension of the C language, implemented not only to add object-oriented capabilities but also to redress some of the weaknesses of the C language. Many added features are orthogonal to object-oriented programming, such as inline expansion of subroutines, overloading of functions, and function prototypes. Because of its origin as an extension of C, its backing by major computer vendors, the perception of it as a nonproprietary language, and the availability of free compilers, C++ seems likely to become the dominant object-oriented language for general use.

C++ is a strongly-typed language developed by Bjarne Stroustrup at AT&T Bell Laboratories. It was originally implemented as a preprocessor that translates C++ into standard C. As a preprocessor, C++ introduced problems for symbolic debuggers, but direct compilers are now available, and symbolic debuggers that support objects with inheritance and dynamic binding are now available. Commercial vendors offer C++ implementations for a variety of operating systems. A C++ compiler with debugger and library are available for no charge from the Free Software Foundation (with restrictions on commercial use).

Unlike several other OO languages, C++ does not contain a standard class library as part of its environment, although the standard AT&T release includes libraries for I/O, coroutine tasking, and complex arithmetic. Class libraries have been implemented by various developers, including a class library developed by the USA National Institutes of Health (NIH), which is achieving wide usage [Gorlen-90]. Class libraries for object-oriented windowing systems include Interviews [Vlissides-88] and ET++ [Weinand-88]. Unfortunately, because C++ provides no guidelines for library organization, different libraries may be incompatible. The emergence of a consensus in favor of a standard foundation class library would be an important asset to C++.

C++ contains facilities for inheritance and run-time method resolution, but a C++ data structure is not automatically object-oriented. Method resolution and the ability to override an operation in a subclass are only available if the operation is declared *virtual* in the superclass. Thus, the need to override a method must be anticipated and written into the origin class definition. Unfortunately, the writer of a class may not expect the need to define specialized subclasses or may not know what operations will have to be redefined by a subclass. This means that the superclass often must be modified when a subclass is defined and places a serious restriction on the ability to reuse library classes by creating subclasses, especially if source code for the library is not available. (Of course, you could declare *all* operations as *virtual*, at a slight cost in memory and function-calling overhead.)

The implementation of run-time method resolution is efficient. For each class, a predefined *struct* is initialized with pointers to each method available to the class. Each object contains a pointer to the method structure for its class. At run-time, a virtual operation is resolved by retrieving the method structure from the object and selecting a member to find the method address. C++ does not support run-time class descriptor objects other than the method pointer structure. C++ 2.0 supports multiple inheritance.

C++ contains good facilities for specifying access to attributes and operations of a class. Access may be permitted by methods of any class (*public*), restricted to methods of subclasses of the class (*protected*), or restricted to direct methods of the class (*private*). In addition, "spot" access can be given to a particular class or function using the *friend* declaration.

As with C, the declaration syntax of C++ is awkward and its grammar is difficult to parse. C++ supports overloaded operators: several methods that share the same name but whose arguments vary in number or type. C++ supports several memory allocation strategies for objects—statically allocated by the compiler, stack-based, and allocated at run-time from a heap. The programmer must avoid mixing objects of different memory types or dangling references may cause run-time failures. Each class can have several *constructor* and *conversion* functions, which initialize new objects and convert between types for assignment and argument passing; these are semantically sound but perhaps somewhat confusing for normal use.

In summary, C++ is a complex, malleable language characterized by a concern for the early detection of errors, various implementation choices, and run-time efficiency at the expense of some design flexibility and simplicity.

15.8.3 Eiffel

Eiffel is a strongly typed object-oriented language written by Bertrand Meyer. Programs consist of collections of class declarations that include methods. Multiple inheritance, parameterized classes (*generics*), memory management, and assertions are supported. A modest class library is provided, including lists, trees, stacks, queues, files, strings, hash tables, and binary trees. For portability, the Eiffel compiler translates source programs into C. Eiffel has good software engineering facilities for encapsulation, access control, renaming, and scope. Eiffel is arguably the best commercial OO language in terms of its technical capabilities.

The focal point of Eiffel is the class declaration, which lists attributes and operations. Eiffel provides uniform access to both attributes and operations by abstracting them into a single concept called a feature. An Eiffel class declaration may include a list of exported features, a list of ancestor classes, and a list of feature declarations. Eiffel does not treat either classes or associations as first class objects.

Eiffel supports memory management through a coroutine which detects objects that are no longer referenced and releases the memory allocated to them. The Eiffel run-time system executes the coroutine whenever the available memory space is low. Several mechanisms are provided to control memory management. Automatic execution of the coroutine may be suppressed through a compiler switch or turned on or off at run-time. For operating systems that

do not support virtual memory, there is a compiler switch to arrange for Eiffel's run-time system to provide automatic paging.

A contractual model of programming is supported by preconditions, postconditions, invariants, and exceptions. A *precondition* is a condition that the caller of an operation agrees to satisfy. A *postcondition* is one the operation itself agrees to achieve. An *invariant* is a condition that a class must satisfy at all stable times. Conditions and invariants are a part of the class declaration and must also be obeyed by all descendent classes. If they are violated at run-time, an exception occurs, which either causes the faulty operation to fail or executes an exception handler for the class if the programmer provides one. Compiler switches provide several levels of error checking. Once an application is debugged you can turn off assertion checking.

15.8.4 CLOS

The Common Lisp Object System (CLOS) is an object-oriented extension of Common Lisp which will soon become an official part of the Lisp language. It resulted from the work of a Lisp standardization group working on the X3J13 ANSII standard. This group studied the many existing object-oriented extensions of Lisp (such as Flavors and CommonLoops), but rather than selecting one of them as a standard, the group decided to formulate a new language based on language features that had proven successful. Because CLOS can be implemented in Common Lisp, it permits experimentation on new concepts to continue, while providing a common standard.

Although it was originally implemented in a hybrid fashion, CLOS is so well integrated with the traditional features of Common Lisp that it has most of the advantages of a "pure" object-oriented language. This is because every data object, including the atoms and lists of Lisp, is a member of a class. Methods for Lisp primitives belong to an inheritance structure. As a result, there is no practical distinction between Lisp primitives and objects.

The programming environment of Common Lisp and CLOS is an interpreter which also allows code to be compiled into a form that is more efficiently executed. Debugging facilities under Common Lisp depend on the specific implementation but are generally excellent. Commercial offerings of Common Lisp with CLOS should be expected to provide completely object-oriented debugger support.

CLOS currently does not have a standard class library. Organizations that use CLOS are currently collecting their own reusable classes, and there is some sharing of classes between organizations. Commercial vendors will undoubtedly provide extension classes such as window packages. One point in favor of CLOS is that the Common Lisp types already present form a common root for whatever new classes are added, so separately developed class libraries are more likely to be consistent.

CLOS provides powerful and flexible inheritance capabilities. Multiple inheritance is supported, and CLOS has rules for resolving the ambiguity resulting from inherited features with the same name.

Polymorphic operations requiring dynamic method resolution can be implemented as generic methods. All arguments to a generic method are explicit; there is no special variable like *self* in Smalltalk. Unlike most object-oriented languages, CLOS uses multiple argu-

ments to resolve methods, allowing the direct implementation of operations that exhibit "multiple polymorphism" in which the specific method for an operation depends on more than one of its arguments (for example, binary arithmetic operations).

CLOS provides a rich collection of metadata that can be accessed and updated at run-time. New classes can be defined, and methods can be added to classes dynamically. These features are a standard part of the language, documented as the "meta-object protocol." Unlike most languages, Lisp-based languages permit new procedures to be constructed at run time out of source statements of other procedures.

Like Common Lisp, CLOS is a weakly-typed language. Native types are provided, and classes behave like types, but there is no requirement to declare the type of a variable, and there is no checking to insure that the use of an object is consistent with its declaration. While achieving maximal design flexibility, weak typing can affect performance and ease of debugging. The optional declarations could be used by the compiler to optimize access, but this optimization depends on the specific implementation.

The concept of encapsulation is not enforced by the CLOS language system. Programmers are encouraged to define and document the public interface of each class and to use only the advertised features of other classes, but there is nothing to prevent the code of one class from directly accessing the implementation details of another class. This lack of enforcement is consistent with Lisp's general policy of providing maximum flexibility and room for experimentation.

15.8.5 OO Database Programming Languages

An OO-DBMS unites two technologies: data base management and object-oriented programming. Object-oriented programming languages (OOPL) are expressive but lack data persistence (data that outlasts the execution of a single job). Conventional DBMSs have data persistence but lack expressibility. OO-DBMSs try to provide both data persistence and expressibility. An OO-DBMS must handle large programs that operate on large data stores.

For conventional applications that combine programs with DBMS access, the developer uses a procedural technique, such as hierarchical functional decomposition or data flow diagrams, to design the overall application. A technique such as Entity-Relationship (see Chapter 12) may then be used to design the database. The database design may be converted to SQL code which is embedded in a language such as C, Pascal, or Cobol.

To say the least, the above scenario is unattractive. The fundamental problem is twofold. First, different paradigms are used to design various parts of the system. Second, the implementation vehicles mismatch. This book targets the design problem by uniformly applying object-oriented models to systems, programming code, and databases. OO-DBMS research is striving to remedy the implementation side by providing one software system that has the richness of programming languages and the practical features of DBMSs.

In general there are two types of database queries: set-oriented and navigational. Relational DBMSs are intended to perform parallel operations on large sets of data. In contrast, OO programming languages are efficient at quickly navigating from one object to another by traversing pointers. A relational DBMS performs navigation by using joins, which are several orders of magnitude slower than pointer traversal. An OO-DBMS must efficiently

perform both types of queries. One important feature of an OO-DBMS is the implicit assumption that the system is oriented towards operations on individual objects and the programmer can expect these to perform well. This is notable mainly because the relational model typically performs badly for single-object operations and navigation between objects [Cattell-86, Maier-86].

The first generation of OO-DBMSs were early products and did not provide the robustness and "industrial strength" users have come to expect of conventional DBMSs. The performance of these systems was disappointing for large applications. GemStone, which is marketed by Servio Logic, is a commercially marketed OO-DBMS. The GemStone language is similar to Smalltalk. The GemStone database has procedural interfaces to C and Pascal. GemStone uses an optimistic concurrency scheme and server-client model and supports single inheritance [Maier-86]. Another early OO-DBMS is ONTOS, marked by Ontologic Corporation. ONTOS has an object-oriented structure accessible by SQL queries and C++ program statements.

ORION is an OO database language prototype by MCC which is written in common LISP. ORION's syntax is similar to Flavors and LOOPS. ORION supports multiple inheritance, versioning, and concurrency with an IBM-like locking protocol. ORION supports set-oriented queries with a relational DBMS flavor. ORION permits some dynamic changes to the database schema. [Kim-88]

Second generation OO-DBMSs are under development by start-up companies whose principals have previous experience developing both object-oriented languages and commercial-quality relational database management systems. There is reasonable hope that one or more of these companies will succeed in producing a second-generation OO-DBMS that can compete effectively with commercial relational DBMSs.

15.8.6 Comparison of Object-Oriented Languages

Figure 15.2 summarizes the discussion of object-oriented languages contained in this section and in the previous section. As with any comparison table of this type, there will always be inaccuracies due to changes in the languages and differences between implementations. Nevertheless, we include this table because we think the benefits of comparing languages at a glance outweighs the shortcomings.

15.9 CHAPTER SUMMARY

This chapter shows how an object-oriented language can be used to implement a design developed using the OMT methodology. Practical advice has been given concerning the implementation of various design constructs and the selection of a language.

In the first part of the chapter we concentrated on language features that are common to most object-oriented languages, using C++, Eiffel, and Smalltalk as examples. Classes in the object model are implemented by declaring classes with the attributes and associations required. Implementation of inheritance is also accomplished by a simple declaration in OO

	C++ 2.0	Smalltalk 80	CLOS	Eiffel	Objective C
Integration of classes with primitive types	hybrid	pure	integrated	integrated	hybrid
Strong type checking	Y	N	N	Y	Y
Ability to restrict access to attributes:					
Control of access from clients	Y	Y	N	Y	Y
Control of access from subclasses	Y	N	N	Y	N
Standard class library	N	Y	N	Y	Y
Parameterized classes	F	—	—	Y	N
Multiple inheritance	Y	N	Y	Y	N
Scoping of class names (packages)	N	N	Y	N	N
Messaging model:					
Single target object	Y	Y	N	Y	Y
Dynamic binding on multiple args	N	N	Y	N	N
Method combination features:					
SUPER concept	N	Y	Y	Y	Y
Before & after methods	N	N	Y	N	N
Assertions and constraints	N	N	N	Y	N
Metadata at run-time	N	Y	Y	N	Y
Garbage collection	N	Y	Y	Y	N
Efficiency:					
Static binding when possible	Y	N	N	Y	Y

Key to table entries:

Y = Yes, the feature is present.
N = No, the feature is not present in common current implementations.
F = Planned in a future release
— = Not Applicable: Parameterized classes are not needed in languages with weak typing.

Figure 15.2 Comparison of commercially available object-oriented languages

languages. Associations are implemented either by using object references (pointers) or by defining a new collection class (*Association*) to serve as a template for creating specific associations in the design.

When writing methods to implement operations, object-oriented programming differs in a few respects from conventional languages. In most current OO languages, the target object is passed to the method as an implicit argument, available within the method under a special identifier (*self*, *this*, or *Current*). Other arguments should be passed as references to objects. Objects can be explicitly allocated or destroyed using special operations provided by the language.

The second part of the chapter discusses some important features which are not common to all OO languages. These features include multiple inheritance, a class library, efficiency of implementation, strong typing, automatic memory management, enforcement of encapsulation, packaging constructs, development environment, metadata accessible at run-time, parameterized classes, assertions and constraints, and persistent objects.

The final part of the chapter is a brief survey of OO languages, including OO database systems. Smalltalk is an interpreted language with weak type checking and garbage collection that is noted for its development environment and extensive class library. Such a language is ideal for rapid development of prototypes but does not support large team efforts well. The C++ language is a hybrid language based heavily on C but with improved type checking and extensions to support OO programming. Efficiency is obtained at the cost of some design flexibility. Eiffel is a strongly-typed language that provides many advanced features such as multiple inheritance, generic classes, automatic memory management, and assertion checking. CLOS is an integrated OO extension to Common Lisp. It provides powerful inheritance capabilities, rich metadata accessible at runtime, and an unusual but very flexible means of defining polymorphic operations. In the rapidly evolving area of OO database systems, the first generation products have been successful in providing basic OO structure accessible from both a query language and a program interface but do not quite meet industrial performance and robustness requirements. Research OO-DBMS have provided more sophisticated features, and a second generation of commercial systems is on the way.

assertion	object-oriented database
class library	object-oriented language
container class	packages (name-space control)
encapsulation	parameterized class
implementation of associations	persistence of data
memory management	programming environment
meta-information	self-reference
method resolution	strong typing

Figure 15.3 Key concepts for Chapter 15

BIBLIOGRAPHIC NOTES

A survey of various approaches to object-oriented language characteristics may be found in [Stefik-86]. A good comparison of specific languages is found in [Micallef-88]. The [Wegner-87] paper constructs a general design space of language characteristics within which existing languages may be classified and new research directions sought.

Descriptions of specific languages may be found in [Stroustrup-84] for C++, [Keene-89] for CLOS, [Cox-86] for Objective-C, [Meyer-88] for Eiffel, [Goldberg-83] for Smalltalk-80, [Schaffert-86] for Trellis-Owl, and [Shah-89] for DSM.

Despite his focus on Eiffel, [Meyer-88] provides a good general coverage of issues in strongly typed object-oriented languages. The same can be said of [Cox-86] for general coverage of languages with weakly typed object systems. Good descriptions of the general purpose objects in a class library as an important supplement to the language itself may be found in [Goldberg-83] and [Gorlen-90].

Specific treatment of unresolved issues in object-oriented languages may be found in the following sources: [Snyder-86] draws attention to the way that some views of inheritance violate encapsulation, [Stein-87] compares delegation and inheritance as means of sharing behavior, and [Meyer-86] contrasts genericity and inheritance.

REFERENCES

[Cattell-86] R.G.G. Cattell, T.R. Rogers. Combining object-oriented and relational models of data. *1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, Calif., Sept. 1986.

[Cox-86] Brad Cox. *Object-Oriented Programming: An Evolutionary Approach*. Reading, Mass.: Addison-Wesley, 1986.

[Goldberg-83] Adele Goldberg, David Robson. *Smalltalk-80: The Language and its Implementation*. Reading, Mass: Addison-Wesley, 1983.

[Goldberg-84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Reading, Mass.: Addison-Wesley, 1984.

[Gorlen-90] Keith Gorlen, Sanfor Orlow, Perry Plexico. *Data Abstraction and Object-Oriented Programming in C++*. Chichester, England: John Wiley & Sons Ltd., 1990.

[Keene-89] Sonya Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Reading, Mass.: Addison-Wesley, 1989.

[Kim-88a] Won Kim, Frederick H. Lochovsky. *Object-Oriented Concepts, Databases, and Applications*. New York: ACM Press, 1988.

[Kim-88b] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F. Garza, Jay Banerjee. Integrating an object-oriented programming system with a database system. *OOPSLA'88* as *ACM SIGPLAN 23*, 11 (Nov. 1988), 142-152.

[Lippman-89] Stanley B. Lippman. *A C++ Primer*. Reading, Mass.: Addison-Wesley, 1989.

[Liskov-87] Barbara Liskov, Alan Snyder, Russell Atkinson, Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM 20*, 8 (August 1977), 564-576.

[Maier-86] David Maier, Jacob Stein, Allen Otis, Alan Purdy. Development of an object-oriented DBMS. *OOPSLA'86* as *ACM SIGPLAN 21*, 11 (Nov. 1986), 472-482.

[Meyer-86] Bertrand Meyer. Genericity versus inheritance. *OOPSLA'86* as *ACM SIGPLAN 21*, 11 (Nov. 1986), 391-405.

[Meyer-88] Bertrand Meyer. *Object-Oriented Software Construction*. Hertfordshire, England: Prentice Hall International, 1988.

[Micallef-88] Josephine Micallef. Encapsulation, Reusability and extensibility in object-oriented languages. *Journal of Object-Oriented Programming 1*, 1 (April/May 1988), 12-38.

[Schaffert-86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, Carrie Wilpolt. An introduction to Trellis-Owl.*OOPSLA'86* as *ACM SIGPLAN 21*, 11 (Nov. 1986), 9-16.

[Shah-89] Ashwin Shah, James Rumbaugh, Jung Hamel, Renee Borsari. DSM: an object-relationship modeling language. *OOPSLA'89* as *ACM SIGPLAN 24*, 11 (Nov. 1989), 191-202.

[Snyder-86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. *OOPSLA'86* as *ACM SIGPLAN* 21, 11 (Nov. 1986), 38-45.

[Stefik-86] Mark Stefik, Daniel G. Bobrow. Object-oriented programming: themes and variations. *The AI Magazine 6*, 4 (1986), 40-62.

[Stein-87] Lynn Andrea Stein. Delegation is inheritance. *OOPSLA'87* as *ACM SIGPLAN 22*, 12 (Dec. 1987), 138-147.

[Stroustrup-84] Bjarne Stroustrup. *The C++ Programming Language*. Reading, Mass.: Addison-Wesley, 1986.

[Stroustrup-88] Bjarne Stroustrup. Parameterized types for C++. USENIX *C++ Conference*. Denver, Colo., October 17-21, 1988.

[Vlissides-88] John M. Vlissides, Mark A. Linton. Applying object-oriented design to structured graphics. *Proceedings of the 1988 USENIX C++ Conference*, Oct. 1988, 81-94.

[Weinand-88] André Weinand, Erich Gamma, Rudolf Marty. ET++—an object-oriented application framework in C++. *OOPSLA'88* as *ACM SIGPLAN 23*, 11 (Nov. 1988), 46-57.

[Wegner-87] Peter Wegner. Dimensions of object-based language design. *OOPSLA'87* as *ACM SIGPLAN 22*, 12 (Dec. 1987), 168-182.

EXERCISES

15.1 (4) Write class declarations for the object diagram in Figure E15.1 in any language that does not directly support multiple inheritance. The application is a simulator that computes the performance of electrical machines.

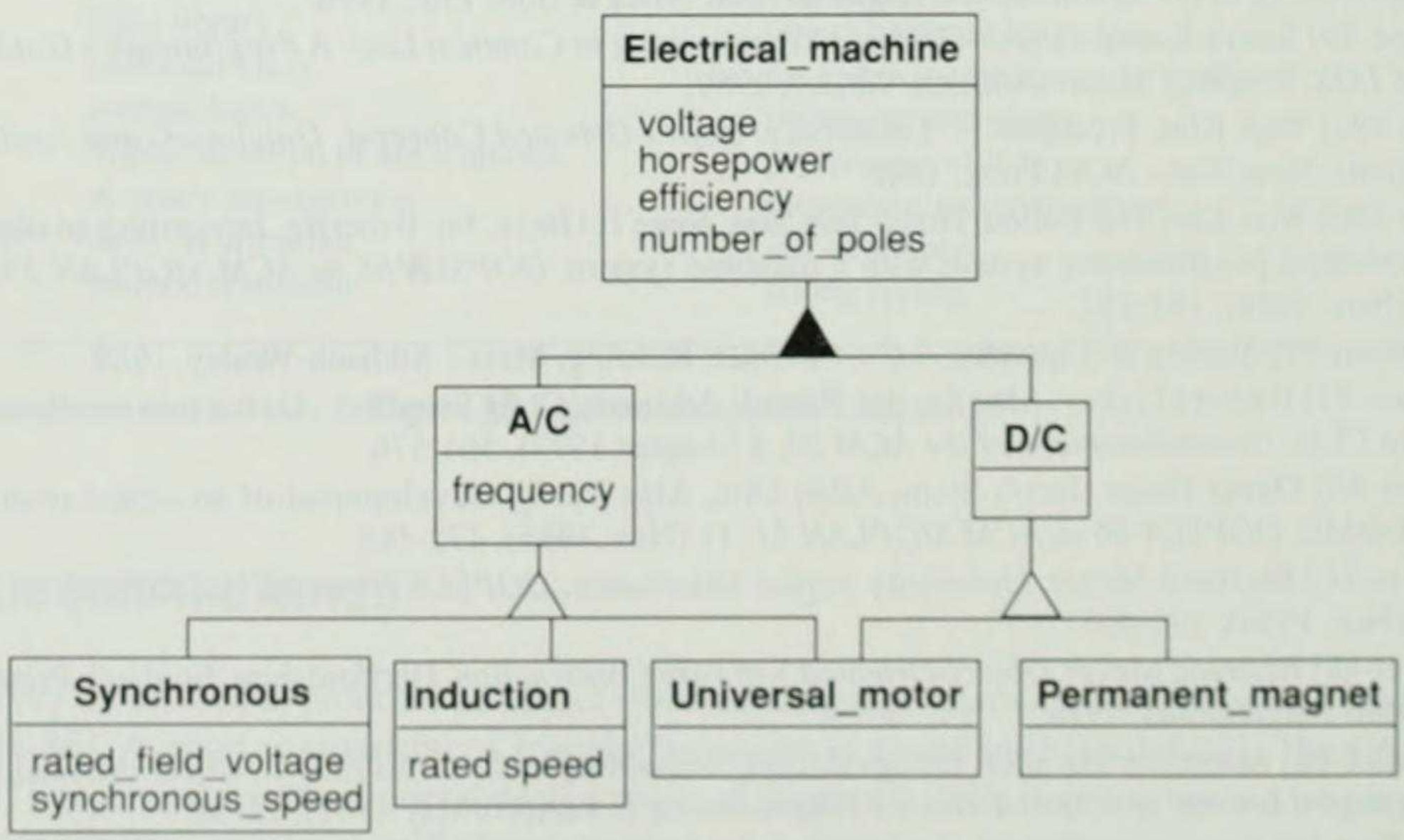


Figure E15.1 Partial taxonomy for electrical machines

15.2 (6) Write code, including class declarations and methods, to implement the following using pointers in any object-oriented language:

a. One-to-one association which is traversed in both directions.
b. One-to-many association which is traversed in the direction from one to many. The association is considered unordered.
c. One-to-many association which is traversed in the direction from one to many. The association is considered ordered.
d. Many-to-many association which is traversed in both directions. The association is considered ordered in one direction, and unordered in the other direction.

15.3 (4) Describe some situations in which strong typing would help you. Describe some circumstances in which strong typing would cause problems.

15.4 (8) Many object-oriented languages have libraries which support the creation and manipulation of container classes such as *Symbol*, *Set*, *Array*, *Dictionary*, and *Association*.

A *Symbol* is a string object that enables you to determine if two strings match by simply checking for equality of the object IDs. The first time a symbol with a given string value is created, a new object ID is allocated. All subsequent attempts to create a symbol with the same string value will return the previously assigned ID.

A *Set* is an unordered collection of unique values. Duplicate values are automatically discarded from a set. Values can be added, deleted, and tested.

An *Array* is an ordered, dynamic, collection of values, indexed by nonnegative integer selector values. The number of elements is variable. Elements can be added or deleted at the end, beginning, or at any point in the middle. An insertion or deletion in the middle causes the positions of subsequent elements to change.

A *Dictionary* is a many-to-one mapping function whose domain and range are objects, including possibly other container objects.

An *Association* is a two-way mapping that can be composed from two dictionaries.

Write pseudocode to construct the following from container classes. List any operations that you assume are defined on the container classes, and explain what they do.

a. A *Sorted_dictionary*. Each entry in the dictionary maps a name to an object. Entries are sorted by name. Needed operations include *insert(name, object)*, which adds an entry to the dictionary; *delete(name)*, which deletes an entry; *find(name)*, which returns the *object* associated with *name*; *find_first()*, which returns the first object in the dictionary; *find_last()*, which returns the last; *find_previous(object)*, which returns the object just previous to *object*; and *find_next_object(object)*, which returns the object which follows *object*. Also needed is an initialization operation which creates a new, empty dictionary. The operation *insert(name, object)* replaces an entry in the dictionary if *name* is already in the dictionary.
b. A *Polygon*. A *Polygon* is a two dimensional object whose x-y coordinates are stored as instances of the class *Point*. Needed operations on the class *Polygon* include *delete()*, which deletes a polygon, and *get_points()*, which returns an array of points which belong to a polygon. Needed operations on the class *Point* include *delete()*, which removes it from its polygon; *append(polygon)*, which adds it to a polygon; and *get_polygon()*, which returns the polygon to which it belongs. Also needed is an initialization operation to create a new polygon from an ordered list of points. Assume that a *Point* belongs to one *Polygon* rather than being shared. If two polygons share a common coordinate, two *Points* are made with the same coordinates.

c. An *Index*. An *Index* is similar to a *Dictionary*, except the mapping is to a set instead of individual objects. An index might be used, for example, to determine all automobiles that were manufactured in a given year. Needed operations include *add(object_selector, object_target)*, which adds a target object to the set associated with the selector; *delete(object_selector, object_target)*, which deletes a target from the set associated with the selector; and *find(object_selector)*, which returns a set of objects which are indexed by *object_selector*. Unlike the *insert* operation in part (a), the *add* operation does not replace existing data but adds to it.

15.5 (5) Describe how you would use tools included in software development environments such as browsers, compilers, interpreters, symbolic debuggers, system builders and change control systems to help solve the following problems:

a. You are using an unfamiliar subroutine in a subsystem library. There is not much documentation, so you would like to quickly find out how it works by experimenting with it.
b. Your program seems to have a bug in it which makes no sense to you as you glare at your source code for one of the program modules. Now you are not sure that the program that you just run was built from the source code that you are looking at. It is possible that you forgot to recompile this module before you ran the linker.
c. The program you are developing quits with a memory fault error. The line at which it fails is different each time you run the program, depending on the data. The problem is repeatable with the same data.
d. You are the leader of a team working on a large software project. You want to make sure that only one person works on a file at a time. The latest version of all modules must be accessible to the group as a whole, but individual users might want private copies for debugging purposes.
e. There are too many steps involved in building the application for you to execute all of them each time you make a change. Furthermore, you have trouble simply remembering all these steps, let alone trying to figure out which ones need to be redone as the result of your latest revision.
f. You have written your own preprocessor. The debugger you are using displays the output of the preprocessor, and you have trouble tracing errors back to the original source code.

15.6 (7) Describe strategies for dealing with the following memory management problems, assuming that automatic garbage collection is not feasible. Your answer should be in the form of guidelines that a programmer could use during coding.

a. *A system for text manipulation*. A common operation is the creation of one large text segment in contiguous memory from an array of several smaller segments. The system is expected to handle a lot of text, and you cannot afford to waste memory. You cannot set an upper bound on either the text length or the array size. The following operations are convenient building blocks which you must provide if they are not available as library routines: Determine the size of a text segment, allocate a segment of memory of a given size, deallocate a previously allocated memory segment, copy text from one memory segment to another, and combine two text segments placing the result in a previously allocated segment of memory. Write pseudocode for a method that combines text, recovering memory space that is no longer used.
b. *A multi pass compiler*. Objects are created dynamically. Each pass examines the objects created on the previous pass and produces objects to be used on the next pass. The computer system on which the compiler will run has a practically unlimited virtual address space and an operating system with a good swapping algorithm. The routines in the run-time library for

allocating and deallocating memory dynamically are inefficient. Discuss the relative merits of two alternatives: (1) Forget about garbage collection all together and let the operating system allocate a large amount of virtual memory. (2) Carefully deallocate memory when objects are no longer referenced.

c. *Banking software or an air traffic control system that runs for a long time.* You have the same computer system and run-time library as described in exercise 15.6(b). Discuss the relative merits of the two approaches.

d. *A subroutine which may create and return an object that uses a large block of memory.* The only operations allowed on the object class are implemented by other subroutines which you are going to write. You do not wish to leave garbage lying around. Discuss the relative merits of the following two approaches: (1) Each time the subroutine is called it destroys the object created the last time it was called, if any. (2) Each time the subroutine is called, it may create a new object. It is up to the calling routine to destroy the object when it is no longer needed by calling a routine that you will write. Consider both situations when the routine that calls your routine is recursive and when it is not.

15.7 (4) Express the following as a constraint that is to be enforced in an application which handles points and polygons: Points are associated with polygons. Every point belongs to exactly one polygon. You do not want a polygon to be linked to a point that has been deleted.

15.8 (7) Describe the attributes and methods of the generic class *Binary_tree* that will support the construction of unbalanced binary trees from any class whose instances can be ordered. A *compare* method must be provided for each class that is included. *Compare* compares two instances of a class, returning *less than*, *equal to*, or *greater than*, depending on the order. Operations on the class *Binary_tree* include *insert(object)* which inserts an object into a tree, *delete(object)* which deletes an object from a tree, *print()* which prints all objects in the tree in order, and *test(object)* which determines if a given object is in a tree. Assume that a *print* method is available for all classes from which trees are constructed. Descriptions of attributes should include data types and whether the attribute is private or public. Descriptions of methods should include data types of arguments and returned values as well as assertions and constraints.

15.9 (7) Write code in any object-oriented language to implement the generic class *Binary_tree* described in exercise 15.8. Test it on the following two classes: *Person* and *Page*. *Person* is ordered by *name* and *Page* is order by *page number*. You will need to write a *compare* method and a *print* method for each class. Make up a few attributes for each class.

15.10 (3) Write code in any object-oriented language to shuffle a deck of cards and to deal 4 hands of 13 cards from the deck. Be sure to create new instances of the classes *Card*, *Deck* and *Hand* from Figure E15.2. The relationship between *Collection_of_cards* and *Card* preserves the order in which cards are placed into collection which is not necessarily according to *suit* and *rank*.

15.11 (3) Using any object-oriented language, implement the aggregation between *Card* and *Collection_of_cards* in Figure E15.2 so that the order in which *Cards* are added to a *Collection_of_cards* is preserved and so that the aggregation can be traversed in both directions.

15.12 (4) Write code in any object-oriented language to implement the *sort* operation on the class *Hand* shown in Figure E15.2. Sort the cards in each suit according to rank in descending order: ace, king, queen, jack, ten down to deuce. Suits should be arranged in a hand in the order: spades, hearts, clubs, and diamonds. There are no arguments to the operation.

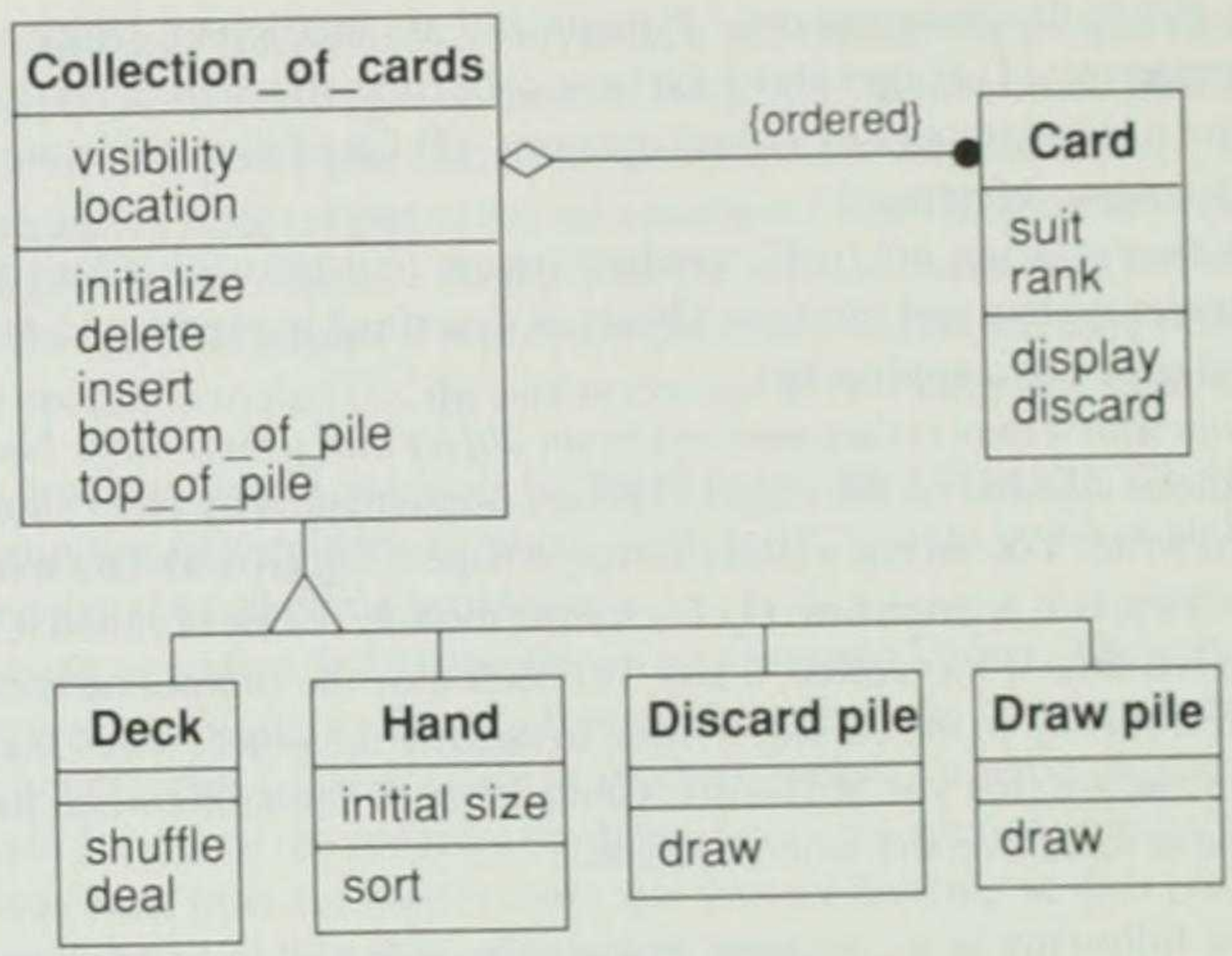


Figure E15.2 Object diagram for a card-playing program

15.13 (4) Implement the *insert* operation in Figure E15.2 with any object-oriented language. There is one argument, *card*, to the operation which inserts the card into the *Collection_of_cards*. A card is inserted at the top of a *Deck*, *Hand*, *Discard pile*, or *Draw_pile*.

You may wish to refer to exercise 8.5 for background for exercises 15.14-15.16.

15.14 (4) Using an object-oriented language, implement all associations involving the classes *Box*, *Link*, *Line_segment*, or *Point* in Figure E15.3. Note that the editor allows links only between pairs of boxes.

15.15 (5) Implement the *cut* operation on the class *Box* in Figure E15.3 using any object-oriented language. Propagate the operation from boxes to attached link objects. Update any associations that are involved. Be sure to recover any memory that is released by the operation. You may assume another routine will update the display.

15.16 (5) Implement a routine using any object-oriented language that will create a logical link between two boxes. (See Figure E15.3.) Inputs to the routine are two boxes and a list of points. The routine should update associations and create object instances as needed. You may assume another routine will update the display.

15.17 (8) Using any object oriented language, implement the following queries on Figure E15.3:

a. Given a box, determine all other boxes that are directly linked to it.
b. Given a box, find all other boxes that are directly or indirectly linked to it.
c. Given a box and a link, determine if the link involves the box.
d. Given a box and a link, find the other box logically connected to the given box through the other end of the link.
e. Given two boxes, determine all links between them.
f. Given a selection and a sheet, determine which links connect a selected box to a deselected box.

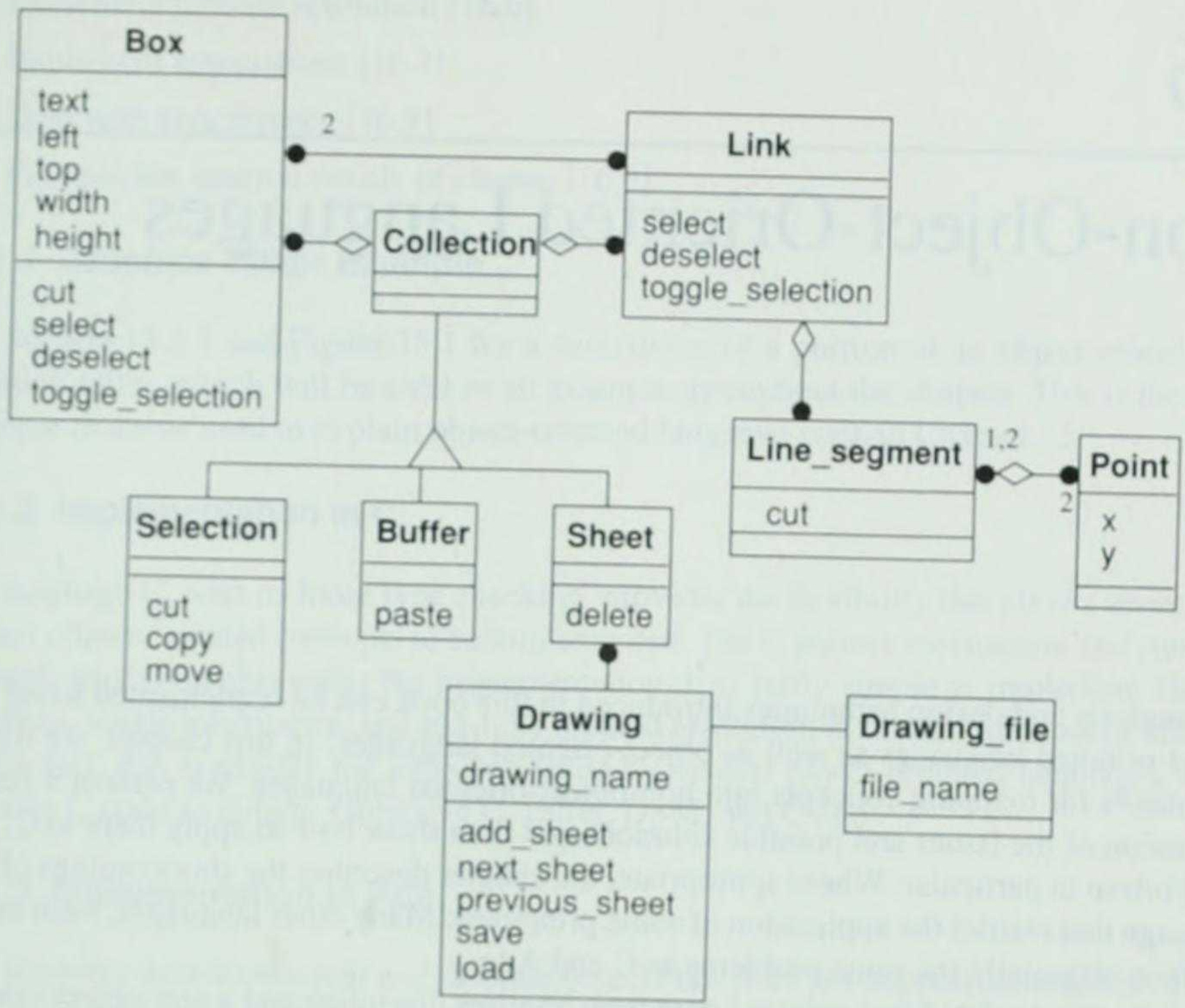


Figure E15.3 Object diagram for a diagram editor

g. Given two boxes and a link, produce an ordered set of points. The first point is where the link connects to the first box, the last point is where the link connects to the second box and intermediate points trace the link.

15.18 (Project) Write a program for simulating evolution, as described in the Computer Recreations column in the May 1989 issue of *Scientific American*, using any object oriented language.

16

Non-Object-Oriented Languages

The analysis and design techniques introduced in this book can be implemented using non-object-oriented languages as well as object-oriented languages. In this chapter, we discuss techniques for mapping concepts into non-object-oriented languages. We present a general discussion of the issues and possible solutions and then show how to apply them to C, Ada, and Fortran in particular. Where appropriate, the chapter describes the shortcomings of each language that restrict the application of some principles. Many other languages, such as Pascal, face essentially the same problems as C and Ada.

Adhering to the object-oriented paradigm requires discipline and a non-object-oriented language offers no indications when you start to go astray. Nevertheless, if you must use a non-object-oriented language, you will benefit from object-oriented analysis and design, even if you lack some capabilities in the implementation language.

After reading this chapter, you will understand how to map an object-oriented design into a non-object-oriented programming language.

16.1 MAPPING OBJECT-ORIENTED CONCEPTS

Implementing an object-oriented design in a non-object-oriented language requires basically the same steps as implementing a design in an object-oriented language [Section 15.1]. The programmer using a non-object-oriented language must map object-oriented concepts into the target language, whereas the compiler for an object-oriented language performs such a mapping automatically. The steps required to implement a design are:

- Translate classes into data structures [16.2]
- Pass arguments to methods [16.3]
- Allocate storage for objects [16.4]
- Implement inheritance in data structures [16.5]

- Implement method resolution [16.6]
- Implement associations [16.7]
- Deal with concurrency [16.8]
- Encapsulate internal details of classes [16.9]

16.1.1 Graphics Editor Example

See Section 15.1.1 and Figure 15.1 for a description of a portion of an object model for a graphics editor, which will be used as an example throughout the chapter. This is the same example that was used to explain object-oriented language code in Chapter 15.

16.1.2 Implementation in C

The language C, with its loose type checking, provides the flexibility that allows several important object-oriented concepts to be implemented. The C pointer mechanism and run-time memory allocation also assist the implementation. It is fairly simple to implement classes, instances, single inheritance, and run-time method resolution in C with little loss of efficiency. (In fact, this is exactly the approach taken by several object-oriented languages which generate C code as output: Objective C, Eiffel, DSM, and C++.)

16.1.3 Implementation in Ada

Ada supports data abstraction and discrete objects but does not support inheritance. Therefore it cannot be considered object-oriented. The main obstacles to a straightforward mapping come from Ada's rigid typing system and lack of procedure pointers. From a practical point of view, though, an object-oriented design constructed using the OMT methodology can be implemented in Ada, although somewhat less easily than in C. Ada offers excellent encapsulation facilities that make possible development of large systems.

16.1.4 Implementation in Fortran

The lack of modern data structuring constructs and dynamic memory allocation makes Fortran difficult, if not impossible, to use for applications requiring complex nonnumeric data structures. On the other hand, Fortran is still widely used for numeric applications, and its provision for complex numbers and multidimensional arrays are unmatched in most "modern" programming languages. Object-oriented analysis and design can be used profitably, but you eventually have to translate data structures into arrays, the only data structuring mechanism in Fortran. The Fortran programmer must manually translate many constructs that would be supported directly by C or Ada.

If your application requires many numerical computations, including matrices and complex numbers, then Fortran does have advantages over most other languages. However, you may consider coding the numerical computations in Fortran and coding the other parts—set up, symbolic manipulation, data network traversal, user interaction—in another language.

The Fortran programs can be considered computational utilities and the non-Fortran programs can manage the overall system.

16.1.5 Other Languages

Pascal is much like Ada in its limitations. It has strong type-checking that is rigid, and it lacks function pointer variables.

Lisp, on the other hand, is such a malleable language that almost anything is possible. The implementation suggestions for C could easily be implemented in Lisp. On the other hand, many Lisp programmers will have access to an object-oriented Lisp, such as CLOS or Flavors, so there is less need to provide LISP workarounds.

16.2 TRANSLATING CLASSES INTO DATA STRUCTURES

Normally you will implement each class as a single contiguous block of attributes—a record structure. Each attribute in a class becomes an element in the record. Each attribute has a declared type, which can be a primitive type, such as *integer*, *real*, or *character*, or can be a structured value, such as an embedded record structure or a fixed-length array.

An object has state and identity and is subject to side effects. A variable that identifies an object must therefore be implemented as a sharable reference and not simply by copying the values of an object's attributes. A reference can be implemented as a memory address or an array index but in any case must permit the sharing (aliasing) of a single object in memory from multiple possible references.

16.2.1 Translating Classes into C Struct Declarations

Each class in the design becomes a C *struct*. Each attribute defined in the class becomes a field of the C *struct*. The structure for the *Window* class is declared as:

```
struct Window
{
    Length xmin;
    Length ymin;
    Length xmax;
    Length ymax;
};
```

Length is a C type (not a class) defined with a C *typedef* statement to provide greater modularity in the definition of values:

```
typedef float Length;
```

In C, an object reference can be represented by a pointer to its object record:

```
struct Window * window;
Length xl = window->xmin;
```

An object can be allocated statically, automatically (on the stack), or dynamically (on the heap) because C can compose a pointer to any object, including one embedded within another structure.

16.2.2 Translating Classes into Ada Record Definitions

The Ada code for a class is similar to the C code. Ada uses a *record* type:

```
type Length is new FLOAT;
type WindowRecord is record
    xmin: Length;
    ymin: Length;
    xmax: Length;
    ymax: Length;
end record;
```

In Ada, an object reference, or pointer, can be represented by an *access type*:

```
type Window is access WindowRecord;
```

We use Ada access types to represent object references when the number of objects of a given class is unbounded. In this case, the object records are allocated at run time using the Ada *new* allocator, as shown below:

```
aWindow: Window := new WindowRecord;
```

Access types are needed for dynamic allocation and for associations between objects. The use of access types and dynamic allocation can be omitted when an object value is embedded within another object and does not have a distinct identity. In any case, the components within the record represent object attributes, and are accessed as follows:

```
xl: Length := aWindow.xmin;
```

16.2.3 Translating Classes into Fortran Arrays

Fortran has no user-defined data structure except the array, so the programmer must simulate records. A class is represented as an implicit group of arrays, one for each attribute in the class. The arrays all have the same dimension, which must be large enough to accommodate all the objects of the class that will exist at any time, since Fortran does not support dynamic memory allocation. The index into the arrays represents an object ID *within the given class*. Object IDs for different classes overlap, so you must know the class of a variable in the program (or determine it as explained later under inheritance). The arrays for a single class can be organized into a common block. For example, the storage for windows in the previous example would look like:

```
COMMON /WINDOW/ XMIN, YMIN, XMAX, YMAX, NWINDOW
REAL XMIN(1000), YMIN(1000), XMAX(1000), YMAX(1000)
INTEGER NWINDOW
```

The programmer must maintain a counter of the number of objects of a given class that have been allocated (NWINDOW) so that new object IDs can be assigned.

Fortran has no way to define new types, so you cannot define *Length* symbolically, but must expand it in terms of primitive data types (such as *REAL*, *COMPLEX*, *INTEGER*, *LOGICAL*, and *CHARACTER*).

Standard Fortran restricts the length of identifiers but many Fortran compilers allow identifier names of 32 or more characters. In our examples, we assume that long names including underscores are allowed. If your compiler requires short names, you must map meaningful names to abbreviations yourself at a considerable cost in readability.

In Fortran, an object can be represented by its index into the attribute arrays for its class. Attributes are accessed by indexing the corresponding array. The program must know the class of the object:

```
INTEGER AWINDOW
REAL X1
X1 = XMIN(AWINDOW)
```

16.3 PASSING ARGUMENTS TO METHODS

Every method has at least one argument, the implicit *self* argument. In a non-object-oriented language, the argument must be made explicit, of course. Methods may also have additional objects as arguments. Some of the arguments may be simple data values and not objects. In passing an object as an argument to a method, a reference to the object must be passed if the value of the object can be updated within the method. If the method is a query that simply extracts information from an object without modifying it, then a call-by-value mechanism can be used if the language permits. Passing all arguments as references is more uniform, however.

We advise a consistent naming convention for method function names. One that we have found useful in C is to concatenate the class name, two underscores, and the operation name. (The two underscores separate the class name from the operation name, each of which may contain single underscores.) Standard Fortran is overly restrictive on names, but many compilers permit long names and the use of underscores. Ada names may not contain double underscores and they are case-insensitive, but names can be as long as desired. Ada variable and argument names must differ from type names.

16.3.1 Passing Arguments in C

In C, an object should always be passed by pointer. Although C permits structures to be passed by value, passing a pointer to an object structure is usually more efficient and provides a uniform access mechanism for both query and update operations.

```
Window__add_to_selections (self, shape)
    struct Window * self;
    struct Shape * shape;
```

16.3.2 Passing Arguments in Ada

In Ada, an object can be passed as an access type:

```
type Window is access WindowRecord;
procedure Window_add_to_selections (self: Window;
                    aShape: Shape);
```

The object access parameter, *self*, is (by default) an *in* parameter because its value will not be changed in the method. If the value of a variable will be updated within the method, it must be passed as an *in out* parameter. Note, however, that the access type (or pointer) referencing an object will not be modified within the method, even though the record-object that it points to may be. An access type permits modification of the attributes of the referenced object.

Access to the same object concurrently from different tasks is likely to cause inconsistencies. If concurrent access to an object is possible, objects should be encapsulated entirely within a task and accessed only from within that task. This will serialize concurrent access, guaranteeing that each operation is completed before the next one is begun. In Section 9.3 we discussed how to identify inherently concurrent tasks in a system.

16.3.3 Passing Arguments in Fortran

In Fortran, an object can be passed as an index into the arrays for its class:

```
SUBROUTINE WINDOW__ADD_TO_SELECTIONS (SELF, SHAPE)
INTEGER SELF, SHAPE
```

An object whose contents will be queried but not updated can be passed as a list of attribute values if desired, although the index is usually shorter and therefore more convenient:

```
FUNCTION CIRCLE__PICK (X0, Y0, RADIUS, X, Y)
LOGICAL CIRCLE__PICK
REAL X0, Y0, RADIUS, X, Y
```

16.4 ALLOCATING OBJECTS

Objects can be allocated statically (at compile time), dynamically (from a heap), or on a stack. Statically allocated objects are implemented as global variables allocated by the compiler. Their lifetime is the duration of the program. They can be useful for system-level objects or constants, but it is poor object-oriented practice to use too many global variables because they defeat modularity.

Most temporary and intermediate objects will be implemented as stack-based variables (such as C *automatic* variables or Pascal *local* variables). The advantage of stack-based variables is that they are automatically allocated and deallocated. The programmer must ensure that no references to a stack-based object remain after the declaring block has been exited. Stack-based variables are not suitable for general objects whose lifetime outlasts the procedure that created them. Using such techniques can lead to subtle bugs which are hard to find.

Dynamically allocated objects are needed when the number of them is not known at compile time. A general object can be implemented as a data structure allocated on request at run time from a heap (a block of global storage managed by the storage allocator). Storage for dynamically allocated objects is requested explicitly by a special operator: *malloc* in C, *new* in Pascal or Ada, *make* in CommonLisp. Once allocated, dynamic objects persist until they are explicitly deallocated, so pointers to them can be stored in other objects. Knowing when to deallocate an object is not always simple, but this problem is shared with many object-oriented languages as well. Some languages, such as Lisp, provide garbage collection, which removes the burden of deallocation from the programmer and avoids the danger of dangling pointers.

16.4.1 Allocating Objects in C

Global objects can be declared as top-level *struct* variables. They can be initialized at compile time:

```
struct Window outer_window = {0.0, 0.0, 8.5, 11.0};
```

When calling a method, the address of the variable (`&outer_window`) must be passed.

Most objects should be allocated dynamically using *malloc* or *calloc*:

```
struct Window * create_window (xmin, ymin, width, height)
    Length xmin, ymin, width, height;
{
    struct Window * window;
    window= (struct Window *)malloc (sizeof (struct Window));
    window->xmin = xmin;
    window->ymin = ymin;
    window->xmax = xmin + width;
    window->ymax = ymin + height;
    return window;
}
```

When an object is no longer needed, it must be deallocated using the C *free* function. Make sure that there are no pointers to it before deallocating it. Also be sure to free any component objects that are pointed to by its instance variables.

Temporary and intermediate objects can be allocated as ordinary C *automatic* variables within a function body or block. When calling a method, the address of the variable must be passed. Stack-based variables cannot be used when their lifetime outlasts the function that created them.

16.4.2 Allocating Objects in Ada

Global objects can be allocated as constant access types and optionally initialized:

```
main_window: constant Window :=
                    new WindowRecord'( 0.0, 0.0, 8.5, 11.0 );
```

Most objects will be allocated dynamically using *new*. Higher-level functions can be written that allocate a new object and initialize some of its attributes:

```
function create_window (xmin, ymin, width, height: Length)
    return Window is
begin
    return new WindowRecord'(xmin, ymin,
                xmin+width, ymin+height);
end;
```

An Ada compiler may provide garbage collection to recover allocated storage that can no longer be referenced. In most cases, however, you must explicitly deallocate access types, just as with C, using the generic function *UNCHECKED_DEALLOCATION*.

16.4.3 Allocating Objects in Fortran

As a programmer you must explicitly allocate new objects from predefined arrays. Programming a dynamic application would be much harder in Fortran than in C or Ada because you must implement the allocation mechanism yourself (by simulating a heap):

```
FUNCTION CREATE_WINDOW (X1, Y1, WIDTH, HEIGHT)
COMMON /WINDOW/ XMIN, YMIN, XMAX, YMAX, NWINDOW
REAL XMIN(1000), YMIN(1000), XMAX(1000), YMAX(1000)
INTEGER NWINDOW
INTEGER CREATE_WINDOW
REAL X1, Y1, X2, Y2
NWINDOW = NWINDOW + 1
XMIN(NWINDOW) = X1
YMIN(NWINDOW) = Y1
XMAX(NWINDOW) = X1 + WIDTH
YMAX(NWINDOW) = Y1 + HEIGHT
CREATE_WINDOW = NWINDOW
RETURN
END
```

16.5 IMPLEMENTING INHERITANCE

There are several ways to implement data structures for inheritance in a non-object-oriented language:

- *Avoid it.* Many applications do not require inheritance. Many other applications have only a few classes requiring inheritance. Those classes not needing inheritance can be implemented as simple records. [Booch-86] describes how to do object-oriented designs in this manner for Ada.
- *Flatten the class hierarchy.* Use inheritance during design but expand each concrete class as an independent data structure during implementation. Each inherited operation must be reimplemented as a separate method on each concrete class. Flattening the hierarchy introduces duplication, but the use of language constructs such as Ada generics or C macros can help. One useful technique is to group some inherited attributes into a record type and embed the record in each concrete class to reduce the number of duplicated lines in each declaration.

- *Break out separate objects.* Instead of inheriting common attributes from a superclass, a group of attributes can be pulled out of all the subclasses and implemented as a separate object with a reference to it stored within each subclass. Grouping attributes under a separate type permits a single method to be written to manipulate them. The subclasses must delegate operations to the referenced object.

These approaches avoid the implementation of inheritance. When inheritance is actually needed, the best implementation depends on the language and the application, as explained below.

16.5.1 Implementing Inheritance in C

To handle single inheritance, embed the declaration for the superclass as the first part of each subclass declaration. The first field of each *struct* is a pointer to a class descriptor object shared by all direct instances of a class. The class descriptor object is a *struct* containing the class attributes, including the name of the class (optional) and the methods for the class. Its format will be discussed later.

The class *Shape* is an abstract class, with concrete subclasses *Box* and *Circle*. The C declarations for classes *Shape*, *Box*, and *Circle* are as follows:

```
struct Shape
{
   struct ShapeClass * class;
   Length x;
   Length y;
};
struct Box
{
   struct BoxClass * class;
   Length x;
   Length y;
   Length width;
   Length height;
};
struct Circle
{
   struct CircleClass * class;
   Length x;
   Length y;
   Length radius;
};
```

A pointer to a *Box* or *Circle* structure can be passed to a C function expecting a pointer to a *Shape* structure because the first part of the *Box* or *Circle* structure is identical to the *Shape* structure. (To be strictly correct, you should *cast* such an argument to be a pointer to *Shape*, but most compilers won't notice the difference anyway.) Because the prefix of the structure is the same as the superclass structure, the superclass method simply ignores the extra fields on the end. For example, a pointer to *Box* is interpreted as a pointer to *Shape* in the following call:

```
struct Box * box;
struct Window * window;
Window__add_to_selections (window, box)
```

The first field of each structure is a pointer to the class descriptor for the actual class of each object instance. This field is needed only if run-time method resolution is to be done. Its form and use are described in Section 16.6.1.

Multiple inheritance cannot be implemented using this approach because a subclass with two superclasses cannot align its attributes with both of them. The best approach is to eliminate multiple inheritance using the workarounds discussed in Chapter 4.

16.5.2 Implementing Inheritance in Ada

Ada's *variant records* can be used to implement inheritance (single inheritance only). A variant record is a record structure with several alternatives. Each record contains a *discriminant*, which is a component to distinguish the alternate forms of the record. Declare a variant record for the root of each class hierarchy, with one variant for each subclass. The attributes in the superclass are common to all the variants. Multiple levels of variant nesting correspond to multiple levels of subclasses. A single method can operate on all variants of a single root class because they are all the same Ada type.

The root classes must be chosen carefully because it is usually inefficient to make everything a variant of class *object*. This approach works well when the hierarchy is shallow. The advantage of variant records is that no trickery is needed to use them, and all access is syntactically and semantically correct:

```
type ItemClass is (GroupClass, BoxClass, CircleClass);
subtype ShapeClass is ItemClass range BoxClass..CircleClass;
type ItemRecord (class: ItemClass) is record
    case class is
        when GroupClass => null;
        when ShapeClass =>
            x: Length;
            y: Length;
            case class is
                when BoxClass =>
                    width: Length;
                    height: Length;
                when CircleClass =>
                    radius: Length;
                when others => null;
            end case;
    end case;
end record;
type Item is access ItemRecord;
subtype Group is Item;
subtype Shape is Item;
subtype Box is Item;
subtype Circle is Item;
```

The record type above represents the inheritance tree from Figure 15.1 consisting of the classes *Item*, *Shape*, *Box*, and *Circle*. When an object is allocated from this record type, it is *constrained* to be a specific variant, representing exactly one of the four classes. The dynamic allocator of a good Ada compiler will allocate only the space required by the attributes of that class of object. The following code creates an object of the *Group* class:

```
item_group: Group := new ItemRecord (GroupClass);
```

16.5.3 Implementing Inheritance in Fortran

Fortran has no user-defined records, therefore variant records are not possible. One approach is to implement a given class hierarchy as a *universal record*. A universal record contains an attribute for every attribute found in any descendent class. Those attributes that do not apply in a particular case are simply ignored. This approach wastes storage but may be considered if the variant part of the record is small.

```
COMMON /SHAPE/X,Y,WIDTH,HEIGHT,RADIUS,NSHAPE
REAL X(1000),Y(1000),WIDTH(1000),HEIGHT(1000)
REAL RADIUS(1000)
INTEGER NSHAPE
```

A less wasteful approach, the Fortran equivalent of the variant record, is to separate a class into subclasses, each implemented as a separate class with its own arrays and object indexes. The original class is represented by a pair of integer arrays: One array stores a code for the subclass and the other stores the index of the object within its corresponding subclass array. The programmer must first assign an integer code to each class in the system. In this example we define class *Item* as either a *Shape* (1000 maximum) or a *Group* (100 maximum). Common block *CLASSES* defines the integer code for each class.

```
COMMON /ITEM/ ITEM_CLASS, ITEM_ID, NITEM
INTEGER ITEM_CLASS(1100), ITEM_ID(1100)
INTEGER NITEM/0/
COMMON /CLASSES/ GROUP, BOX, CIRCLE
INTEGER GROUP/1/, BOX/2/, CIRCLE/3/
```

When a new object is created, an index value must be allocated from the superclass as well as the subclass. For example, the following code creates a new circle:

```
FUNCTION CREATE_CIRCLE (X0, Y0, RADIUS0)
common blocks for ITEM, SHAPE, and CLASSES go here
INTEGER CREATE_CIRCLE
NSHAPE = NSHAPE + 1
X(NSHAPE) = X0
Y(NSHAPE) = Y0
RADIUS(NSHAPE) = RADIUS0
NITEM = NITEM + 1
ITEM_CLASS(NITEM) = CIRCLE
ITEM_ID(NITEM) = NSHAPE
CREATE_CIRCLE = NITEM
END
```

This same technique of representing an object in a generic situation by a pair of values, one indicating the class and the other indicating a class-specific ID, can be used by Ada, Pascal, and other languages.

16.6 IMPLEMENTING METHOD RESOLUTION

In an object-oriented language, a single operation may be implemented by many methods (polymorphism), depending on the run-time class of the object. Run-time method resolution is one of the main features of an object-oriented language that is lacking in a non-object-oriented language. Method resolution can be handled in several ways:

- *Avoid it.* If each operation is defined only once in the class hierarchy and not overridden, then there is no polymorphism and no need for run-time method resolution. Methods can still be inherited; all subclasses share the methods of a superclass, but they cannot override them. This advice is not as outrageous as it may seem. Casual overriding of operations to change their behavior is semantically questionable because a subclass should not modify the meaning of an inherited operation (Section 4.3.1).
- *Resolve methods at compile time.* If the class of each object is known at compile time, then the correct method can be determined and called directly, avoiding the need for run-time method resolution. It is sufficient to know that the object class is a descendent of the lowest class that overrides the method. In many applications, the classes of most objects are known at compile time. Of course, if the class hierarchy is changed or a new method is defined, then the programmer must manually reevaluate the method resolution and substitute a new function call.
- *Resolve methods at run time.* If you have a collection of objects of mixed classes to which you need to apply an abstract operation, then dynamic method resolution is required. You must test the class of the object to determine the correct method. An object-oriented language automatically makes this test. The C language supports an elegant and efficient technique of performing methods resolution using a predefined structure containing method pointers. Other languages, such as Ada and Fortran, do not support method pointers, so you must use *case* statements within a single dispatch method.

16.6.1 Method Resolution in C

Any methods that you can resolve at compile time can be implemented as straight C function calls. Many operations are implemented only once as methods and never overridden, so they do not need run-time method resolution. For example, all methods of *Window* are unique. The most general approach, however, is to define a class descriptor object for each class containing a pointer to the method function for each operation visible from the class, including inherited operations. Each class descriptor is a C *struct* containing all the operations defined in the class or inherited from a superclass. Each class descriptor contains inherited operations from its superclass as its first part, just as the *struct* for an object instance contains inherited attributes. The following code shows the declaration for the class descriptors for

Item, *Shape*, *Box*, and *Circle*. We append the word *Class* to the class name in declaring the *struct* for the class descriptor.

```
struct ItemClass
{
    char * class_name;
    void (* cut) ();
    void (* move) ();
    Boolean (* pick) ();
    void (* ungroup) ();
};
struct ShapeClass
{
    char * class_name;
    void (* cut) ();
    void (* move) ();
    Boolean (* pick) ();
    void (* ungroup) ();
    void (* write) ();
};
struct BoxClass
{
    char * class_name;
    void (* cut) ();
    void (* move) ();
    Boolean (* pick) ();
    void (* ungroup) ();
    void (* write) ();
};
struct CircleClass
{
    char * class_name;
    void (* cut) ();
    void (* move) ();
    Boolean (* pick) ();
    void (* ungroup) ();
    void (* write) ();
};
```

The class descriptor *struct* defines the names of the operations visible to a class, but you still have to define and initialize a class descriptor object for each class. Each descriptor object is a global variable, the single instance of its class descriptor *struct*. (We use the same name for the class descriptor object as the C *struct* because they belong to separate C name spaces and cannot be confused.) You must initialize each field of the class descriptor object with the name of the C method function defined or inherited by the class. You can determine the correct method by examining the object model. For example, class *Box* inherits operation *move* from class *Shape* but overrides operations *pick* and *write* with its own methods:

```
struct BoxClass BoxClass =
{
    "Box",
    Shape__cut,
    Shape__move,
    Box__pick,
    Shape__ungroup,
    Box__write
};
struct CircleClass CircleClass =
{
    "Circle",
    Shape__cut,
    Shape__move,
    Circle__pick,
    Shape__ungroup,
    Circle__write
};
```

If a class has class attributes, they can also be stored in the class descriptor as additional fields. For example, we store the name of each class in the class descriptor object for use during debugging, printing, or other uses.

Note that class descriptor objects are needed only for concrete classes and are unnecessary for abstract classes, such as *Shape*. The only use of the class descriptor objects is to store the methods and class variables available to objects. When a new object instance is created, the address of its class descriptor object is stored as the *class* field in the object record. The information about the class of an object, including its name, class attributes, and methods, can be obtained at run time from the objects's *class* field.

We omitted operations *draw* and *erase* from the class descriptor. These operations are implemented as methods only in the *Shape* class; they call the *write* operation with either the foreground or background color. Because they are not overridden, they can be implemented as direct function calls. Of course, we can include them in the class descriptor for uniformity and future extensions.

The *class* field of each object must be initialized with a pointer to the class descriptor:

```
struct Circle * create_circle (x0, y0, radius0)
    Length x0, y0, radius0;
{
    struct Circle * new_circle;
    new_circle =
          (struct Circle *) malloc (sizeof (struct Circle));
    new_circle->class = &CircleClass;
    new_circle->x = x0;
    new_circle->y = y0;
    new_circle->radius = radius0;
    return new_circle;
}
```

If an operation must be resolved at run time, the class descriptor object is used to determine the correct C function. The class descriptor object is obtained from the object and indexed by the name of the operation. For example, to call the *pick* operation on an unknown *Shape* requires the following code:

```
struct Shape *shape;
Length x, y;
Boolean status;
status = (*shape->class->pick) (shape, x, y);
```

Note that the target object must be included twice, once to find the method and once as the first argument of the method. Dynamic method resolution requires two more memory accesses and one more add than a direct function call. Compared to the overhead of calling an average function, the added cost is really insignificant.

16.6.2 Method Resolution in Ada

Ada permits overloading subprogram names based on the types of the subprogram arguments, but overloading must be resolved at compile time, and cannot be used for cases requiring run-time type discrimination. Ada has no procedure pointers, so a method table cannot be constructed. The best approach for dynamic method resolution in Ada is to define a "dispatch method" for each operation, a single procedure shared by all classes that is called whenever run-time method resolution on the operation is required. The dispatch method contains a *case* statement that tests the actual class of the object (represented by the discriminant of the variant record) and calls the corresponding implementation method directly. If you add a new subclass requiring a new method, you have to add it to the case statement in the dispatch method, but none of the client code that uses the operation need be modified. By using dynamic method resolution only when required, the number of dispatch methods can be reduced to a minimum.

The following code shows the implementation of the *move* operation by two specific methods followed by the more general dispatch method. We assume that the subclasses have been implemented as a variant record and declared as Ada subtypes:

```
procedure Shape_move (self: Shape; dx, dy: Length) is
begin
    self.x := self.x + dx;
    self.y := self.y + dy;
end;
procedure Group_move (self: Group; dx, dy: Length) is
    the_list: ItemList;
begin
    the_list := self.items;
    for i in 1..the_list.count loop
        move (the_list.values(i), dx, dy);
    end loop;
end;
```

```
procedure move (self: Item; dx, dy : Length) is
begin
    case self.class is
        when ShapeClass => Shape_move (self, dx, dy);
        when GroupClass => Group_move (self, dx, dy);
        when others => null;
    end case;
end;
```

The component named *items* in the *ItemRecord* implements the association between a group and its items, as described later in Section 16.7. The *ItemList* type is a dynamic array of items.

16.6.3 Method Resolution in Fortran

Methods for objects whose class is known at compile time can be resolved to direct procedure calls. The remaining objects must have class numbers stored in them (see the common block *CLASSES* in Section 16.5.3). A dispatch procedure can be written for each operation, with the class number and object index of each object as parameters. The dispatch procedure contains a computed goto or a conditional on each possible class value to call the correct method:

```
      FUNCTION PICK (CLASS, ID, PX, PY)
      LOGICAL PICK
      LOGICAL GROUP_PICK, BOX_PICK, CIRCLE_PICK
      INTEGER CLASS, ID
      GOTO (100, 200, 300) CLASS
      PICK = .FALSE.
      RETURN
  100 PICK = GROUP_PICK (ID, PX, PY)
      RETURN
  200 PICK = BOX_PICK (ID, PX, PY)
      RETURN
  300 PICK = CIRCLE_PICK (ID, PX, PY)
      RETURN
      END
```

16.7 IMPLEMENTING ASSOCIATIONS

Implementing associations in a non-object-oriented language presents the same two possibilities as in an object-oriented language: mapping them into pointers or implementing them directly as association container objects. We have discussed these choices as part of object design in Section 10.7. To summarize their implementation:

- *Mapping associations to pointers.* The traditional approach to implementing binary associations is to map each role of an association into an object pointer stored as a field of the

source object record. Each object contains a pointer to an associated object (if the multiplicity is "one" or "zero-one") or a pointer to a set of associated objects (if the multiplicity is greater than one). A set may be implemented using any appropriate available data structure—often a linked-list or array, but a hash table or binary tree may be used when the greater execution efficiency warrants the extra programming effort.

The association may be implemented in one direction or in both directions. If it is traversed in only a single direction, it may be implemented as a pointer from one object to another. If it is traversed in both directions, it must be implemented using pointers in both associated objects; the two pointers must be kept mutually consistent. It is best not to directly update the pointers within general application code but to write a utility procedure to add or delete links by updating both cross-linked pointers. You need not be so careful when accessing values from the association—the pointers can be accessed directly from either object—although there are maintenance advantages to hiding the implementation within an access procedure.

• *Implementing association objects.* An association can be implemented directly as a data structure. If an association relates more than two classes then it cannot be mapped into pointers, and a separate object must be used in any case. In its basic form (and its mathematical definition), an association is simply a set of records, each containing one object ID for each associated class. The simplest approach is to implement an association as an array or list of records or record pointers. To traverse a binary link, the list is searched to find a link in which one field is equal to the source object, and the value of the other field is returned as the target object.

The access of values from an unordered list is inefficient because the list must be linearly searched. A more efficient implementation is to sort the list on one or more key fields or to hash the elements on one or more key fields. Such a data structured is said to be *indexed* on the key fields. If the association is to be traversed or accessed from more than one direction, then multiple indexes must be built and maintained. Whenever an element is added, modified, or deleted, all the indexes must be updated accordingly. Depending on the relative numbers of updates and accesses in each direction of traversal, it is more efficient to either index the association on a single key (and search it linearly when an access occurs on a different key) or to index the association on multiple keys (and update several indexes when a link changes).

16.7.1 Implementing Associations in C

A binary association is usually implemented as a field in each associated object, containing a pointer to the related object or to an array of related objects. For example, the many-to-one association between *Item* and *Group* would be implemented as:

```
struct Item
{
    struct ItemClass * class;
    struct Group * group;
};
```

```
struct Group
{
    struct GroupClass * class;
    int item_count;
    struct Item ** items;
};
```

Other data structures, such as a linked list or a hash table, can also be used to store sets of objects. In this example, a group is created from a set of selected items. The memory for the *items* pointer can be allocated all at once, since the number of items in a group does not change. If it were possible to add a single new item to a group, then both pointers must be updated:

```
Group__add_item (self, item)
    struct Group * self;
    struct Item * item;
{
    item->group = self;
    self->items = (struct Item **) realloc (self->items,
            ++self->item_count * sizeof (struct Item *));
    self->items [self->item_count-1] = item;
}
```

You can build more sophisticated data structures to avoid calling *realloc* more than necessary. It is useful to have available a library of generic container objects, such as variable-length arrays, lists, and hash tables.

16.7.2 Implementing Associations in Ada

A binary association is normally implemented in Ada as mutual pointers between two objects. A pointer is implemented as an Ada *access* type. For example, the many-to-one association between *Item* and *Group* would be implemented by adding *group* and *items* to the fields of the *Item* type:

```
type ItemListRecord (capacity: Positive);
type ItemList is access ItemListRecord;

type ItemRecord (class: ItemClass) is record
    theGroup: Group := null;
    case class is
        when GroupClass =>
            items: ItemList := null;
        when ShapeClass =>
            -- [Code deleted: See Section 16.5.2]
    end case;
end record;

type ItemVector is array (Positive range <>) of Item;
```

```
type ItemListRecord (capacity: Positive) is record
    count: Natural := 0;
    values: ItemVector (1..capacity);
end record;
```

The type *ItemListRecord* is can be allocated with any initial size. The collection of items in a group can be expanded without bound if the ItemList is reallocated at a larger size when its capacity is outgrown.

The collection of related objects could instead can be stored as a linked list in which each item in the set contains a link to the next item in the set. In a linked list every object that may appear in a set must have an additional field to store the link to the next object in the set. The related object contains an access pointer to the first object in the list.

16.7.3 Implementing Associations in Fortran

The same approaches are possible in Fortran as in other languages. Represent an association by mutual pointers. An additional integer array is added to each class to hold the ID of an associated object. In the case of multiplicity "many," a set of related values is needed. Because Fortran does not have dynamic memory allocation, a linked list is usually the easiest way to represent a set of objects. To the common block *GROUP*, we add array *ITEMS* containing the index of the first item in the list of items related to the group object. To the common block *ITEM*, we add array *NEXT_IN_GROUP* to link together the items in one list of related items and also array *GROUP* containing the index of the related group. Null relationships are indicated by an index of zero.

```
COMMON /ITEM/ITEM_CLASS,ITEM_ID,NEXT_IN_GROUP,GROUP, NITEM
INTEGER ITEM_CLASS(1100),ITEM_ID(1100),NEXT_IN_GROUP(1100)
INTEGER GROUP(1100)
INTEGER NITEM/0/
COMMON /GROUP/ITEMS,NGROUP
INTEGER ITEMS(100),NGROUP
```

16.8 DEALING WITH CONCURRENCY

Most languages do not explicitly support concurrency. Even in the languages that support concurrency, such as Ada, the cost of a task is usually large compared to a statement or a procedure call, so it cannot be used freely. Although many algorithms could be implemented in parallel if suitable machines and languages were available, algorithms can generally be implemented sequentially with no loss of power and with less danger of programming errors. Concurrency is usually needed only when the external interactions force it, that is, when more than one external event can occur, and the behavior of the program depends on their timing.

Multiple tasks are required to implement true concurrency. Note that interrupts are logically equivalent to multiple tasks because the flow of control in the interrupt is independent of the flow of control in the main program. If an object is treated as a passive piece of data

accessible to two or more tasks concurrently, then inconsistencies can occur because each task will have an incorrect picture of the state of the object. If, however, each object is "owned" by one task, then there is little danger of conflict in the use of data because an object cannot develop two concurrent threads of control. The use of shared data should therefore be avoided, and communication between separate tasks should occur through messages exchanged by tasks.

In a language lacking concurrent constructs, such as C or Pascal, there is no danger of violating these rules because the only way to create concurrency is to ask the operating system to create a parallel task. If the operating system supports interrupt routines, usually implemented by calling a procedure in the address space of the main program, then the interrupt procedure should not modify any objects accessible to the main program but should merely store its data in a reserved location, set a software flag, and allow the main program to come to a clean synchronization point for further processing.

In a language with tasking, such as Ada, there is danger of interference among concurrent tasks. The danger can be reduced by assigning each global variable to a single task and by allocating any dynamically created objects to a single task at a time. Communications among tasks should use the defined synchronization operations, such as the Ada entry call and *accept* statements.

There exist techniques, such as the use of semaphores, for sharing direct access to data among more than one task. These techniques are equivalent to simulating a task without actually creating one. They are less straightforward than the use of actual tasks and should be avoided in most cases.

16.9 ENCAPSULATION

Encapsulation of data representation and method implementation is one of the major themes of object-oriented programming. Object-oriented languages provide constructs to encapsulate implementation. Some of this encapsulation is lost when the programmer must manually translate object-oriented concepts into a non-object-oriented language, but you can still take advantage of the encapsulation facilities provided by the language.

16.9.1 Encapsulation in C

C has a reputation for encouraging loose programming style harmful to encapsulation. Nevertheless, you can take the following steps to improve encapsulation:

- Avoid the use of global variables.
- Package the methods for each class into a separate file. Only include the declarations for classes whose internal structure you need, such as ancestor classes of the current class. Do not access the fields of objects of different classes; call an access method instead.
- Treat objects of other classes as type "`void *`"; while technically illegal, most compilers treat all pointers the same (but watch out for word length or alignment problems on some machines).

16.9.2 Encapsulation in Ada

Ada provides strong enforcement of encapsulation by distinguishing between the external view of a package (the package specification) and the internal view of its implementation (the package body). The package specification and body may be compiled separately.

Private types provide further support for encapsulation. A type that is declared in the package specification is visible outside the package and is therefore said to be "exported" by the package. Ada provides mechanisms for declaring a *private* type whose implementation is hidden. External clients of a private type must access objects of that type only through operations defined by the package. The implementation of the type and its operations can be changed without impacting the clients.

A common style of Ada programming is to implement a class as a package that exports a private type. This kind of package is sometimes called a *type-manager* because it encapsulates the implementation of a type and operations that can be performed on it. Instances of the class can have attributes if the type is a record type. Operations on the class are implemented as subprograms that are visible in the package specification. Class variables can be implemented as variables declared in the context of the package specification or its body. Sometimes several classes are put in the same package to allow mutual visibility. The following code gives an example of this style of implementation, representing the *Item* class and its subclasses, *Shape*, *Box*, and *Circle* as private types exported by an Ada package.

```
package Item_pkg is
    type Length is new Float;
    type Item is private;
    subtype Shape is Item;
    subtype Box is Item;
    subtype Circle is Item;
    subtype Group is Item;

    function create_box (x, y, height, width: Length)
                return Box;
    function create_circle (x, y, radius: Length)
                return Circle;
    function create_group return Group;
    procedure add_item (self: Group; new_item: Item);
    procedure move (self: Item; dx, dy : Length);
    procedure draw (self: Item);
        -- [ Additional subprograms deleted ]
private
    type ItemClass is (GroupClass, BoxClass, CircleClass);
    subtype ShapeClass is ItemClass
                range BoxClass..CircleClass;
    type ItemRecord (class: ItemClass);
    type Item is access ItemRecord;
end Item_pkg;
```

The package specification above reveals only the external interface of the *Item* class and its subclasses. The declarations within the private section of the specification are hidden from external clients but are needed by the compiler in compiling external accesses. The implementation of these classes is hidden in the package body, which can be compiled separately:

```
package body Item_pkg is
    type ItemListRecord (capacity: Positive);
    type ItemList is access ItemListRecord;

    type ItemRecord (class: ItemClass) is record
       -- [ Code deleted: See Section 16.5.2 ]
    end record;
    type ItemVector is array (Positive range <>) of Item;
    type ItemListRecord (capacity: Positive) is record
       count: Natural := 0;
       values: ItemVector (1..capacity);
    end record;
       -- [ Subprogram bodies have been deleted ]
end Item_pkg;
```

Using the variant record implementation of inheritance requires that all subclasses of a given class be together in the same package because all subclasses are subtypes of the parent class. In Ada, strong type distinctions are not made between subtypes of a common base type. For purposes of type checking and overload resolution by the compiler they are viewed as the same base type.

16.9.3 Encapsulation in Fortran

The common blocks containing the storage of attributes for a class should be known only by methods on the class involved. Other classes should access an object only by its index value. All access to attributes of the class should be through an access procedure defined by the class. Because of the lack of pointers in Fortran, encapsulation is adequate. If a separate data manager is written in another language, encapsulation is even better because all objects in Fortran would be treated as encapsulated objects represented as coded integers.

16.10 WHAT YOU LOSE

Use of an object-oriented or non-object-oriented language is not a matter of functionality. By using the mappings described above, you can translate any object-oriented construct into a non-object-oriented language. Computational power is never an issue because any universal language can compute anything computable.

The real issue with languages is not power but expressiveness, convenience, protection from errors, and maintainability. An object-oriented language makes writing, maintaining,

and extending programs easier and safer because it performs tasks that the non-object-oriented language programmer must perform manually. These include:

- *Expressiveness:* The non-object-oriented programmer must map object-oriented operations, such as method calling or subclass declaration, into explicit operations which are often ugly.
- *Convenience:* The programmer must manually traverse the class hierarchy when calling methods or passing arguments. If the class hierarchy changes, then the programmer must manually reevaluate the traversals.
- *Error protection:* The programmer must ensure that all methods are included in a dispatch method or dispatch structure. The programmer must initialize a new object with its class. The programmer must avoid accessing internal attributes of other classes.
- *Maintainability:* If changes are made to the object declarations, the programmer must determine their effects on the code and modify the code accordingly. An object-oriented language provides and enforces modularity of classes that prevents changes from propagating through the entire program. The non-object-oriented programmer requires discipline to achieve similar modularity without the help of the language.

Nevertheless, if you must use a non-object-oriented language, we feel that an object-oriented design will simplify your task and provide greater flexibility and extensibility if you are willing to program in a disciplined manner.

16.11 CHAPTER SUMMARY

Object-oriented designs can be implemented in conventional programming languages but require programming discipline. Classes can be implemented as records in any modern language. In Fortran, classes should usually be implemented as implicit collections of arrays. Method arguments should be address pointers or array indexes to permit shared access to an object by different access paths (aliasing). Objects should be allocated from a heap, if possible. In Fortran the programmer must preallocate storage for the maximum number of objects expected. Inheritance can be simulated easily in C by duplicating the structure of a superclass in a subclass and simply passing a subclass object to a superclass method. Inheritance can be implemented with greater difficulty using variant records in strongly-typed languages such as Ada and Pascal. Run-time method resolution is clean and efficient in C, using function pointers stored in a class descriptor object for each class. In Ada or Fortran, case statements within a single dispatch method must be used for run-time method resolution. In most applications, many operations can be resolved at compile time, and run-time method resolution can be reserved for special cases. Associations are usually implemented as pointers from one object to another. Sets of related objects can be implemented as arrays, linked lists, or other data structures. An association can also be implemented as a distinct object containing a set of paired values. Concurrency is handled by operating system calls in most

languages except Ada, which contains explicit support for concurrent tasks. If concurrency exists, every object should be "owned" by a single task to prevent run-time conflicts. Ada contains excellent encapsulation facilities; other languages contain some facilities for encapsulation, but they rely upon the discipline of the programmer.

There is no support in non-object-oriented languages to generate object-oriented data structures and enforce proper use of the constructs. In particular, changes to code may force the programmer to manually retranslate the object-oriented constructs. Before deciding to accept these sacrifices, a design team should investigate using an object-oriented language such as C++ or Eiffel. A transition from C to C++ is not difficult for a programming team, yet offers object-oriented programming support. Nevertheless, if you must use a non-object-oriented language, use of object-oriented design will make your job easier. Any mappings that you must make are conceptual mappings that you must make in any case to use the language, and object-oriented notation makes them explicit.

implementation of concurrency	implementation of classes
encapsulation	implementation of inheritance
implementation of associations	implementation of method resolution

Figure 16.1 Key concepts for Chapter 16

BIBLIOGRAPHIC NOTES

Two workstation software libraries use C language object-based approaches to provide tools for creating window-based applications. SunView [Sun-86] uses attribute keywords and variable length argument lists to set and access the properties of windows, frames, panels, buttons, and sliders. The X Window toolkit intrinsics library [McCormack-88] allows software developers to use *widgets* that define the appearance and behavior of buttons, sliders, and menus. Attributes are called *resources*. An inheritance mechanism provides for the addition of new widgets by software developers.

Jacky [Jacky-86] describes a medical radiation therapy planning system implemented in Pascal. Objects are represented as Pascal *records*, using the fields in the record as attributes. A message passing mechanism in Pascal provides run-time procedure binding.

The Ada language is covered in [Barnes-89], which can serve either as a reference or as a tutorial manual for serious readers. For a concise description of graphical object-oriented design for Ada, see [Booch-86]. A good discussion of object-oriented programming in Ada is found in [Seidewitz-87], which includes treatment of the usefulness and limitations of generics. A taxonomy of general purpose data structures can be found in [Booch-87], showing Ada implementations that use generics. Before applying the Ada techniques, the reader is advised to obtain [AdaLRM], the official language reference manual.

REFERENCES

[AdaLRM] ANSI/MIL-STD-1815A. *Military Standard--Ada Programming Language*, Ada Joint Program Office, U.S. Department of Defense.

[Barnes-89] John G. P. Barnes. *Programming in Ada, 3rd edition*. Reading, Mass.: Addison-Wesley, 1989.

[Booch-86] Grady Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, March, 1986.

[Booch-87] Grady Booch. *Software Components with Ada*. Redwood City, Calif.: Benjamin/Cummings, 1987.

[Jacky-86] Jonathan Jacky, Ira Kalet. An object-oriented approach to a large scale scientific application. *OOPSLA'86* as *ACM SIGPLAN 21*, 11 (Nov. 1986) 368-376.

[McCormack-88] Joel McCormack, Paul Asente, Ralph R. Swick. *X Toolkit Intrinsics - C Language Interface*. MIT Project Athena, 1988.

[Seidewitz-87] Ed Seidewitz. Object-oriented programming in Smalltalk and Ada. *OOPSLA'87* as *ACM SIGPLAN 22*, 12 (Dec. 1987) 202-213.

[Sun-86] *SunView Programmer's Guide*, Sun Microsystems, Mt. View, Calif., 1986.

EXERCISES

Any code requested in the following exercises should be written in a non-object-oriented language. Many of the exercises are taken from the previous chapter, so refer to the figures there.

16.1 (5) Implement the following library objects:

a. A *Dictionary*, which is a one-way many-to-one mapping function whose domain and range are objects.

b. A *Set*, which is an unordered collection of unique values. Duplicate values are automatically discarded from a set. Values can be added, deleted, and tested.

c. A *VariableArray*, which is an ordered, dynamic, collection of values, indexed by nonnegative integer selector values. The number of elements is variable. Elements can be added or deleted at the end, beginning, or at any point in the middle

16.2 (6) Write code to implement the following:

a. One-to-one association which is traversed in both directions.

b. One-to-many association which is traversed in the direction from one to many. The association is considered unordered.

c. One-to-many association which is traversed in the direction from one to many. The association is considered ordered.

d. Many-to-many association which is traversed in both directions. The association is considered ordered in one direction and unordered in the other direction.

16.3 (6) Write library functions *allocate(size)*, which allocates and returns a block of memory, and *deallocate(block)*, which recovers an unused block of memory. These functions are useful in implementing object-oriented designs in a non-object-oriented language.

16.4 (4) Write code for the ordered aggregation between *Collection of cards* and *Card* in Figure E15.2. Operations include insertion, deletion, and traversing in both directions.

16.5 (4) Implement all associations involving the classes *Box*, *Link*, *Line segment*, or *Point* in Figure E15.3 using any technique discussed in this chapter.

16.6 (7) Write code for the *cut* operation on the class *Box* in Figure E15.3. Propagate the operation from boxes to attached link objects. Update any associations that are involved. Be sure to recover any memory that is released by the operation. You may assume another routine will update the display.

16.7 (7) Prepare a routine that will create a logical link between two boxes. (See Figure E15.3.) Inputs to the routine are two boxes and a list of points that define the line segments of the link. The routine should update associations and create object instances as needed. You may assume another routine will update the display.

16.8 (8) Implement the following queries on Figure E15.3:

a. Given a box, determine all other boxes that are directly linked to it.
b. Given a box, find all other boxes that are directly or indirectly linked to it.
c. Given a box and a link, determine if the link involves the box.
d. Given a box and a link, find the other box logically connected to the given box through the other end of the link.
e. Given two boxes, determine all links between them.
f. Given a selection and a sheet, determine which links connect a selected box to a deselected box.
g. Given two boxes and a link, produce an ordered set of points. The first point is where the link connects to the first box, the last point is where the link connects to the second box, and intermediate points trace the link.

16.9 (7) Implement the classes in Figure E15.1. Note the multiple inheritance.

16.10 (8) Discuss how to implement the following classes described in exercise 15.4, using a non-object-oriented language:

a. *Sorted_dictionary*
b. *Polygon*
c. *Index*

16.11 (8) Write code in any non-object-oriented language to implement the generic class *Binary_tree* described in exercise 15.8. Make up at least three classes to test it. You will need to write a *compare* method and a *print* method for each class.

16.12 (Project) Write a program for simulating evolution, as described in the Computer Recreations column in the May 1989 issue of *Scientific American*.

17

Relational Databases

The object-oriented paradigm is versatile. It not only provides a sound basis for designing systems and programming code but can also be used to design databases. The use of an object-oriented design transcends the choice of database. You can design hierarchical, network, relational, and object-oriented databases. Object-oriented designs are efficient, coherent, and less prone to the update problems that plague many other database design techniques. As a side benefit, the use of a uniform design technique improves integration of database and programming language code.

You design a database by first performing the analysis steps described in Chapter 8 and constructing an object model. The remaining methodology chapters (9 and 10) primarily concern design of programming code and apply to database design to a lesser extent. This chapter resumes where Chapter 8 ends and considers implementation issues. How can we map an object model to database structures and tune the result for fast performance? This chapter also includes a brief introduction to databases for new readers.

Chapters 15 and 18 complement this chapter. Chapter 15 includes a section on object-oriented DBMS that have developed from object-oriented programming languages. Chapter 18 discusses automation of the object-model-to-database mapping process.

The coverage of this chapter is biased towards relational DBMS for the following reasons. Relational DBMS are gaining popularity at the expense of hierarchical and network DBMS. Relational DBMS are increasing their advantage in functionality and flexibility and are catching up in performance. Object-oriented DBMS look promising but have not yet reached the commercial mainstream. Logic DBMS also seem promising but are even further away from mass market acceptance.

17.1 GENERAL DBMS CONCEPTS

A *Data Base Management System (DBMS)* is a computer program for managing a permanent, self-descriptive repository of data. This repository of data is called a *database* and is stored in one or more files. There are many reasons why developers use a DBMS:

- *Crash recovery*. The database is protected from hardware crashes, disk media failures, and some user errors.
- *Sharing between users*. Multiple users can access the database at the same time.
- *Sharing between applications*. Multiple application programs (presumably related) can read and write data to the same database. A database is a neutral medium that facilitates communication between free-standing programs.
- *Security*. Data can be protected against unauthorized read and write access.
- *Integrity*. You can specify rules that data must satisfy. A DBMS can control the quality of its data over and above facilities that may be provided by application programs.
- *Extensibility*. Data may be added to the database without disrupting existing programs. Data can be reorganized for faster performance.
- *Data distribution*. The database may be partitioned across various sites, organizations, and hardware platforms.

The life cycle for most database applications includes the following steps.

1. Design the application.
2. Devise a specific architecture for coupling the application to a database.
3. Select a specific DBMS to serve as a platform.
4. Design the database. Write DBMS code to set up the proper database structures.
5. Write programming language code to compensate for DBMS shortcomings, provide a user interface, validate data, and perform computations. Many DBMS provide productivity tools to simplify routine applications.
6. Populate the database with information.
7. Run the application. Query and update the database as needed.

Of course, reality is more complex than this list implies, and there is feedback between steps. Often the DBMS platform is chosen first.

The most important and difficult task for many database applications is the database design. The design of the accompanying programming code is usually much easier. You should design a database for many of the same reasons that you should design any computer program: careful design of software before coding improves the quality and reduces the cost. A database design is often referred to as a *data model* or *schema*.

In general there are two approaches to database design. The first approach is attribute driven: Compile a list of attributes relevant to the application and synthesize groups of attributes that preserve functional dependencies. The other approach is entity driven: Discover entities that are meaningful to the application and describe them. In a typical design, there are ten times fewer entities than attributes, so entity design is much more tractable. Object modeling is a form of entity design.

The *three schema architecture*, summarized in Figure 17.1, is the standard architecture for a family of related database applications. This architecture was originally proposed by the ANSI/SPARC committee on DBMS. The basic idea is that a database design should

comprise three layers: the external, conceptual, and internal schemas. Each external schema is a database design from the perspective of a single application. The external schema is a view or abstraction of the global, overall conceptual schema. The external schema isolates applications from most changes in the conceptual schema. To a large extent, future evolution in the conceptual schema can be resolved within the external-conceptual schema interface. The conceptual schema is a database design from the perspective of an enterprise. The conceptual schema integrates related applications and hides the peculiarities of the underlying DBMS. The internal schema deals with the limitations and features of a specific DBMS. The internal schema level consists of actual DBMS code required to implement the conceptual schema.

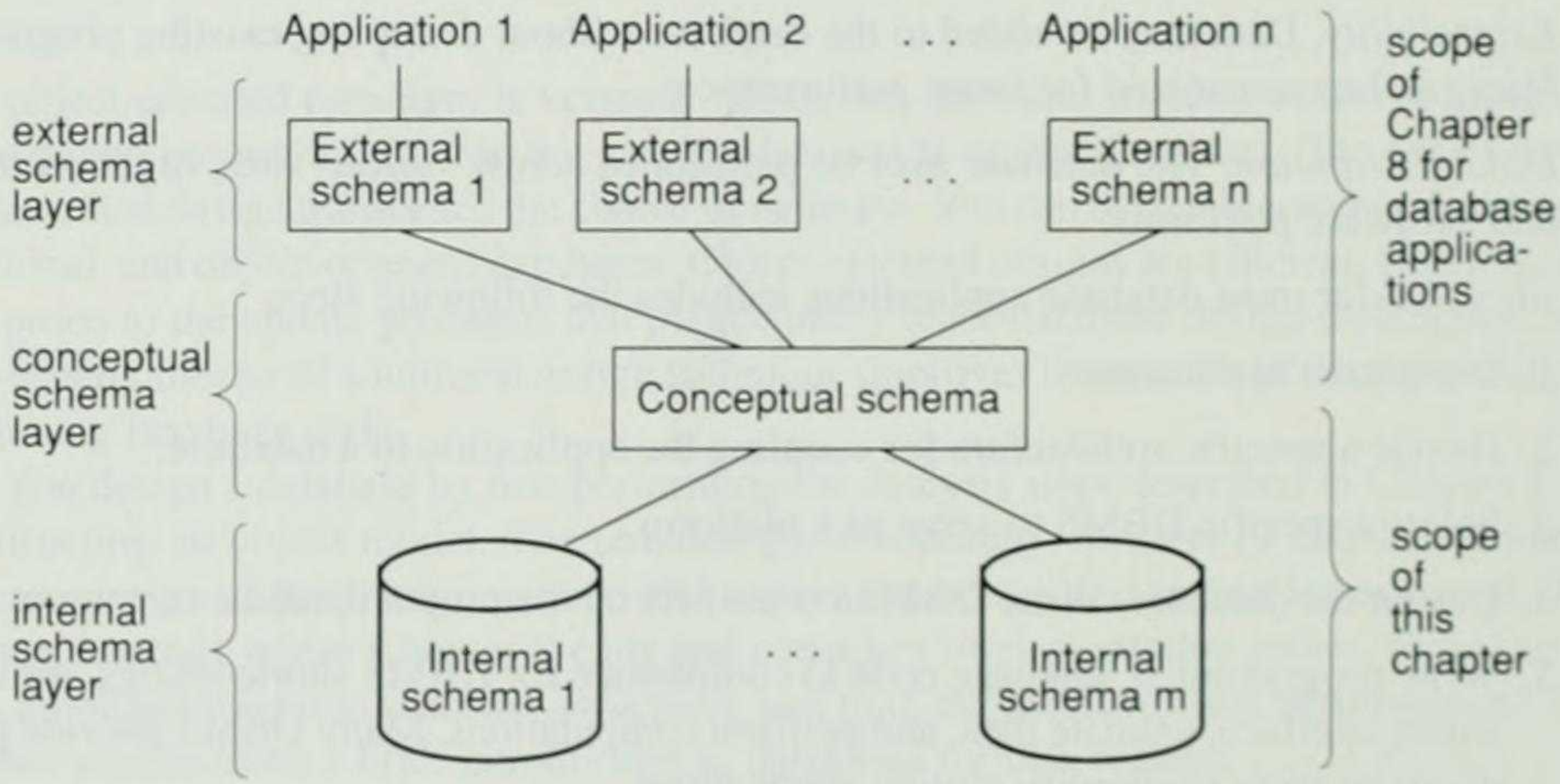


Figure 17.1 ANSI/SPARC three schema architecture

Object modeling (discussed in Chapter 8) is useful for designing both the external and conceptual schema. You should construct one object model for each external schema and another object model for the conceptual schema. This chapter explains how to translate object models into DBMS code.

17.2 RELATIONAL DBMS CONCEPTS

The *relational data model* was invented by E.F. Codd and is based on one simple concept — the table. A *relational DBMS* (RDBMS) is a computer program for managing these tables. A RDBMS, as defined by Codd, has three major parts:

- Data that is presented as tables
- Operators for manipulating tables
- Integrity rules on tables

We will consider each of these parts in turn. Readers familiar with RDBMS may skip this section.

17.2.1 RDBMS Logical Data Structure

A relational database logically appears as simply a collection of tables. Tables have a specific number of columns and an arbitrary number of rows. The columns of tables are called *attributes* and directly correspond to attributes in object models. The rows are called *tuples* and correspond to object instances and links. A *simple value* is stored at each table row and column intersection.

Relational database theory dictates that each attribute must be assigned a domain. A *domain* is a set of legal values. Domains carry more information than just a data format and permit greater semantic checking. For instance, domains can be used to prevent operations on incompatible attributes, such as adding a cost to a weight. The concept of domain is similar to strong typing in a programming language. Unfortunately, most RDBMS do not support domains and only support simple data formats like number, date, and character string.

Each value in a table must belong to the domain of its attribute or be null. *Null* means that an attribute value is unknown or not applicable for a given row. There are complex theoretical issues concerning null values that often cause problems for real applications.

RDBMS use various devices for speeding access to tables, as literal tables are much too slow for practical needs. These tuning devices are transparent and not visible in the commands for reading and writing to tables. The RDBMS decides when tuning information is helpful in processing a query, and if so, automatically uses it. The RDBMS automatically updates tuning information whenever the corresponding tables are updated. Indexing, hashing, and sorting are common tuning techniques.

17.2.2 RDBMS Operators

SQL has become the most popular language for RDBMS, as well as an ANSI and ISO standard. Unfortunately, SQL is far from an ideal language and has many technical flaws. For example, SQL violates modern principles of language theory. The scope of the SQL standard is small and incomplete; it does not address important issues such as performance tuning and programming productivity tools. (See [Date-87] for a detailed critique of the SQL language.) Nevertheless, we will use SQL in this book, since it is a standard.

SQL provides operators for manipulating tables. The SQL *select* statement queries tables. The syntax of the *select* command looks something like:

```
SELECT attribute-list
FROM table-1, table-2, ...
WHERE predicate-is-true
```

Logically table-1, table-2, and any others are combined into one temporary table. The attribute list specifies which columns should be retained in the temporary table. The predicate expression specifies which rows should be retained. The contents of the temporary table are

returned as the answer to the query. Additional SQL commands create tables, insert rows into tables, delete rows from tables, and perform other functions.

Interactive SQL commands are set-oriented; they operate on entire tables rather than individual rows or values. SQL provides a similar language for use with application programs through an awkward, row-at-a-time interface. RDBMS can deal with entire tables, but most programming languages cannot.

17.2.3 RDBMS Integrity

An important aspect of RDBMS, one that is often overlooked, is support for integrity. Most RDBMS lack proper support for integrity. The two aspects of integrity in Codd's model are entity integrity and referential integrity. Entity integrity dictates that each table have exactly one primary key. A *primary key* is a combination of one or more attributes whose value unambiguously locates each row in a table. (The primary key is always a candidate key. Candidate key was defined in Chapter 4.) In Figure 17.2, *person-ID* is the primary key of the *Person* table; *company-ID* is the primary key of the *Company* table.

Person table

person-ID	person-name	address	company-ID
1	Jim Smith	314 Olive St.	1001
5	Moe Brown	722 Short St.	1002
999	Jim Smith	1561 Main Dr.	1001
14	Jane Brown	722 Short St.	null

Company table

company-ID	company-name	address
1001	Ajax Widgets	33 Industrial Dr.
1002	AAA liquors	724 Short St.
1003	Win-more Sports	1877 Broadway

person-ID is the primary key of Person table.
company-ID is the primary key of Company table.
company-ID is a foreign key in Person table.

Figure 17.2 Primary key and foreign key

Referential integrity requires that the RDBMS keep each foreign key consistent with its corresponding primary key. A *foreign key* is a primary key of one table that is embedded in another (or the same) table. In Figure 17.2 *company-ID* is a foreign key in the *Person* table. It would not be permissible to change Moe Brown's *company-ID* to 1004 since 1004 is not defined in the *Company* table. If the row for Ajax Widgets was deleted in the *Company* table, then both Jim Smith rows in the *Person* table must have their *company-ID* set to null. The link between foreign key and primary key forms a frequent navigation path between tables.

The SQL standard and commercial RDBMS implementations are slowly moving towards support for referential integrity. Referential integrity is useful when mapping object

models to tables. Figure 17.2 happens to be a one-to-many association that has been converted to table form (see Section 17.3.4).

17.2.4 Normal Forms

Normal forms are rules developed to avoid logical inconsistencies from table update operations. Each normal form prohibits a form of redundancy in table organization that could yield meaningless results if one table were updated independently of other tables. There are multiple levels of normal form. Each higher level of normal form adds a constraint to the normal form below it. As the database designer satisfies higher normal form, tables tend to become fragmented; normal forms improve database consistency at the cost of added navigation and slower query execution. You should not inadvertently violate normal forms but may occasionally do so for a good reason, such as performance.

A table is in *first normal form* when each attribute value does not contain a repeating group. Figure 17.3a has a row with two entries under *equipment name* and thus violates first normal form. Restating the dual entries as two rows in Figure 17.3b satisfies first normal form.

A table is in *second normal form* when it satisfies first normal form and each row has a primary key. Each nonprimary key field must fully depend on the primary key. Figure 17.3b violates second normal form because *equipment manufacturer* and *manufacturer address* depend on the full primary key while *plant manager* only depends on part of the primary key. Figure 17.3c splits off the partial dependency into another table and satisfies second normal form.

A table is in *third normal form* when it satisfies second normal form and each nonprimary key attribute directly depends on the primary key. Figure 17.3c violates third normal because there is a transitive dependency; *manufacturer address* depends on *equipment manufacturer* which in turn depends on the primary key. Third normal form requires direct dependence on the primary key. Once again, splitting into two tables resolves the violation. Figure 17.3d satisfies third normal form.

There are higher normal forms that are usually not needed. [Kent-83] contains a more thorough explanation of normal forms. The motivation for observing normal form becomes much more compelling when there are thousands or millions of rows, instead of the few shown in Figure 17.3. The next section discusses how object models tend to preserve normal form.

17.2.5 Views

A *view* is a virtual table that is dynamically computed as needed. A view does not physically exist but is derived from one or more underlying tables. In theory, views are the means for deriving external schema from conceptual schema for the ANSI three schema architecture. In practice, views are less useful. Commercial RDBMS usually support reading through views but seldom support writing through views. There are subtle semantic issues associated with writing through views that most RDBMS avoid [Keller-86].

(a) Violates 1st normal form.

plant name	equipment name	plant manager	equipment manufacturer	manufacturer address
ethylene	final cooler, feed heater	Jim Smith	ABC exchanger	1247 Locust
styrene	feed pump	Bill Gunn	XYZ pumps	432 Broadway
styrene	feed heater	Bill Gunn	ABC exchanger	1247 Locust

(b) Satisfies 1st normal form. Violates 2nd normal form.

plant name	equipment name	plant manager	equipment manufacturer	manufacturer address
ethylene	final cooler	Jim Smith	ABC exchanger	1247 Locust
ethylene	feed heater	Jim Smith	ABC exchanger	1247 Locust
styrene	feed pump	Bill Gunn	XYZ pumps	432 Broadway
styrene	feed heater	Bill Gunn	ABC exchanger	1247 Locust

Primary key: (plant-name, equipment-name)

(c) Satisfies 2nd normal form. Violates 3rd normal form.

plant name	plant manager
ethylene	Jim Smith
styrene	Bill Gunn

Primary key: (plant-name)

plant name	equipment name	equipment manufacturer	manufacturer address
ethylene	final cooler	ABC exchanger	1247 Locust
ethylene	feed heater	ABC exchanger	1247 Locust
styrene	feed pump	XYZ pumps	432 Broadway
styrene	feed heater	ABC exchanger	1247 Locust

Primary key: (plant-name, equipment-name)

(d) Satisfies 3rd normal form.

plant name	plant manager
ethylene	Jim Smith
styrene	Bill Gunn

Primary key: (plant-name)

plant name	equipment name	equipment manufacturer
ethylene	final cooler	ABC exchanger
ethylene	feed heater	ABC exchanger
styrene	feed pump	XYZ pumps
styrene	feed heater	ABC exchanger

Primary key: (plant-name, equipment-name)

equipment manufacturer	manufacturer address
ABC exchanger	1247 Locust
XYZ pumps	432 Broadway

Primary key: (equipment-manufacturer)

Figure 17.3 Normal forms

17.3 RELATIONAL DATABASE DESIGN

So far this chapter has reviewed DBMS in general and RDBMS. We now shift our focus towards database design. As stated earlier, we only consider relational database design, primarily because RDBMS technology is mature and dominates the marketplace.

17.3.1 Extended Three Schema Architecture for Object Models

Figure 17.4 shows how object modeling relates to the three schema architecture. First, you should formulate object models for the external and conceptual schema. Then, you should translate each object model to ideal tables, that is, the table model. Views and/or interface programs connect external tables to conceptual tables. Conceptual tables convert to internal schema. We will now study each part of the extended three schema architecture in detail.

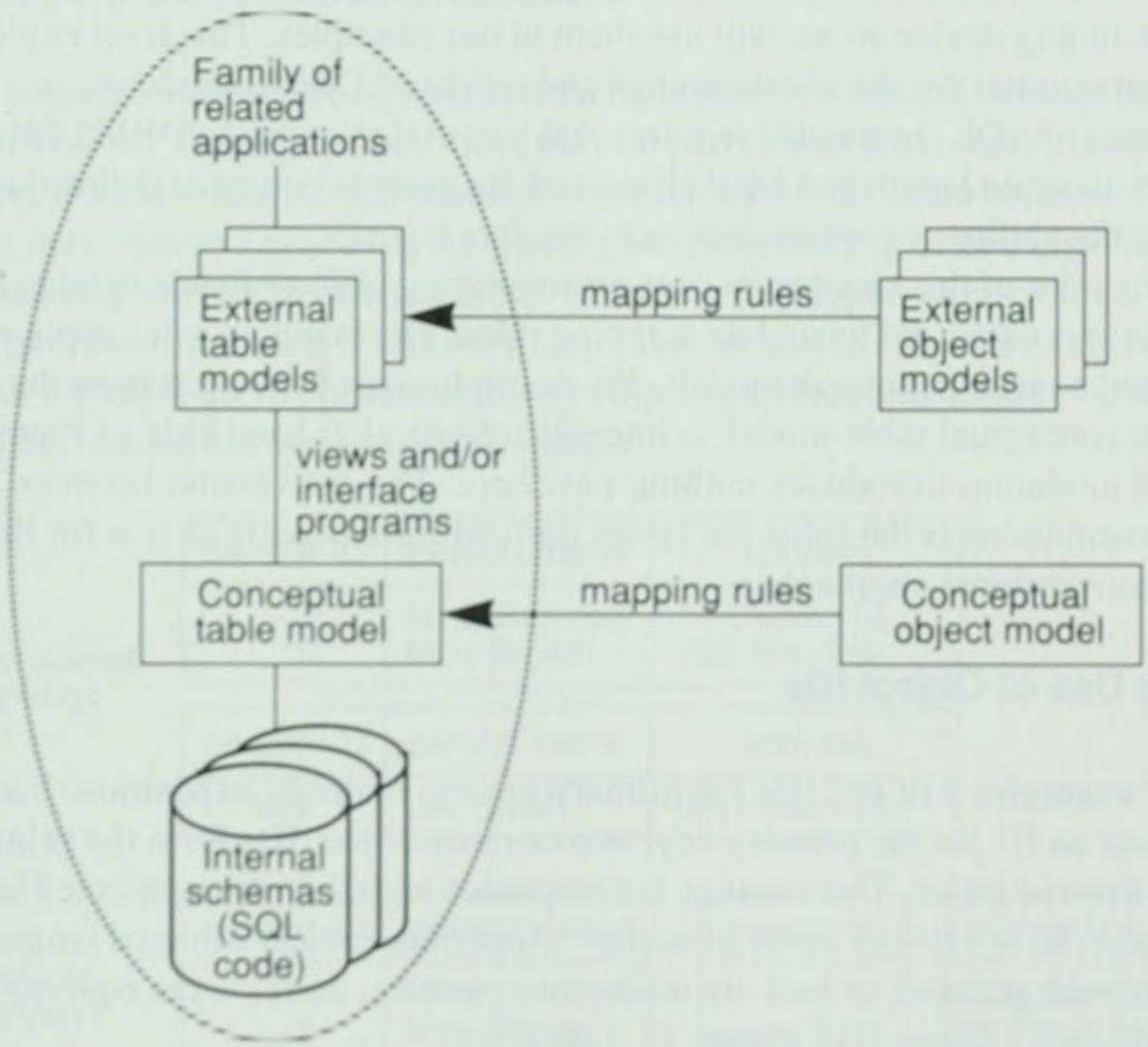


Figure 17.4 Object modeling and the three schema architecture for RDBMS

The object models focus on logical data structure. Each object model consists of many classes, associations, generalizations, and attributes. Object modeling promotes deep, abstract thinking about a problem unencumbered by implementation details. Object models are effective for communicating with application experts and reaching a consensus about the important aspects of a problem. Object models help developers achieve a coherent, understandable, efficient, and correct database design.

Each table model consists of many ideal tables. These ideal tables are generic and DBMS-independent. Ideal tables abstract common characteristics of RDBMS implementations. The table model decouples DBMS idiosyncrasies from object model to table model mapping rules. This improves documentation and eases porting.

In order to translate from an object model to ideal tables, you must choose among several mapping alternatives. For example, there are two ways to map an association to tables and four ways to map a generalization (discussed later in this chapter). You must also supply details that are missing from the object model, such as the primary key and candidate keys for each table and whether each attribute can be null. Attributes in candidate keys usually should not be null; the data modeler may choose to require values for additional attributes. You must assign a domain to each attribute and list groups of attributes subject to frequent access.

The internal schema of the three schema architecture consists of SQL commands that create the tables, attributes, and performance-tuning structures. Indexes are the most popular performance-tuning device so we will use them in our examples. This level exploits the features and compensates for the shortcomings and quirks of DBMS products.

Generation of SQL commands requires that you assign tables to DBMS files and honor DBMS restrictions on length and legal characters for names. You must define domains if the RDBMS has the ability or convert domains into data types.

The remainder of this chapter focuses on converting object models to table models. We will study various cases and formulate mapping rules. The mapping rules apply equally well to external and conceptual object models. We deemphasize conversion from the external table model to conceptual table model to internal schema (left hand side of Figure 17.4) because object modeling introduces nothing new here. The conversion between the various tabular representations is the same for tables derived from objects as it is for those derived from more conventional approaches.

17.3.2 The Use of Object IDs

Most of our examples will use IDs for primary keys to simplify exposition. Each class-derived table has an ID for the primary key; one or more object IDs form the primary key for association-derived tables. This strategy is compatible with the object-oriented language notion that objects have identity apart from their properties. Object-oriented languages implement identity with pointers or look-up tables into pointers; an ID is the equivalent database construct. Figure 17.2 uses IDs; Figure 17.3 does not.

There are benefits to using IDs. IDs are immutable and completely independent of changes in data value and physical location. The stability of IDs is particularly important for associations since they refer to objects. Contrast this with referring to objects by name. Then changing a name requires update of many associations. IDs provide a uniform mechanism for referencing objects.

On the other hand, IDs have disadvantages. Generating IDs is a nuisance, for which RDBMS provide no inherent support. For example, it is awkward to track previously allocated IDs and reclaim deleted IDs for reuse. Object-oriented languages usually avoid ID re-

use by giving IDs many bits. Databases cannot be so cavalier, since they are specifically intended to be long lived.

IDs undermine the original intent of RDBMSs. RDBMS theory emphasizes that data is located and manipulated based on attribute values. In one sense, it certainly is permissible to define IDs as attributes and adopt a protocol for handling them. But on the other hand, an ID really is not a value and is an implementation artifact that RDBMSs are trying to eliminate.

So when should you use IDs? Do not use IDs for applications where users directly access the database. People think in terms of descriptive properties such as names and not in terms of artificial numbers. The advantages of IDs may prevail when database access is restricted via programs. Restricted access often occurs because application software is needed to compensate for DBMS deficiencies, enforce integrity, and provide a user interface. We will discuss object-to-table mapping rules that apply, regardless of whether IDs are used.

17.3.3 Mapping Object Classes to Tables

Each class maps to one or more tables. (Also a table may correspond to more than one class. See Section 17.3.4.) The objects in a class may be partitioned horizontally and/or vertically. For instance, if a class has many instances of which a few are often referenced, horizontal partitioning may improve efficiency by placing the frequently accessed objects in one table and the remaining objects in another table. Of course, an application will not benefit from horizontal partitioning unless it knows which table to search. Similarly, if a class has attributes with different access patterns, it may help to partition the objects vertically. Figure 17.5 shows horizontal and vertical partitioning.

Horizontal partition

person-ID	person-name	address
1	Mike Rumrow	14 Center St.
5	Moe Brown	722 Short St.

person-ID	person-name	address
999	Jim Smith	1561 Main Dr.

Vertical partition

person-ID	person-name
1	Mike Rumrow
5	Moe Brown
999	Jim Smith

person-ID	address
1	14 Center St.
5	722 Short St.
999	1561 Main Dr.

Figure 17.5 Horizontal and vertical partitioning of tables

In Figure 17.6 an object class converts to one table. Class *Person* has attributes *person name* and *address*. The table model lists these attributes and adds the implicit object ID. As part of formulating the table model, we add details. We specify that *person ID* cannot be null since it is a candidate key. We decide that *person name* should not be null; a name must be entered for every person. (*Person name* is not a candidate key because two persons may have

the same name.) Attribute *address* may be null. We assign a domain to each attribute, specify the primary key for each table, and note frequently accessed groups of attributes.

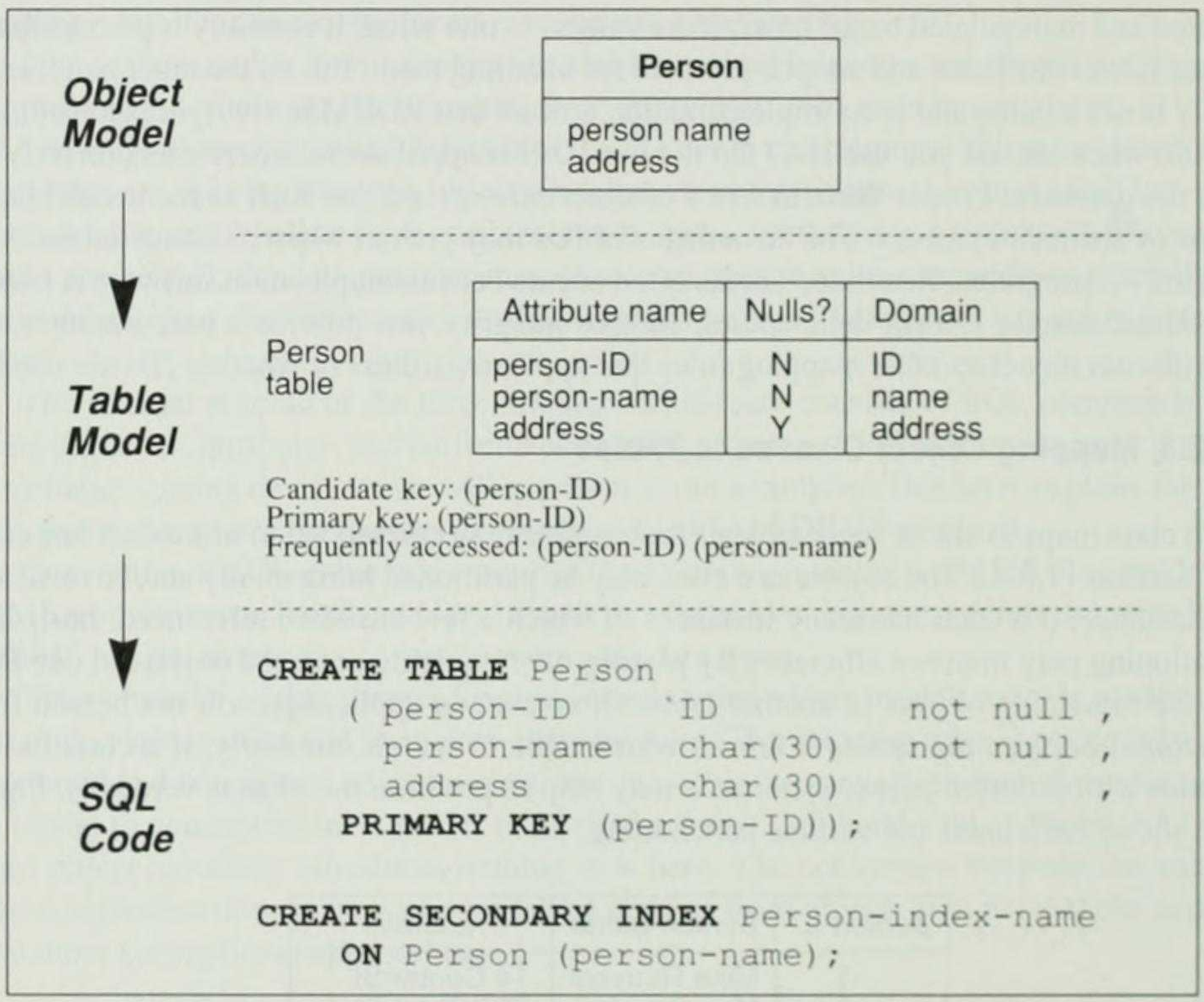


Figure 17.6 Mapping class to table

The SQL code creates the *Person* table. (The SQL standard currently supports referential integrity. However many commercial products do not yet fully support this aspect of the standard.) The index on *person-name* ensures quick retrieval for this attribute since it is frequently accessed. The SQL level also maps domains to data types.

17.3.4 Mapping Binary Associations to Tables

In general, an association may, or may not, map to a table. It depends on the type and multiplicity of the association and the database designer's preferences in terms of extensibility, number of tables, and performance trade-offs.

Figure 17.7 shows that a many-to-many association always maps to a distinct table. This schema satisfies third normal form. The primary keys for both related classes and any link attributes become attributes of the association table. Attributes *company ID* and *person ID* combine to form the only candidate key for the *Owns-stock* table. In general, an association may be traversed starting from either class so both *company ID* and *person ID* could be fre-

quently accessed. The foreign key clauses of the SQL code indicate that each *Owns-stock* tuple must reference a *Company* and *Person* that have been defined in their respective tables.

An association table always sets the foreign keys from the related objects to not null; by definition, a link between two objects requires that both objects be known. If a given pair of objects does not have a link, we omit an entry in the association table.

Many RDBMS do not semantically support primary keys; the workaround is to define unique indexes. (A RDBMS that explicitly supports primary keys may also generate a unique index as a side effect.) We use the primary key clause in our examples because it is part of standard SQL and we expect it to eventually become widely adopted. The primary key may subsume a secondary index. For example, if the *Owns-stock* primary key in Figure 17.7 is implemented as a unique index on *company-ID*, *person-ID*, the secondary index on *company-ID* would be superfluous.

Figures 17.8, 17.9, and 17.10 show two options for mapping a one-to-many association to tables. You may create a distinct table for the association or bury a foreign key in the table for the many class. We do not show SQL code for this example and some others because SQL code is voluminous, and the table model already makes the desired point. The advantages of merging an association into a class are:

- Fewer tables
- Faster performance due to fewer tables to navigate

The disadvantages of merging an association into a class are:

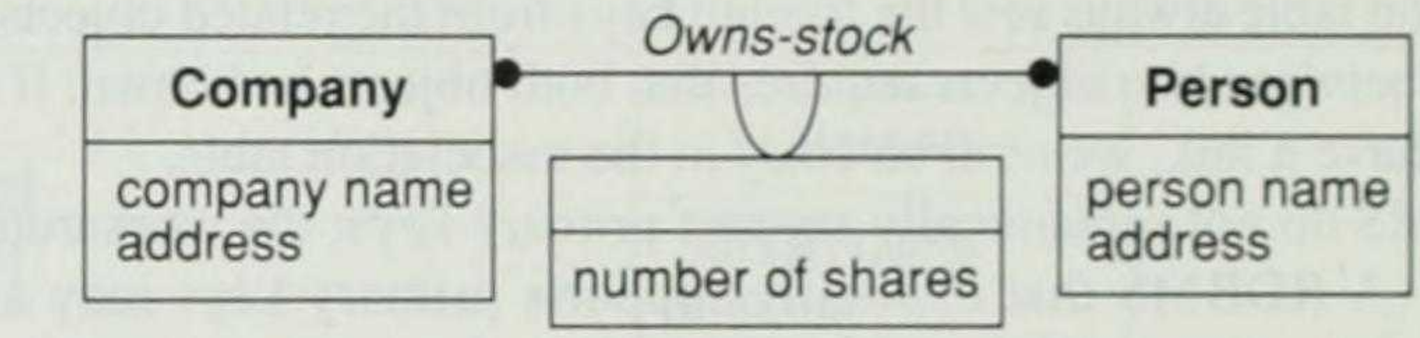
- *Less design rigor.* Associations are between independent objects of equal syntactic weight. In general, it seems inappropriate to contaminate objects with knowledge of other objects. This point is related to the argument in favor of encapsulation for object-oriented languages.
- *Reduced extensibility.* It is difficult to get multiplicity right on the first few design passes. One-to-one and one-to-many associations may be externalized. Many-to-many associations must be externalized.
- *More complexity.* An asymmetrical representation of the association complicates search and update.

The final decision on whether to collapse an association into a related class depends on the application.

You may also collapse a one-to-one association into an object table or merge even further and store both objects and the association all in one table.* Chapter 3 used *a country has a capital city* to illustrate a one-to-one association. *Country*, *Capital city*, and the association could all be stored in one table. Merging into a single table improves performance and reduces database storage at the cost of less extensibility and possible violation of third normal form.

---

* Be careful with collapsing cyclical associations. For example, if classes *A*, *B*, and *C* are related by one-to-one associations *A::B*, *B::C*, and *C::A*, it may not be correct to fold all three classes and three associations into one table. For example, the instances *a1::b1*, *b1::c1*, and *c1::a2* require that the cycle be implemented with at least two tables.



Object Model

Company table (similar to Person table in Figure 17.6) ...

Person table from Figure 17.6 ...

Table Model

Owns-stock table

Attribute name	Nulls?	Domain
company-ID	N	ID
person-ID	N	ID
number-of-shares	Y	share-count

Candidate key: (company-ID, person-ID)
Primary key: (company-ID, person-ID)
Frequently accessed: (company-ID) (person-ID)

SQL Code

```
Create table and indexes for Company ...
    (similar to that for Person)

Create table and indexes for Person
    (Figure 17.6) ...

CREATE TABLE Owns-stock
    ( company-ID          ID        not null ,
      person-ID           ID        not null ,
      number-of-shares    integer            ,
    PRIMARY KEY (company-ID, person-ID),
    FOREIGN KEY (company-ID) REFERENCES Company,
    FOREIGN KEY (person-ID) REFERENCES Person);

CREATE SECONDARY INDEX Owns-stock-index-company
    ON Owns-stock (company-ID);

CREATE SECONDARY INDEX Owns-stock-index-person
    ON Owns-stock (person-ID);
```

Figure 17.7 Mapping many-to-many association to tables

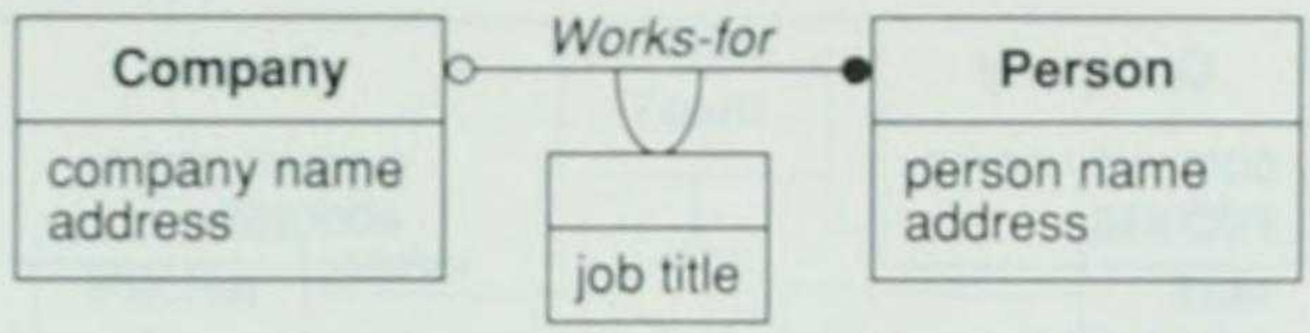


Figure 17.8 Object model for one-to-many association

Company table (similar to Person table in Figure 17.6) ...

Person table from Figure 17.6 ...

Works-for table

Attribute name	Nulls?	Domain
company-ID	N	ID
person-ID	N	ID
job-title	Y	title

Candidate key: (person-ID)
Primary key: (person-ID)
Frequently accessed: (company-ID) (person-ID)

Figure 17.9 Table model for one-to-many association—distinct association table

Company table (similar to Person table in Figure 17.6) ...

Person table

Attribute name	Nulls?	Domain
person-ID	N	ID
person-name	N	name
address	Y	address
company-ID	Y	ID
job-title	Y	title

Candidate key: (person-ID)
Primary key: (person-ID)
Frequently accessed: (person-ID) (person-name) (company-ID)

Figure 17.10 Table model for one-to-many association—buried foreign key

Note that for a one-to-many association you may also collapse both classes and the association all into one table. However, usually this is undesirable and may violate second normal form.

The hollow ball multiplicity symbol in Figure 17.8 indicates that a person need not be employed. In Figure 17.11, all persons within the scope of the model must be employed. This figure illustrates the issue of existence dependency that we mentioned in Section 4.6. Here, *Person* objects know about *Company* objects since each *Person* object requires a *Company* object. There is less benefit to having a distinct association table. The only change in the table model for the buried foreign key case is that *company ID* cannot be null.

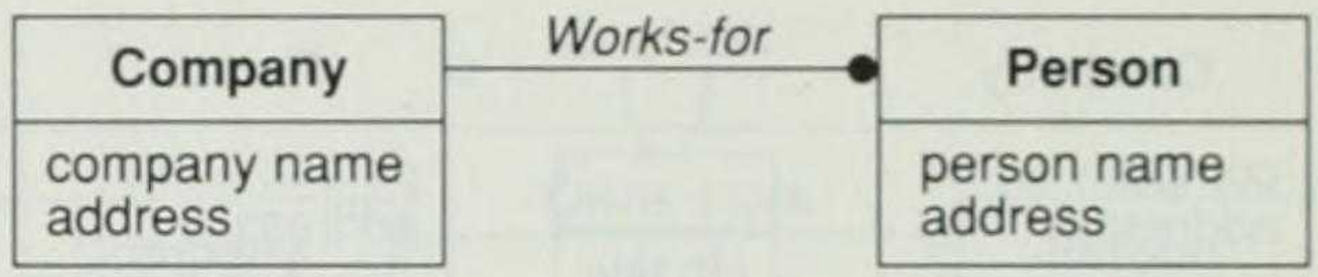


Figure 17.11 Object model for association with existence dependency

17.3.5 Mapping Ternary Associations to Tables

Figure 17.12 shows an RDBMS implementation for a ternary association. Note that we show a table for each class participating in the ternary association even for a class that may be trivial like *Year*. Here, if the *Year* table merely pairs *year-ID* with *year*, we could perform a minor optimization: Discard the *Year* table and replace *year-ID* by *year* in the ternary table.

Figure 17.12 does not exhibit existence dependency. This certainly seems to make sense for the pitching example. A pitcher, team, and year have independent existence apart from whether a pitcher happens to pitch for a team in a given year. For a ternary association where a dependency between an object class and the ternary association is important, it is probably best to promote the ternary association to an object class (see Section 3.3.2) and handle the dependency as in Figure 17.11. The promotion of the ternary association to a class gives each ternary instance an ID that can then be referenced by the dependent object.

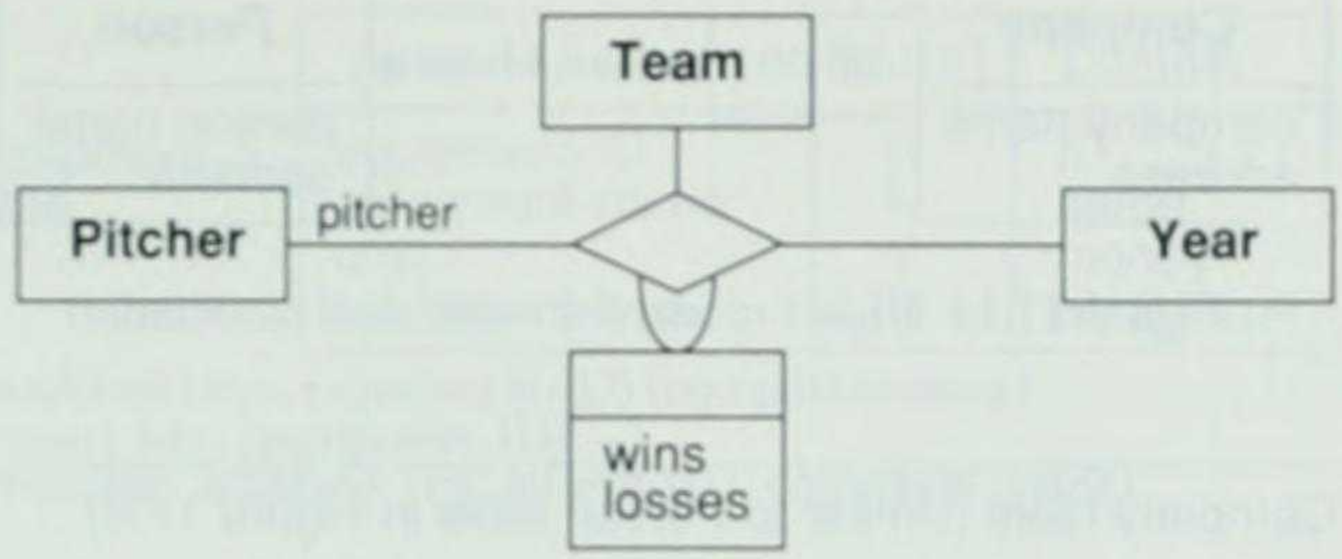
Note that the *Person* table refers to *person-ID* while the ternary table refers to *pitcher-ID*. The ternary table reflects that a person assumes the role of pitcher. *Pitcher-ID* is a more meaningful attribute name for the ternary table than *person-ID*. The object model shows that these two names refer to the same entity and would be expected to share a common navigational path. The role name of a class should be used for attributes buried in association tables. Figure 17.12 shows how to handle roles with SQL code. (We could have used roles in some of our earlier examples. For instance, in Figure 17.7 person assumes the role of shareholder. In Figure 17.8 and Figure 17.11 a company is the employer and a person is the employee.)

Figures 17.13 and 17.14 discuss one more association construct—qualification. A company has many persons serving as officers. Most offices within a company are held by one person, such as President and Treasurer. A few offices, such as member of the board of directors, may be served by many persons. A person may have many company-office combinations. For instance, a person may be an officer of more than one company or hold multiple offices for the same company.

The *Company* and *Person* tables are similar to earlier examples. The association table is more interesting. The association table has three attributes: the primary key for *Company*, the primary key for *Person*, and the qualifier *office*. None of these attributes can be null since they are all an essential part of the qualified association. All three attributes must appear in the primary key and candidate key since the association is many-to-many after qualification.

We have not explicitly discussed aggregations in any of our database examples. The design of aggregation tables follows the same mapping rules as for associations.

Object Model



Candidate key: (pitcher-ID, team-ID, year-ID)

Table Model

Person table ... Team table ... Year table ...

Ternary table

Attribute name	Nulls?	Domain
pitcher-ID	N	ID
team-ID	N	ID
year-ID	N	ID
wins	Y	games
losses	Y	games

Candidate key: (pitcher-ID, team-ID, year-ID)
Primary key: (pitcher-ID, team-ID, year-ID)
Frequently accessed: (pitcher-ID) (team-ID) (year-ID)

SQL Code

```
CREATE TABLE Person
    ( person-ID      ID      not null ,
      and other attributes ... ,
    PRIMARY KEY (person-ID));

also create table for Team and Year

CREATE TABLE Person-Team-Year-ternary
    ( pitcher-ID     ID      not null ,
      team-ID        ID      not null ,
      year-ID        ID      not null ,
      wins           integer          ,
      losses         integer          ,
    PRIMARY KEY (pitcher-ID, team-ID, year-ID),
    FOREIGN KEY (pitcher-ID) REFERENCES Person,
    FOREIGN KEY (team-ID) REFERENCES Team,
    FOREIGN KEY (year-ID) REFERENCES Year);

also create indexes ...
```

Figure 17.12 Mapping ternary association to tables

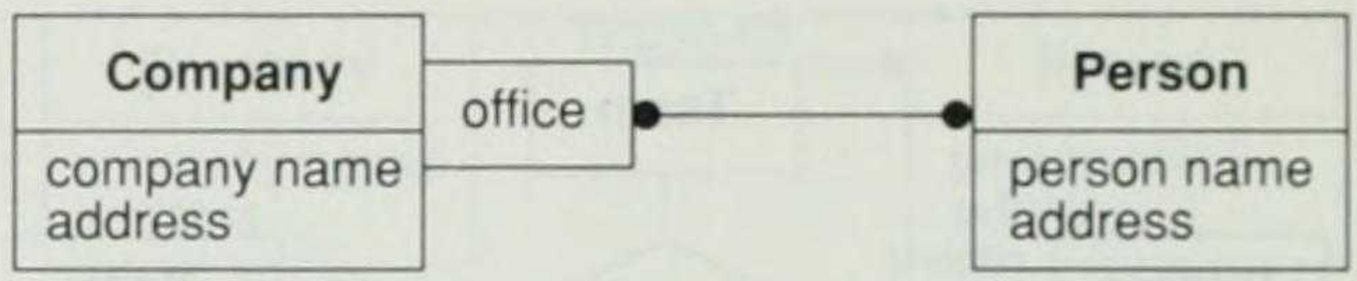


Figure 17.13 Object model for a qualified association

Company table (similar to Person table in Figure 17.6) ...

Person table from Figure 17.6 ...

Qualified association table

Attribute name	Nulls?	Domain
company-ID	N	ID
person-ID	N	ID
office	N	office-name

Candidate key: (company-ID, person-ID, office)
Primary key: (company-ID, person-ID, office)
Frequently accessed: (person-ID) (company-ID, office)

Figure 17.14 Table model for a qualified association

17.3.6 Mapping Generalizations to Tables

There are four approaches to mapping generalizations to tables. We will use Figure 17.15 as our basis for exploring these strategies. We begin by discussing single inheritance.

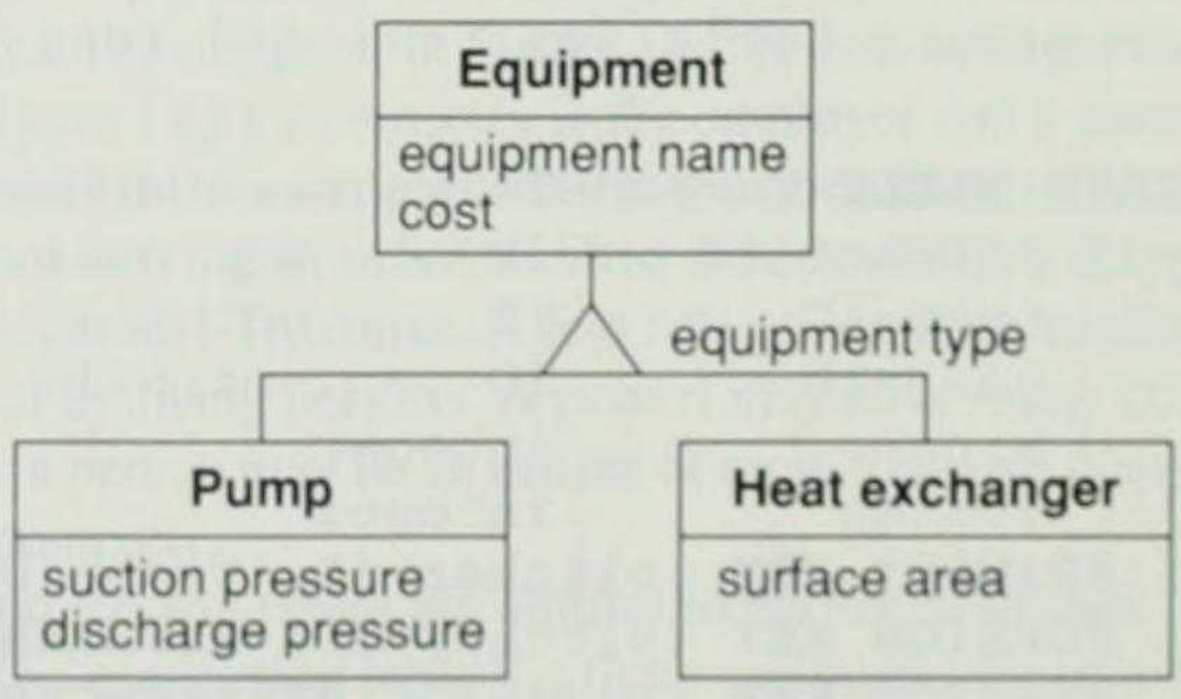


Figure 17.15 Object model for generalization

Figure 17.16 shows the normal approach. The superclass and the subclasses each map to a table. The identity of an object across a generalization is preserved through the use of a shared ID. Thus *product pump* may have one row in the *Equipment* table with ID 101 and

Equipment table

Attribute name	Nulls?	Domain
equipment-ID	N	ID
equipment-name	N	name
cost	Y	money
equipment-type	N	equip-type

Candidate keys: (equipment-ID) (equipment-name)
Primary key: (equipment-ID)
Frequently accessed: (equipment-ID) (equipment-name)

Pump table

Attribute name	Nulls?	Domain
equipment-ID	N	ID
suction-pressure	Y	pressure
discharge-pressure	Y	pressure

Candidate key: (equipment-ID)
Primary key: (equipment-ID)
Frequently accessed: (equipment-ID)

Heat exchanger table

Attribute name	Nulls?	Domain
equipment-ID	N	ID
surface-area	Y	area

Candidate key: (equipment-ID)
Primary key: (equipment-ID)
Frequently accessed: (equipment-ID)

Figure 17.16 Table model for generalization—superclass and subclass tables

another row in the *Pump* table also with ID 101. This approach is logically clean and extensible. However, it involves many tables, and superclass to subclass navigation may be slow. You could navigate the tables as follows:

1. The user supplies an equipment name.
2. Find the *Equipment* row that corresponds to equipment name.
3. Retrieve the *equipment ID* and *equipment type* for this row.
4. Go to the subclass table indicated by *equipment type*, and find the subclass row with the same ID as the *Equipment* row.

For example, the user may specify equipment name "product pump." The application looks in the equipment table and finds that product pump has ID 101 and equipment type pump. The application then searches the *Pump* table and retrieves additional data for ID 101.

Figure 17.17 shows SQL code for Figure 17.16. Note that SQL cannot enforce the partitioning indicated by the generalization discriminator. You could store *product pump* in both the *Pump* and *Heat exchanger* tables, and SQL would permit it. The basic problem is the SQL has little support for integrity constraints. You would need to write special application

code to enforce the generalization partition and use SQL permissions to block interactive access or access through other programs.

```
CREATE TABLE Equipment
    ( equipment-ID          ID          not null ,
      equipment-name        char(30)    not null ,
      cost                  money                ,
      equipment-type        char(10)    not null ,
    PRIMARY KEY (equipment-ID));

CREATE SECONDARY INDEX  Equipment-index-name
    ON Equipment (equipment-name);

CREATE TABLE Pump
    ( equipment-ID          ID          not null ,
      suction-pressure      real                 ,
      discharge-pressure    real                 ,
    PRIMARY KEY (equipment-ID),
    FOREIGN KEY (equipment-ID) REFERENCES Equipment);

Create table for Heat exchanger ...
    (similar to that for Pump)
```

Figure 17.17 SQL code for generalization—superclass and subclass tables

Figure 17.18 and Figure 17.19 are alternate mapping approaches. They are motivated by the desire to eliminate superclass-to-subclass navigation and thus speed performance. However, the improved performance incurs a price.

Figure 17.18 illustrates the "many subclass approach." This approach eliminates the superclass table and replicates all the superclass attributes in each subclass table. You might use this approach if a subclass has many attributes, the superclass has few attributes, and the application knows what subclass to search. Figure 17.18 observes third normal form but is less satisfying than the standard approach. You cannot enforce uniqueness of equipment names across subclass tables since RDBMS do not provide indexes that span tables.

The "one superclass table approach" shown in Figure 17.19 brings all subclass attributes up to the superclass level. Each record in the superclass table uses attributes pertinent to one subclass; the other attribute values are null. The table in this figure violates third normal form; *equipment-ID* or *equipment-name* is the primary key, but attribute values also depend on *equipment type*. This may be a useful approach if there are only two or three subclasses with few attributes.

The best way to handle generalization relationships that exhibit multiple inheritance from disjoint classes (see Section 4.4) is to use the standard approach shown in Figure

17.16—one table per superclass, one table per subclass. The best way to handle multiple inheritance from overlapping classes is to use one table for the superclass, one table for each subclass, and one table for the generalization relationship (the fourth approach for mapping generalizations to tables). Multiple inheritance infrequently occurs, so usually it is not worth the effort to try to optimize the mapping.

Pump table

Attribute name	Nulls?	Domain
equipment-ID	N	ID
equipment-name	N	name
cost	Y	money
suction-pressure	Y	pressure
discharge-pressure	Y	pressure

Candidate keys: (equipment-ID) (equipment-name)
Primary key: (equipment-ID)
Frequently accessed: (equipment-ID) (equipment-name)

Heat exchanger table

Attribute name	Nulls?	Domain
equipment-ID	N	ID
equipment-name	N	name
cost	Y	money
surface-area	Y	area

Candidate keys: (equipment-ID) (equipment-name)
Primary key: (equipment-ID)
Frequently accessed: (equipment-ID) (equipment-name)

Figure 17.18 Table model for generalization—many subclass tables

Equipment table

Attribute name	Nulls?	Domain
equipment-ID	N	ID
equipment-name	N	name
cost	Y	money
equipment-type	N	equip-type
suction-pressure	Y	pressure
discharge-pressure	Y	pressure
surface area	Y	area

Candidate keys: (equipment-ID) (equipment-name)
Primary key: (equipment-ID)
Frequently accessed: (equipment-ID) (equipment-name)

Figure 17.19 Table model for generalization — one superclass table

17.3.7 Summary of Object Model to Table Mapping Rules

Mapping object classes to tables

- Each class maps to one or more tables. (Similarly, a table may correspond to more than one class if they are connected with a one-to-one or one-to-many association.) [Figure 17.6]

Mapping associations to tables

- Each many-to-many association maps to a distinct table. [Figure 17.7]
- Each one-to-many association maps to a distinct table or may be buried as a foreign key in the table for the many class. [Figures 17.8 through 17.11]
- Each one-to-one association maps to a distinct table or may be buried as a foreign key in the table for either class.
- For one-to-many and one-to-one associations, if there are no cycles, you have the additional option of storing the association and both related objects all in one table. Be aware this may introduce redundancy and violate normal forms.
- Role names are incorporated as part of the foreign key attribute name. [Figure 17.12]
- N-ary (n>2) associations map to a distinct table. Sometimes, it helps to promote an n-ary association to a class. [Figure 17.12]
- A qualified association maps to a distinct table with at least three attributes — the primary key of each related class and the qualifier. [Figure 17.13, Figure 17.14]
- Aggregation follows the same rules as association.

Mapping single inheritance generalizations to tables

- The superclass and each subclass map to a table. [Figure 17.16, Figure 17.17]
- No superclass table, superclass attributes are replicated for each subclass. [Figure 17.18]
- No subclass tables, bring all subclass attributes up to the superclass level. [Figure 17.19]

Mapping disjoint multiple inheritance to tables

- The superclass and each subclass map to a table. [Figure 17.16, Figure 17.17]

Mapping overlapping multiple inheritance to tables

- The superclass and each subclass map to a table; the generalization relationship also maps to a table.

17.3.8 Another Approach for Mapping Object Models to Tables

Our approach to mapping object structures to tables has been to carry forward as much object model semantics as possible to RDBMS tables. Then we can avail ourselves of inherent RDBMS query, update, and integrity facilities. This is the approach that we describe in this chapter.

Another approach to mapping object models is to store <entity-name, key, attribute-name, value> tuples. Then, one table can store all classes, associations, and generalizations for an entire object model. For example, the object "John Smith" would be stored as <person, 123, person-name, "John Smith">, <person, 123, address, "456 Ocean Drive">, and <person, 123, salary, $25000>. Many knowledgebase-to-DBMS translators use this approach.

In general, putting all entities in one table is not a good approach to relational database design. The one table approach subverts an important benefit of databases: a database is more than just a repository of data, and is supposed to be self-descriptive. A database not only stores data but also stores the structure of its data (metadata). In a sense, the one table approach removes metadata from the database and puts it into the application code that must decipher the one large table.

17.4 ADVANCED RELATIONAL DBMS

Research is currently underway on DBMS more powerful than the conventional SQL RDBMS described in this chapter. Section 15.8.5 discussed one vein of research: object-oriented programming languages with persistent data. This section describes another approach: advanced RDBMS.

An explicit goal of advanced RDBMS is to make as few changes as possible to the relational model. The primary thrust is to extend the relational model with new data types, operators, and access methods. For example, a mathematical application may require complex numbers as a data type. A decision table application may require new operations that check a decision table for completeness and consolidate rows for cases where a decision criterion is irrelevant. Special access methods like a grid search would help spatial applications.

The rationale is that advanced applications demand a wide range of capabilities. No single DBMS can implement the features needed to satisfy all applications. Thus an advanced RDBMS must provide a support system that enables application developers to add custom features. The architecture is open as contrasted with the closed architecture of conventional RDBMS.

Some advantages of the advanced RDBMS approach are:

- *Definitely adds to existing RDBMS functionality.* It preserves the traditional strengths of RDBMS: many simultaneous users, large quantities of data, reliability, distributed data management, and programming support tools.
- *Integrates well with existing relational databases.* It provides for smooth flow of data between engineering and business applications.
- *Data sharing.* The database is truly a central repository and not wedded to any particular programming language or application.

Some potential disadvantages are:

- *Performance.* Is even an augmented RDBMS capable of efficient operations on individual objects?

- *Functionality*. Does the RDBMS paradigm interfere with the ability to deliver needed capabilities? If so, will these limitations cause a problem for real applications?
- *Security*. An open architecture may make it more difficult to protect data against unauthorized reading and writing.

POSTGRES is one example of the advanced RDBMS approach. POSTGRES is a prototype at UC Berkeley and the successor to the INGRES RDBMS. POSTGRES adopts the "one database approach"—one database that can be extended to serve a variety of applications. Supported data types include variable length data, QUEL (a query language similar to SQL) queries, and procedures. POSTGRES provides facilities for active databases, such as triggers and inferencing with forward and backward chaining [Stonebraker-86].

EXODUS is a prototype at the University of Wisconsin. EXODUS adopts the "database generator approach"—a host of custom DBMS each serving an application niche. EXODUS provides kernel facilities for use by all applications and a set of tools to aid the database implementor in generating the custom portion of each DBMS [Carey-86].

17.5 CHAPTER SUMMARY

A Data Base Management System (DBMS) is a computer program that is designed to provide general purpose functionality for storing, retrieving, and controlling access to permanent data. A DBMS protects data against accidental loss and makes it available for sharing. An entire branch of computer science is devoted to studying DBMS issues and answering questions such as:

- What paradigm is most conducive to representing database structure?
- What kind of language provides the most natural interaction?
- How can a database capture more of the true meaning of its data?
- How should a database be organized for maximum performance and flexibility?

Several DBMS paradigms are available: hierarchical, network, relational, and object-oriented. Hierarchical and network DBMS bring the conceptual DBMS close to the underlying physical data structures. Thus they are efficient but difficult to use and fading in the marketplace. Relational DBMS dominate today's marketplace. Relational DBMS present the database at a higher level of abstraction than hierarchies and networks and are easier to use. Relational DBMS implementations are improving in performance as they mature and use smarter optimization techniques. Object-oriented DBMS and/or advanced relational DBMS may be the wave of the future. Much research is underway that is motivated by the lessons learned from current relational DBMS. Many theoretical, performance, and practical issues must be resolved before we learn the fate of these new approaches.

Object-oriented concepts provide an excellent basis for modeling hierarchical, network, relational, and object-oriented DBMS. Object models permit developers to think about a problem at a high, abstract level and yet rest assured that the resulting design can be easily

and practically implemented. The following simple rules enable designers to convert an object model to relational DBMS tables.

- An object class maps to a table.
- An association maps to a table.
- A generalization maps to a superclass table plus a series of subclass tables.

We presented various alternatives that deviate from the basic conversion formula. These alternatives trade-off performance, integrity, and extensibility.

aggregation (in RDBMS schema)	object (in RDBMS schema)
association (in RDBMS schema)	primary key
class (in RDBMS schema)	qualified association (in RDBMS schema
foreign key	schema
generalization (in RDBMS schema)	table
integrity of data	ternary association (in RDBMS schema)
normal forms	three schema architecture

Figure 17.20 Key concepts for Chapter 17

BIBLIOGRAPHIC NOTES

Many good books explain DBMS and relational DBMS principles. C.J. Date has written numerous books [Date-81] [Date-83] [Date-86] [Date-87]. Date is a polished writer and has deep insights into relational DBMS technology derived from his past work for IBM. [Loomis-87a] is another good DBMS reference.

Our approach to database design is not all that much different from other authors. After all, there is only so much a person can say about mapping to simple tables. Chapter 12 of this book reviews other approaches to database design. Part of [Loomis-87a] discusses database design but this has been superseded by her more recent work [Loomis-87b]. Our work improves on other approaches with: a better modeling notation, an intermediate table model, more attention paid to integrity, and more thorough mapping rules.

Our treatment of relational database design is detailed but is not complete. We make no provision for handling versions and alternatives; we have not decided how to model them. We do not address database security (SQL GRANT command) and how to analyze dynamic interaction with a database (transaction modeling). We have been automating the conversion of object models to relational DBMS schema, but this work is not described here. (A user interactively draws object models on the CRT screen, annotates them with database implementation decisions, and then asks the system to automatically generate a relational DBMS schema definition. Chapter 18 describes some of our early work in this area.) [Oertly-89] discusses evolutionary database design—how to accommodate changes in the structure of a database that is already populated with data. Oertly's treatment is compatible with our work.

REFERENCES

[Blaha-88] Michael Blaha, William Premerlani, James Rumbaugh. Relational database design using an object-oriented methodology. *Communications of the ACM 31*, 4 (April 1988), 414-427.

[Carey-86] M. Carey, D. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J. Richardson, E. Shekita. The architecture of the EXODUS extensible DBMS. *International Workshop on Object-Oriented Database Systems*, Pacific Grove, Calif., September 1986.

[Date-81] Chris J. Date. *An Introduction to Database Systems, Third Edition*. Reading, Mass.: Addison-Wesley, 1981.

[Date-83] Chris J. Date. *An Introduction to Database Systems, Volume II*. Reading, Mass.: Addison-Wesley, 1983.

[Date-86] Chris J. Date. *Relational Database: Selected Writings*. Reading, Mass.: Addison-Wesley, 1986.

[Date-87] Chris J. Date. *A Guide to the SQL Standard*. Reading, Mass.: Addison-Wesley, 1987.

[Keller-86] Arthur M. Keller. The role of semantics in translating view updates. *IEEE Computer 19*, 1 (January 1986), 63-73.

[Kent-83] William Kent. A simple guide to five normal forms in relational database theory. *Communications of the ACM 26*, 2 (February 1983), 120-125.

[Loomis-87a] Mary E.S. Loomis. *The Database Book*. New York: Macmillan, 1987.

[Loomis-87b] Mary E.S. Loomis, Ashwin V. Shah, James E. Rumbaugh. An object modeling technique for conceptual design. *European Conference on Object-Oriented Programming*, Paris, France, June 15-17, 1987, published as *Lecture Notes in Computer Science, 276*, Springer-Verlag.

[Oertly-89] Fredy Oertly, Gerald Schiller. Evolutionary database design. *Fifth International Conference on Data Engineering*, Los Angeles, 1989.

[Stonebraker-86] Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. *ACM SIGMOD'86*, Washington, D.C., May 28-30, 1986.

EXERCISES

17.1 (6) Four different object diagrams for directed graphs are shown in Figure E17.1 through Figure E17.4. A directed graph is constructed from named edges and vertices. Each directed edge in a graph is an arrow that starts at one vertex and ends at another vertex. Any number of edges may be connected to the same vertex. More than one edge may connect a given pair of vertices, and an edge may connect a vertex to itself.

Figure E17.1 depicts a graph as a many-to-many association between vertices. Directionality is indicated by the role names *from* and *to*. Edges are treated as link attributes. Figure E17.2 depicts a graph as a many-to-many association between edges. Vertices are link attributes. The qualifier, *end*, is an enumerated type with possible values of *to* and *from* indicating which ends of the edges are connected. Figure E17.3 treats both vertices and edges as objects. A graph is described by connections between vertices and edges. Two named associations, *To* and *From*, are used to store connections, one for each end of an edge. Figure E17.4 depicts connections with a qualified association. Each end of an edge connects to exactly one vertex. Once again, *end* is an enumerated type.

Which diagram most accurately models a graph? Discuss in detail the relative merits of each diagram. Consider what happens if more than one edge connects a given pair of vertices. What if an edge connects a vertex with itself? What happens if only one edge is connected to a vertex?

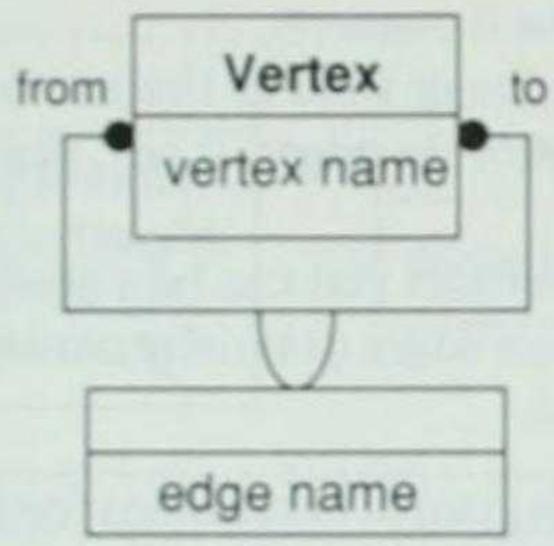


Figure E17.1 Object diagram of directed graphs with edge name as a link attribute

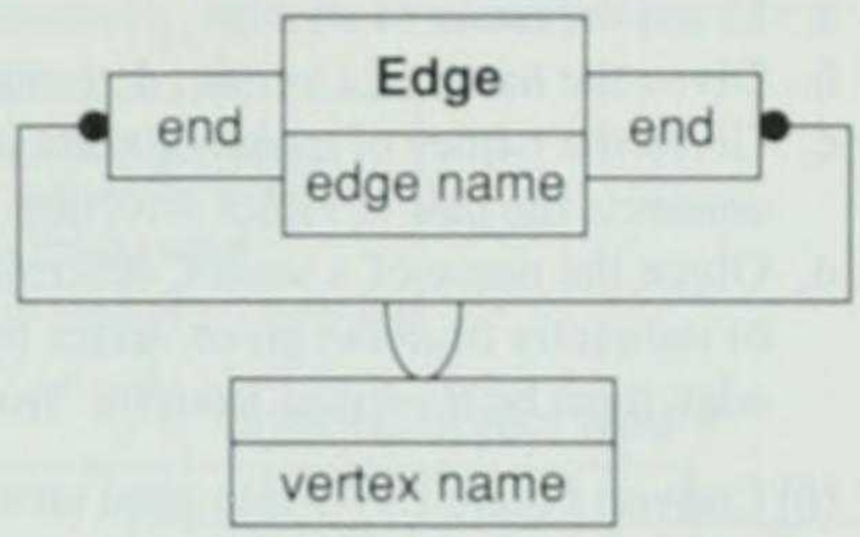


Figure E17.2 Object diagram of directed graphs with vertex name as a link attribute

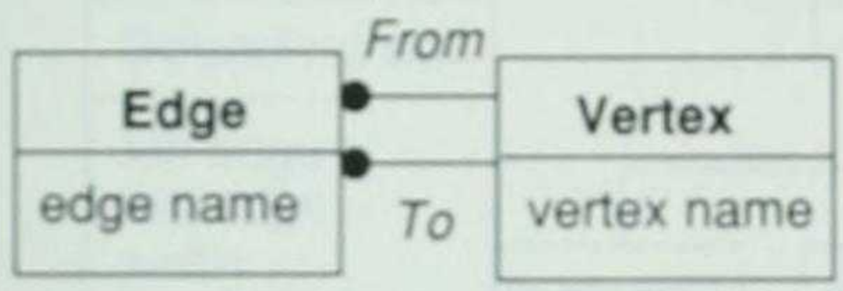


Figure E17.3 Object diagram of directed graphs with associations for each end of an edge

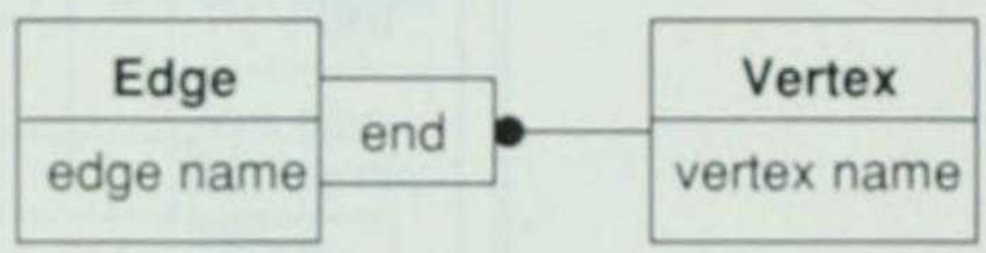


Figure E17.4 Object diagram of directed graphs with an association for connections

17.2 (5) Convert each diagram from the previous exercise into ideal tables (the table model described in the chapter).

17.3 (4) Convert the ideal tables for the object diagrams in Figure E17.3 and Figure E17.4 into SQL commands to create empty database tables and indexes. Use your judgement to supply information that is missing from the object diagrams and needed for the conversion process.

17.4 (4) Populate the database tables created by the SQL commands of exercise 17.3 for the directed graph in Figure E17.5.

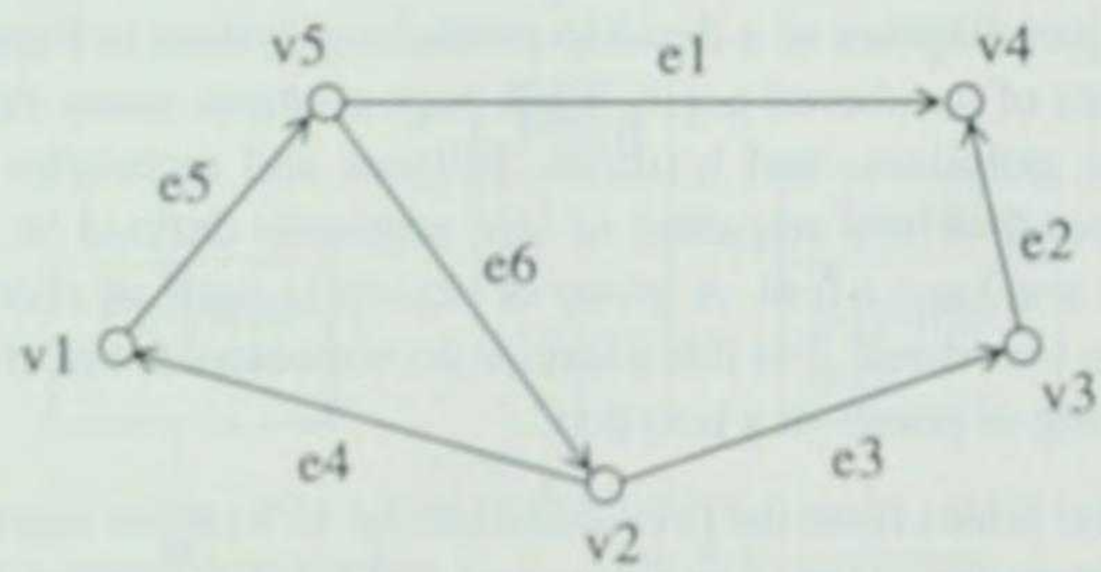


Figure E17.5 Sample directed graph

17.5 (6) For the object diagram in Figure E17.4, prepare queries for the following. You may express queries in SQL or in your own language. In some cases, you may need to embed a query within a procedure.

a. Given the name of an edge, determine the two vertices that it connects.
b. Given the name of a vertex, determine all edges connected to or from it.
c. Given the names of a pair of vertices, determine the name of the edges, if any, that directly connects the pair in either direction.
d. Given the name of a vertex, determine the names of all vertices that can be visited directly or indirectly from the given vertex by traversing one or more edges (transitive closure). Each edge must be traversed from its "from" end to its "to" end.

17.6 (6) Convert Figure E17.6 into ideal tables. This is an object diagram of expressions formed from constants, variables, and arithmetic operators. Unary minus is not allowed. An expression is a binary tree of terms.

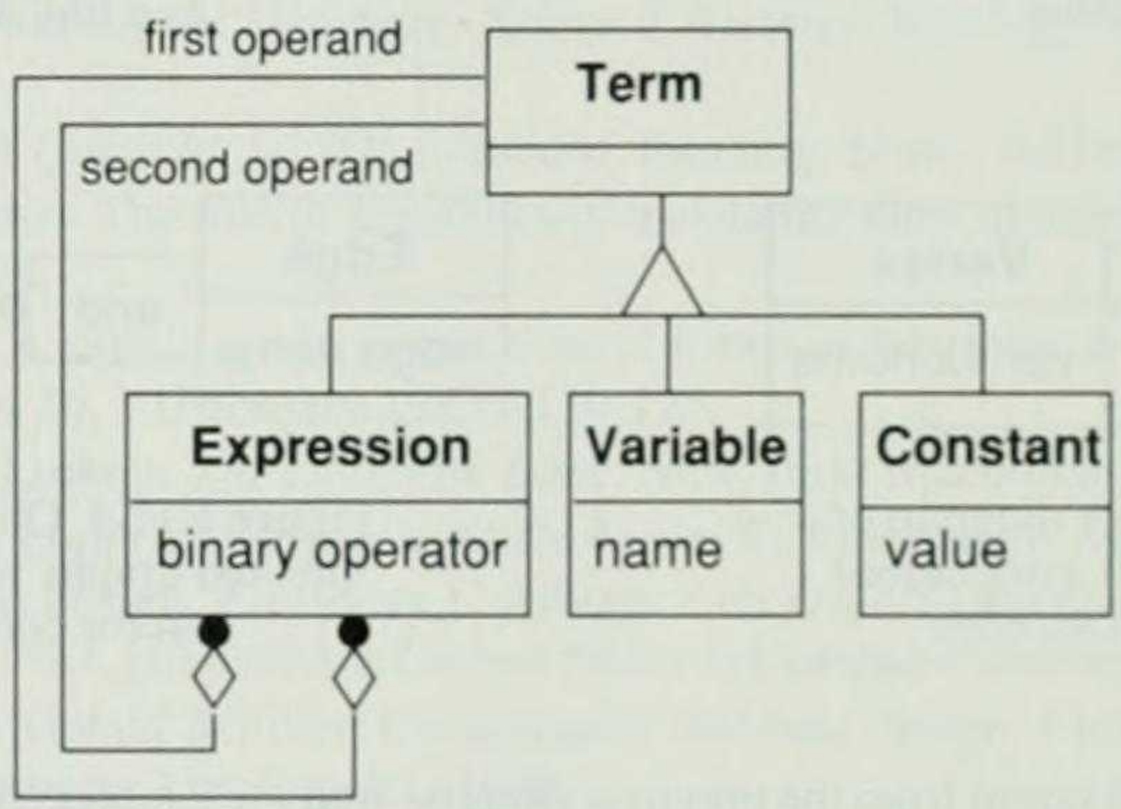


Figure E17.6 Object diagram for expressions

17.7 (4) Convert the ideal tables to RDBMS SQL commands for the previous exercise.

17.8 (5) Indicate the contents of the database tables created by the SQL commands of the previous exercise for the expression $(X + Y/2) / (X/3 - Y)$. Consider the parentheses in establishing the precedence of operators; otherwise, ignore them in populating the database tables.

17.9 (7) Convert the object diagram of a desktop publishing system in Figure E17.7 to ideal tables. A document consists of numbered pages. Each page contains many drawing objects, including ellipses, rectangles, polylines, and textlines. Ellipses and rectangles are embedded within a bounding box. A polyline is a sequence of line segments defined by vertex points. Textlines originate at a point and have a font. A group of objects is itself an object. Treat all associations and aggregations as unordered. For this exercise do not consider the ordering of pages in a document or the ordering of points in a polygon.

17.10 (6) Revise your ideal tables from the previous exercise to treat the association between *Polyline* and *Point* and the aggregation between *Document* and *Page* as ordered. That is, given a polyline, it must be possible to query the database to retrieve the points in the correct order, and given a document it must be possible to scan its pages in order.

17.11 (5) Modify the ideal tables produced in exercise 17.9 to reflect the revision to the object diagram shown in Figure E17.8. Discuss the relative merits of making the revision.

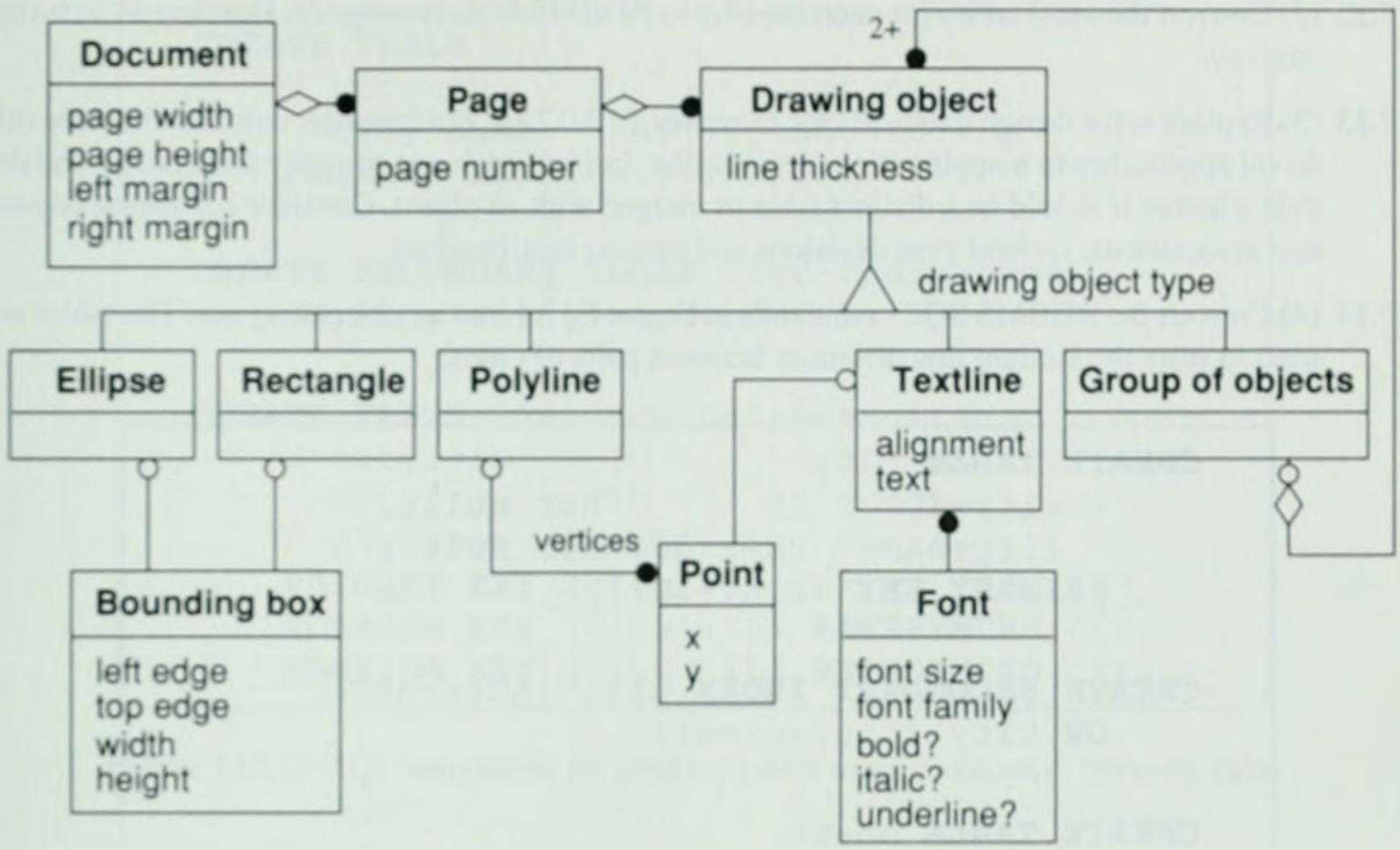


Figure E17.7 Object diagram of a desktop publishing system

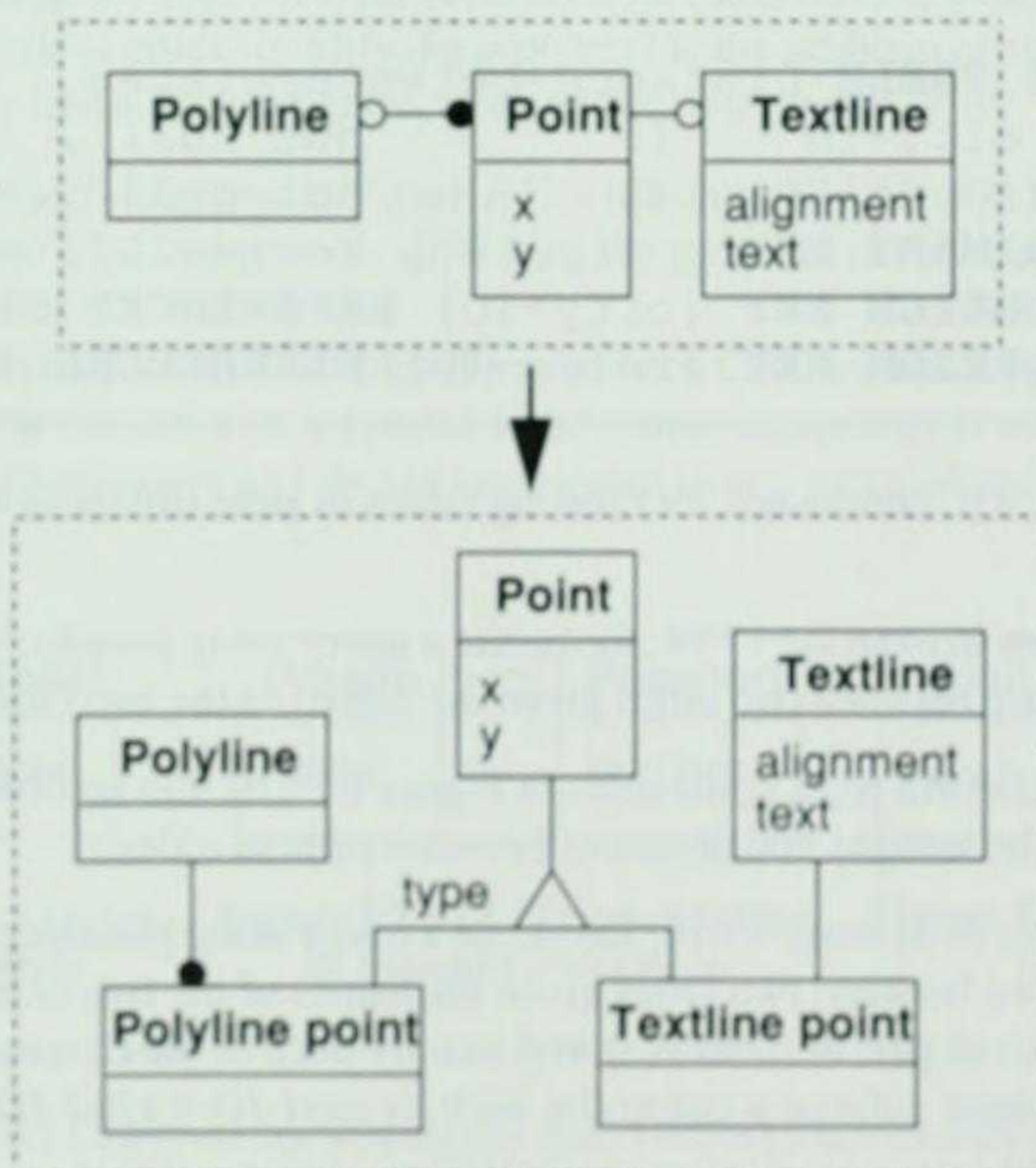


Figure E17.8 Generalization of point to eliminate zero-or-one multiplicity

17.12 (5) Convert the ideal tables for exercise 17.9 to RDBMS SQL commands. Develop an indexing strategy.

17.13 (5) Explore some design trade-offs for exercises 17.9-17.12. For instance, consider the three different approaches to mapping the generalization. Look at each one-to-many association, and decide whether it should be a distinct table or merged with an object. Consider collapsing one-to-one associations. Defend your decisions and discuss ramifications.

17.14 (4) Convert the RDBMS SQL commands in Figure E17.9 into an object diagram. The tables are used to store the straight line distances between pairs of cities.

```
CREATE TABLE City
    ( city-ID     ID        not null ,
      city-name   char(30)  not null ,
    PRIMARY KEY (city-ID));

CREATE SECONDARY INDEX City-index-name
    ON City (city-name);

CREATE TABLE Route
    ( route-ID    ID          not null ,
      distance    real                 ,
    PRIMARY KEY (route-ID));

CREATE TABLE Distance-between-cities
    ( city-ID     ID          not null ,
      route-ID    ID          not null ,
    PRIMARY KEY (city-ID , route-ID),
    FOREIGN KEY (city-ID) REFERENCES City,
    FOREIGN KEY (route-ID) REFERENCES Route);
```

Figure E17.9 SQL commands for creating tables to store distances between cities

17.15 (4) Using the tables in exercise 17.14, formulate a query using pseudocode or SQL that will determine the distance between two cities given the names of the two cities.

17.16 (4) Convert the RDBMS SQL commands in Figure E17.10 into an object diagram. The tables are used to store the straight line distances between pairs of cities.

17.17 (4) Using the tables in exercise 17.16, formulate a query using pseudocode or SQL that will determine the distance between two cities given the names of the two cities. Assume that the distance between a given pair of cities is stored exactly once in the *Distance-between-cities* table. (The application must enforce a constraint such as *city1-ID* < *city2-ID* so that the distance is only entered once.)

17.18 (5) Discuss the relative merits of the two approaches in the previous four exercises for storing distance information.

```
CREATE TABLE City
    ( city-ID     ID        not null ,
      city-name   char(30)  not null ,
    PRIMARY KEY (city-ID));

CREATE SECONDARY INDEX City-index-name
    ON City (city-name);

CREATE TABLE Distance-between-cities
    ( city1-ID     ID        not null ,
      city2-ID     ID        not null ,
      distance     real      not null ,
    PRIMARY KEY (city1-ID , city2-ID),
    FOREIGN KEY (city1-ID) REFERENCES City,
    FOREIGN KEY (city2-ID) REFERENCES City);
```

Figure E17.10 SQL commands for creating tables to store distances between cities

17.19 (5) Discuss the similarities and differences between the database tables used to store edge and vertex information in exercises 17.1- 17.5 and the tables used to store distance information between cities in exercises 17.14 -17.18. How does fact that there is exactly one straight line distance between a pair of cities simplify the problem? Is the problem of storing distances between cities more nearly like a directed graph or an undirected graph? Why?

17.20 (5) Convert the object diagrams in Figure E17.11, Figure E17.12, and Figure E17.13 into ideal tables and then into SQL commands. How does the presence of the zero-one multiplicity influence the conversion?

Figure E17.13 is a portion of an object model for a CAD system. A piece of equipment may be indicated on the screen with a special icon; minor equipment is not shown on the screen. Some icons are graphic-only and do not correspond to any piece of equipment.

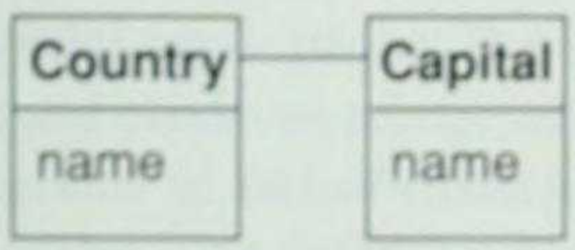


Figure E17.11 Object diagram of capitals of countries

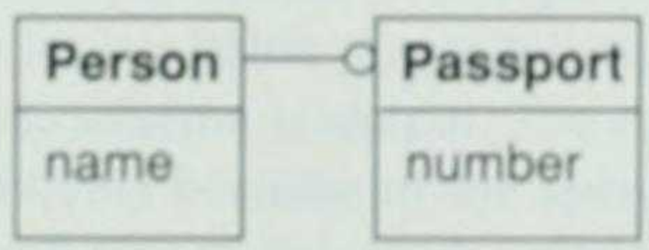


Figure E17.12 Object diagram of passport ownership

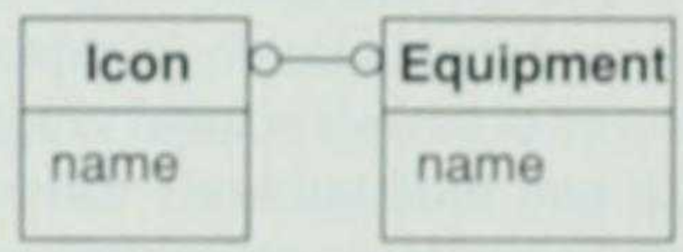


Figure E17.13 Object diagram for CAD system

17.21 (5) Develop an object diagram for storing the status of forks and philosophers for the dining philosopher's problem. (See exercise 3.22.) Convert your diagram to RDBMS SQL commands. Show possible contents of the database tables for the situation in which each philosopher has exactly one fork.

17.22 (8) Convert the diagram in Figure E17.14 into RDBMS SQL commands.

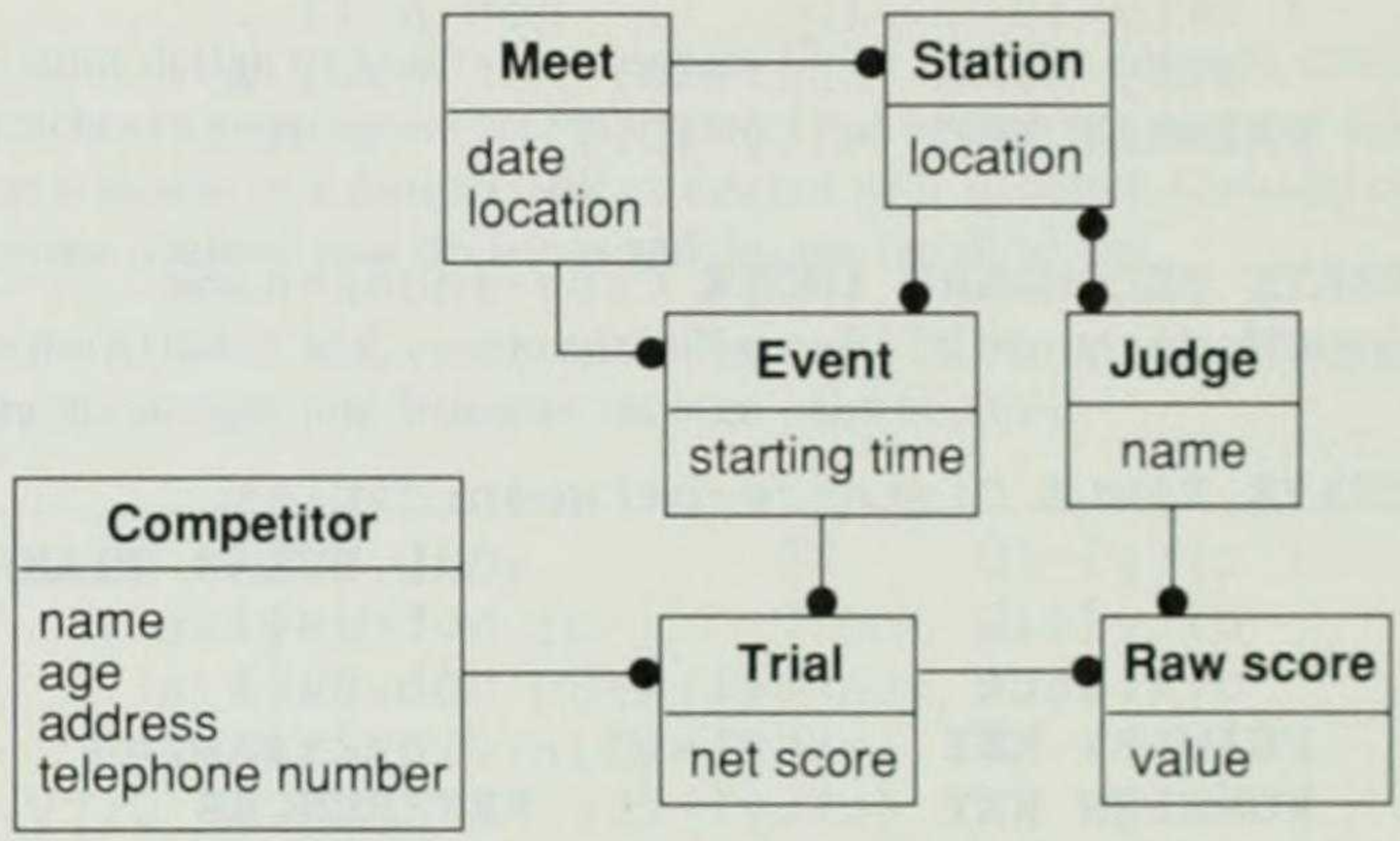


Figure E17.14 Partial object diagram for a scoring system

17.23 (5) What new features should be added to SQL to provide better support for OO concepts?

18

Object Diagram Compiler

Compilers provide a natural application for the OMT methodology. Most compilers are batch programs that consist of one or more passes. Each pass works towards the goal of transforming the input to the output. For example, a C compiler may accept an input file of ASCII characters. Successive passes may parse into tokens, abstract the intended semantics, generate code, and then optimize code. Users expect that a compiler will carefully check its input during processing; the more errors the compiler can detect, the fewer errors will be encountered during program execution.

This chapter describes the development of a compiler for object diagrams for bill-of-material applications. The compiler is a batch program that accepts an object diagram as input and produces relational DBMS schema as output. The compiler input is limited to object diagrams of bill-of-material problems because of the special error checking built into the compiler. The compiler has served two purposes: It solved the immediate problems of the target applications and it was an important milestone in our continuing work on automating the OMT methodology.

The *Object Diagram Compiler* illustrates several concepts mentioned earlier in the book. The compiler architecture consists of four passes that step through a series of internal models. There are many relationships and functional dependencies within a model but few between models. As Chapter 9 notes, this kind of multipass architecture is common for compilers. The design of the compiler also touches upon some database issues discussed in Chapter 17 and metadata concepts described in Chapter 4.

18.1 BACKGROUND

General Electric has several internal applications that configure product assemblies. The business practice for these applications is that the customer specifies product function; the manufacturer must devise a corresponding product. For example, a customer might order a lawn mower that is lightweight, inexpensive, and energy efficient. The manufacturer may fill the order with a mower that has a 1 horsepower gasoline-powered engine, 45 cm diameter aluminum deck, and nonmulching blade. (GE does not manufacture lawn mowers; we use the lawn mower example for convenience.)

The list of parts chosen by a manufacturer to build a product is called a bill-of-material (BOM). A BOM is a tree of the direct and indirect parts that compose an assembly. The literature often refers to BOM as parts explosion problems. BOM are often printed out as indented parts lists, where the indentation corresponds to the depth in the hierarchy. Figure 18.1 shows a sample BOM for a lawn mower. The first value listed after each part type is the part number. Thus lawn mower *LM16G* has engine *E1*, deck *D16*, and so forth. Part properties follow the part number.

```
lawn mower: LM16G
    engine: E1, 1HP, gas powered
        piston: P1, 5 cm length, .5 cm diameter
        carburetor: CARB9
        wires: WI5, quantity 4, oil resistant
        ...
    deck: D16, 45 cm diameter, aluminum
    blade: B16, 40 cm long, non-mulching
    wheels: WH3, solid rubber, 8 cm diameter
    ...
```

Figure 18.1 A portion of a bill-of-material (BOM)

In the past, most product assemblies have been manually designed by human experts. [Blaha-90] describes a new, automated approach to generating BOMs. The new approach uses a relational database to reverse engineer past product designs and extracts the implicit design rules for mapping specifications to choices of parts. The details of the new approach are beyond the scope of this book.

This chapter describes a portion of the new BOM configuration system: a subsystem called the *Object Diagram Compiler*. First, we draw an object diagram of a BOM parts explosion with an ordinary graphics editor by carefully placing many lines, triangles, boxes, text strings, and other primitives. The *Object Diagram Compiler* then processes the object diagram and ultimately generates DBMS commands that create tables and indexes. Whenever we change the object diagram, we rerun the compiler and generate new DBMS commands. Thus the object diagram is a living document. The *Object Diagram Compiler* simplifies documentation and DBMS schema generation by using the same source document for both. Our success with the *Object Diagram Compiler* has motivated further work towards developing more comprehensive tools for automating the OMT methodology.

18.2 PROBLEM STATEMENT

As shown in Figure 18.2, the objective of the *Object Diagram Compiler* is to translate BOM object diagrams into database commands and check for input errors.

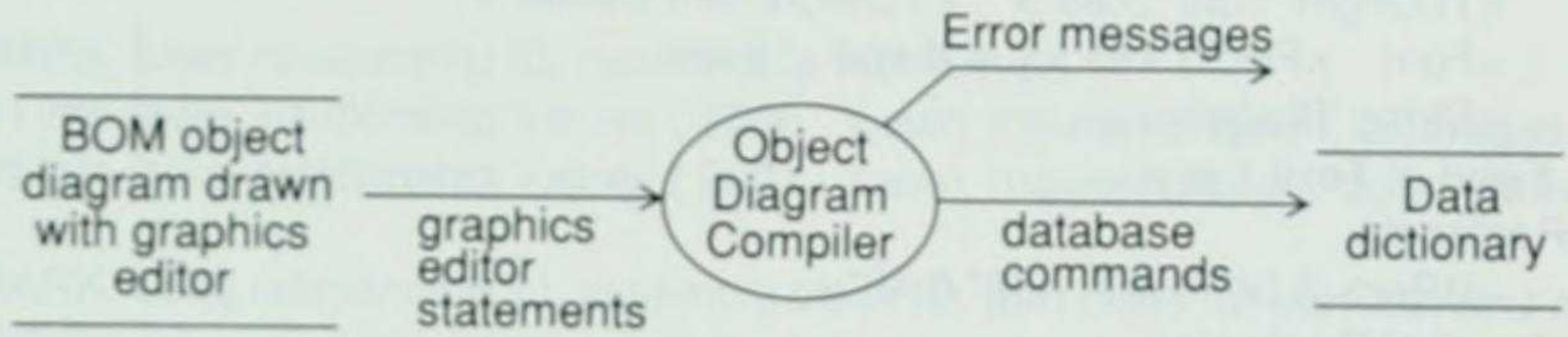


Figure 18.2 Functional model of *Object Diagram Compiler*

18.2.1 Compiler Input

The compiler must read an ASCII description of a BOM object diagram produced by a general purpose graphics editor. We used FrameMaker,* but others such as MacDraw and Interleaf are also suitable. The graphics editor must be geometric-shape-based and not pixel-based. The compiler input is analogous to the source code of a conventional programming language compiler.

Figure 18.3 is a sample BOM object diagram. A lawn mower has a blade and engine. Each blade and engine may be associated with several lawn mowers. (Figure 18.3 refers to catalog descriptions of parts and not the physical parts. See Section 4.5, *Metadata*, for details.) *Lawn mower*, *Blade*, and *Engine* are described by attributes. Figure 18.4 shows some of the corresponding Framemaker language for the image of the *Blade* class.

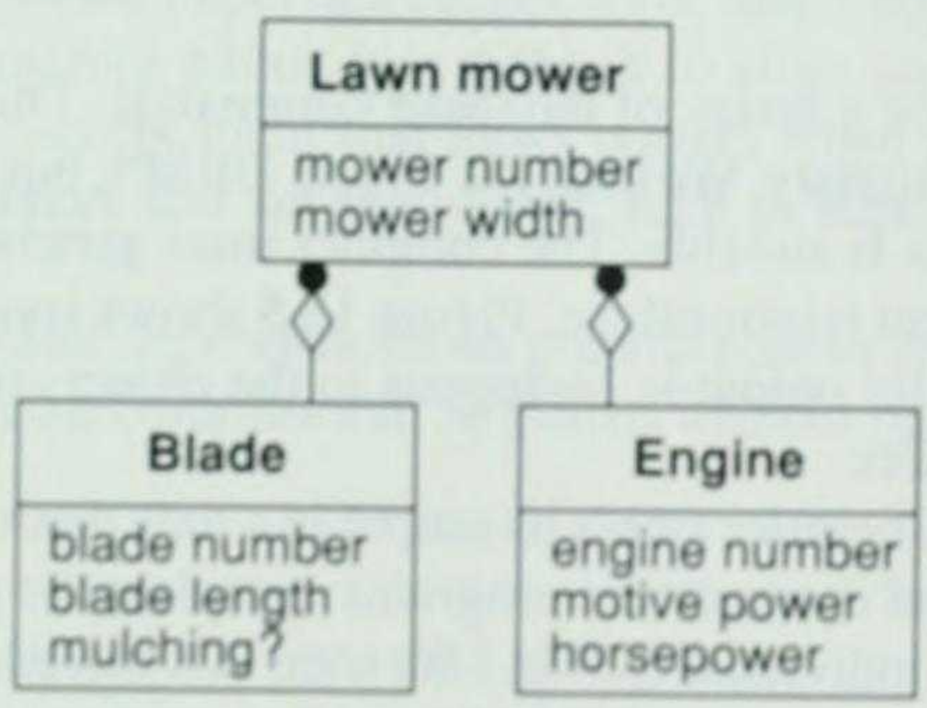


Figure 18.3 Sample BOM object diagram

The *Object Diagram Compiler* must support basic object modeling constructs: classes, attributes, binary associations, binary aggregations, and generalizations. The compiler does not need to support object modeling constructs that are not needed for our BOM applications, such as qualified associations, n-ary relationships, and operations.

* FrameMaker is a desktop publishing system that integrates word processing, graphics, page layout, and other features. We only used the graphic editing capabilities in the *Object Diagram Compiler*.

```
<Rectangle
    <BRect  3.06" 5.65" 1.06" 0.28">
> # end of Rectangle
<TextLine
    <TLOrigin  3.58" 5.83">   <TLAlignment Center >
    <Font   <FBold Yes >   > # end of Font
    <String 'Blade'>
> # end of TextLine
<Rectangle
    <BRect  3.06" 5.93" 1.06" 0.49">
> # end of Rectangle
<TextLine
    <TLOrigin  3.14" 6.08">   <TLAlignment Left >
    <Font   <FPlain Yes >   > # end of Font
    <String 'blade number'>
> # end of TextLine
<TextLine
    <TLOrigin  3.14" 6.22">   <TLAlignment Left >   <String 'blade length'>
> # end of TextLine
<TextLine
    <TLOrigin  3.14" 6.36">   <TLAlignment Left >   <String 'mulching?'>
> # end of TextLine
```

Figure 18.4 Sample input for *Object Diagram Compiler*

18.2.2 Compiler Output

The compiler must produce a series of database commands. This output is executed as a script file to load a data dictionary. We used the Oracle DBMS, but any relational DBMS that supports the SQL language is suitable. The compiler must generate association names and determine how to implement relationships. Figure 18.5 shows some output that corresponds to Figure 18.3. The compiler output is analogous to the object code of a conventional programming language compiler.

The *Object Diagram Compiler* stores its output in a data dictionary custom designed for BOM applications. (Several downstream programs complete the process of creating the tables and indexes that store individual BOMs.) We used this custom dictionary rather than the DBMS dictionary because we required BOM specific extensions. Section 18.3.6 describes our data dictionary.

18.2.3 Miscellaneous Requirements

The *Object Diagram Compiler* was subject to further requirements on development time and performance. The compiler had to be developed quickly—in four months or less. It was part of the larger BOM generation project, and we did not want to divert our attention. Our requirements on execution time were more lax. An execution time of two hours or less is ad-

```
REMARK  Load object class ID, object class name, table name
Insert into meta_object_table Values ( 1000 , 'Lawn mower' , 'Lawn_mower' );
Insert into meta_object_table Values ( 1001 , 'Blade' , 'Blade' );
Insert into meta_object_table Values ( 1002 , 'Engine' , 'Engine' );

REMARK  Load relationship ID, relationship name, type
Insert into meta_relationship Values ( 2000 , 'Lawn_mower#Engine' , 'aggregation' );
Insert into meta_relationship Values ( 2001 , 'Lawn_mower#Blade' , 'aggregation' );

REMARK  Load relationship ID, assembly class ID, part class ID, part min and max
multiplicity
Insert into meta_aggregation Values ( 2000, 1000, 1002, 1, 1 );
Insert into meta_aggregation Values ( 2001, 1000, 1001, 1, 1 );

.....
```

Figure 18.5 Sample output from *Object Diagram Compiler*

equate because the compiler is a batch program that is only run at system set-up time. The compiler is not used for the day-to-day generation of BOMs. (As it turned out, because of skillful software engineering, performance was much better than required!)

The compiler must detect input errors but need not correct the errors. For instance, class names must be unique, have less than 31 characters, and begin with an alphabetic character. The compiler must not abort when it encounters user errors. The compiler must accept bad input and keep running so that the user can detect multiple errors from a single run.

The compiler must tolerate imperfect connections. Since object diagrams are prepared with a general purpose graphics editor, it is difficult to align and precisely connect everything. The compiler must accept input that looks correct when printed on paper. Also the compiler must detect mistakes that are not visible, such as superimposed drawing components.

The compiler must isolate dependencies on graphic editors and relational DBMS within one or two modules. Then the compiler can be ported without major rewriting.

18.3 ANALYSIS

Analysis is the first step towards solving the problem statement. It quickly became apparent that the difference in level of abstraction between input and output was too great to span in a single step. We decided that it would be best to use several passes. This conclusion is not surprising, since many compilers require multiple passes.

The disadvantage of a multipass approach is that the developer must prepare multiple object models and design multiple compilation phases. The advantage of multiple passes is that complexity is addressed in a divide and conquer fashion. The resulting code is easier to port since graphic editor and DBMS dependencies are isolated in small portions of code.

Figure 18.6 elaborates the functional model of the *Object Diagram Compiler* shown in Figure 18.2. Each ellipse represents a compiler pass. The three data stores at the bottom of the figure denote intermediate object models. The remainder of this section describes the compiler passes and intermediate object models. There is no significant dynamic model for the compiler, since it is a batch transformation.

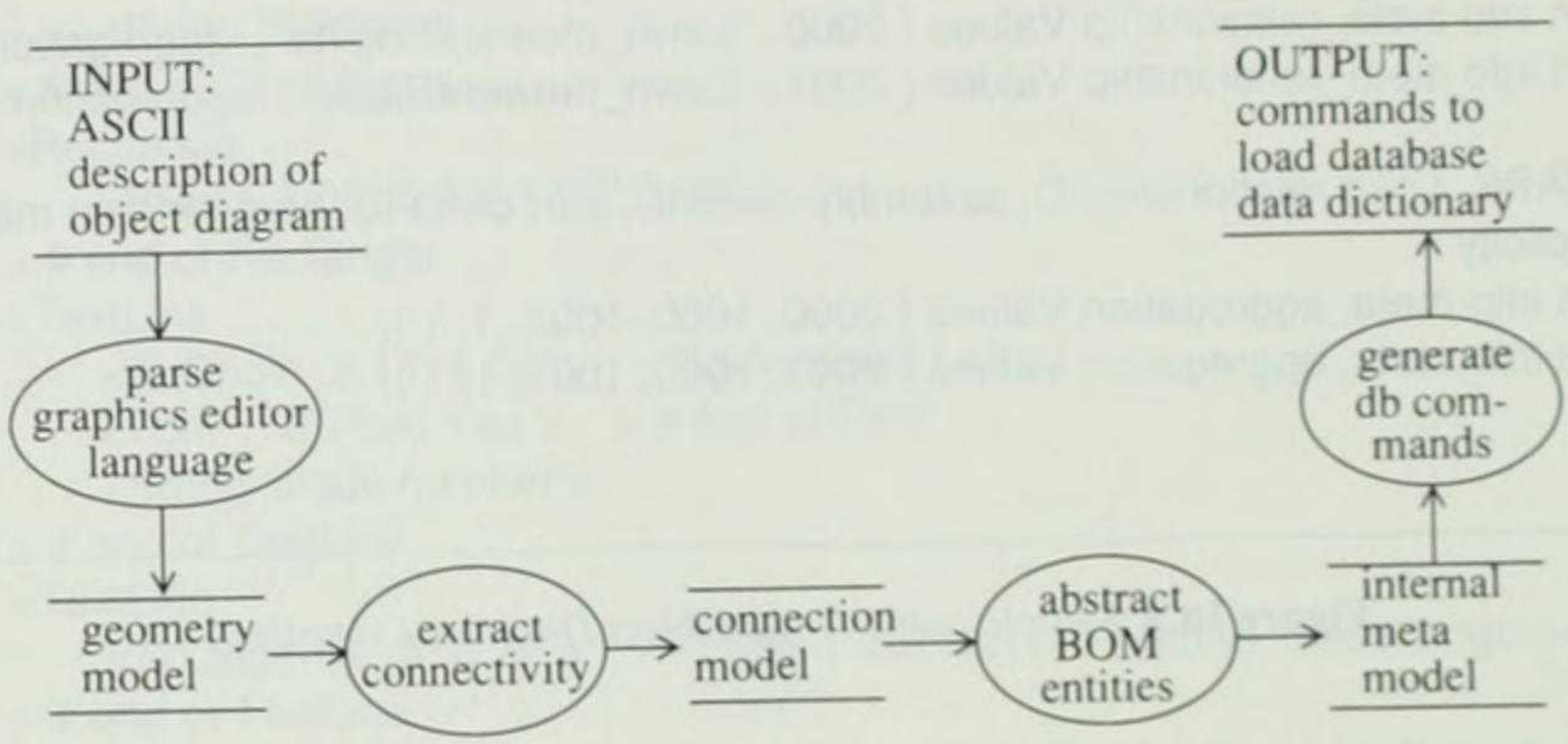


Figure 18.6 Functional model of *Object Diagram Compiler*

18.3.1 Parse Graphics Editor Language

The first pass reads in the raw diagram Framemaker file generated by the graphics editor (a sample is shown in Figure 18.4) and builds the geometry model. The first pass corresponds to lexical analysis in a conventional programming language compiler. This pass required little analysis effort beyond figuring out a correct BNF syntax for the graphics editor language. Our development of a BNF grammar for the graphics editor was aided by vendor documentation.

18.3.2 Geometry Object Model

Figure 18.7 summarizes the geometry model. We have omitted some classes, associations, and attributes in order to conserve space and simplify explanation. The geometry model represents a diagram as a series of graphic primitives and is a convenient form for further processing. The geometry model regards an object diagram as simply a picture. The graphical objects (ellipses, rectangles, polylines, polygons, and textlines) coexist but have no interrelationships. We derived the geometry model directly from the BNF grammar of the graphics editor language.

The graphics editor that we used organizes components by page. A *Textline* is a single line of text that may be placed anywhere on a page. A *Textline* has a font and alignment and originates at a point. Each *Font* describes text size and special features such as italics, bold, and underlining. The geometry model includes *page number* for use in error messages.

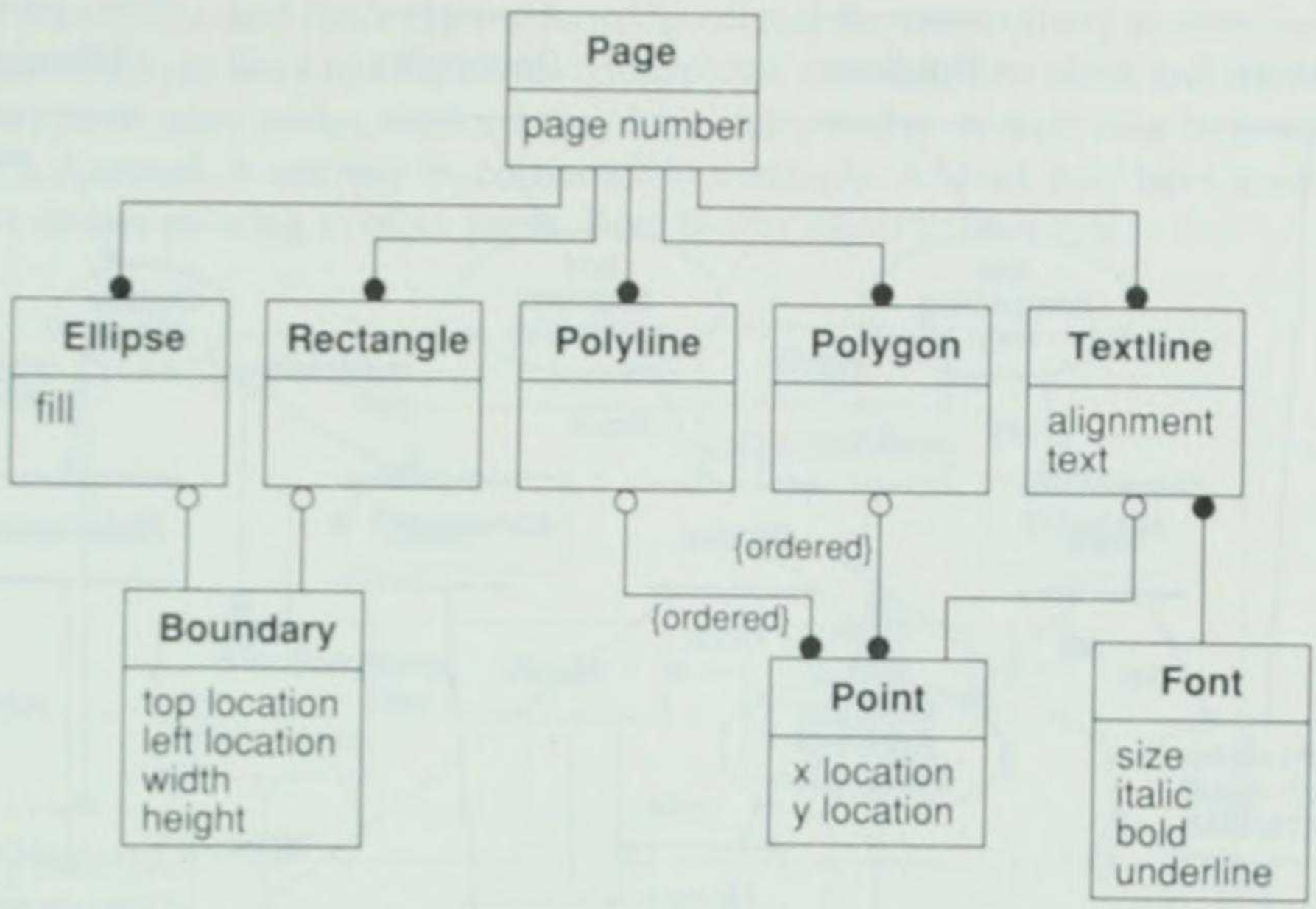


Figure 18.7 Geometry object model

Ellipse, *Rectangle*, *Polyline*, and *Polygon* capture graphical information in the object diagram. Ellipses have a location and a bounding rectangle. An ellipse may have a hollow fill or solid fill. Rectangles have a location, width, and height. Polylines and polygons are composed of straight line segments that connect a sequence of points. A polyline is an arbitrary sequence of lines; a polygon is a closed geometric figure.

The geometric model for our lawn mower example (Figure 18.3) contains the following objects:

- 6 rectangles,
- 11 textlines (3 bold font, 8 normal font),
- 2 polygons (diamonds),
- 2 ellipses (which happen to be circles) with solid fill, and
- 2 polylines (vertical lines between *Lawn-mower* and *Blade*, *Lawn-mower* and *Engine*).

18.3.3 Extract Connectivity

The second pass consolidates raw geometry, classifies sizes and shapes, and detects intersections and attachments of geometrical primitives. We call the output from the second pass the connection model.

Figure 18.8 presents part of the data flow diagram for this pass. *Ellipses*, *Rectangles*, *Polylines*, *Polygons*, and *Textlines* from the geometry model are input data stores in the functional model. Every class and association in the connection model is the output of one or

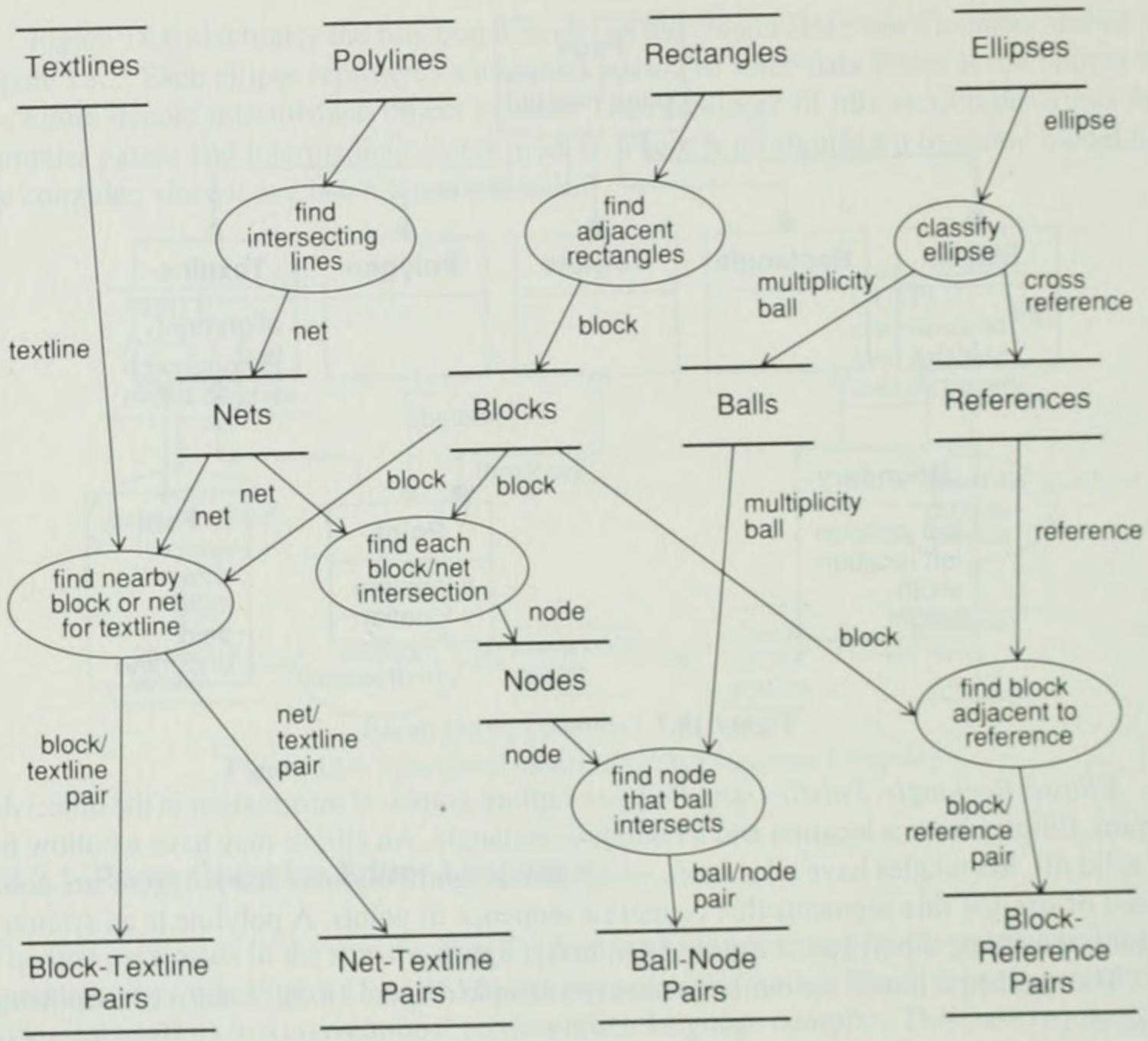


Figure 18.8 Connectivity extraction functional model

more functional model processes. Recall that during analysis we do not specify algorithms but specify the effects of processes.

In Figure 18.8, groups of contiguous rectangles are recognized as blocks. Groups of intersecting lines are recognized as nets. Each block and net intersection is found and stored as a node. Ellipses are classified into multiplicity balls or sheet cross references. Each multiplicity ball is associated with a node; each reference is associated with a block. Each textline is bound to its nearest block or net.

18.3.4 Connection Object Model

The connection model, summarized in Figure 18.9, bridges the geometry model and the internal meta model. Classes that are shaded are carried forward from the geometry model. The connection model regards an object diagram as a topology, a collection of blocks connected with nets. Blocks and nets have attached geometry and text.

The *Block*, *Net*, and *Node* classes form the core of the connectivity model. Each block has a name; each net has a name. A node corresponds to exactly one block and one net. Each block may have many nodes; each net may have many nodes. A node may be connected to a ball or a diamond. A net may be connected to a triangle. A block may have several cross reference circles pointing to other pages. Both blocks and nets have text.

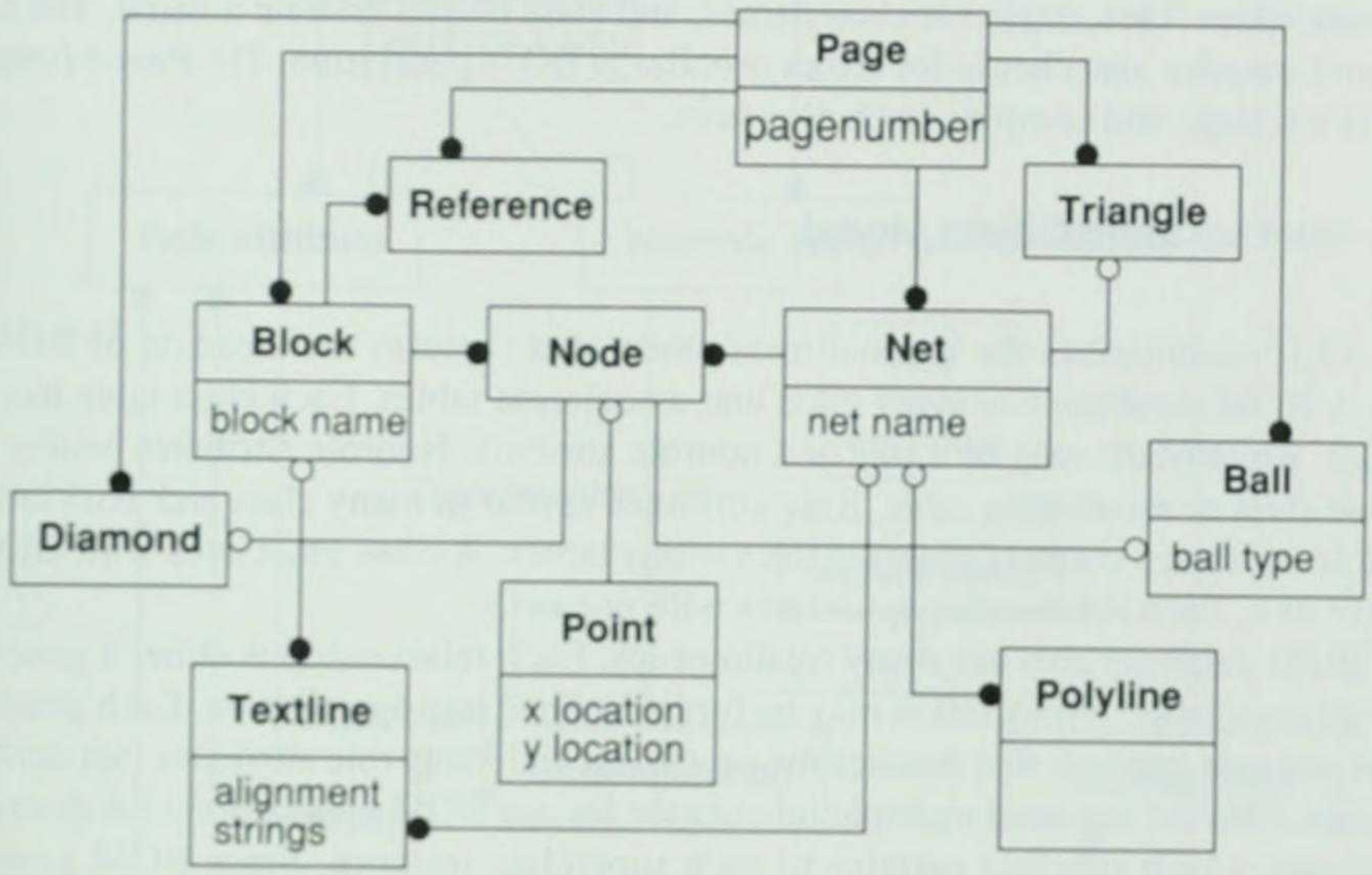


Figure 18.9 Connection object model

The connection model for our lawn mower example contains the following objects:

- 3 blocks named *Lawn-mower*, *Blade*, and *Engine*,
- 2 nets named *Lawn-mower#Blade* and *Lawn-mower#Engine*,
- 4 nodes,
- 2 diamonds each associated with a node, and
- 2 solid balls each associated with a node.

The *Lawn-mower* block is associated with two textlines: *mower-number* and *mower-width*. Similar associations occur for *Blade* and *Engine*.

18.3.5 Abstract BOM Entities

The third pass raises the level of abstraction. The graphical symbols in a BOM diagram have more meaning than their mere shape. A block denotes a class; a net represents a generalization or association. Pass 3 coalesces cross references to classes, verifies uniqueness of

names, and classifies relationship types. Pass 3 applies hard coded rules for mapping relationships to tables (see Chapter 17). The second and third passes correspond to semantic analysis in a conventional programming language compiler.

The functional model for this pass is complicated by extensive error checking. Much of this error checking derives from using a general purpose graphics editor to construct the input object diagrams. In general, the user can make many diagramming errors, such as dangling association lines, duplicate class names, and class blocks without a name. The *Object Diagram Compiler* also checks for errors peculiar to BOM generation. The Pass 3 functional model is too large and complex to display here.

18.3.6 Internal Meta Object Model

Figure 18.10 summarizes the internal meta model that captures the meaning of BOM diagrams. A BOM database has many class and association tables. Each class table has many attributes. An attribute may be a role or a nonrole attribute. Nonrole attributes belong to exactly one class or association table. Role attributes appear in many class and association tables. Role attributes connect together the various tables. A class associates with all blocks that refer to it. Each relationship associates with one net.

A BOM database also has many relationships. Each relationship is either a generalization or an association. Association may be further refined into aggregation. Each generalization has one role attribute that denotes the superclass and many role attributes that denote the subclasses. (We did not need multiple inheritance for our BOM applications.) A discriminator indicates which subclass pertains to each superclass instance. Since BOM generation only uses binary associations, each association has exactly two role attributes. An association may have zero or more nonrole attributes.

The internal meta model for our lawn mower example contains the following objects:

- 3 class tables named *Lawn-mower*, *Blade*, and *Engine*,
- 0 association tables. (Our hard coded rules bury both associations in the *lawn-mower* example. See Section 18.4.),
- 3 role attributes: *mower-number*, *blade-number*, and *engine-number*,
- 5 nonrole attributes,
- 2 aggregations (Both associations are aggregations.), and
- 0 generalizations, and 0 link values.

18.3.7 Generate Database Commands

The fourth and final pass transforms the internal meta model into the database meta model. This process is simple because the internal meta model is similar to the database meta model. The last pass is analogous to code generation in a conventional programming language compiler and produces the desired output of DBMS commands.

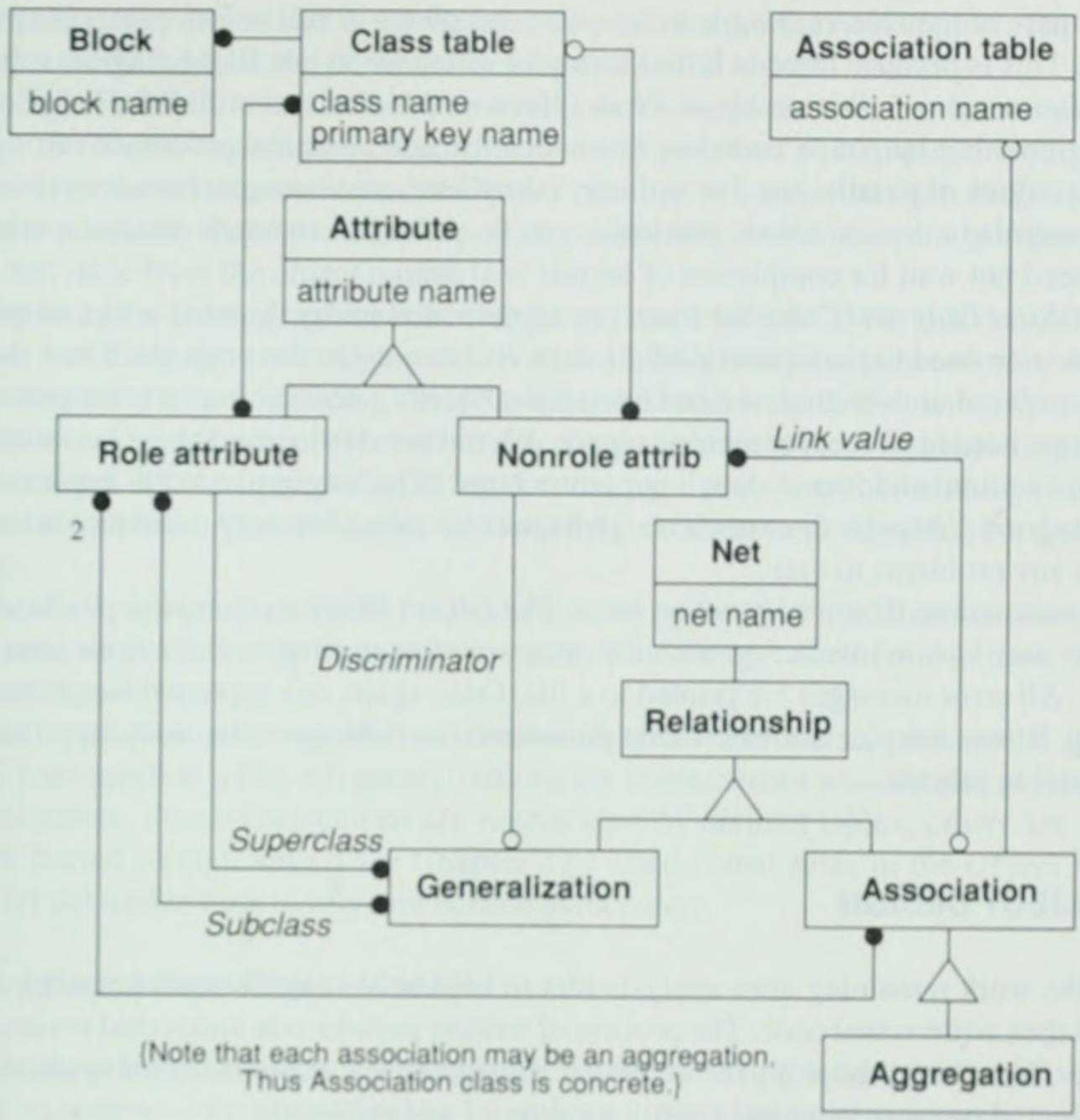


Figure 18.10 Internal meta model

18.4 SYSTEM DESIGN

The choice of system archifecture for the *Object Diagram Compiler* was straightforward and directly follows from the overall functional model in Figure 18.6. We chose to implement the program as a batch transformation as discussed in Chapter 9.

The *Object Diagram Compiler* consists of a sequence of passes that transform between layers. We chose a closed architecture where each layer can only access the objects in itself and the immediate preceding layer. The advantage of a closed architecture is increased information hiding which simplifies debugging, extension, and porting. In general, the disadvantage of a closed architecture (relative to an open architecture) is loss of efficiency. For the *Object Diagram Compiler*, efficiency is good because of the careful design of each pass. Our object models show these linkages between object models as shaded rectangles. Thus *Class table* in the internal meta model refers to *Block* in the connection model.

Each pass of the *Object Diagram Compiler* completes in full before proceeding to the next pass. This is because there is little locality of reference in our BOM diagrams. For example, a *Lawn mower* class on Sheet 1 can refer to a *Blade* class on Sheet 27. In contrast most programming language code has more locality, and compiler processes can operate with some degree of parallelism. For ordinary compilers, once enough characters have been read to recognize a lexical token, the token can be passed to semantic analysis; semantic analysis need not wait for completion of lexical analysis.

The *Object Diagram Compiler* manages all data in memory. Input is a file; output is a file. There is no need to make intermediate data visible outside the program. Since the program has a closed architecture, we considered deallocating objects from earlier passes that are no longer needed to recover memory space. We did not deallocate objects because of the extra logic required and fear of dangling pointer bugs. (The language used to implement the *Object Diagram Compiler* does not have garbage collection.) Memory consumption has not caused us any problems to date.

Implementation of control is not an issue. The *Object Diagram Compiler* is a batch program. The user has no interaction with the program after starting it; there is no interactive debugger. All error messages are printed to a file. Once again, our approach was dictated by simplicity. It was simpler not to provide an interactive debugger; the batch approach has worked well in practice.

18.5 OBJECT DESIGN

Most of the work remaining after analysis was to implement associations, prepare pseudocode, and then write actual code. The process of writing pseudocode uncovered several analysis flaws. We revised the analysis as needed. Design was a straightforward application of object-oriented programming and mostly a matter of adding detail.

18.5.1 Functional Models

In this problem, as in many batch transformations, functional dependencies can be expressed as an acyclic directed graph among data stores containing sets of objects. Each data store contains all the objects of a given class, such as *Rectangles* and *Nodes*, or all the links of a given association, such as *Ball-Node Pairs*. The data flow diagram can be constructed by examining each class and association in the output model and determining the input classes and associations that it depends on.

In many cases the final algorithm can be written down directly without actually drawing a data flow diagram. We found that a "marking" approach aids formulation of functional algorithms. A mark represents a class, association, or attribute that has been computed at a given point in the algorithm. First all input classes and associations are marked on the object diagram. Then all classes and associations in subsequent stages of the batch transformation are examined. Pseudocode for each attribute and association is written in terms of previously marked classes and associations, and the corresponding output attribute or association is marked. When all the attributes of a class are marked, then the entire class is marked. When

the entire output object model is marked, the functional algorithm is complete. This marking technique works because the dependencies form a partial order.

In Figure 18.8, *Blocks* can be computed from *Rectangles*. *Balls* can be computed from *Ellipses*. *Nets* can be computed from *Polylines*. *Nodes* can be computed from *Nets* and *Blocks*. After these computations, *Ball-Node Pairs* can be computed from *Balls* and *Nodes*. We did not actually draw the data flow diagram because the functional algorithm is so directly derivable from the object model.

The first pass (parse graphics editor language) reads the ASCII file that contains a BOM diagram and loads the geometry model. We used UNIX compiler generation tools for this pass to improve our productivity. This pass required little design effort and no pseudocode.

The second pass (extract connectivity) scans the geometry model and produces the connection model. Most of our design effort for this pass was concerned with devising graph traversal algorithms. As an example, Section 18.5.3 describes our algorithm for finding connections.

The third pass (abstract BOM entities) scans the connection model and generates the internal meta model. Like the second pass, most of the design effort for this pass concerned choice of algorithms.

The fourth pass (generate database commands) scans the internal meta model and writes DBMS commands to a file. All generalizations are implemented with distinct superclass and subclass tables. Some associations are represented by distinct tables; others are captured through buried foreign keys. (See Chapter 17.) Hard coded rules in the *Object Diagram Compiler* determine how to implement each association.

18.5.2 Intermediate Object Models

An important design decision was the choice of programming language. Based on past favorable experience, we decided to use an in-house object-oriented programming language called DSM [Shah-89]. DSM's class library provided us with several convenient choices for handling associations. DSM can implement associations with foreign keys, sets, arrays, tables, and a built-in generic association type. Our choice of technique was determined by the type of access required. We frequently used foreign keys and arrays. For example, Figure 18.11 shows a common way of implementing a one-to-many association.

A *Polygon* object may access points through the array of points stored in the *points* attribute. A *Point* object may access a polygon through the *parent* attribute. Since points may belong to exactly one of several different types of objects, the data type of parent is "object." "Object" indicates that the value is an object but does not specify the class.

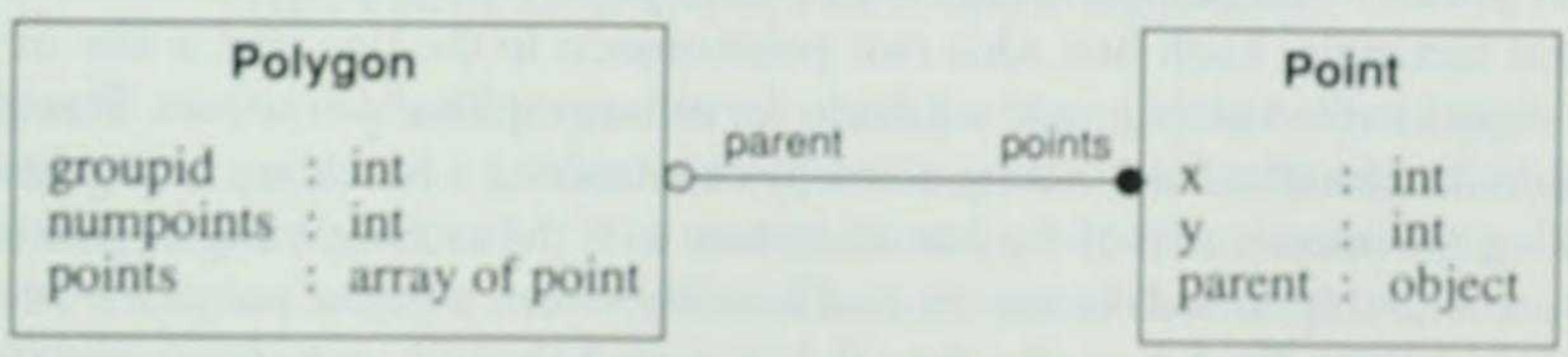


Figure 18.11 Implementation of a one-to-many association

The internal meta model captures the intended meaning of object diagrams. Initially, this model was similar to Figure 18.10. However, much of the internal meta model was not needed for implementation, so we simplified it during design. In the end, the internal meta model was a simple mapping of blocks and nets into classes and relationships, as shown in Figure 18.12.

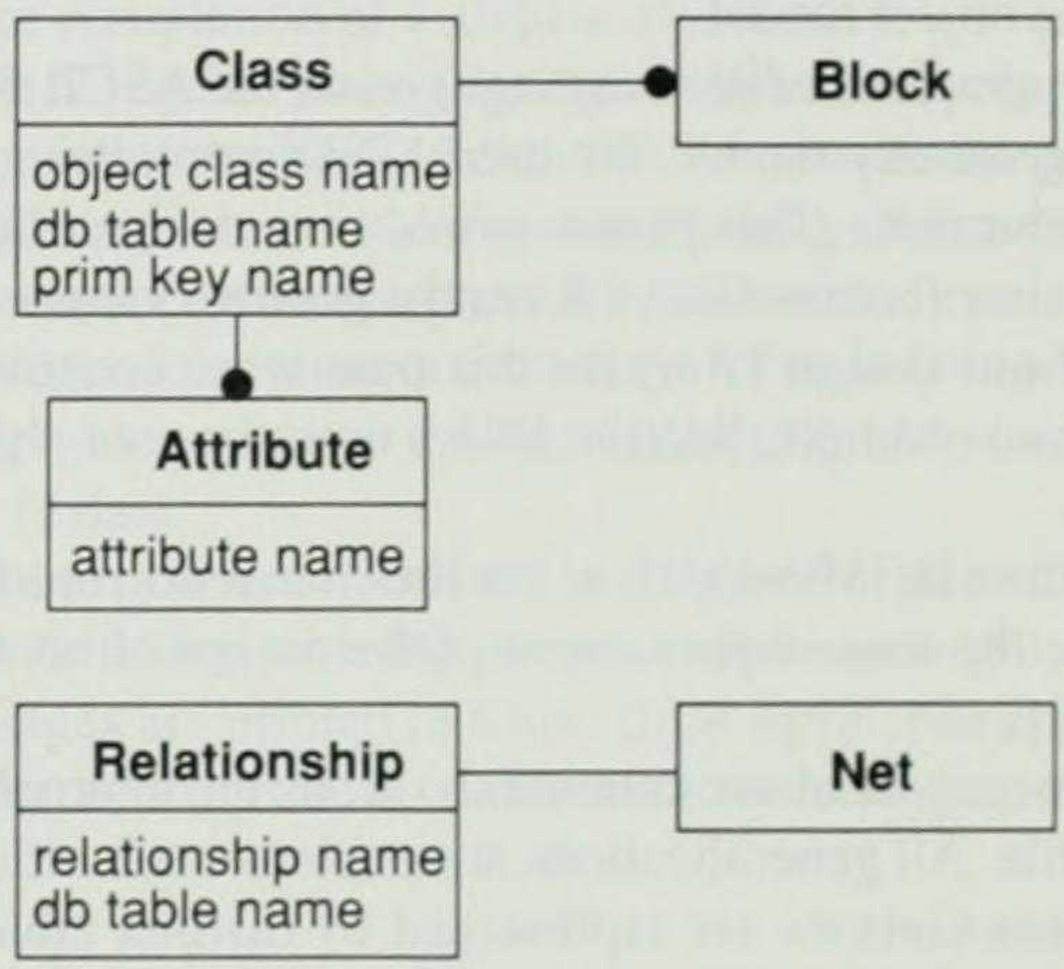


Figure 18.12 Internal meta object model

The structure of the database meta model is similar to Figure 18.10. We hand converted the database meta model into database tables to bootstrap the system. The *Object Diagram Compiler* was only intended to compile BOM diagrams, thus it cannot compile itself.

18.5.3 Connectivity Algorithm

A commonly occurring theme in the *Object Diagram Compiler* is determining connections. Lines connect to rectangles, ellipses, and each other. Text is contained in rectangles and is adjacent to lines. Rectangles are adjacent to each other. During design we observed that each of these separate problems could be reduced to a special case of a more general problem: Find the intersection of a bounding box with a bag of points.

For each page and element type, we construct bags of points as follows. (A bag is the same as a mathematical set, except it may have duplicates.) For each rectangle, we add the four corner points to the rectangle bag; each corner point is an object that contains a pointer to its parent rectangle. Each line adds two point objects to the line bag; a line of text adds one point object to the text bag; and similarly for other graphical primitives. Then for example, we find all rectangles intersecting a line by constructing a bounding box around the line and checking the intersection of the bounding box with the rectangle bag of points.

We used a doubly sorted dictionary to efficiently search a bag of points for intersections with a bounding box. Each bag of points is first sorted on the x-coordinate and then on the y-coordinate. Figure 18.13 shows an object model of our point access structure. Three important operations are: Insert a point, delete a point, and find points.

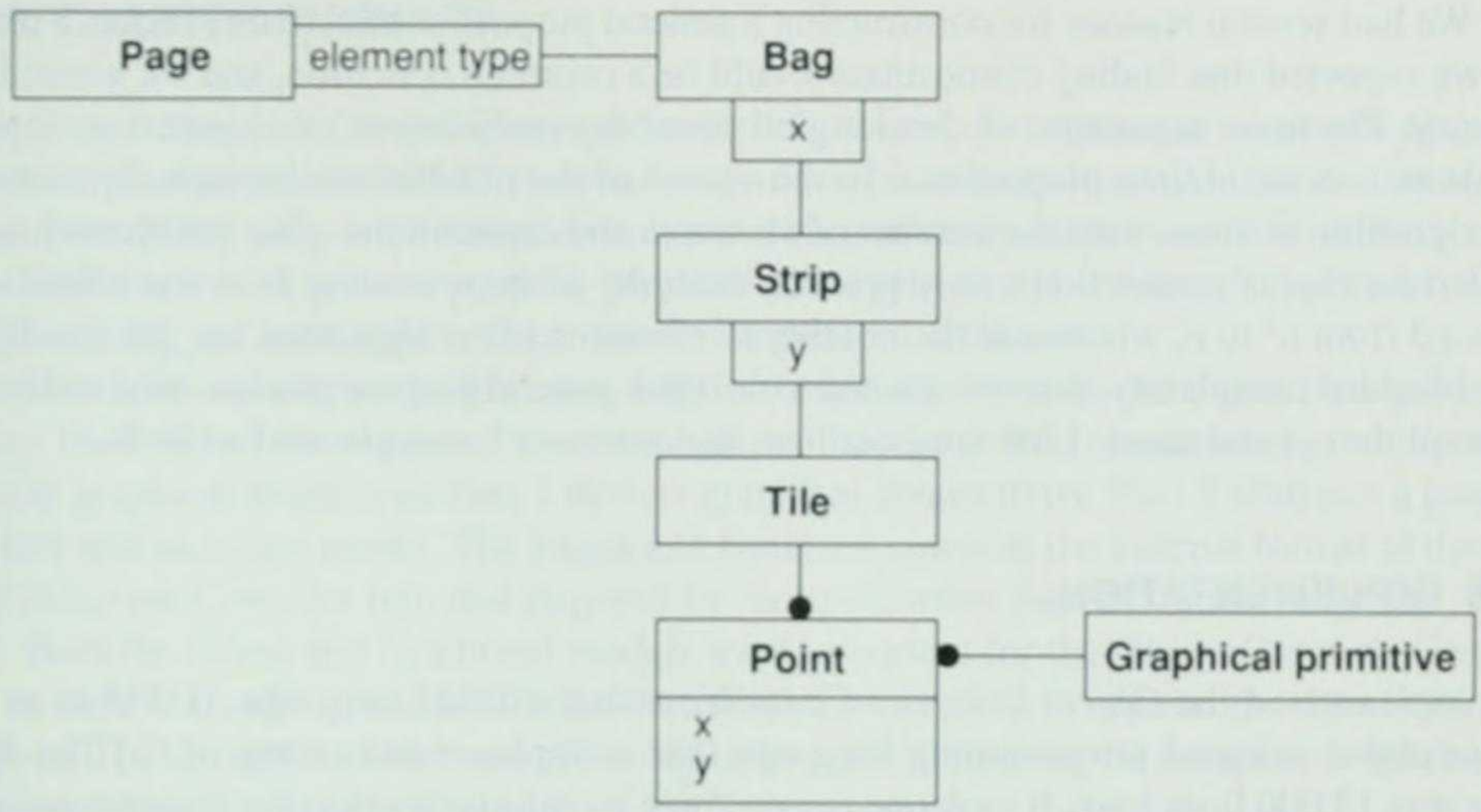


Figure 18.13 Object model of point access structure

To insert a point, first find the strip to which the point belongs. Try for an exact match on the x-coordinate in the qualified association between *Bag* and *Strip*. If none is found, create a new strip and add it to the association. Next, find the tile to which the point belongs by trying for an exact match on the y-coordinate in the qualified association between *Strip* and *Tile*. If none is found, create a new tile. Finally, associate the point with the tile.

Tiles are not of uniform size. Figure 18.14 has twelve points, seven strips, and seventeen tiles. (Each tile has a point at the lower left hand corner; when points coincide, one tile corresponds to multiple points.) A tile is merely a useful abstraction for organizing points in a search space for efficient interval searches.

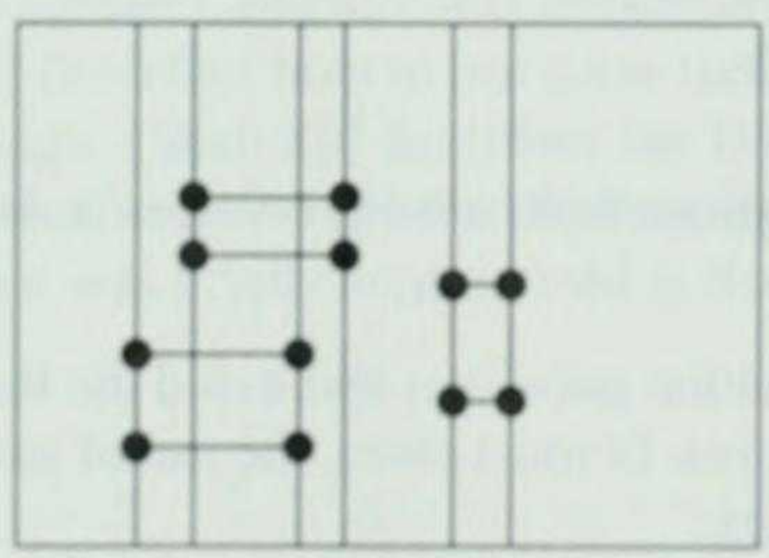


Figure 18.14 Strips and tiles for rectangle bag of points

Deletion is similar, except tiles and strips are deleted as they become empty.

Searching starts with the x-values of the bounding box. We implemented a special operation on the qualified association that returns all strips between the minimum and maximum values. Then, we queried the y-qualified association to determine all tiles in the strip between the y-values of the bounding box. Finally, all points associated with the tiles were added to the collection of points found. We convert the collection of points into a set of graphical primitives by traversing the association between point and graphical primitive.

We had several reasons for constructing a general purpose connectivity routine. First of all, we expected that finding connections would be a common operation, and we wanted efficiency. The naive algorithm of checking all possible combinations of elements on a page yields an execution time proportional to the square of the number of elements. In contrast, our algorithm is linear with the number of elements and connections. (Our problems have a limited number of connections, so in practice the order of the execution time was effectively reduced from n^2 to n, where n is the number of elements.) Our algorithm has the disadvantage of added complexity; thus we wanted to design a general purpose routine once and reuse it for all the special cases: Line intersect line, line intersect rectangle, and so forth.

18.6 IMPLEMENTATION

We implemented the *Object Diagram Compiler* with the DSM language. (DSM is an in-house object-oriented programming language that is implemented on top of C.) The final code was 13,000 lines long. It took one person three months to develop the diagram compiler. Six weeks were consumed by analysis and design and six weeks by coding and debugging. About twenty bugs were found, each of which took only a few minutes to find and correct.

The compiler is presently being used for several BOM generation applications. One application has a current input of more than 100 classes and 150 associations yielding a graphics editor file with over 13,000 lines. The compiler runs fast, compiling 5,000 lines per minute. This was a pleasant surprise because we initially feared that using an object-oriented language might degrade performance. The *Object Diagram Compiler* fully satisfied its requirements. Our success with the compiler has motivated work on more comprehensive software to support the OMT methodology.

18.7 LESSONS LEARNED

We draw the following conclusions from our experiences in developing the *Object Diagram Compiler*.

- Stick to the object-association paradigm and avoid the temptation to use stacks, pointers, and other data structures. In most cases, the use of associations eliminates the need for complex data structures.
- Use of a limited number of paradigms and a rich library of container classes greatly cut the number of bugs and increased our software development productivity.
- When making enhancements, don't just patch the design, rethink the analysis. Wherever we violated this rule, we wasted time and made errors. Think through the consequences of your "enhancement" deeply and not superficially.
- Batch software is much easier to write than interactive software; batch software does not need to contend with user interface and dialogue considerations.

18.8 CHAPTER SUMMARY

The *Object Diagram Compiler* accepts as input a textual description of an object diagram. The compiler produces DBMS commands that can be executed to load data dictionary tables. We call this program a compiler because it must not only transform its input, but it must also detect errors. A general purpose graphics editor provides no inherent support for object diagramming, so the compiler must detect many error conditions.

We chose a multipass architecture with intermediate object models; each pass gradually raises the level of abstraction. Pass 1 recognizes that the ASCII characters in the input file denote graphical primitives. Pass 2 detects graphical connectivity. Pass 3 abstracts a graphic picture into an object model. The fourth and final pass converts the internal format of the *Object Diagram Compiler* into that required by the application data dictionary tables.

Both the object and functional models were important for the *Object Diagram Compiler*, as there is much data to handle and substantial processing to convert the data from pass to pass. The dynamic model was relatively unimportant because the compiler is batch software and has no user interaction.

The design and implementation of the *Object Diagram Compiler* proceeded rapidly and were simply a matter of detail. The most significant tasks were to implement associations, prepare pseudocode, write the actual code, and then debug. Probably, our most important design decision was choosing the DSM object-oriented programming language. DSM has a rich class library that simplifies programming. We also decided to use UNIX compiler generation tools to parse the input. The *Object Diagram Compiler* completely satisfied its requirements. The compiler had excellent performance, despite the lax requirement in this area.

BIBLIOGRAPHIC NOTES

The first pass in our *Object Diagram Compiler* was implemented with UNIX compiler construction tools. [Schreiner-85] describes how to use these tools. [Aho-79] describes the theoretical basis of compiler design. [Shah-89] describes the DSM language that was used to implement the *Object Diagram Compiler*. [Blaha-90] describes the larger BOM configuration generation application that was briefly summarized in Section 18.1.

REFERENCES

[Aho-79] A.V. Aho, J.D. Ullman. *Principles of Compiler Design, Third Edition*. Reading, Mass.: Addison-Wesley, 1979.

[Blaha-90] M.R. Blaha, W.J. Premerlani, A.R. Bender, R.M. Salemme, M.M. Kornfein, C.K. Harkins. Bill-of-Material Configuration Generation. *Sixth International Conference on Data Engineering*, February 5-9, 1990, Los Angeles, CA.

[Schreiner-85] Axel T. Schreiner, H. George Friedman. *Introduction to Compiler Construction with UNIX*. Englewood Cliffs, New Jersey: Prentice Hall, 1985.

[Shah-89] Ashwin Shah, James Rumbaugh, Jung Hamel, Renee Borsari. DSM: an object-relationship modeling language. *OOPSLA'89* as *ACM SIGPLAN 24*, 11 (Nov. 1989), 191-210.

EXERCISES

18.1 (3) An object model compiler that supports a subset of the OMT notation uses the meta model in Figure E18.1. The compiler reads object models and generates instances of classes and associations (objects and links). Which of the following constructs does the meta model support: object classes, methods, object attributes, generalization, aggregation, binary association, ternary association, link attributes, qualified associations, role names, association names? If a construct requires something that is missing from the meta model, the construct is not supported.

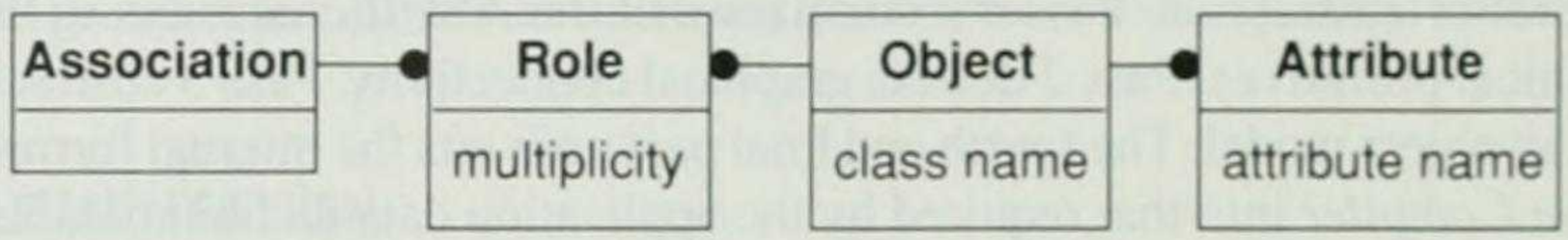


Figure E18.1 Meta model for an object model compiler

18.2 (5) Figure E18.2 is an input to the compiler described in exercise 18.1. The output of the compiler can be thought of as an instance diagram. A partially completed instance diagram corresponding to Figure E18.2 is shown in Figure E18.3. The links are correct, but some names and values are missing. Fill them in.

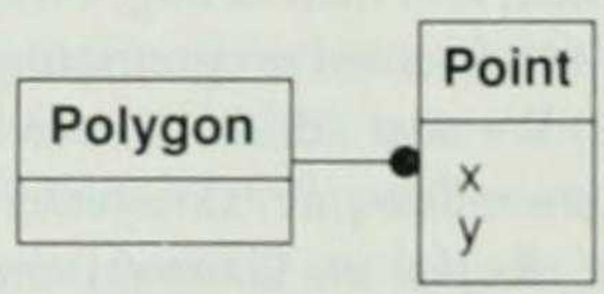


Figure E18.2 Compiler input object model

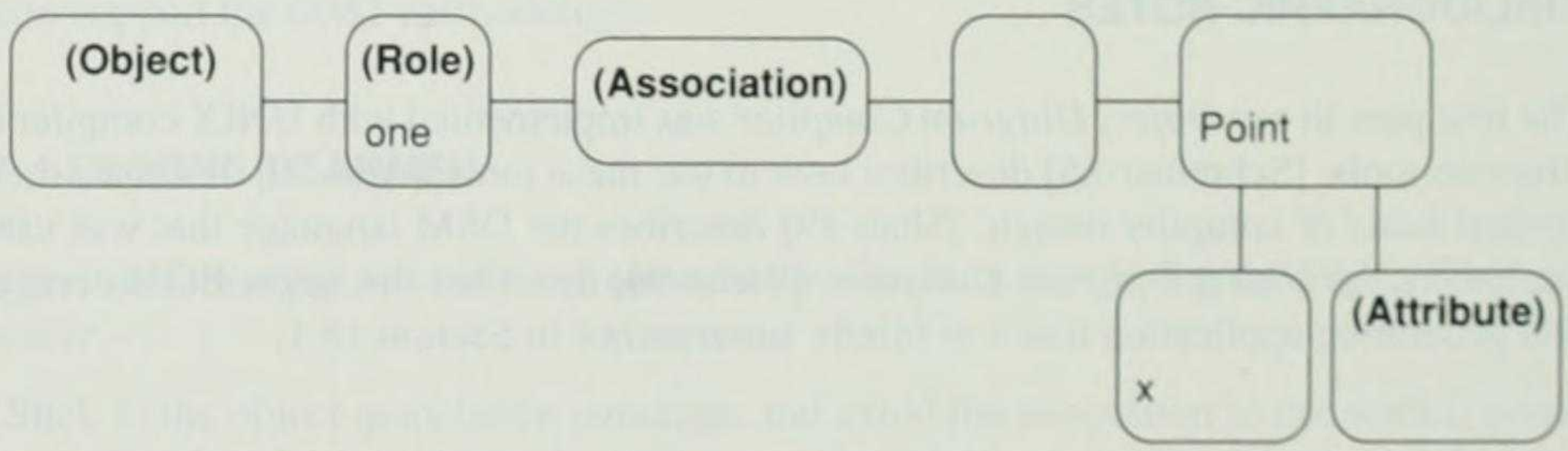


Figure E18.3 Partially completed compiler output instance diagram

18.3 (7) The meta model itself is fair game as input to the compiler. Figure E18.4 is the corresponding partially completed output instance diagram. The links are correct. Finish filling in names and values.

18.4 (9) Extend the meta model from Figure E18.1 to include generalization and aggregation relationships as well as relationship names. Make sure the meta model itself is valid input for the compiler.

18.5 (9) Prepare an instance diagram for the output of the compiler using the object diagram you prepared for the previous exercise as input.

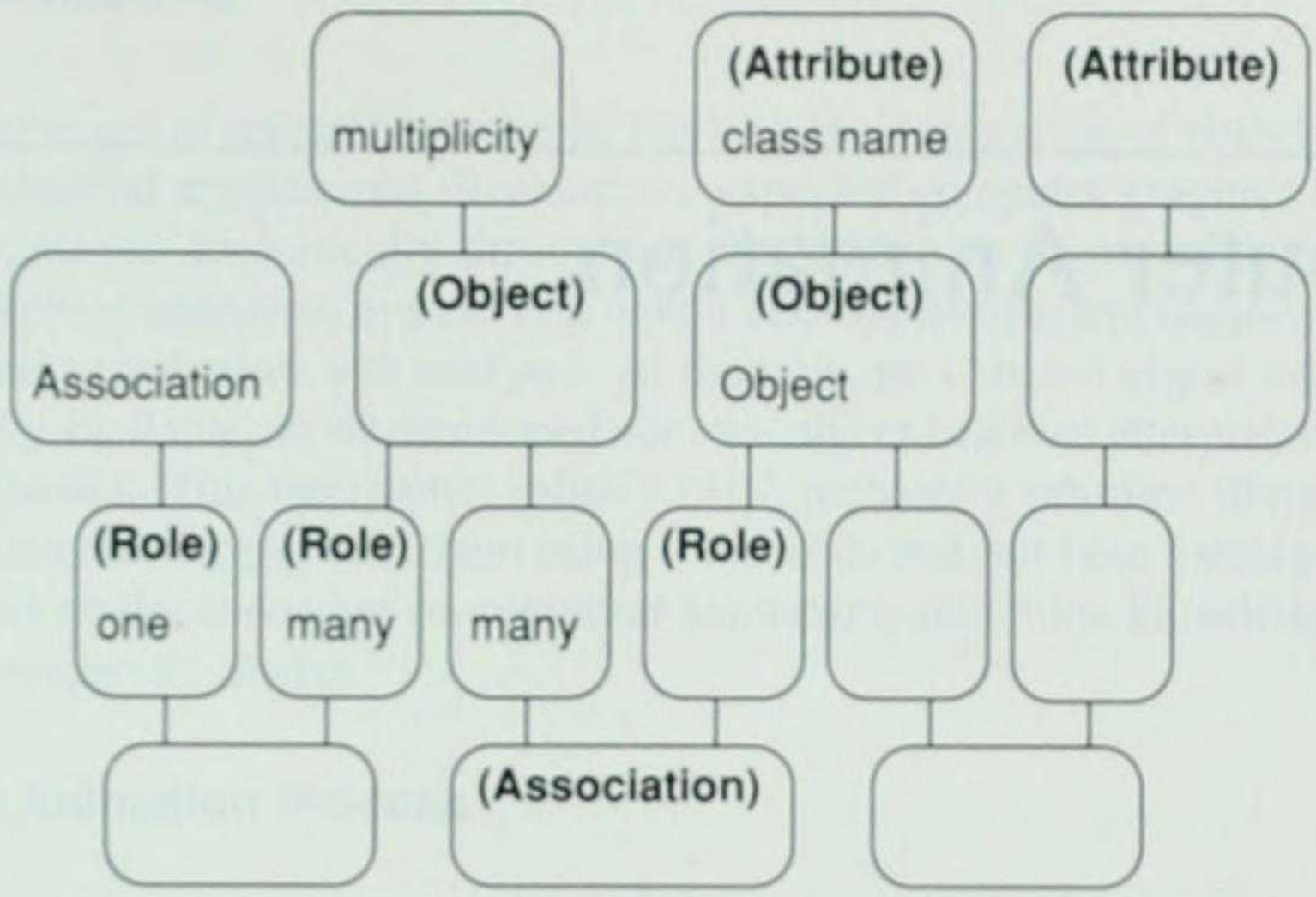


Figure E18.4 Partially completed compiler output instance diagram

19

Computer Animation

Computer animation systems create sequences of images, which when shown one after the other, create the illusion of motion. Computer animation systems require an extensible design so that new applications and new graphics hardware can be easily added.

This chapter describes a three-dimensional computer animation system. The system, *OSCAR*, the *O*bject-oriented *SC*ene *A*nimato*R*, produces high quality film and video sequences of the results of scientific and engineering calculations and experiments. *OSCAR* automates the creation, control, and management of 3-D computer-generated animation sequences. Using an object-oriented script language as its user interface, *OSCAR* controls analysis, modeling, rendering, display, and filming. Interfaces have been developed for scientific analysis programs in the areas of molecular modeling, robotics, mechanisms analysis, and fluid mechanics. The object-oriented design has produced a system that lends itself to interfacing with existing and future applications.

The *OSCAR* application demonstrates how an object-oriented design can be successfully implemented with a non-object-oriented language. *OSCAR* is written in C using an object-oriented development environment that includes macros and run-time support for inheritance, instancing, and message passing. *OSCAR* includes an interpreter that permits users to create instances from the animation classes in the system and to control these instances by sending messages. Computer animation systems have a dynamic simulation architecture as discussed in Chapter 9.

19.1 BACKGROUND

OSCAR is the result of research, started in 1984, on the application of object-oriented technology to industrial applications. Because we expected computer graphics systems to increase performance dramatically throughout the 1980s, we proposed developing a 3-D computer graphics animation system that would allow scientists and engineers to study the results of their experiments and analyses. At the time, no C-based object-oriented systems were generally available, so we developed our own object-oriented interpreter to support the animation research. This interpreter, called *LYMB*, includes a run-time library for creating objects and communicating with them using commands and run-time message passing. The interpreter has no dependencies on computer animation; animation knowledge is held with the animation-specific classes.

19.1.1 The Animation Process

OSCAR simulates the steps in the traditional, manual animation process. As in all disciplines, animation has its own terminology. We briefly describe the animation process, defining enough jargon for the reader to understand the *OSCAR* system.

The user of a computer animation system acts as the producer, writer, and director of the final animation sequence. This sequence may be played back on a computer graphics terminal or recorded onto movie film or video tape. The producer manages the overall film production, keeping schedules, assigning tasks, and organizing resources. The writer creates a script based on the requirements of the customer. The director controls the animation, positioning the props, actors, cameras, and lights.

The animation process proceeds as follows:

1. *Story.* The writer, working with a customer, develops a narrative story that describes the actors and their roles, including their appearance, dialog, and actions. Writing is an artistic process and is difficult to assist with the computer. The actors may include inanimate objects.

2. *Story board.* A story board is a graphic synopsis of the animation, illustrating the appearance of the animation and the flow of the story. The story board contains a representative drawing for each major point in the story. The art director normally produces the story board.

3. *Script.* The script contains the detailed positioning and movement of the actors, cameras, and lights in the animation. Some computer animation systems provide scripting languages. Since the script may change over the course of production, a good language allows the script to be easily modified when minor animation changes are required.

4. *Simulations.* Experiments and simulations often provide the motions of the animation for scientific applications. Scientists and engineers familiar with the application domain set up and execute these simulations. Simulations can be computation-intensive.

5. *Models*. Each prop and actor in an animation must have a geometric model that describes its appearance. This model is translated, during the rendering process, from a mathematical description into a portion of a 2-D image.

6. *Preview*. Before committing the animation sequence to film, a fast preview of the animation helps establish critical timing and the overall look of the final product. Producing high quality film output is usually costly, so changes to the simulation should be made as early as possible.

7. *Render*. Rendering produces shaded images for later display by applying algorithms to the geometric model, surface properties, and lighting. The render operation corresponds to assembling a single frame of a conventional animation. Rendering realistic effects such as multiple light sources, texture, shadows, translucency, and refraction can be computation-intensive.

8. *Recording*. After rendering, the images are recorded on either film or video, one frame at a time.

A typical animation application is three-dimensional molecular modeling. First a batch simulation is run on a supercomputer to compute the position of each atom in a vibrating molecule over a period of time. The actors are the atoms in the molecules and their chemical bonds. The animator prepares a script viewing the molecule from interesting angles. Rapid previewing is essential because it is hard to choose the right viewing angles without seeing the results. The final animation is run at full resolution, and a film is produced. Actually exposing a film of several minutes' duration can take a full day of computer time.

19.2 PROBLEM STATEMENT

The goal of the graphics animation system is to provide an automated system for the efficient creation, control, and management of 3-D computer generated animation sequences. The system provides facilities to record animation sequences on 16mm film or video. The system also provides interfaces between diverse simulation, modeling, and rendering systems. Realizing that the application of computer animation is broad, the system must adapt to new applications with a minimal amount of new software.

The following requirements were derived from knowledge of the conventional movie making process and the scientific and engineering analysis process. The movie animation metaphor used is that of stop-frame animation. In a stop-frame animation system, frames are recorded one at a time. All participants in the animation are changed slightly and another frame is exposed.

The user input consists of text files that contain animation scripts. Scripts can reference a library of complex actions, as well as define new actions. The scripting language serves as a base for animation description and documentation of an animation.

Each animation consists of one or more scenes. Within a given scene, the script can control any participant in the animation including actors, cameras, and lights. Control is provided through lists of actions that the director groups into cues. Cues are active for a user-specified duration and specify actions applied to objects in the animation. Any valid statement in the animation language can be an action. For example, an actor might be told to move at a constant velocity during the scene. Each scene specifies the renderer that produces its frame images.

Actors are the active participants in the animation. They have shape, position, orientation, and appearance properties, such as color and texture. For example, in a planetary exploration animation, actors would include planets, rings, satellites, rockets, stars, and the sun. The specification of actor motions must be flexible enough to allow rotations and scaling about arbitrary axes and origins. The visibility of individual actors must also be controllable. Often, it is useful to group actors together, so the system must provide a mechanism to allow actors to be collected and controlled enmasse.

The geometry of an actor is kept in a geometric model. Models range from rigid motions of simple geometric shapes to complicated deformations of complex objects. Since modeling is a complicated and tedious task, the system must be able to use models that already exist. Also, users may have invested large amounts of training in the use of specific modelers. Modelers should quickly produce low-resolution models during previewing so that the computation time to produce a high-resolution model is only be required for the detailed film.

All animation is viewed through a camera. Multiple cameras can exist, but only one is active in a scene at one time. The user can place the camera in the environment and control its aim and orientation. Different camera movement effects like panning, zooming, and tracking are required.

Multiple lights can be used to illuminate the animation. Lights can be turned on and off and their intensity can be controlled. Lights also have geometry—spherical, cylindrical, point, or flood.

Engineers and scientists who use the system will rely on their own analysis programs to provide much of the physics of the animation. Therefore, the animation system must provide a way to interface to these external programs. These programs should not be modified just to interface with the animation system, so the interface burden must be borne by the animation system.

The system must provide a facility to track completion of frames and where they reside. A frame is one image in an animation sequence. A frame can be built from one rendered frame or a combination of rendered frames. A post-production capability allows the user to recall frames, combine them with other frames, and manipulate them on a recoding device.

It is too time consuming to specify the incremental motion of each actor in an animation for each frame, so the system must provide a way to move objects along easily specified paths. Keyframing is an animation technique that lets an animator specify complex motion by giving a few key frames and letting a junior animator fill in the intervening frames. A computerized animation system should have a keyframing capability in which the system interpolates the animation.

19.3 ANALYSIS

19.3.1 Object Model

Now, following the analysis steps in Chapter 8, we extract objects from the requirements and our knowledge of computer animation and computer graphics. Notice that the terminology from the problem statement is used in the object model. Many of the properties we specify for the objects come from our intuitive feeling for what an object should look like and how it should behave.

Identifying Object Classes

We start by listing candidate object classes:

- actor
- camera
- light
- scene
- cue
- renderer
- sequence
- frame

We follow four classes through analysis: actors, cues, renderers, and scenes.

- *Actors* are the participants in our animation. Our actors can represent a variety of objects we wish to model, so we have try to determine what properties are important to all actors, regardless of the application. In a finite element analysis, an actor could be an element mesh of a turbine blade. For molecular modeling, both atoms and bonds are actors.
- *Cues* control the actors, cameras, and lights in the animation. Cues are temporal objects, changing the state of the animation with time. Each cue is a series of events. For example, a cue might tell a ball to start rolling across the room or pan a camera across the stage. Any time dependent action can be affected by a cue.
- *Renderers* produce the individual frames in the animation. Rendering is still an active research area in computer graphics, so we must take care not to restrict ourselves to a particular renderer, such as scan line, ray trace, radiosity, or wire frame. The challenge is to characterize the properties of renderers that we expect all renderers to have. *Renderer* is an abstract class meant to be overridden by subclasses.
- *Scenes* control the animation process, bringing together cues and renderers. They usually describe a coherent set of actions within the animation.

Identifying Associations and Attributes

Here we define associations and attributes in the model:

- Each *actor* has a model that describes its geometry. Modeling is an active research area, and there is no one modeling technique that can be used in all situations. Also, the requirements state that existing modelers have to be supported, so *Model* is an open-ended abstract class.

  Actors also have position, orientation, color, and visibility. An actor's orientation is with respect to some origin. We group positional properties, such as translate, rotate, and scale, under a general 3-D transform. Actors also have light transmission properties which we break out as a separate object as new variations may be invented.

- *Cues* are time sensitive. Each cue has a time interval during which it is active. Usually, the user wants to do something special when an interval is entered and when it is exited. We call actions that should be executed at the start of the interval *start actions* and those that should be executed when the interval expires *end actions*. While a cue is active, it executes *tick actions* at the time resolution of its enclosing scene.

- *Renderers* need to know what actors, cameras, and lights are in a given scene so that they can create images. A user may not want to render every frame of an animation and should be able to specify a rendering resolution. The user also must be able to specify a spatial resolution that tells the renderer how big an image to create and where that image lies in the camera's view space. Graphics people call this spatial resolution the *viewport* of a rendering. The background color of the rendered image is also a property of renderers.

- A *scene* must know what cues and renderers are required to create the animation sequence. Like a cue, a scene also has starting and ending actions that perform initialization and termination.

Identifying Operations

Most classes in the *OSCAR* animation system have interesting operations.

- *Actors.* We certainly will want to *translate*, *rotate*, and turn an actor *on* and *off*.
- *Cues.* We invoke a cue with a *tick* operation that includes the current time within the scene. The cue performs all its tick actions.
- *Renderers.* A *render* operation causes a renderer to create one frame of animation.
- *Scenes.* A *start* operation begins a scene's execution. The scene in turn steps through its time interval, invoking cues and rendering frames.

Identifying Inheritance

Scenes and cues both have time intervals during which they are active, as well as start, tick, and end actions. We group *Scene* and *Cue* under the abstract superclass *Sequencer*. Scenes also have a clock and a time resolution; cues do not need these attributes as they are driven by scenes.

Lights, cameras, and actors all have position, orientation, and graphical transformation operations which we group under the abstract superclass *Transformable object*.

Figure 19.1 shows the final object model for the *OSCAR* system.

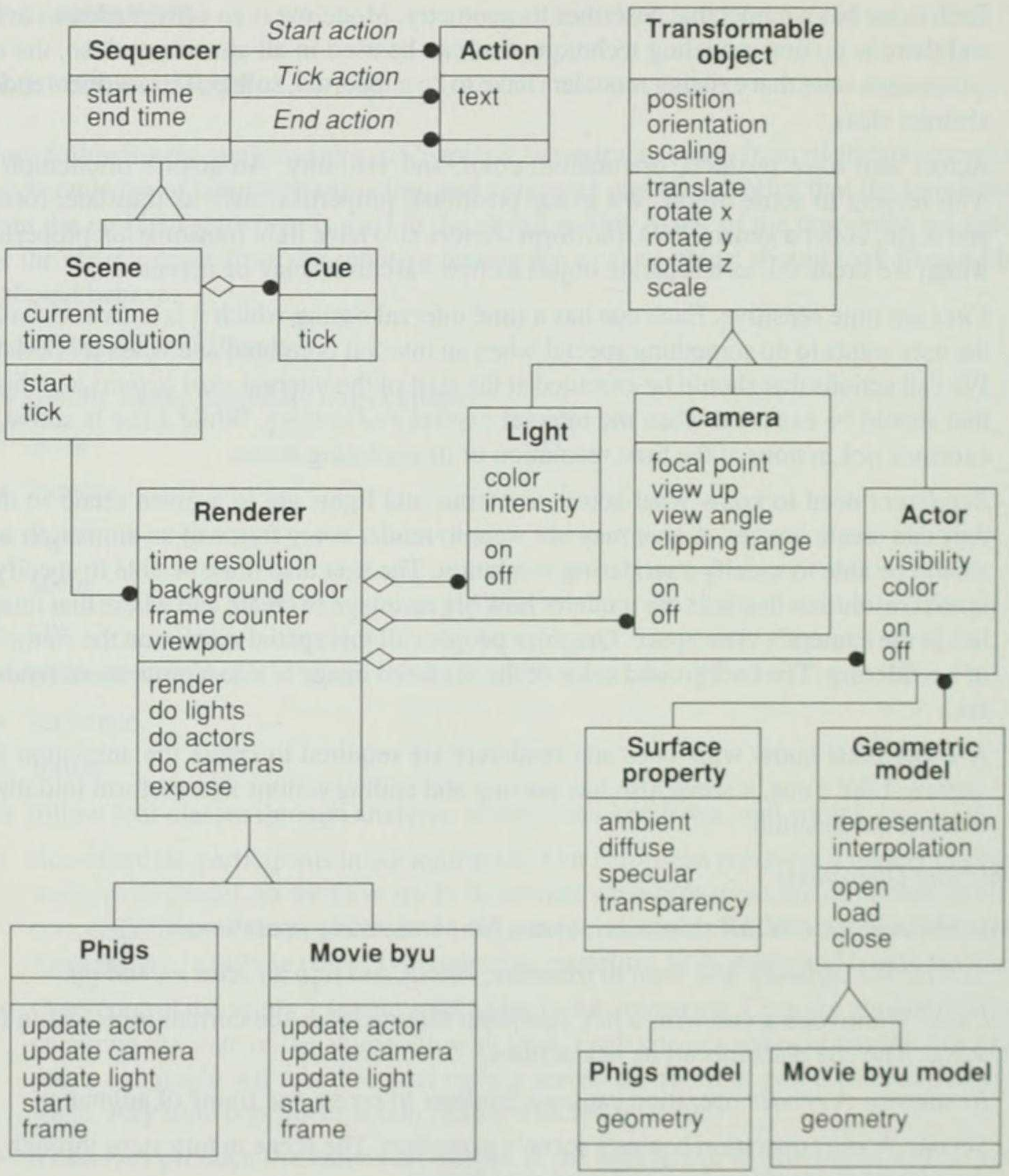


Figure 19.1 Object model for *OSCAR* system

19.3.2 Dynamic Model

The dynamics of an animation are specified by a scene, which encapsulates a series of user-defined events. Figure 19.2 and Figure 19.3 show the dynamic model for *Scene* and *Cue*. A scene repeatedly enters the *tick* state, each time incrementing the current time and sending *tick* events to its cues and *render* events to its renderers. A cue is similar, but it receives tick events from the scene.

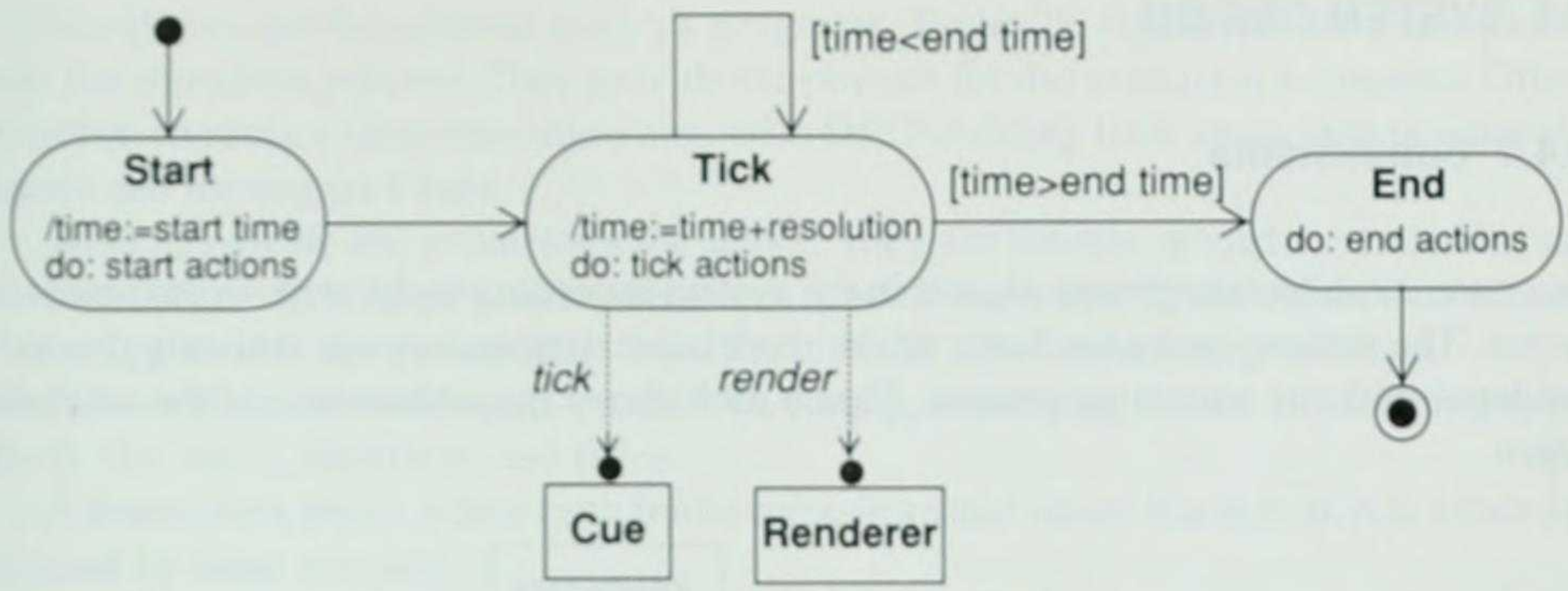


Figure 19.2 State model for *Scene*

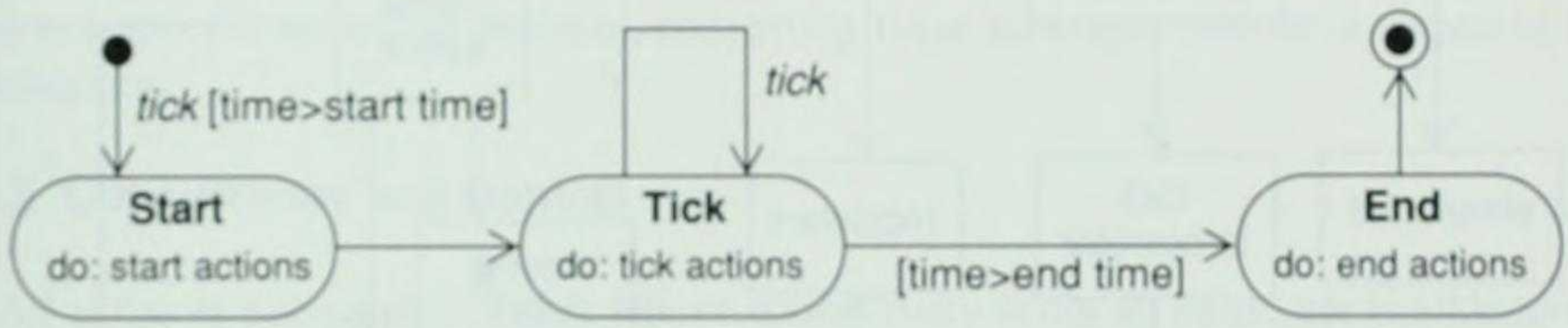


Figure 19.3 State model for *Cue*

19.3.3 Functional Model

The most computation-intensive functional model is that for the renderer. A functional model for a renderer is shown in Figure 19.4.

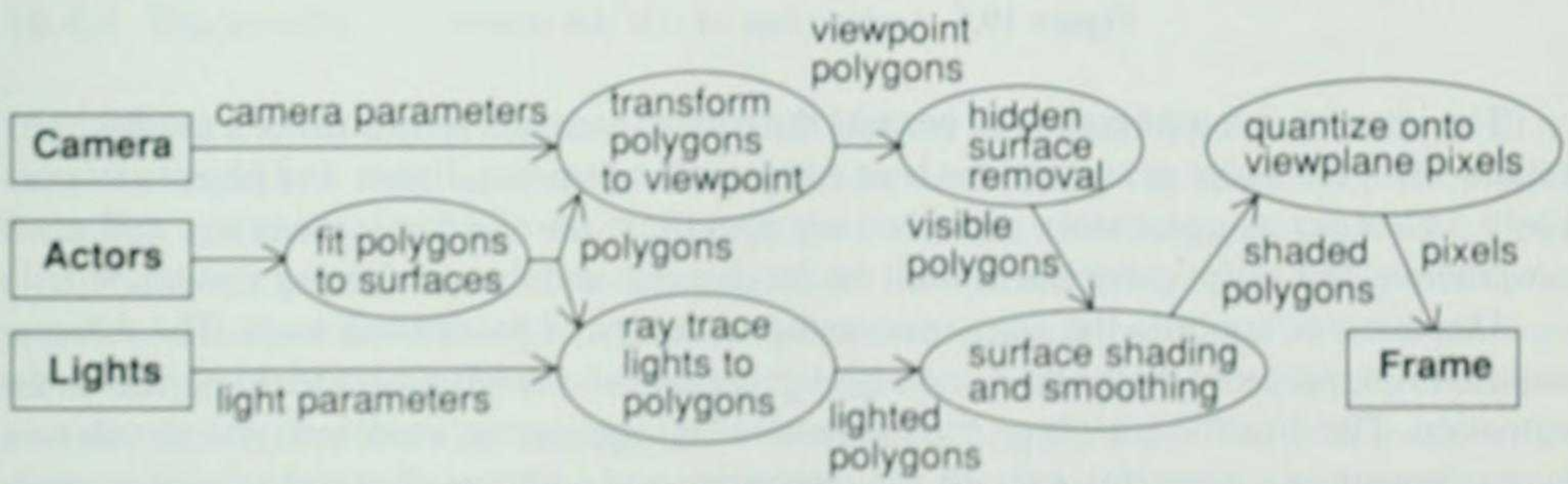


Figure 19.4 Functional model of a renderer

19.4 SYSTEM DESIGN

19.4.1 Subsystems

The system architecture groups objects in the system according to the steps in the animation process. The anthropomorphic flavor of the model descriptions helps to maintain the correspondence with the animation process. Figure 19.5 shows the architecture of the animation system.

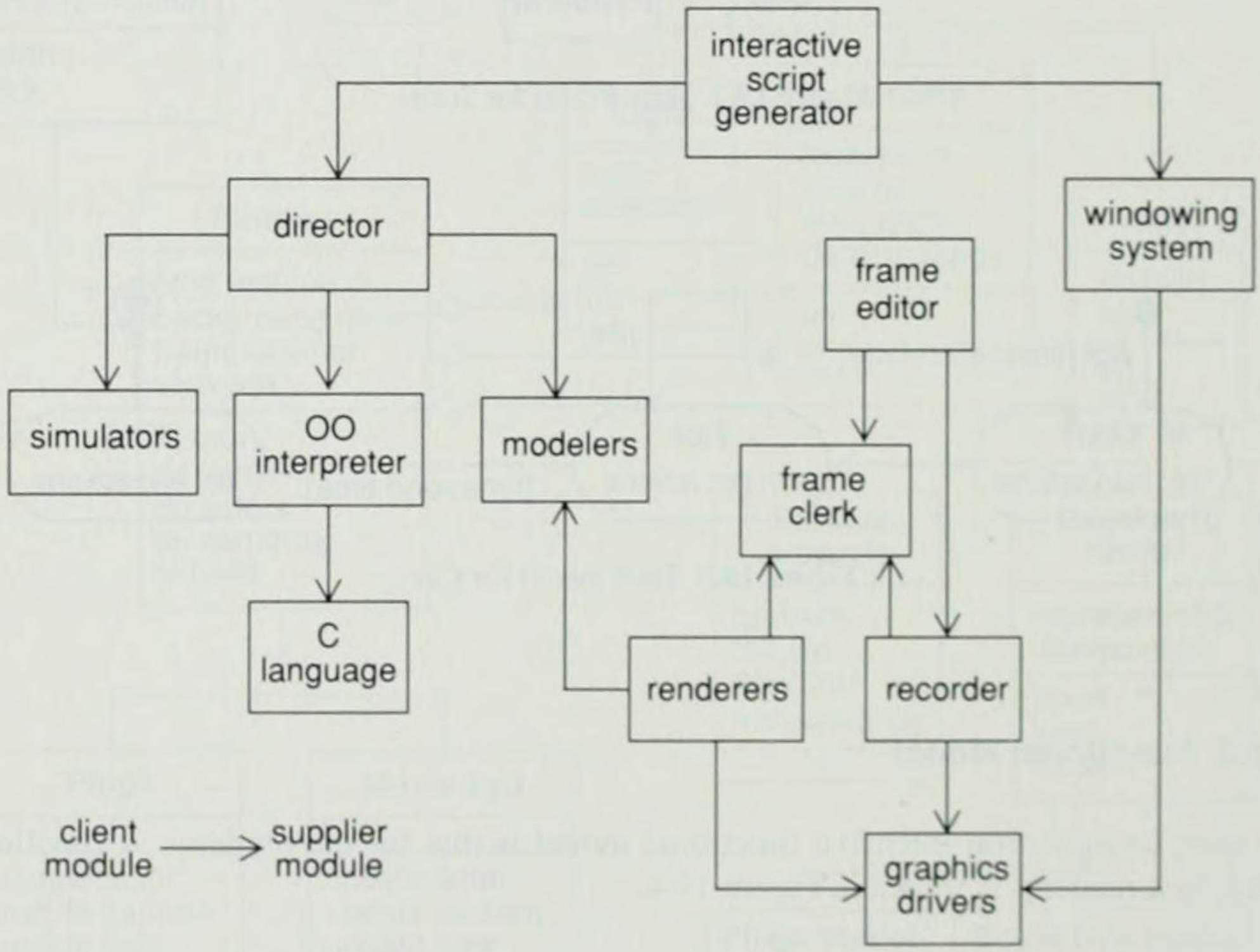


Figure 19.5 Architecture of *OSCAR* system

The interactive script generator permits the user to generate scripts using a graphical interface. With the script generator, the user can position cameras, lights, and objects interactively. Other script generators provide easy specification of complex camera and actor movements. The script generator is built on the director and the windowing system.

The director controls the animation using a variety of predefined tools. The director reads a script, prepared by the user or a script generator, and sends messages to objects in the animation. The director is built on the interactive OO interpreter, modelers, and simulators. By implementing a powerful, easy-to-use interpreter and a library of useful animation operations, we avoid the need to provide a special animation language.

Simulators are the external analysis programs. Typically, simulators are run separately from the animation process. They provide the physics for the animation sequences. Often, a simulator controls a complete animation, with *OSCAR* doing little more than positioning a camera and turning on a light.

Modelers create the geometries for actors. They are loosely related to renderers, since some renderers can only process models from specific modelers. Often the models are made from polygons, but the system must be able to handle other modeling primitives.

Frame editor objects do post-production editing, combining frames and creating special effects like wipes, dissolves, and fades.

A frame clerk tracks where each frame came from and where it is stored. It is a data store accessed by other modules.

Renderers take actor, camera, and light information to produce a single frame of animation. External renderers run in batch mode, producing their images from command files created by *OSCAR*. Other renderers are built within the animation system itself, but all renderers respond to the same protocol.

Recorder objects take finished frames and commit them to film or video. They also respond to a specific recording protocol, converting these messages into device-specific control functions.

19.4.2 Concurrency and Control

OSCAR is run as a sequential, batch job, so concurrency is not an issue, nor is allocation of tasks to processors. Control is implemented procedurally by explicitly sending messages to objects within iterative loops simulating scenes.

19.4.3 Data Formats

Scripts are written as sequences of operations in the OO interpreter language, rather than in any special format. The format of the frame clerk must be given, but we omit it here. The format of the simulators, modelers, and renderers is specified by their suppliers and is beyond our control, so we must accept them as they are.

19.4.4 Trade-offs

The director and the interpretive language must be easy to use and flexible, but they will not consume much run time, as most of the execution time is spent in predefined simulators, modelers, and renderers. Thus we chose to make the interpretive language highly symbolic. Operations are encoded as text strings which are interpreted at run time using an explicit search of the class hierarchy for methods. Operands are passed using variable-length argument lists and text keywords for flexibility. Set-up operations use several levels of indirection for flexibility. We also incorporated several debugging aids. The efficiency of the director is less than 50% but does not matter because the director consumes a small portion of the overall *OSCAR* execution time.

19.5 OBJECT DESIGN

During object design, we begin to shift our focus from the external requirements to the computer implementation. We extend the object models we derived during analysis to include associations and attributes that objects require for internal processing. Most of the design, however, is a simple and direct implementation of the analysis model.

19.5.1 Actors

To retain the state of an actor's position and orientation, we introduce a new class: the *Matrix*. *Camera* and *Light* also implement much of their behavior with *Matrix*. Each instance of *Matrix* is a 4-by-4 transformation matrix that captures the geometry of a *Transformable object* and provides an efficient basis for implementing motion. Translate, rotate, and scale are simply special cases of matrix multiplications on the 4-by-4 matrix. Successive transformations can be combined by multiplying their respective matrices. Each *Transformable object* delegates its behavior to a *Matrix* object.

19.5.2 Cues

The only external operation on a cue is *tick*. The operation is a direct implementation of the dynamic model:

1. If clock has just entered the active interval for a cue, execute *start actions*.
2. If clock is within the active interval for a cue, execute *tick actions*.
3. If clock has just left the active interval for a cue, execute *end actions*.

This simple algorithm gives the animation system control over any object in the system. The power comes from the ability to use any statement in the language as an action. Later, in some examples, we show how the cue appears to the user.

19.5.3 Scenes

The only external operation on a scene is *start*, which steps through a scene as follows:

1. Execute *start actions*.
2. Set *current time* to *start time*.

As long as *current time* is less than *end time*:

3. Increment *current time* by *resolution*.
4. Execute scene *tick* actions.
5. Send *tick* operation to each cue in the scene.
6. Send *render* operation to each renderer in the scene.

Once the scene's *current time* exceeds its *end time*:

7. Execute scene *end actions*.

19.5.4 Renderers

In the *Renderer* class, the *render* operation is the only complicated external operation. *OSCAR* supports many renderers, but all rendering is controlled by a uniform protocol. The render protocol determines whether a frame should be rendered, based on the renderers *time resolution* property. Then it positions the cameras, lights, and actors; the frame is built; and an *expose* operation requested. Finally, the time of the render operation is noted and the frame counter incremented. An abstract class, called *Renderer*, executes the rendering protocol. Specific renderers are subclasses of this abstract class and must implement any operations not implemented by the abstract class (noted as subclass responsibility).

The render protocol is as follows:

Render operation

1. If the frame counter is not a multiple of the rendering time resolution, just increment the frame counter and return.
2. If this is the first frame to render, perform the *start* operation (subclass responsibility).
3. Perform *cameras* operation.
4. Perform *lights* operation.
5. Perform *actors* operation.
6. Perform *frame* operation (subclass responsibility).
7. Perform *expose* operation.
8. Update render time.
9. Increment frame counter.

Operations for Cameras, Lights, and Actors

These operations check their list of objects to see if any have been changed since the last render operation. For each changed object, they perform *update_camera*, *update_light*, or *update_actor* operations with the object name as an argument. Unmodified objects use the image from the previous frame. It is necessary to store the current image for each object as an optimization to avoid recomputation.

Next we describe the subclass responsibility operations for a typical renderer called Phigs+. Phigs+ is a graphics standard for three-dimensional displays that have a hierarchical display structure.

1. *Start:* If the system is not initialized, open it, create a workstation, and set its update mode to deferred.
2. *Update_camera* queries the named camera for its viewing parameters and converts the *OSCAR* camera parameters into Phigs+ parameters to form a view orientation matrix. It inserts this matrix into the workstations view table.
3. *Update_light* calculates the light direction vector by asking the named light for its focal point and position. It also gets the color of the light and stores the direction and color in the light table for the workstation.

4. *Update_actor* checks to see whether its model has changed since the named actor has changed. If the model has changed, it applies a load operation to the model. If this actor is not in a Phigs+ actor group, the actor is added to the group and a segment is opened. Then update_actor queries the status of the actor's position, color, orientation, and creates the appropriate structure elements. Finally, the segment is closed and posted to the workstation.

5. *Frame* sets the background for the workstation, and tells the workstation to render all segments.

These five operations are all that are required to add a new renderer to the system. Each operation performs one task that does not interact with the other tasks. The order of the operations is enforced by the abstract renderer class.

19.6 IMPLEMENTATION

OSCAR is implemented using an object-oriented interpreter based on the C language. The interpreter implements creation of classes and objects, single inheritance, and operations coded as text strings. Its syntax is similar to Smalltalk, with some extensions. The user interface consists of a parser that interprets operations of the form:

object_name message_sequence

A message sequence consists of messages with optional arguments. The suffix of the message determines whether a message requires arguments. The following suffixes illustrate message semantics:

? requests the value of an instance variable

= sets an instance variable

: has arguments but does not set an instance variable

@ defines an index

+, -, /, and * terminate arithmetic operations

Messages to one object can be concatenated on a statement. A typical statement is:

```
Actor new: aBox position=(0,5,0)
        rotate_x:30 color=(1,0,1) on!;
```

Actor is a class that responds to the *new:* message by creating an instance of itself with the name *aBox*. Classes, when they receive *new:* messages, return a pointer to the new instance that will receive the rest of the messages in the statement. This new actor receives the *position=* message, causing its position instance variable to be set to (0,5,0). The remaining messages rotate *aBox* 30 degrees about its x axis, set its color to magenta (red=1, green=0, blue=1), and turn it on.

The system's knowledge of the problem domain, here computer animation, is kept within the classes and not in the language.

19.6.1 Sample Animation

A simple animation illustrates the power of the animation system and the language. This short animation shows a bouncing ball that starts at rest and moves along a path that simulates gravity. Here, we do not use an accurate scientific simulation of gravity but rather use a sine curve to simulate the motion.

First we need an actor for the ball. We start it at (0,0,0), make it red, and give it a model called ball_model.

```
actor new: ball position=(0,0,0)
        color=(1,0,0) model= ball_model;
```

Now we need a modeler to describe the physical model of the ball. For this example, we use a Phigs+ modeling primitive for a sphere.

```
Phigs_sphere new: ball_model radius=.5;
```

Phigs_sphere is a predefined class that creates a Phigs+ compatible representation of a sphere.

An *OSCAR* camera is similar to a physical camera. We give it a position, a focal point, an up direction, angle for the lens (view angle), and turn it on.

```
camera new: camera_1 position=(10,0,80) focal_point=(10,0,0)
        view_up=(0,1,0) view_angle=30 on!;
```

We also need lights to illuminate the scene. We place a light at the camera with the following statement:

```
light new: light_1 position=[camera_1 position?]
        focal_point=[camera_1 focal_point?] intensity=.8 on!;
```

Here we used a message sequence with square brackets to take the results of one message, *position?*, to use as an argument for another, *position=*. This powerful construct is used throughout computer animation. Even though the light object does not know about cameras, the parser provides a simple mechanism to communicate state information from one object to another.

Now we create a renderer, an object that will take the actors, cameras, and lights and create one frame of animation.

```
Phigs new: aRenderer lights=light_1 cameras=camera_1
        actors=ball;
```

Next we define the cue that controls the motion of the ball. In this animation, the ball will bounce once in six seconds. Although there are many ways to do this, we use half a sine curve to simulate the effects of gravity. To make the action more interesting we add a bit of horizontal motion. The cue will start at time 0 and last for 6 seconds.

```
cue new: bounce
        start_time=0 end_time=6
        start_actions="scalar new: x; scalar new: y;"
        tick_actions=("x = 20 / 6 * [bounce time?];",
        "y = 30 * [bounce time?] sin! * 10;",
        "ball position= [x ?], [y ?], 0);");
```

Scalar is a scalar number class. The message "=" sets the scalar instance's *value*, while the message "/" divides the current value by its argument. Likewise, the *sin!* message changes the scalar's current value to the sine of its value. To get the value of the scalar, we send a "?" message. Here, using our simple message passing parser, we have added an arithmetic capability to the animation system. Remember, a cue sends messages to the parser whenever the cue receives a message and the time is within the cue's start and end times.

The last part of our script defines the scene.

```
scene new: scene_1
      start_actions= "! Scene 1 Starts"
      renderers=aRenderer cues=bounce
      end_actions="!Scene 1 Complete"
```

The scene starts when it receives a *start!* message and runs until its clock reaches its duration.

```
scene_1 start!;
```

An interesting variation of the script places the camera's focal point at the ball's position, a "follow the bouncing ball" effect. This will keep the ball in the center of the frame. To appreciate this, we should place a few balls of varying size around on the floor of the room, which we add as follows:

```
actor new: room model=room_model;
actor new: ball1 model=ball_model position=(5,2,0)
      scale=(.5,.5,.5);
actor new: ball2 model=ball_model position=(5,7,0)
      scale=(.2,.2,.2);
actor new: ball3 model=ball_model position=(3,5,0)
      scale=(1.7,1.7,1.7);
aRenderer actors+ (room, ball1, ball2, ball3);
```

The *follow* cue keeps the camera's focal point on the ball:

```
cue new: follow duration=[bounce duration]
      tick_actions="aCamera focal_point=[aBall position?];"
```

Then we add this cue to the scene:

```
scene_1 cues+ follow time=0 start!;
```

Animations often evolve this way: one cue at a time. We might get more sophisticated and simulate the ball squashing against the floor by changing its z scale factor. Another cue might pan the camera around the room while the ball bounces.

19.7 LESSONS LEARNED

OSCAR made its first film in December of 1984 when it consisted of twenty five classes including actors, cameras, lights, cues, and renderers. Since its initial implementation, *OSCAR* has grown to contain over ninety classes, fifty eight of which are subclasses of the original classes. Thus over half the classes share code with other classes.

During this project, we made several observations:

- Applying object modeling to animation produces a natural user interface with terminology that is familiar to experts in the application domain.
- The abstraction step of the design is critical and requires the most effort.
- The object-oriented approach partitions a complex system into manageable pieces.
- The system is less fragile than others we have written. We make changes and additions without fear of breaking *OSCAR*.

19.8 CHAPTER SUMMARY

OSCAR is a three-dimensional computer animation system which produces high quality film and video sequences of the results of scientific calculations and experiments. *OSCAR* is extensible and able to accommodate new applications and advances in graphics technology. *OSCAR* was designed using the object-oriented paradigm and implemented in the C language. The jargon used in constructing *OSCAR* mimics that used in the traditional, manual animation process: scene, cue, light, camera, and actor. Since the structure of *OSCAR* parallels the application domain, the resulting software is straightforward to use and understand. *OSCAR* adopts the "stop-frame" metaphor. Each frame is constructed and recorded one at a time; the resulting frames are played rapidly in succession to create the illusion of motion.

The object, dynamic, and functional models are all significant for *OSCAR*. However, most of *OSCAR*'s development effort dealt not with the static data structure but with designing and programming various operations. *OSCAR*'s architecture closely follows that of the object model; most classes in *OSCAR* are autonomous and correspond to entities in the real world. To a large extent, the object design phase just added detail to the analysis models. Nevertheless there was some conversion of application concepts to computer concepts, such as the introduction of class *Matrix* to implement *Camera* and *Light* behavior.

BIBLIOGRAPHIC NOTES

Computer animation is an active research area in the computer graphics community. [Laybourne-78] describes the conventional animation process including topics on stop-frame animation, cartoon animation, and Disney animation. [Magnenat-Thalmann-85] describes the animation process and several computer animation systems. The ACM Special Interest Group on Computer Graphics, SIGGRAPH, holds an annual technical conference on computer graphics and publishes proceedings each year in the summer. [Rogers-85] describes various rendering algorithms. [Weinstock-86] shows how to do computer animation on personal computers. [Hayward-84] defines common computer animation terminology and techniques. [Gaskill-85] describes techniques that help make a good movie, whether it is computer generated or shot in your back yard with a portable video camera.

REFERENCES

[Gaskill-85] A. L. Gaskill, D. A. Englander. *How to Shoot a Movie and Video Story*. Dobbs Ferry, New York: Morgan and Morgan, 1985.

[Hayward-84] S. Hayward. *Computers for Animation*. Boston, Mass.: Focal Press, 1984.

[Laybourne-78] K. Laybourne. *The Animation Book*. New York: Crown, 1978.

[Lorensen-89] W.E. Lorensen, B. Yamrom. Object-Oriented computer animation. *Proceedings of IEEE NAECON*, Dayton, Ohio, May 1989, Volume 2, 588-595.

[Magnenat-Thalmann-85] N. Magnenat-Thalmann, D. Thalmann. *Computer Animation: Theory and Practice*. Berlin: Springer Verlag, 1985.

[Rogers-85] D.F. Rogers. *Procedural Elements for Computer Graphics*. New York: McGraw-Hill, 1985.

[Weinstock-86] N. Weinstock. *Computer Animation*. Reading, Mass.: Addison-Wesley, 1986.

EXERCISES

19.1 (7) Discuss the feasibility of using the object-oriented approach to extend *OSCAR* to the following applications. In each case, extend the object model and describe any operations that you add.

a. A system to add color to existing black and white movies. Input to the system is a digitized movie. The user colors objects in selected frames and the system colors the same objects in other frames.

b. A display for a flight simulator for training pilots. The system displays the view from the cockpit of an airplane based on the position and orientation of the plane and a landscape database.

c. A video games display system. The system displays a background and several moving objects under the control of the game.

19.2 (4) Identify the actors in the following animation domains:

a. Depiction of structure and transformations of molecules during chemical interactions. Molecules are represented by ball and stick figures, where colored balls represent atoms.

b. Animated television network logos, shown on television during station breaks. Combining the best of art and computer science, these animations are visually pleasing.

c. Analysis of mechanisms. Mechanical components, such as gears, cams, and pistons are modeled in motion.

19.3 (5) Describe how *OSCAR* could be used to simulate the glider described in exercise 12.2.

19.4 (4) Compare the design of *OSCAR* with that of the system described in chapter 18. Be sure to discuss the differences in use of methods, global/local scope of operations, system architecture, and relative importance of object modeling, functional modeling, and dynamic modeling.

19.5 (7) Using the syntax defined in this chapter, animate a scene in which you are viewing a bowling alley from the perspective of a bug sitting at the bottom of a finger hole in a bowling ball as it rolls down an alley.

20

Electrical Distribution Design System

A computer-aided design (CAD) tool is a software system specifically designed to automate all or part of a design activity. CAD tools typically have an interactive user interface and are graphics intensive. CAD tools are currently used in applications such as drafting, chemical process design, mechanical design and analysis, HVAC (heating, ventilation, and air conditioning), integrated circuit chip design, printed circuit board layout, and piping and instrumentation design.

This chapter describes the *O*ne-*LI*ne diagram *E*ditor (OLIE), a CAD tool for designing electrical power distribution systems. The system permits an engineer to lay out a power distribution system as a network of components connected by buses and circuits. The editor is controlled by fixed menus, pop-up menus, and direct input to the diagram.

As with most user interface applications, OLIE is dominated by the dynamic model. The object model is also significant, but the functional model is relatively unimportant. The most difficult aspect of developing OLIE was the requirement to integrate with several existing subsystems: a graphics system, a database manager, and a menu system. These subsystems had various quirks, flaws, and performance overheads, which OLIE had to overcome in order to provide lively and natural user interaction. OLIE illustrates the interactive interface architecture described in Chapter 9.

20.1 BACKGROUND

A "one-line diagram" [Russell-78] is a schematic drawing of an electrical power system that uses a concise, standard notation accepted by all power engineers. One-line diagrams get their name from the fact that only one phase of a three phase system is shown. Standard symbols are used to represent components of a power system, such as transformers, circuit breakers, generators, fuses, and switches [ANSI-75]. Circuits and buses connect the standard symbols. The distinction between circuits and buses is based on electrical voltage drop. Circuits are connections with measurable voltage drop (especially if there is a short circuit on

the power system) and are typically implemented with wires, cables, or transmission lines. Buses are connections with no appreciable voltage drop and are usually implemented with large cross section aluminum, copper bars, or pipes.

A sample portion of a one-line diagram is shown in Figure 20.1. Two high voltage (138 KV) buses are shown. Arrows at the top of the diagram indicate sources of power, such as a utility. Arrows at the bottom indicate loads. Circuit breakers are shown as squares; transformers are indicated by zig-zag lines. The small solid dots distinguish connecting lines from overlapping lines. Four separate sources are connected to the two high voltage buses through nine circuit breakers. By selecting which circuit breakers are open and which ones are closed, the power system operator can configure the power supply to the three transformers in several ways. This would allow one bus to be disconnected during construction of additional circuits, for example. The loads on transformer 1 and transformer 2 can be served by either transformer by closing the breaker that connects the two secondary buses.

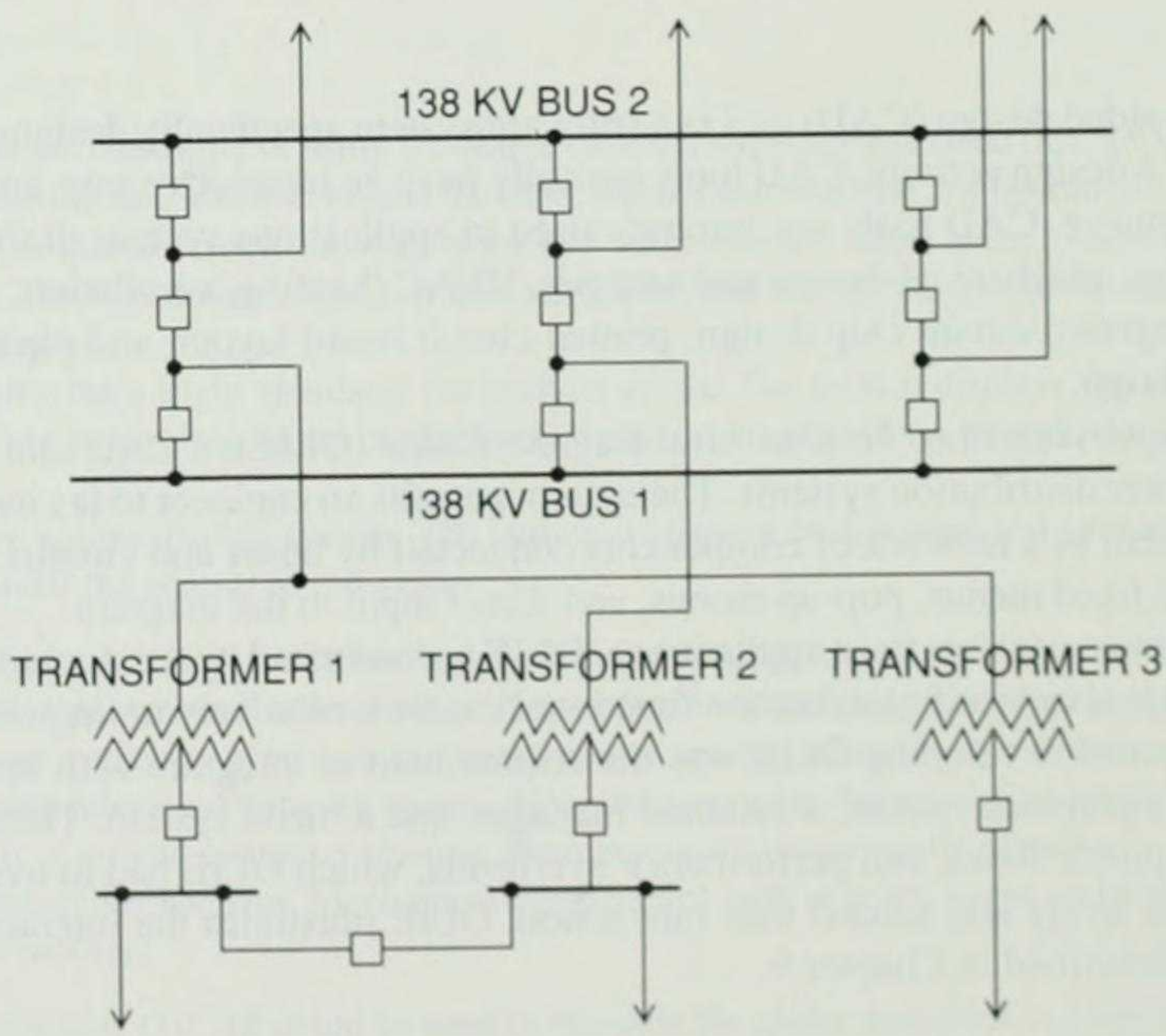


Figure 20.1 Sample one-line diagram

The motivation for developing OLIE derives from recent advances in computing technology. Engineers of various disciplines would like to replace the old analysis software developed twenty years ago that operates in batch mode and requires that input be submitted as card images. A more natural approach to specifying a simulation is to draw a diagram on the screen and annotate the drawing with desired design conditions. This new approach permits menu driven interaction and possibly interaction with the simulation program during its execution. For example, an engineer can check on the convergence of a simulation or change design conditions as a computation proceeds.

20.2 PROBLEM STATEMENT

Several sources were used to determine functional requirements, including marketing studies, customer requests, examination of other graphics editors, and the experience of the OLIE software designer as an electric power engineer. Marketing studies envisioned that OLIE would integrate with other computer-aided design and engineering software tools used in architecture, engineering, and construction (AEC) applications through common project databases, as shown in Figure 20.2.

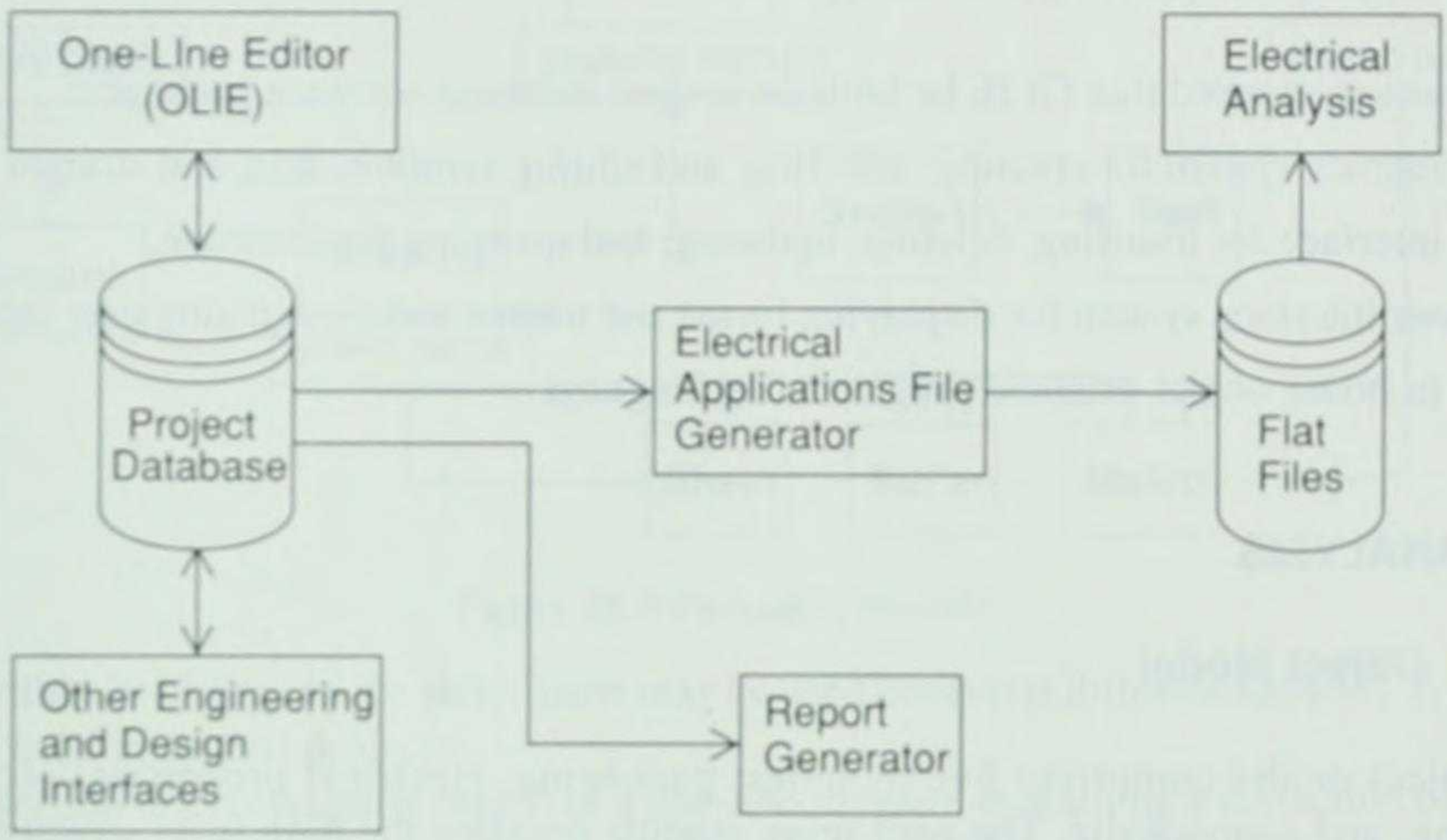


Figure 20.2 Integration of OLIE with other applications

20.2.1 Functional Requirements

OLIE must store its data in a database to be managed by a database management system (DBMS). At first, this may seem to be an improper "requirement" and appear to be a possible decision that could result from analysis of the "true" requirements. However, the OLIE software is not intended to be an end in itself but merely one of a suite of tools that electrical power engineers would use to perform their work. Thus OLIE must interface to other power engineering programs. The requirements analysis for this larger system lead to the conclusion that OLIE must use a database.

By definition, OLIE must provide basic commands for editing one-line diagrams, such as create, pan, zoom, drag, move, copy, pick, and put. OLIE must have a lively response so that it does not interrupt the user's train of thought. The user interface must be natural to engineers who are not computer experts.

OLIE must permit data to be shared among users, application programs, and projects. The system must permit portions of a design to be locked for periods of extended work. These extended duration locks permit work that may last for days or weeks, in contrast to the ordinary database transactions that complete in a matter of seconds. A typical OLIE user

has several active projects at a time; multiple engineers may work on each project. Libraries of information, such as drawing symbols, span many projects.

OLIE must recover from storage media failures, computer failures, and software errors. The software must be portable to other hardware, operating systems, and software subsystems, including new graphics hardware. OLIE must be extensible to include new functions and new application programs and be easily maintained. Engineers must be able to generate reports about a power distribution system in several formats.

20.2.2 Architectural Requirements

Management required that OLIE be built on several in-house software packages:

- a graphics system for creating, selecting, and editing symbols, text, and straight lines,
- an interface for inserting, deleting, updating, and querying the database,
- a user interface system for displaying forms and menus and recognizing user input, and
- an in-house object-oriented programming language.

20.3 ANALYSIS

20.3.1 Object Model

The object model comprises five modules: packaging, electrical properties, geometry, grouping, and connectivity. The packaging module contains the high level structure for a project: A project consists of multiple one-line diagram drawings, each of which is further organized into one or more sheets. The electrical properties module defines attributes for each electrical component. Geometry describes the appearance of electrical items on the screen. Grouping describes the construction of composite items from simple items. The connectivity module captures the correspondence between a diagram and an electrical network.

The actual modules have much detail that is not shown; here we just show the basic structure. We have shaded classes that are shared between modules.

Packaging Module

The packaging module (Figure 20.3) hierarchically organizes data for user access. The class OLIE *root* has a single instance that provides a context for library and project names. A customer may have one or more projects active at the same time, each identified by project name. Within each project there are several drawings, each qualified by a drawing name. A drawing name is unique within a project but may be reused in other projects. A drawing has multiple sheets that are also named.

There may be several symbol and macro libraries. Each project is assigned one symbol library and one macro library. A symbol is an icon that has meaning within the context of a one-line diagram; thus for instance, parallel zig-zag lines denote a transformer. A macro is a user defined grouping of symbols, buses, and circuits. A macro or a symbol is qualified by

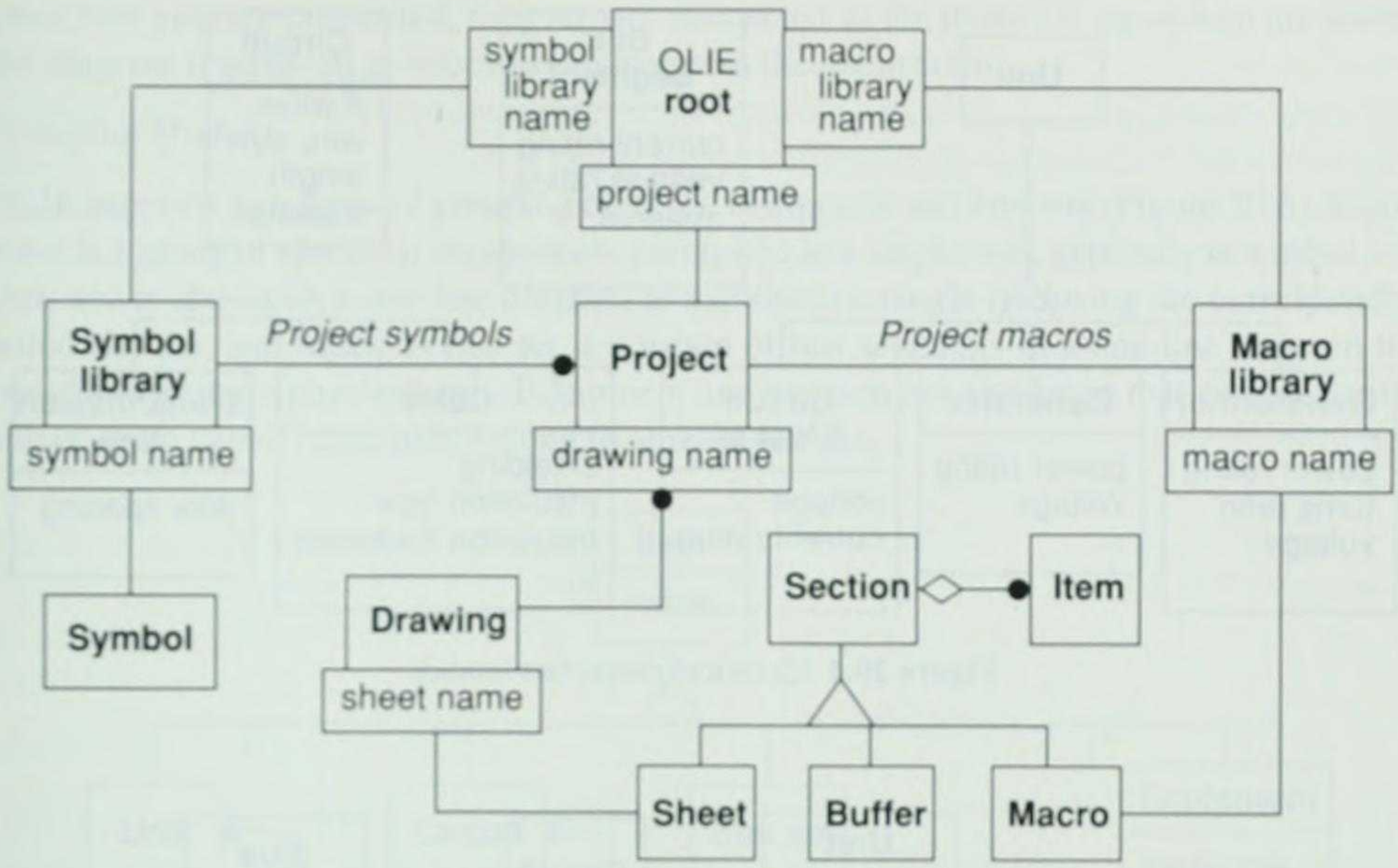


Figure 20.3 Packaging module

name within its library, so the same name may be used in several different libraries. A library may be used in several projects.

A section is a rectangular region of a one-line diagram containing a collection of items, such as symbols, buses, and circuits. A section is the smallest unit of access control to a diagram, as different sections can be edited by different engineers concurrently. Sheets, buffers, and macros are all sections. A sheet is the lowest level in the project hierarchy containing a portion of a one-line diagram. A buffer contains a portion of a one-line diagram involved in an editing operation. Since sheets, buffers, and composites all contain the same type of items, generalizing them into a section permits code reuse.

Electrical Properties Module

The electrical properties module (Figure 20.4) defines electrical properties for components in a power distribution system. The purpose of this module is to define attributes of classes; this module contains no associations. Circuits come in two varieties, cables (underground) and transmission lines (above ground), each inheriting common attributes from circuit. Bus segments have simple structure. Units come in many different varieties, such as transformers and circuit breakers. The attributes in this module do not appear on the one-line diagram; their values are entered by the engineer on pop-up forms.

Geometry Module

The geometry module (Figure 20.5) describes the layout of a one-line diagram on the screen. Circuits and bus segments, collectively called path items, are drawn as polylines (connected line segments) with user set width. Units are represented with arbitrary symbols chosen from

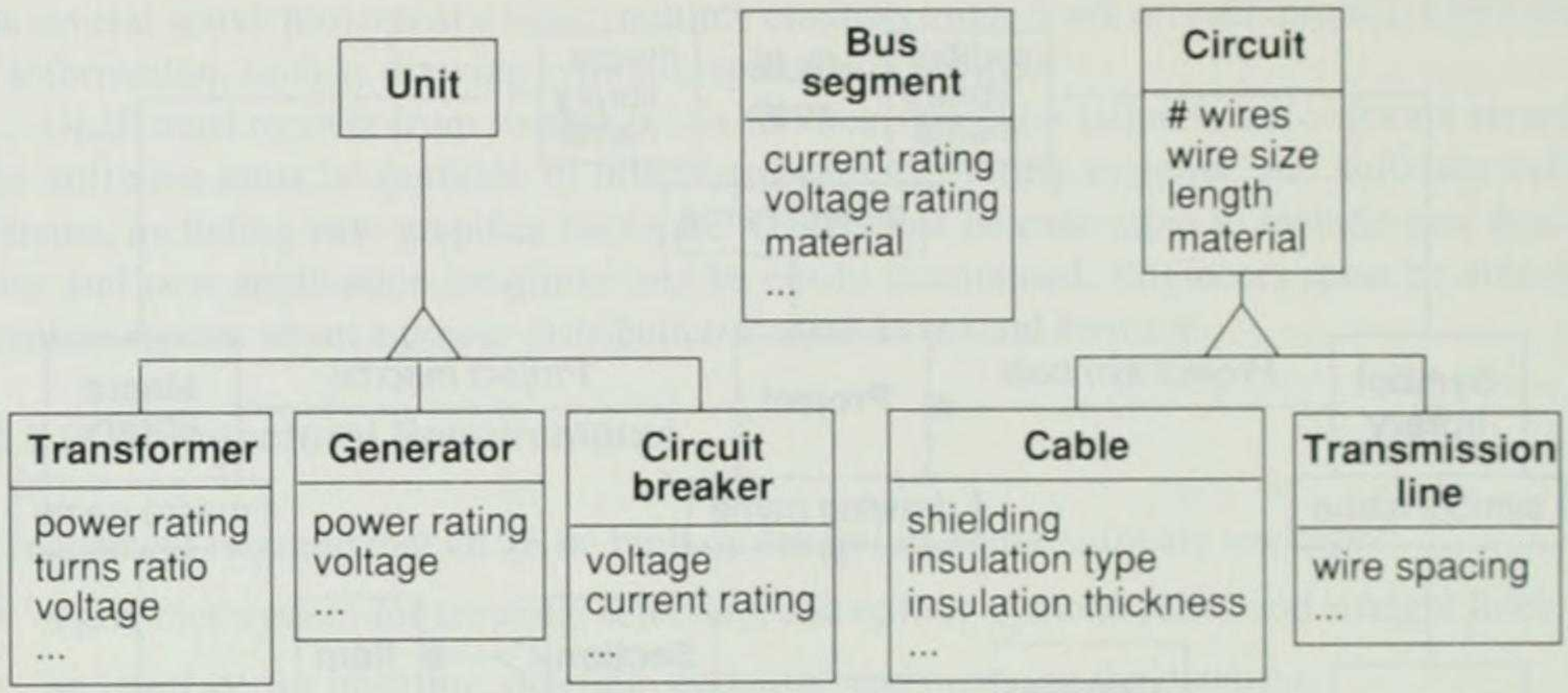


Figure 20.4 Electrical properties module

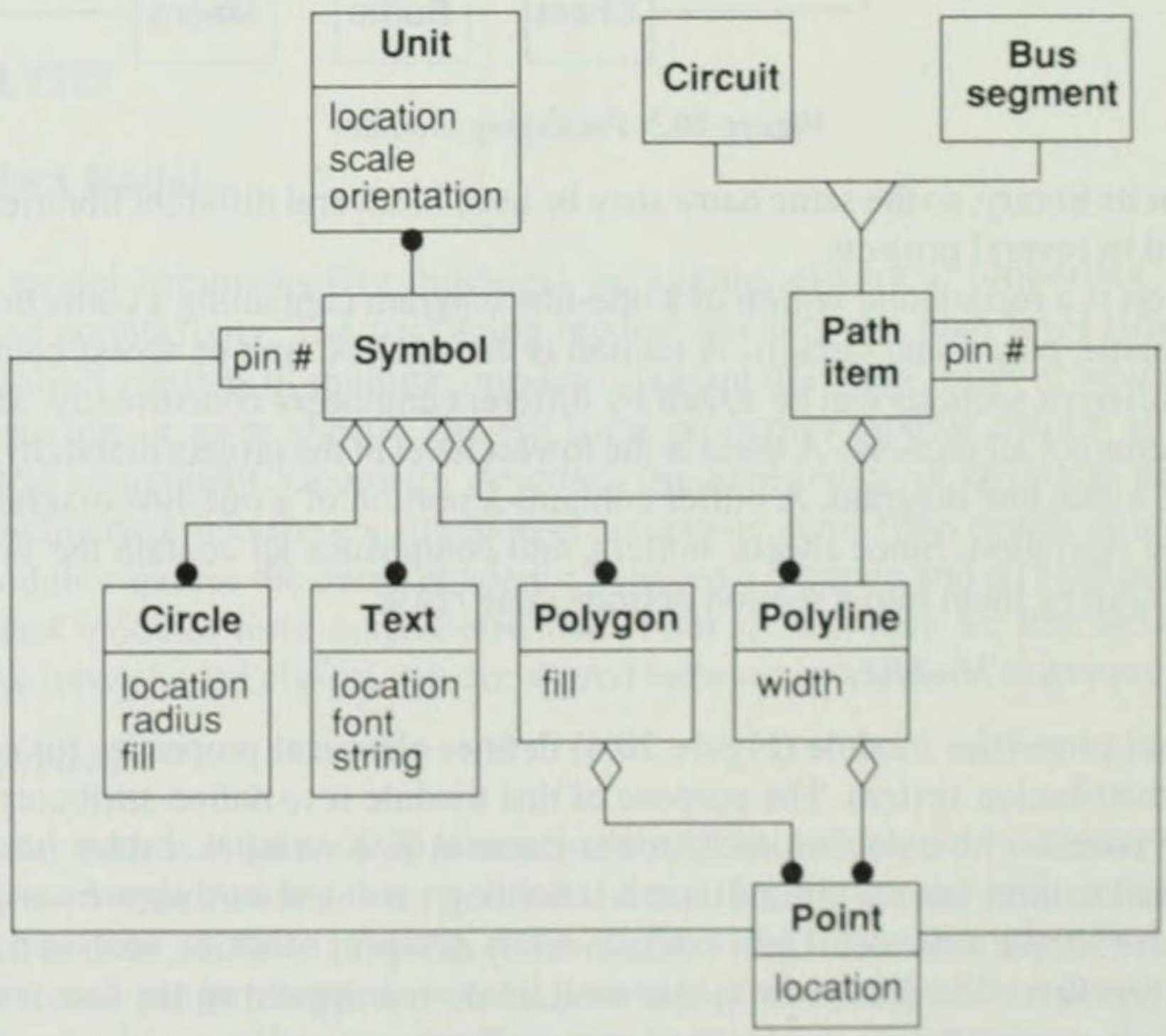


Figure 20.5 Geometry module

the symbol library. Each symbol is constructed from geometrical shapes. Each symbol has a fixed number of predefined connection points, or pins, which can be connected to pins of other symbols on the diagram. Each path item has two pins, one at each end of the polyline.

Once two pins are connected, they remain connected as the items are moved on the screen; the diagram is adjusted as necessary to maintain the connection.

Grouping Module

OLIE supports two types of grouping of items, equipment and bus sets (Figure 20.6). Equipment is a group of electrical components purchased as a single unit, generally in a metal cabinet, and is shown on a one-line diagram as a dashed rectangle enclosing the corresponding units, circuits, and buses. A bus set is a group of bus segments in a one-line diagram that constitute a single physical unit. Equipment and bus sets are similar in that each is a set of one or more items. Items may belong to at most one set.

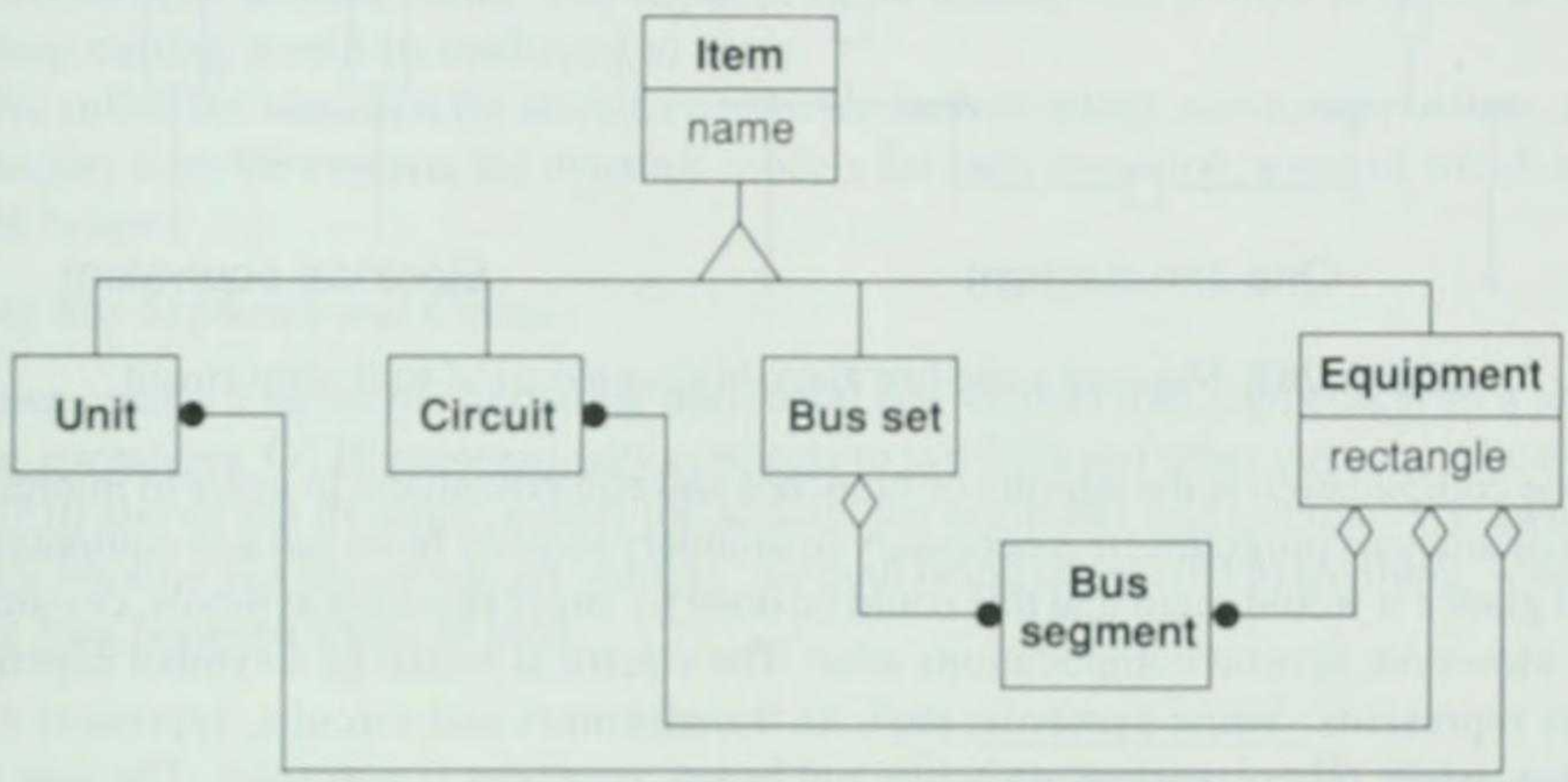


Figure 20.6 Grouping module

Connectivity Module

The connectivity module (Figure 20.7) relates the physical connections in the diagram to the electrical connections in the power system. The one-line diagram editor and electrical analysis programs each have a slightly different view of a power system. The editor regards a one-line diagram as consisting of an interconnected network of symbols, buses, and circuits. Editing operations such as move, copy, and cut must quickly determine broken connections and buses and circuits that are stretched or compressed. Electrical analysis programs view a power system as an electrical circuit: a branch and equipotential representation, similar to

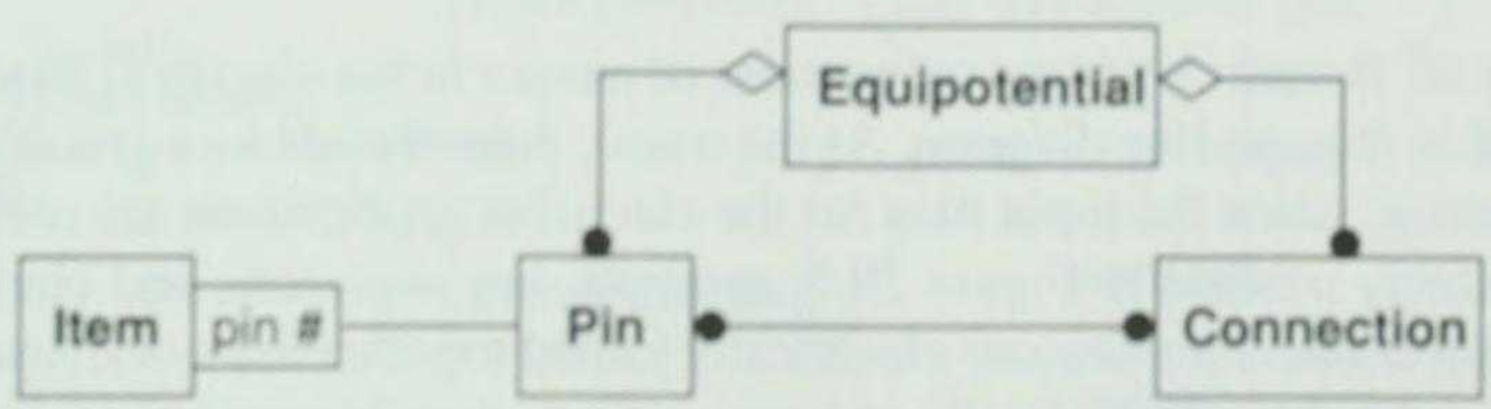


Figure 20.7 Connectivity module

an edge-vertex graph. A branch is a two port circuit element that relates electrical current through the branch to the voltage difference across the branch. Ports of branches are connected together in an equipotential. All ports connected in an equipotential are at the same voltage. The sum of the currents entering an equipotential is zero.

The mapping from symbols, buses, and circuits in one-line diagrams to branches and equipotentials as an equivalent electrical circuit is not trivial. A simple example is shown in Figure 20.8. With the circuit breakers closed, the two buses are equipotential.

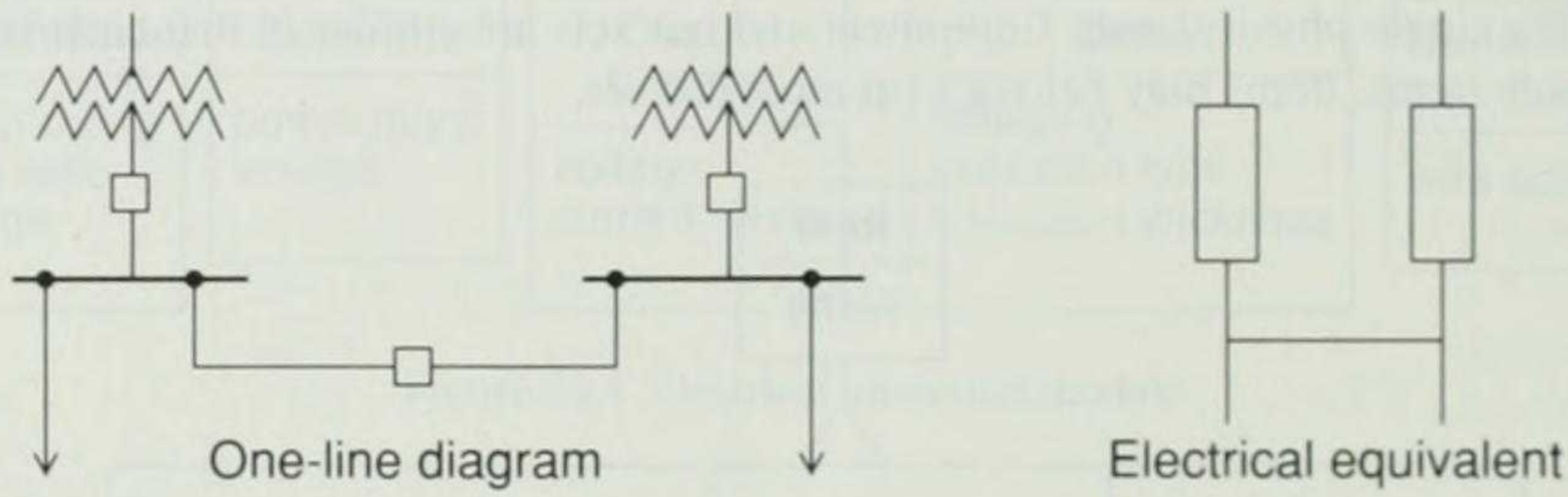


Figure 20.8 Mapping a one-line diagram to an electrical equivalent circuit

One complication is the naming of branches and equipotentials. In order to interpret the results of analysis programs, it is necessary to uniquely identify branches and equipotentials. At first glance it would seem that this could be done by simply naming symbols, circuits, and buses. However, several complications arise. The electrical model of a symbol depends on what it represents. Some symbols, such as transformers and circuits, represent circuit branches, while others, such as switches and buses, represent connections. The user could inadvertently assign more than one name to an equipotential, by naming the buses on either side of a switch, as shown in Figure 20.9.

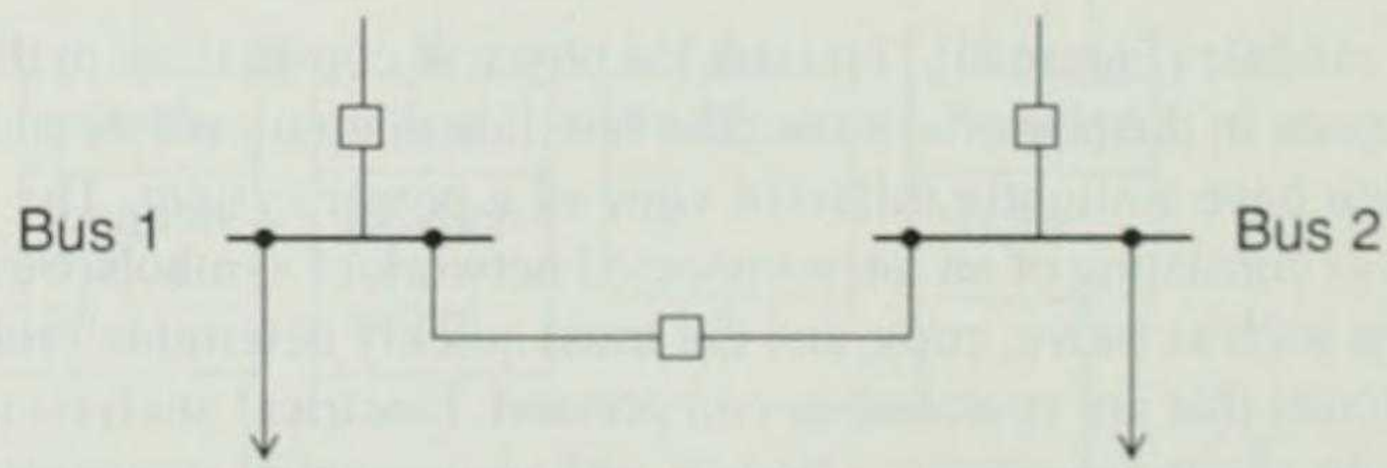


Figure 20.9 Ambiguity in assignment of names to buses

Ideally, OLIE should check the uniqueness of names in the electrical model any time a name is created in the one-line diagram. At the worst, there should be a graceful way of correcting the situation when the input files for the electrical applications are created.

In the geometry module in Figure 20.5, symbols, bus segments, and circuits have pins for making connections. Pins that are electrically joined together are associated with a common connection. Equipotentials, pins, and connections map between the representation of the one-line diagram as symbols, buses, and circuits and the branches and nodes. Branches are not explicitly shown in the model, because they are easy to determine.

20.3.2 Dynamic Model

Since OLIE is an interactive editor, the dynamic model and the user interface are essentially the same thing. After revisions to the object model became infrequent, we began designing the user interface. We examined other graphic editors and selected the features with the best "look and feel." We grouped user interactions by function, such as display control, editing, traversing the project hierarchy, and so on, and developed scenarios.

Then we designed menus and specified mouse behavior. The left mouse button selects items from the diagram or commands from fixed menus. The middle button extends selections. The right button select commands from pop-up menus. Sometimes, selecting a menu item would raise another menu. The depth of menu nesting was limited to three, as we felt that deep nesting would be confusing to users.

We culled the scenarios for atomic operations, such as select, move, copy, delete, mirror, and display text. We constructed dynamic models for each operation, some of which are discussed below.

Adding Bus Segments and Circuits

The user creates a series of alternating horizontal and vertical line segments with a series of mouse operations. OLIE automatically connects to symbols and other circuits or buses. Figure 20.10 shows the dynamic model for adding bus segments and circuits. The two differ slightly because bus segments can "dangle" without being connected to anything, while each circuit runs between a pair of pins.

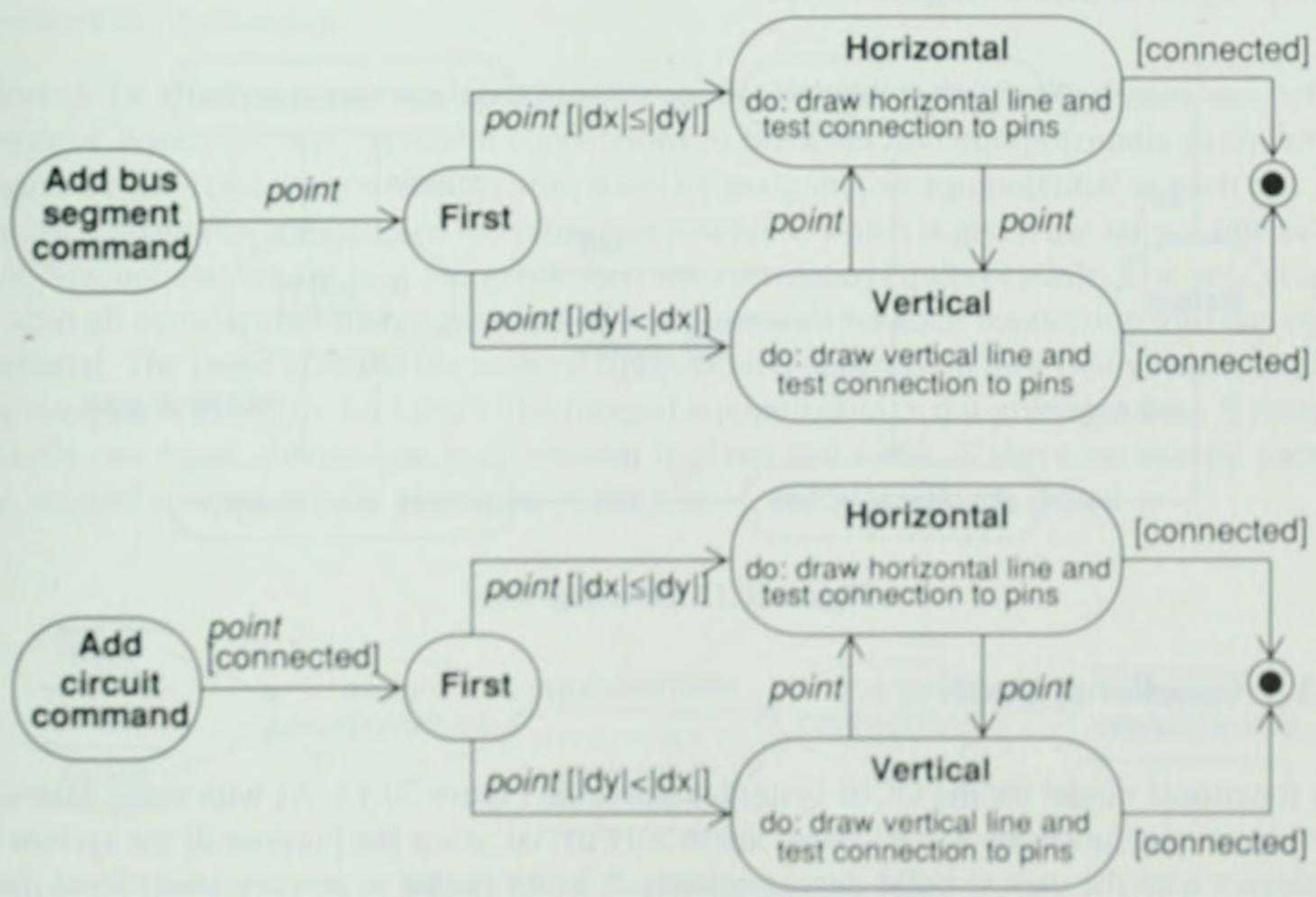


Figure 20.10 Adding bus segments and circuits

Adding Items

Items can be symbols or macros. They are identified by name from the library, followed by a location at which the image of the item is drawn on the diagram as shown in Figure 20.11. Multiple copies of a symbol can be added by clicking on a series of points. The user must indicate when to stop adding items by choosing *done*.

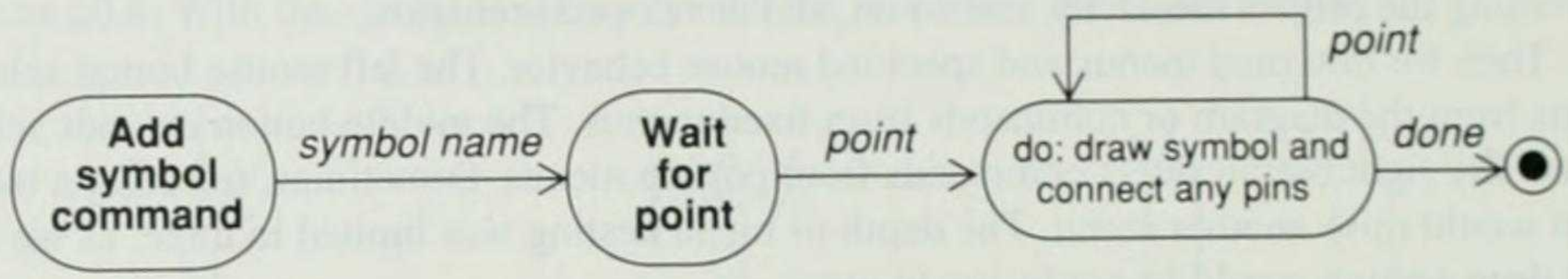


Figure 20.11 Adding symbols

Selection of Geometry

In Figure 20.12, buses, circuits, symbols, and text are selected individually or in groups for further operations. The left mouse button starts a new selection while the middle button adds items to an existing selection. If either button is pressed and released, then an item (if any) near the mouse point is selected. If either button is pressed, held, slid, and released, all items (if any) enclosed in the rectangle between the down point and up point are selected. The details of press-hold-slide-release are not shown on the dynamic model. Such details belong to the model of the graphics driver that captures mouse events. Instead, the diagram shows *left* and *left region* as distinct logical events.

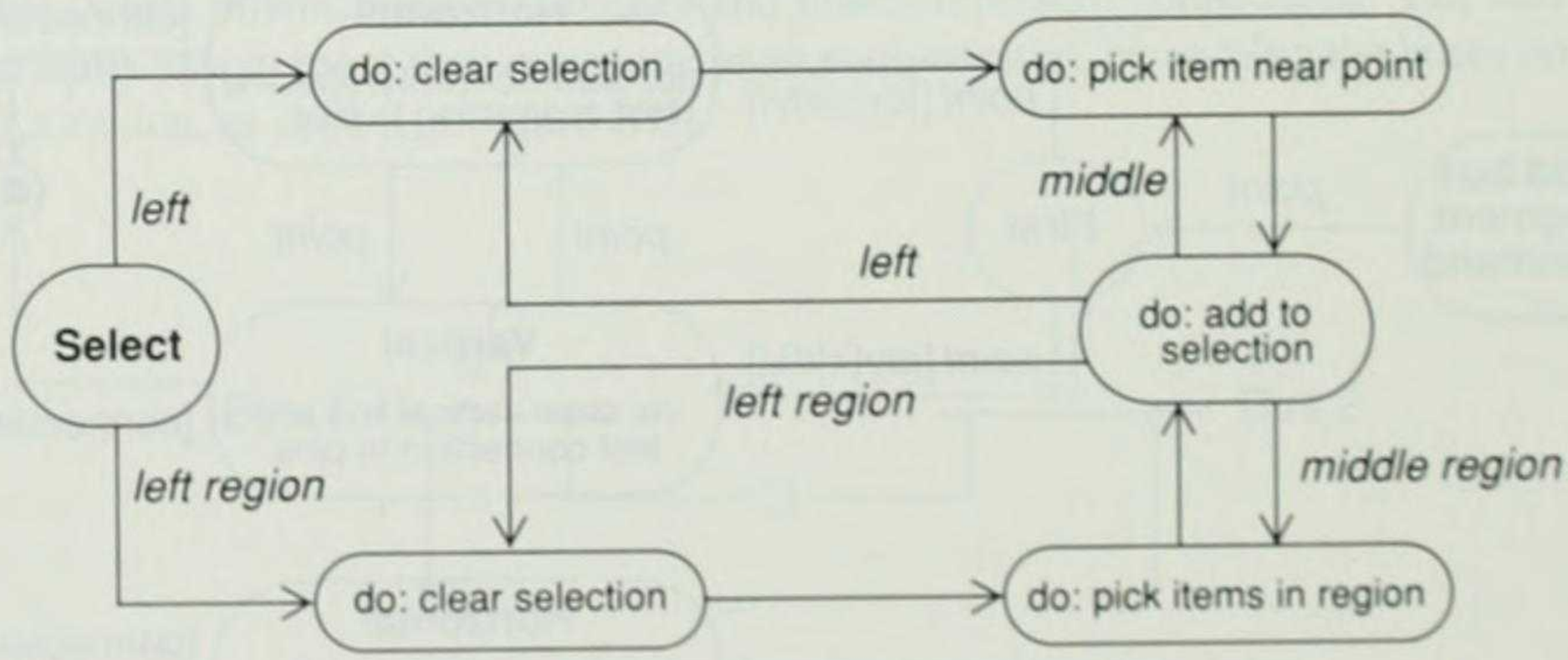


Figure 20.12 Selecting items

20.3.3 Functional Model

The functional model for the OLIE system is shown in Figure 20.13. As with many interactive editors, the functions in the system are mostly trivial, since the purpose of the system is to interact with the user to build data structures. It is not useful to prepare lower-level data

flow diagrams for the editor or project control. The electrical analysis is the only significant part of the functional model. Electrical analysis is performed by external stand-alone programs, so their internal functional models are generally inaccessible (it is useful to prepare a list of the inputs to each one, however). The extract connectivity process is the only part of the analysis that is part of the OLIE system proper.

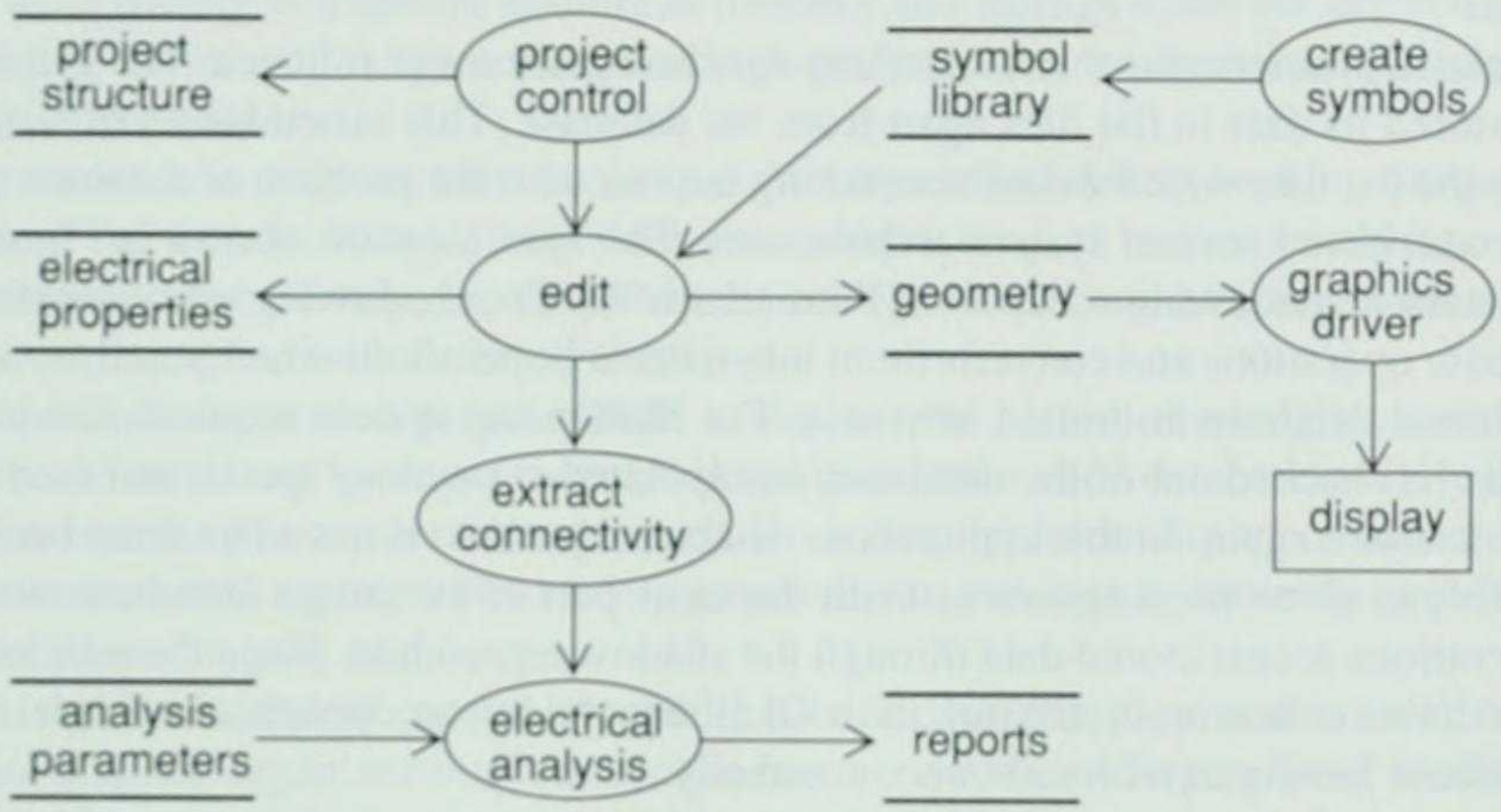


Figure 20.13 Functional model of OLIE system

Connectivity Extraction

Most of the effort in preparing input files for application programs derives from converting symbols, buses, circuits, pins, and connections to branches and equipotentials as shown in Figure 20.14. The conversion algorithm starts by assigning an equipotential to each pin. All pins of a given bus are assigned the same equipotential, which is given the same name as the bus. Symbol and circuit pins are given their own unnamed equipotentials. The next step is to scan all connections, merging the set of equipotentials for each connection into one equipotential. The name given to the merged equipotential depends on how many names are being merged. If there are no names, the merged equipotential is not given a name. If there is exactly one name, the merged equipotential is given that name. If there are several names, the merged equipotential is given one of the names, and a warning is issued.

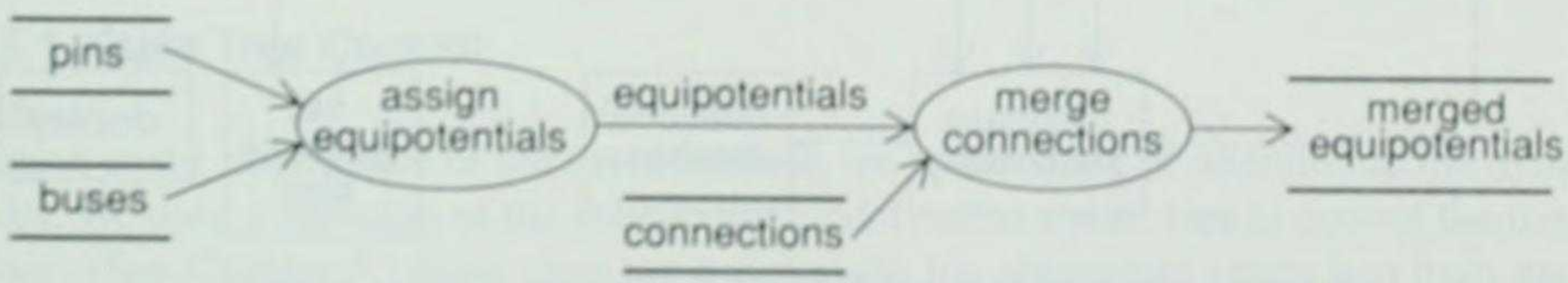


Figure 20.14 Connectivity extraction

20.4 SYSTEM DESIGN

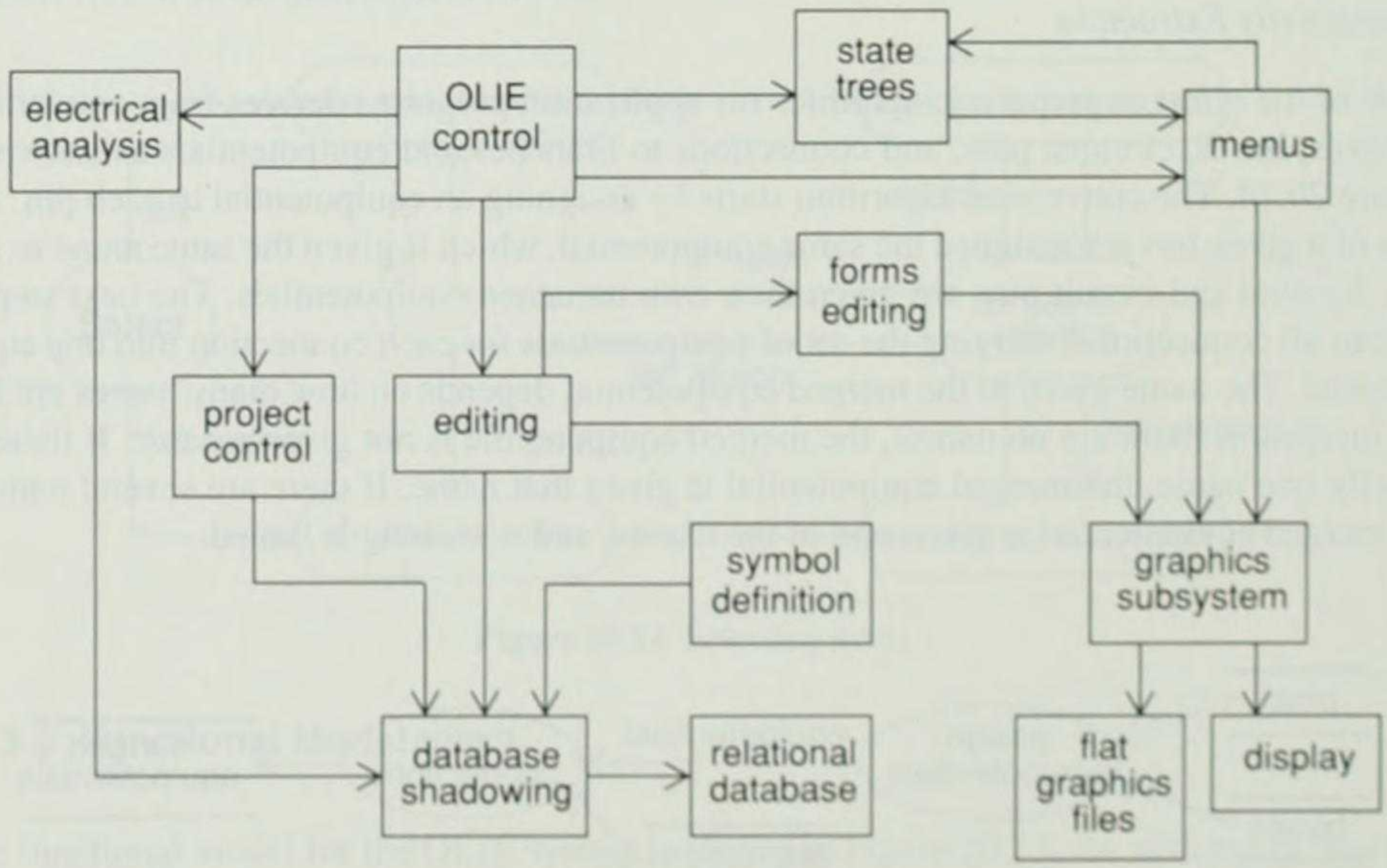
We feared that it would not be possible to simultaneously satisfy the requirements that we store one-line diagrams in a database, support data sharing, and provide a fast, lively user interface. We considered the possibility of OLIE directly interacting with the database, but benchmark testing revealed that response would be too slow to support a lively user interface.

A related problem was that of database synchronization and rollback. The graphics subsystem stored its data in flat files apart from the database. This raised issues of how to synchronize the flat files with the database, taking into account the problem of database rollback.

We considered several system architectures. The final solution, shown in Figure 20.15, used a database shadowing subsystem [Premerlani-90]. The shadowing subsystem intercepts all database operations and converts them into memory operations when possible, accessing the relational database at limited intervals. The shadowing system requires that chunks of data must be checked out of the database, manipulated at memory speed, and then checked into the database again. In this application, database checkout is not a handicap because it is undesirable to allow two engineers to edit the same part of the design simultaneously.

Operations access stored data through the shadowing module. Since the relational database maintains concurrency control, each OLIE execution can operate as a single-user program without having to worry about concurrency.

Operations access the graphics display through the graphics subsystem, which displays the one-line diagram and stores graphical information in flat files. The editing module main-



Arrows show dependence of client module on supplier module.

Figure 20.15 Architecture of OLIE system

tains the consistency of the database and the flat files. Graphical information is stored redundantly in the database and the flat files to boost performance. Normally, the graphics system stores graphical information in the flat files and the shadowing system stores graphical and other information in the database. In the event that the information in the flat files becomes inconsistent with the database, the shadowing system rebuilds the flat files through the graphics system.

We were willing to trade-off additional memory and storage space for performance and functionality. Users of this system require high-performance workstations anyway, so this trade-off is reasonable.

The names in a diagram must be unique. We decided to defer checking uniqueness of names until the user is done editing. Incremental checking of names would have greatly complicated the concurrency constraints among multiple users, and a survey of users showed that they were satisfied with deferred checking.

Database checkout also permits OLIE to support the notion of a *long transaction*, that is a transaction that lasts for days or weeks, in contrast to the usual database activity that transpires in a matter of seconds. An engineer's work may be internally inconsistent during a long transaction; thus engineers can freely pursue their work and at the same time OLIE can satisfy database integrity requirements.

Our implementation of control adopts the event driven paradigm and is based on an explicit state diagram organized into a tree, called a *state tree* (see Chapter 5 and Section 20.5). All input events cause state transitions within the tree. Menus are enabled and disabled by the state tree control; in turn, user menu selections generate input events interpreted by the control. Transitions in the tree force the execution of actions, such as cut, paste, and move. We implemented control with state trees, instead of Harel's statecharts (we regard statecharts as a superior approach) for historical reasons. We had an implementation of state trees available in-house; OLIE was developed before we became aware of Harel's work.

20.5 OBJECT DESIGN

Object design was straightforward, given the analysis and architecture decisions. The graphics, database, state tree, menuing, forms editing, and electrical analysis subsystems were all predefined. The main work with these subsystems was to properly set up input parameters. Considerable effort went into the layout of menus, forms, and item attributes, but we will not reproduce the details here. The bulk of the work involved the editor: the high-level interactive flow of control (policy) and the low-level operations (implementation).

20.5.1 State Tree Control

Because of the importance of the user interface, we paid particular attention to the dynamic model. We used a variation of the state-event model called a state tree to control the user interface. (See Chapter 5.) State trees are a technique for organizing states into trees and attaching events to tree nodes to share common structure and behavior, permitting a layered decomposition of control structures and the elimination of redundant information. Events at-

tached to states specify how events are interpreted and can be inherited by substates, just like attributes and operations on object classes. Entry and exit actions attached to states specify the effect of entering and leaving a state. Because states are organized into a tree, a substate cannot be entered without its superstate being entered so that the entry (and exit) actions are inherited as well.

A state tree is equivalent to a Harel state diagram in which subtrees are nested states within the superstate representing a tree node. The entry and exit actions on the tree nodes are equivalent to entry and exit actions on the nested state contours.

State trees naturally support a menu driven interface. Each menu entry causes a transition to a given state. Any menu entry can be selected in any state because the exit actions of the current state allow it to be cleaned up regardless of the user command.

The state tree used in OLIE consists of two separate trees: a main tree and an auxiliary tree. The main tree contains states normally encountered during user interaction. All main states are mutually exclusive; the user can only execute one main command at a time. The auxiliary tree handles user interrupts without losing the context of the main tree. For example, OLIE has pan and zoom commands that can be picked at any time, even in the middle of other operations. Pan and zoom commands retain control until the user exits from them, at which time control returns to the main tree.

One feature of state trees that OLIE exploited is that menu organization does not have to reflect the state structure. This eliminates the problem frequently encountered in other approaches in which a series of "quit" or "exit" commands are needed in changing context.

20.5.2 Main Tree

A portion of OLIE's main tree is shown in Figure 20.16. The tree is organized according to command functionality with subdivisions within each kind of command.

Only some states are shown. When the program starts, it enters the *Initialize* state, which displays the fixed menus, performs initialization, and transfers control to the *Select sheet* state, which allows the user to select an initial sheet for editing. *Utilities* substates create reports, composites, plots, or the files needed by electrical analysis programs. *Add geometry* substates create items in one-line diagrams. The *Exit* state queries the user to save any changes that have been made. The exit command can be picked by the user at any time, because any command in progress is cleaned up by the exit functions on the state tree path from the current state to the *Exit* state. *Layout* substates perform editing operations such as pick, put, move, and copy. *Annotate* substates pop-up text forms to enter electrical parameter values.

20.5.3 Auxiliary Tree

The auxiliary tree, shown in Figure 20.17, contains states for commands that can interrupt other commands. *Pan-zoom* is used for changing the viewpoint of the one-line diagram. *Display control* changes display options such as the visibility of a grid or text. *Select geometry* provides control for selecting elements of the one-line diagram. *Orient-scale* is used to change the orientation or size of selected elements.

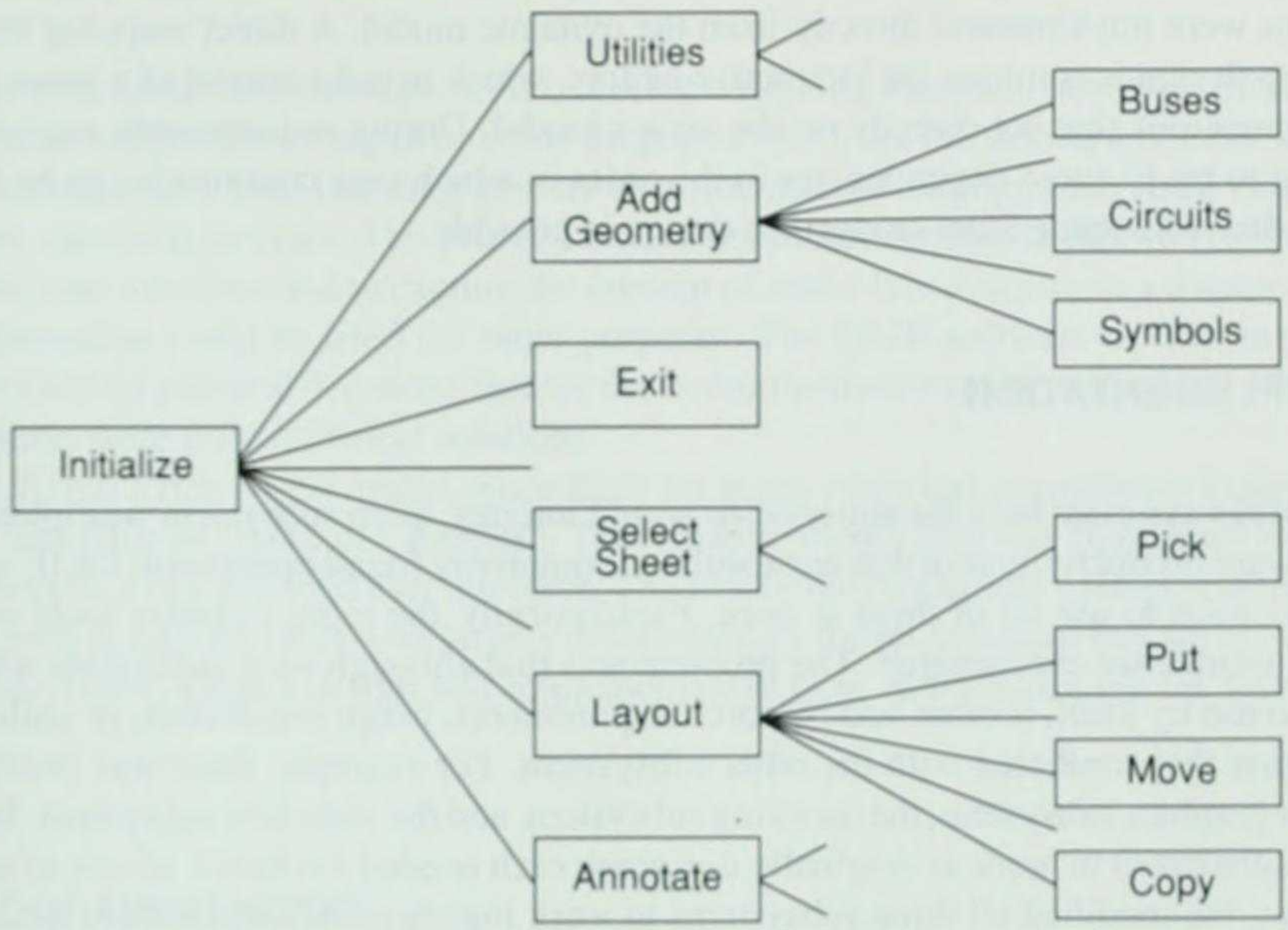


Figure 20.16 Main state tree

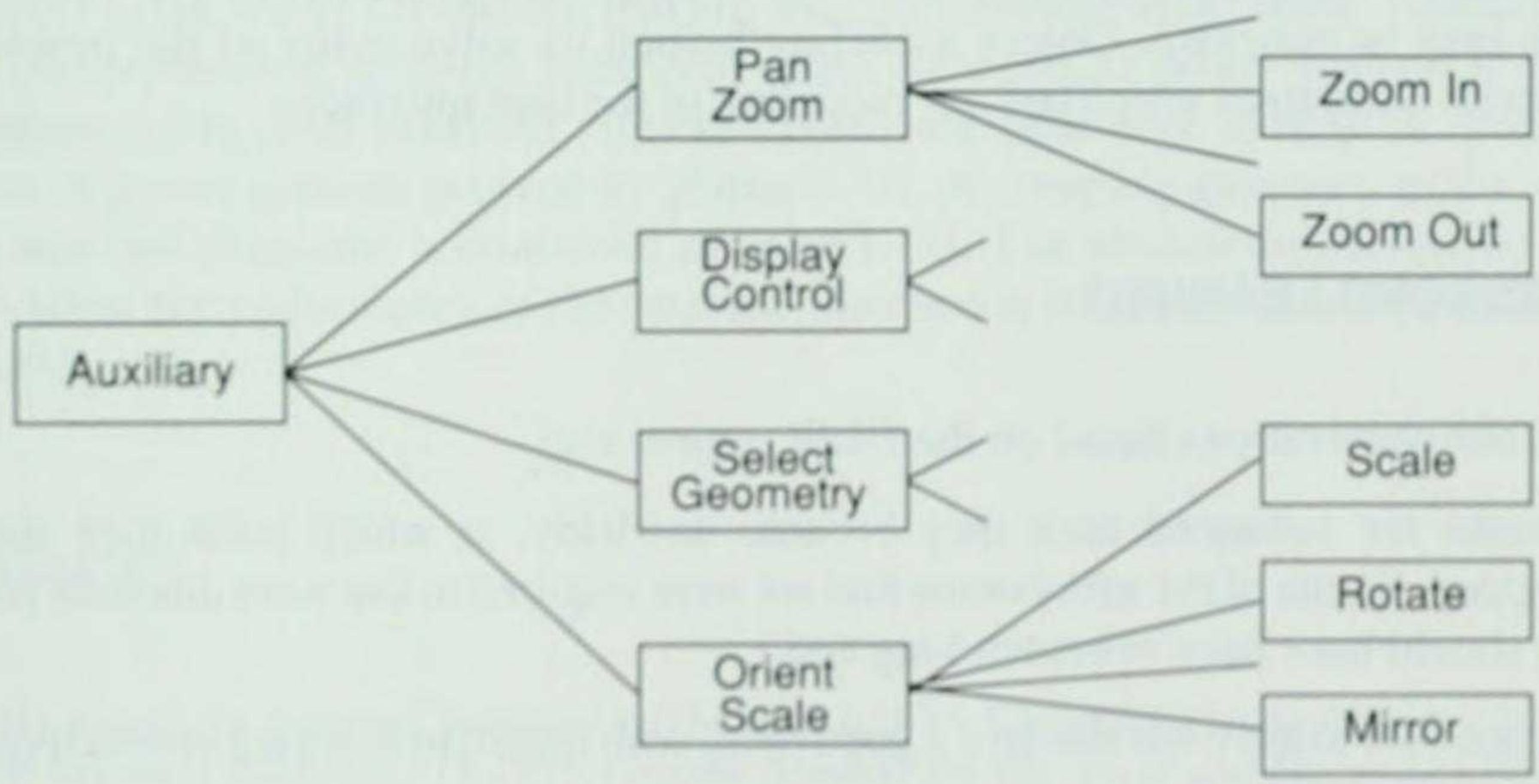


Figure 20.17 Auxiliary state tree

20.5.4 Low-level Operations

Objects were implemented directly from the analysis model in an object-oriented language using association objects to directly implement associations. The same objects are stored in memory and in the database (the shadowing subsystem uses memory to cache the database),

EXERCISES

20.1 (8) Relational database operations (insert, delete, update) in OLIE are buffered in memory with a checkout/check-in scheme. When a user wishes to edit a sheet, OLIE locks the sheet and reads all records from the database belonging to the sheet. Each record is converted into an object instance in memory and tagged with a state that is used to determine what should be done to update the database when the user saves editing changes or checks the sheet back in. As the user edits the sheet, OLIE updates the tags. Possible states are *Same*, *Different*, *Not Inserted Yet*, and *Not Deleted Yet*. *Same* indicates that the object instance in memory contains the same data as the record in the database. *Different* means the instance has different data than the corresponding record. *Not Inserted Yet* indicates the instance is due to be inserted into the database. *Not Deleted Yet* means the instance is due for deletion from the database. When objects are copied from the database, they are placed in the *Same* state and remain in that state unless they are updated or deleted. Newly created objects are in the *Not Inserted Yet* state. Objects that need to be deleted are placed in the *Not Deleted Yet* state. Each time an object is accessed, its new state is determined from the operation and the old state. The operations are *insert*(new data), *delete*(object), *update*(object), *save*(set of objects), and *load*(portion of database). Some operations may undo others. For example, an insertion can be elided with a subsequent deletion.

Figure E20.1 is a partially completed state diagram for updating tags. Initial and final states are missing as well as activities, actions, events, and so forth. (Refer to Chapter 5 for a discussion of initial and final states, which in this case are used to allocate and deallocate memory used for objects.) Complete the diagram. Be sure to show what needs to be done for a save operation. Some helpful hints: objects in the *Same* state do not require any action during a *save* operation, since the data in memory is the same as that in the database. Certain operations are illegal on objects in certain states and need not be shown. An *update* operation on an object in the *Not Inserted Yet* state leaves it in the same state, since the database does not yet know about the object. The only legal operation on an object in the *Not Deleted Yet* state is *save*, which requires the corresponding record in the database to be deleted and the object to be terminated.

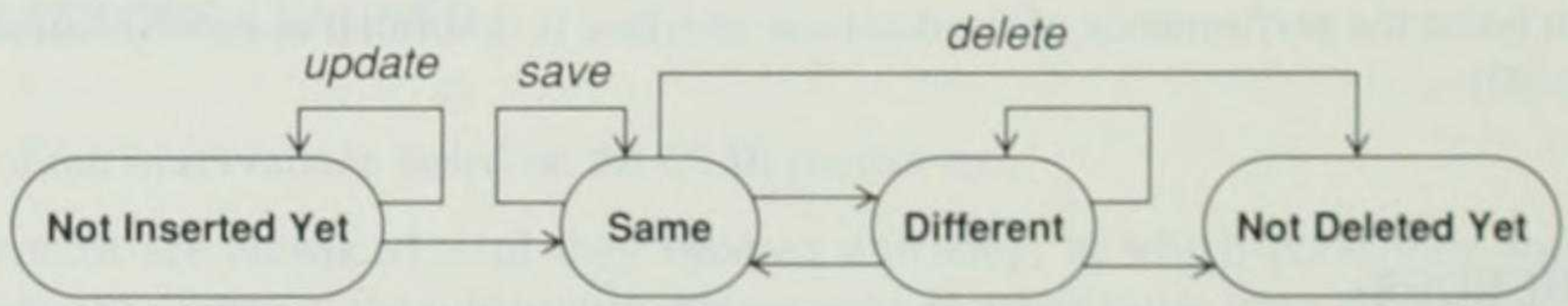


Figure E20.1 Partially completed state diagram for a system to buffer database operations by shadowing records with object instances

20.2 (7) A commonly occurring problem in the design of graphical diagram editors which preserve connectivity is that of detecting bridging networks. For example, if some symbols in a network are moved, some lines in the network may require adjustment to accommodate the relative displacement between symbols that are being moved and those which are not. Three ways of modeling connectivity are shown in Figure E20.2, Figure E20.3, and Figure E20.4. In each case, networks may connect many symbols, and symbols may be connected to many networks. A given network may be connected several times to a given symbol, and more than one network may be connected to a symbol at the same place. In Figure E20.2, the object class, *Port*, is the place on a symbol where a connection is made or the end of a branch of a network where it is connected

to a symbol. Figure E20.3 shows the result of eliminating the *Port* class. A connection is the attachment point of a network to a symbol. Figure E20.4 is a further simplification.

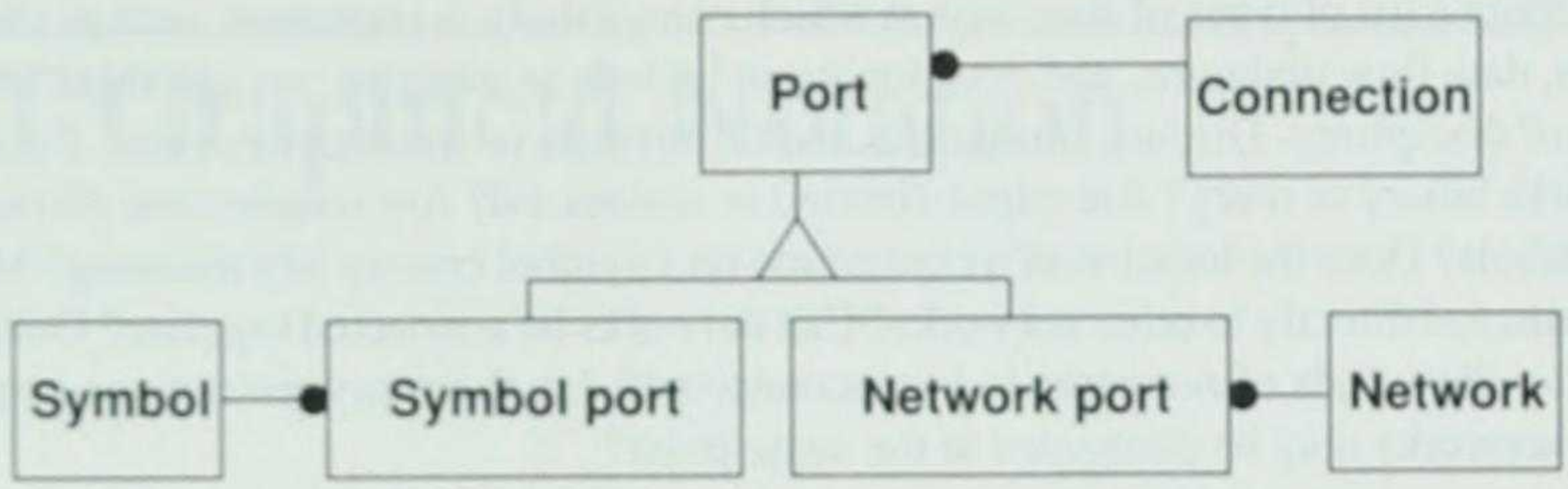


Figure E20.2 Connectivity object diagram

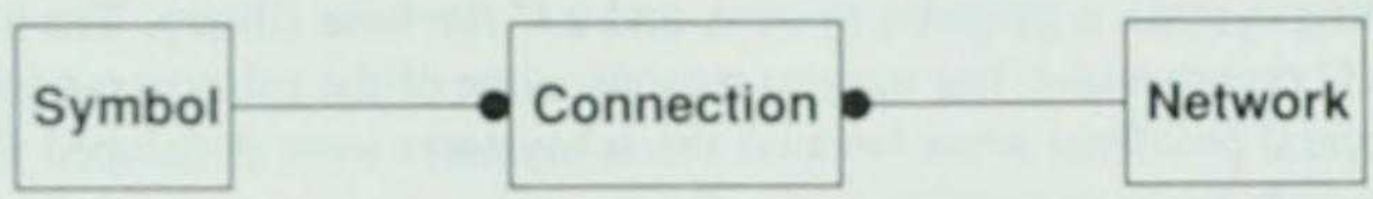


Figure E20.3 Connectivity object diagram

Figure E20.4 Connectivity object diagram

For each diagram, describe how you would accomplish the following operations. If an operation cannot be carried out using one of the diagrams, explain why not.

a. Two sets of symbols are given, a set of all symbols and a set of symbols that have been selected for a moving, cutting, or copying. The output of the operation will be three sets that partition the set of all networks: The set of networks that are connected only to selected symbols, the set of networks that are connected only to unselected symbols, and the set of bridging networks that are connected to both selected and unselected symbols.
b. Given a symbol, find any networks that form more than one connection with the symbol.
c. Given a network, find all symbols that are connected to it.
d. Make a copy of a set of selected symbols. Also copy networks which are connected entirely to symbols which are selected. Ignore bridging networks.
e. Determine all symbols that are connected directly or indirectly to a given set of symbols (transitive closure).

20.3 (6) The dilemma of how to make copies frequently arises in the design of graphics editors. Methods can be broadly categorized into shallow copy and deep copy. Shallow copy methods simply create a reference to the object(s) being copied. Deep copy involves copying the contents of the object(s) being copied, including any objects(s) that it refers to. The choice depends on the application. For each of the following copy operations, discuss the relative merits of the two approaches:

a. Copy of a library object, such as symbols or macros, into a sheet (see Figure 20.3).
b. Copy of sheet from an old project into a new one. After the sheet is copied, it will be edited. Editing changes to the new sheet should not affect the old sheet.
c. Copy of selected *items* from one place on a sheet to another.

EXERCISES

20.1 (8) Relational database operations (insert, delete, update) in OLIE are buffered in memory with a checkout/check-in scheme. When a user wishes to edit a sheet, OLIE locks the sheet and reads all records from the database belonging to the sheet. Each record is converted into an object instance in memory and tagged with a state that is used to determine what should be done to update the database when the user saves editing changes or checks the sheet back in. As the user edits the sheet, OLIE updates the tags. Possible states are *Same*, *Different*, *Not Inserted Yet*, and *Not Deleted Yet*. *Same* indicates that the object instance in memory contains the same data as the record in the database. *Different* means the instance has different data than the corresponding record. *Not Inserted Yet* indicates the instance is due to be inserted into the database. *Not Deleted Yet* means the instance is due for deletion from the database. When objects are copied from the database, they are placed in the *Same* state and remain in that state unless they are updated or deleted. Newly created objects are in the *Not Inserted Yet* state. Objects that need to be deleted are placed in the *Not Deleted Yet* state. Each time an object is accessed, its new state is determined from the operation and the old state. The operations are *insert*(new data), *delete*(object), *update*(object), *save*(set of objects), and *load*(portion of database). Some operations may undo others. For example, an insertion can be elided with a subsequent deletion.

Figure E20.1 is a partially completed state diagram for updating tags. Initial and final states are missing as well as activities, actions, events, and so forth. (Refer to Chapter 5 for a discussion of initial and final states, which in this case are used to allocate and deallocate memory used for objects.) Complete the diagram. Be sure to show what needs to be done for a save operation. Some helpful hints: objects in the *Same* state do not require any action during a *save* operation, since the data in memory is the same as that in the database. Certain operations are illegal on objects in certain states and need not be shown. An *update* operation on an object in the *Not Inserted Yet* state leaves it in the same state, since the database does not yet know about the object. The only legal operation on an object in the *Not Deleted Yet* state is *save*, which requires the corresponding record in the database to be deleted and the object to be terminated.

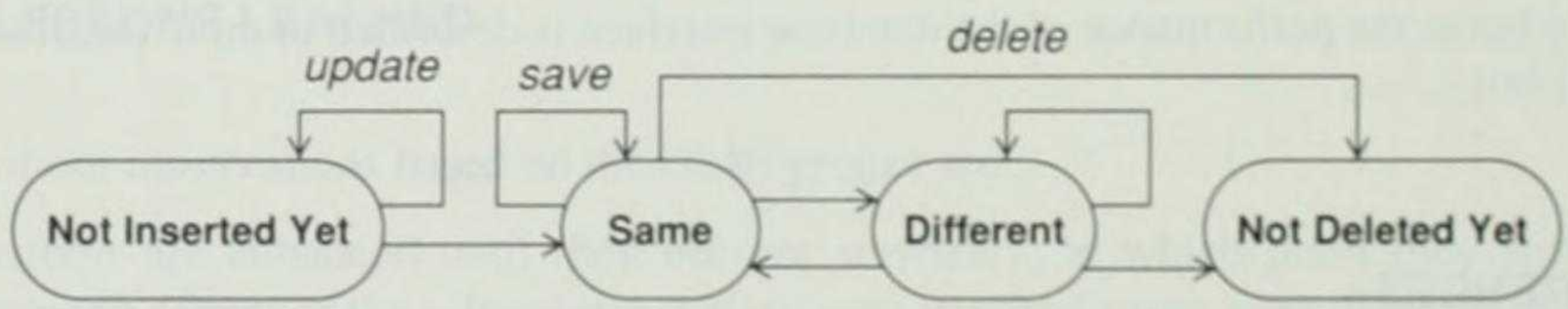


Figure E20.1 Partially completed state diagram for a system to buffer database operations by shadowing records with object instances

20.2 (7) A commonly occurring problem in the design of graphical diagram editors which preserve connectivity is that of detecting bridging networks. For example, if some symbols in a network are moved, some lines in the network may require adjustment to accommodate the relative displacement between symbols that are being moved and those which are not. Three ways of modeling connectivity are shown in Figure E20.2, Figure E20.3, and Figure E20.4. In each case, networks may connect many symbols, and symbols may be connected to many networks. A given network may be connected several times to a given symbol, and more than one network may be connected to a symbol at the same place. In Figure E20.2, the object class, *Port*, is the place on a symbol where a connection is made or the end of a branch of a network where it is connected

to a symbol. Figure E20.3 shows the result of eliminating the *Port* class. A connection is the attachment point of a network to a symbol. Figure E20.4 is a further simplification.

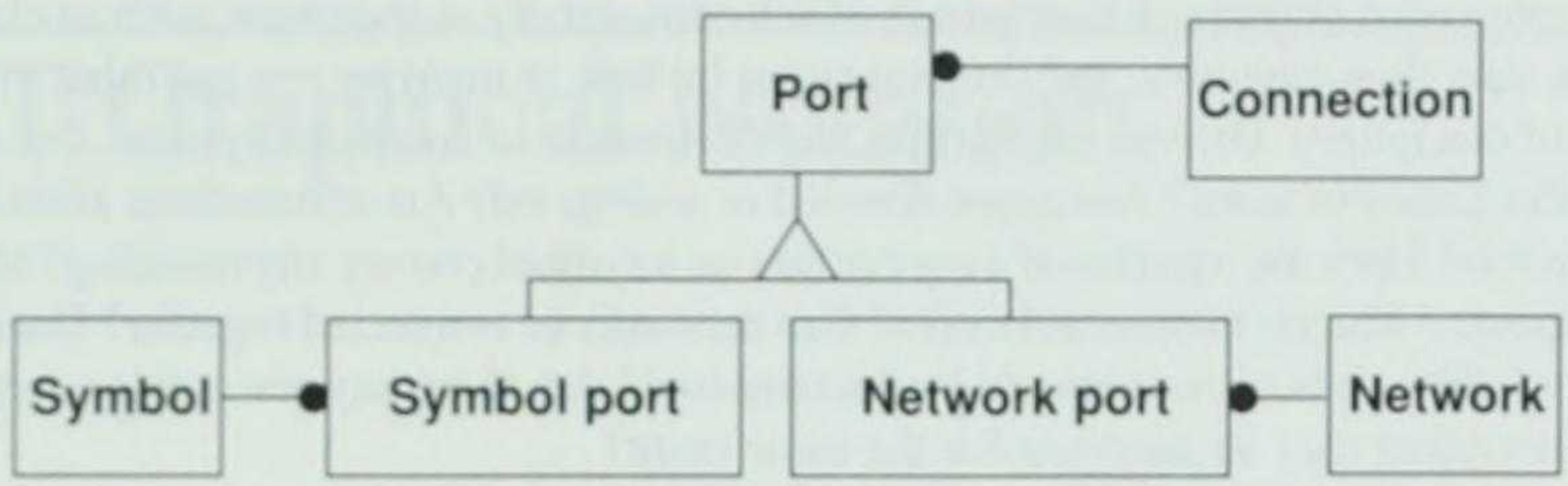


Figure E20.2 Connectivity object diagram

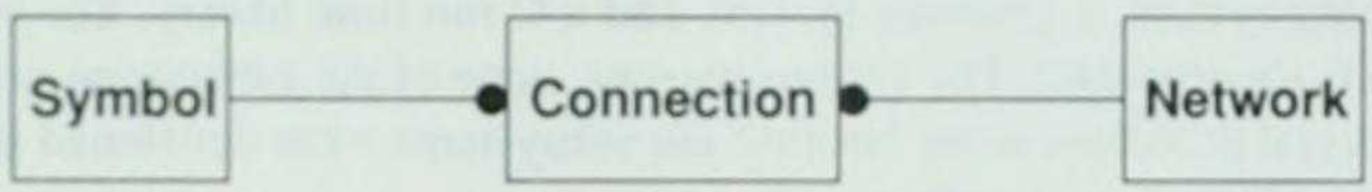


Figure E20.3 Connectivity object diagram

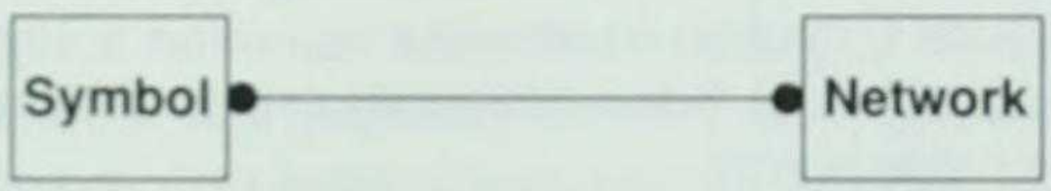


Figure E20.4 Connectivity object diagram

For each diagram, describe how you would accomplish the following operations. If an operation cannot be carried out using one of the diagrams, explain why not.

a. Two sets of symbols are given, a set of all symbols and a set of symbols that have been selected for a moving, cutting, or copying. The output of the operation will be three sets that partition the set of all networks: The set of networks that are connected only to selected symbols, the set of networks that are connected only to unselected symbols, and the set of bridging networks that are connected to both selected and unselected symbols.
b. Given a symbol, find any networks that form more than one connection with the symbol.
c. Given a network, find all symbols that are connected to it.
d. Make a copy of a set of selected symbols. Also copy networks which are connected entirely to symbols which are selected. Ignore bridging networks.
e. Determine all symbols that are connected directly or indirectly to a given set of symbols (transitive closure).

20.3 (6) The dilemma of how to make copies frequently arises in the design of graphics editors. Methods can be broadly categorized into shallow copy and deep copy. Shallow copy methods simply create a reference to the object(s) being copied. Deep copy involves copying the contents of the object(s) being copied, including any objects(s) that it refers to. The choice depends on the application. For each of the following copy operations, discuss the relative merits of the two approaches:

a. Copy of a library object, such as symbols or macros, into a sheet (see Figure 20.3).
b. Copy of sheet from an old project into a new one. After the sheet is copied, it will be edited. Editing changes to the new sheet should not affect the old sheet.
c. Copy of selected *items* from one place on a sheet to another.

20.4 (5) Modify the state trees in Figure 20.16 and Figure 20.17 so that the pick, put, move, and copy operations can be performed at any time, including in the middle of adding geometry to a sheet.

20.5 (6) Prepare a list of types of diagrams in which connectivity is important, such as electrical schematics, data flow diagrams, and decision trees. Include as many as you can think of from a wide range of disciplines. Discuss similarities and differences in connectivity rules. For example, are networks binary or n-ary? Are edges directed or undirected? Are connections allowed anywhere on symbols? Does the location of a connection on a symbol convey any meaning? May networks be connected directly to other networks? Can networks be connected together? Does it make any sense to allow ends of networks to be unconnected? Are there any restrictions concerning how many networks may be connected at the same point?

20.6 (7) The system described in this chapter was built on top of 6 existing software subsystems: a relational database, an object-oriented language with a container class library, a windowing system, a menuing system, a graphics system, and a C run-time library. The object-oriented language was a C preprocessor. For various reasons, none of the subsystems could be modified in any way. Several problems arose because the subsystems were developed independently. Discuss workarounds for each problem:

a. Name clashes. During software development, the linker frequently complained about multiply defined symbols caused by the same name being used in two or more of the subsystem libraries, including the C run-time library.
b. Preprocessor incompatibilities. Some subsystems used preprocessor constructs that were illegal for the other preprocessors.
c. Memory allocation. Each subsystem used dynamic memory allocation. There were several different strategies used. Some of them did not check to see if memory was actually allocated successfully, resulting in memory fault errors.
d. Interrupt processing. Some subsystems performed input/output synchronously, others asynchronously. Two subsystems intercepted keyboard interrupts.

20.7 (7) Repeat the previous exercise, this time assuming that you have a free hand to change any of the subsystems.

Appendix A

OMT Graphical Notation

All of the graphical notations that are used to build the object model, dynamic model, and functional model have been summarized on the inside covers of the book. Those four pages should be useful as a quick reference while constructing or reading diagrams. We caution you not to simply give those four pages to a novice and expect that person to understand them. To understand the concepts being represented, refer to the chapters of Part 1. To learn how to apply the notation and concepts within the software development life cycle, the chapters in Part 2 should be consulted. If you find you need to review any of the concepts represented graphically here, use the index to locate the explanation in the book.

With the exception of the label for each construct and a few descriptive comments, all of the diagram elements, text names, and punctuation symbols shown are intended to be part of the notation itself. The names in these diagrams (such as *Class*, *attribute-1*, *operation*, and *event-2*) have been chosen to indicate what kind of element they are examples of. You may wish to modify the syntax of names and the declarations of attributes and signatures to make them consistent with the syntax of your implementation language.

Most of the items shown are optional, especially during early stages of modeling. Even in design, it is unwise to overspecify by including names and notations that are not needed. For example, when an association is labeled by its role names, it is usually not necessary to give the association itself a name. We have not indicated which elements are optional because we wanted to show only the actual OMT notation wherever possible, without obscuring it with an additional meta-notation.

Similarly, there are a few constructs that you may never need to use. We have placed the more important constructs toward the top of each page. In most of our work we use only about one third of the notation, but the additional constructs are occasionally necessary.

Appendix B

Glossary

abstract class a class that cannot have direct instances but whose descendants can have instances.

abstract operation an operation defined but not implemented by an abstract class. The operation must be implemented by all concrete descendent classes.

abstraction a mental facility that permits humans to view real-world problems with varying degrees of detail depending on the current context of the problem.

action (in dynamic modeling) an instantaneous operation. Actions are associated with events and are usually formal in nature.

activity (in dynamic modeling) an operation that takes time to complete. Activities are associated with states and represent real-world accomplishments.

actor object (in a DFD) an active object that drives the data flow graph by producing or consuming values.

aggregation a special form of association, between a whole and its parts, in which the whole is composed of the parts.

analysis a stage in the development cycle in which a real-world problem is examined to understand its requirements without planning the implementation.

ancestor class a class that is a direct or indirect superclass of a given class.

architecture the overall structure of a system, including its partitioning into subsystems and their allocation to tasks and processors.

assertion a statement about some condition or relationship that can be either true or false at the time that it is tested. (Contrast with *constraint* and *invariant*.)

association a relationship among instances of two or more classes describing a group of links with common structure and common semantics.

attribute a named property of a class describing a data value held by each object of the class.

automatic transition (in dynamic modeling) an unlabeled transition that automatically fires when the activity associated with the source state is completed.

batch transformation a sequential input-to-output transformation, in which inputs are supplied at the start and the goal is to compute an answer; there is no ongoing interaction with the outside world. (Contrast with *continuous transformation*.)

cached data data that is redundant because it can be derived from other data.

call-by-reference a language mechanism that passes arguments to a procedure by passing the address of each argument rather than its value. (Contrast with *call-by-value*.)

call-by-value a language mechanism that passes arguments to a procedure by passing a copy of the data values. If an argument is modified, the new value will not take effect outside of the subroutine that modifies it. (Contrast with *call-by-reference*.)

candidate key a minimal set of attributes that uniquely identifies an instance or link.

class a description of a group of objects with similar properties, common behavior, common relationships, and common semantics.

class attribute an attribute whose value is common to a class of objects rather than a value peculiar to each instance.

class descriptor an object representing a class itself, containing a list of attributes and methods as well as the values of any class attributes. Class descriptors are implemented in some, but not all, languages. A class descriptor is an instance of a *metaclass*.

class diagram an object diagram that describes classes as a schema, pattern, or template for many possible instances of data. (Contrast with *instance diagram*.)

class operation an operation on a class, rather than on instances of the class. An instance creation operation is a common example.

class variable (in Smalltalk) an attribute of a class descriptor object; a *class attribute*.

classification a grouping of objects with the same data structure and behavior.

client a system component that calls upon the services provided by another component. The component providing the service is a *supplier*.

coherence a property of an entity, such as a class, an operation, or a module, such that it is organized on a consistent plan and all its parts fit together toward a common goal.

conceptual schema (in a relational database) a design from the perspective of an entire enterprise.

concrete class a class that can have direct instances.

concurrent two or more tasks, activities, or events whose execution may overlap in time.

condition (in dynamic modeling) a Boolean function of object values valid over an interval of time.

constraint a functional relationship between objects, classes, attributes, links, and associations; a statement about some condition or relationship that must be maintained as true. (Contrast with *assertion* and *invariant*.)

constructor (in C++) an operation that initializes a newly created instance of a class. (Contrast with *destructor*.)

container class a class of *container objects*. Examples include sets, arrays, dictionaries, and associations.

container object an object that stores a collection of other objects and provides various operations to access or iterate over its contents.

continuous transformation a system in which the outputs actively depend on changing inputs and must be periodically updated. (Contrast with *batch transformation*.)

contour (in a state diagram) picture of a state that can contain substates. The contour for a state totally encloses the contours of its substates.

control the aspect of a system that describes the sequences of operations that occur in response to stimuli.

control flow (in a DFD) a Boolean value that affects whether a process is executed.

database a permanent, self-descriptive repository of data that is managed by a DBMS.

database management system (DBMS) a computer program that manages a permanent, self-descriptive repository of data.

data dictionary a textual description of each class, its associations, attributes, and operations.

data flow (in a DFD) the connection between the output of one object or process and the input to another.

data flow diagram a graphical representation of the functional model, showing dependencies between values and the computation of output values from input values without regard for when or if the functions are executed.

data store (in a DFD) a passive object that stores data for later access.

DBMS (acronym) *database management system*.

delegation an implementation mechanism in which an object, responding to an operation on itself, forwards the operation to another object; (in object-oriented languages) a mechanism in which methods may be attached directly to instances and where the method resolution is performed by searching a chain of instance pointers, rather than by searching a class hierarchy.

derived association an association that is defined in terms of other associations.

derived attribute an attribute that is computed from other attributes.

descendent class a class that is a direct or indirect subclass of a given class.

destructor (in C++) an operation that cleans up an existing instance of a class that is no longer needed. (Contrast with *constructor*.)

DFD (acronym) *data flow diagram*.

dictionary a class of container object that maps a value of one type into a value of another type, possibly the same type; a lookup table. Mathematically, a discrete function from a domain to a range.

direct instance an object that is an instance of a class but not an instance of any subclass of the class.

discriminator an attribute of enumeration type that indicates which property of a class is being abstracted by a particular generalization.

domain (in a database) the set of legal values for an attribute in a database; (mathematics) the set over which a function or relation is defined.

dynamic binding a form of method resolution that associates a method with an operation at run time, depending on the class of one or more target objects.

dynamic model a description of aspects of a system concerned with control, including time, sequencing of operations, and interaction of objects.

dynamic simulation a system that models or tracks objects in the real world.

encapsulation a modeling and implementation technique that separates the external aspects of an object from the internal, implementation details of the object (also called *information hiding*).

entity-relationship (ER) diagram a graphical representation that shows entities and the relationships between them.

entity integrity (in a relational database) a property of a database such that each table has exactly one primary key.

event (in dynamic modeling) something that happens instantaneously at a point in time.

event attribute data values conveyed by an event from one object to another.

event trace a diagram that shows the sender and receiver of events and the sequence of events.

extensibility a property of software such that new kinds of objects or functionality can be added to it with little or no modification to existing code.

extension (in generalization) the addition of new features by a subclass.

external schema (in a relational database) a design from the perspective of a single application.

feature either an attribute or an operation of a class.

fire (in dynamic modeling) to cause a transition to occur.

first normal form (in a relational database) a property of a schema such that an attribute cannot contain a repeating group.

fixed aggregate an aggregate with a predefined number and types of components. (Contrast with *variable aggregate.*)

foreign key (in a relational database) a primary key of one table that is embedded in another (or the same) table.

functional model description of aspects of a system that transform values using functions, mappings, constraints, and functional dependencies.

garbage collection a language mechanism for automatically deallocating data structures which can no longer be accessed and are therefore not needed.

generalization the relationship between a class and one or more refined or specialized versions of it.

generics (see *parameterized classes*)

guard condition (in dynamic modeling) a Boolean expression that must be true in order for a transition to occur.

guarded transition (in dynamic modeling) a transition that fires only if a guard condition is true.

hybrid object-oriented language a language that has both object-oriented types (classes) and non-object-oriented types (primitive types).

identity a distinguishing characteristic of an object that denotes a separate existence of the object even though the object may have the same data values as another object.

implementation a stage in the development cycle in which a design is realized in an executable form, such as a programming language or hardware.

implementation method (style) a method that implements specific computations on fully-specified arguments, but does not make context-dependent decisions. (Contrast with *policy method.*)

index a data structure that maps one or more attribute values into the objects or database table rows that hold the values, usually for optimization purposes.

indirect instance an object that is an instance of a class and also an instance of a subclass of the class.

information hiding (see *encapsulation*)

inherently concurrent two objects that can receive events at the same time without interacting.

inheritance an object-oriented mechanism that permits classes to share attributes and operations based on a relationship, usually generalization.

instance an object described by a class.

instance diagram an object diagram that describes how a particular set of object instances relate to each other. (Contrast with *class diagram.*)

instance variable (in Smalltalk) an attribute.

instantiation the process of creating instances from classes.

integrity (in a relational database: see *entity integrity* and *referential integrity*)

interactive interface a system that is dominated by interactions between the system and agents, such as humans, devices, or other programs.

internal schema (in a relational database) the actual code required to implement the conceptual schema.

invariant a statement about some condition or relationship that must always be true. (Contrast with *assertion* and *constraint*.)

iterator a language construct that controls iteration over a range of values or a collection of objects.

join class (in multiple inheritance) a class with more than one superclass.

lambda transition (see *automatic transition*)

layer a subsystem that provides multiple services, all of which are at the same level of abstraction, built on subsystems at a lower level of abstraction. (Contrast with *partition*.)

leaf class a class with no subclasses. It must be a concrete class.

link an instance of an association; a physical or conceptual connection between objects.

link attribute a named data value held by each link in an association.

message (in Smalltalk) invocation of an operation on an object, comprising an operation name and a list of argument values.

metaclass a class describing other classes.

metadata data that describes other data.

method the implementation of an operation for a specific class.

method caching an optimization of method searching in which the address of a method is found the first time an operation is applied to an object of a class and then stored in a table attached to the class.

method resolution (in a programming language) the process of matching an operation on an object to the method appropriate to the object's class.

methodology (in software engineering) a process for the organized production of software using a collection of predefined techniques and notational conventions.

model an abstraction of something for the purpose of understanding it before building it.

module a coherent subset of a system containing a tightly bound group of classes and their relationships.

multiple inheritance a type of inheritance that permits a class to have more than one superclass and to inherit features from all ancestors.

multiplicity the number of instances of one class that may relate to a single instance of an associated class.

normal form (in a relational database) a set of rules that reduce consistency problems from table updates.

null (in a relational database) a special value that denotes an attribute value which is unknown or not applicable for a given row.

object a concept, abstraction, or thing with crisp boundaries and meanings for the problem at hand; an instance of a class.

object design a stage of the development cycle during which the implementation of each class, association, attribute, and operation is determined.

object diagram a graphical representation of the object model showing relationships, attributes, and operations. (See *instance diagram* and *class diagram*, which are the usual special cases. However, models with metadata do not permit such a dichotomy.)

object model a description of the structure of the objects in a system including their identity, relationships to other objects, attributes, and operations.

Object Modeling Technique an object-oriented development methodology that uses object, dynamic, and functional models throughout the development life cycle. Abbreviated *OMT*.

object-oriented a software development strategy that organizes software as a collection of objects that contain both data structure and behavior. Abbreviated *OO*.

object-oriented development a software development technique that uses objects as a basis for analysis, design, and implementation.

object-oriented programming language a language that supports objects (combining identity, data, and operations), method resolution, and inheritance.

OMT (acronym) *Object Modeling Technique*.

OO (acronym) *object-oriented*.

OO-DBMS (acronym) *object-oriented database management system*.

OOPL (acronym) *object-oriented programming language*.

operation a function or transformation that may be applied to objects in a class.

origin class the topmost class that defines an attribute or operation.

overloading (in a language) binding the same name to multiple operations whose signatures differ in number or types of arguments. A call to an overloaded operation is resolved at compile time based on the types of the calling arguments.

override to define a method for an operation that replaces an inherited method for the same operation.

package (in a programming language such as Ada) a syntactic block with a well-defined interface used to control visibility of its contents.

parameterized classes a template for creating real classes that may differ in well-defined ways as specified by parameters at the time of creation. The parameters are often data types or classes, but may include other attributes, such as the size of a collection. (Also called *generic classes*.)

partition a subsystem that provides a particular kind of service. A partition may itself be built from lower level subsystems. (Contrast with *layer*.)

peer two or more system components that are mutually interdependent for services. (Contrast with *client* and *supplier*.)

persistent data data that outlasts the execution of a particular program.

pointer an attribute in one object that contains an explicit reference to another object. Pointers are implementation constructs corresponding to associations.

policy the making of context-dependent decisions.

policy method (style) a method that makes context-dependent decisions, switches control among other methods, combines and parameterizes calls to lower-level methods, and checks for status and error, but which calls on other methods for detailed computations. (Contrast with *implementation method*.)

polymorphism takes on many forms; the property that an operation may behave differently on different classes.

postcondition a condition that the operation itself agrees to achieve.

precondition a condition that the caller of an operation agrees to satisfy.

primary key (in a relational database) a combination of one or more attributes whose value unambiguously locates each row in a table.

private (referring to an attribute or operation of a class) accessible by methods of the current class only. (Contrast with *public*.)

process (in a DFD) something that transforms data values.

programming-in-the-large the creation of large, complex programs with teams of programmers.

propagation the automatic application of an operation to selected objects in a network when the operation is applied to some starting object in the network.

protected (referring to an attribute or operation of a class in C++) accessible by methods of any descendent of the current class. (Contrast with *public* and *private*.)

protocol specification of the semantics of an operation, including its signature, a description of the function performed by the operation, and any preconditions or postconditions.

public (referring to an attribute or operation of a class) accessible by methods of any class. (Contrast with *private*.)

qualified association an association that relates two classes and a qualifier; a binary association in which the first part is a composite comprising a class and a qualifier, and the second part is a class.

qualifier an attribute of an object that distinguishes among the set of objects at the "many" end of an association.

query operation an operation that returns or computes a value without modifying any objects.

range (mathematics) the set over which the results of a function are defined.

real-time system an interactive system for which time constraints on actions are particularly tight or in which the slightest timing failure cannot be tolerated.

recursive aggregate an aggregate that contains, directly or indirectly, an instance of the same kind of aggregate.

referential integrity (in a relational database) a property of a database such that each foreign key is consistent with its corresponding primary key.

reflective a property of a system such that it can examine its own structure dynamically and reason about its own state.

relation (mathematics) a set of tuples, usually from a list of specified domains; (relational database) a table in the database.

relational database a database managed by a relational DBMS.

relational DBMS a computer program that provides an abstraction of relational tables to the user. It must provide three kinds of functionality: present data in the form of tables, provide operators for manipulating the tables, and support integrity rules on tables.

restriction (in generalization) a constraint that a subclass places on the value of an attribute contained in a superclass.

robust a property of software such that it does not fail catastrophically when some of its design assumptions are violated.

role one end of an association.

role name a name that uniquely identifies one end of an association.

scenario (in dynamic modeling) a sequence of events that occur during one particular execution of a system.

schema the structure or template of the data in a database.

second normal form (in a relational database) property of a schema such that it is in first normal form and each row has a primary key.

self (in Smalltalk) the default name of the target object of a method. (Corresponds to *this* in C++ and *Current* in Eiffel.)

service a group of related functions (or operations) that work together to provide a functional capability.

sheet (in an object model) the mechanism for breaking large object models into a series of pages.

signature (for an attribute) the attribute's type; (for an operation) the number and types of its arguments and the type of its result.

single inheritance a type of inheritance in which a class may have only a single superclass.

specialization the creation of subclasses from a superclass by refining the superclass.

SQL a standard language for interacting with relational DBMS.

state the values of the attributes and links of an object at a particular time.

state diagram a directed graph in which nodes represent system states and arcs represent transitions between states.

strong typing a property of a programming language such that the type of each variable must be declared. (Contrast with *weak typing*.)

structure chart a tree-shaped graph in which the nodes represent procedures and the arcs represent caller/callee relationships.

subclass (in generalization) a refined version of another class, the superclass. An instance of the subclass is also an instance of the superclass.

subsystem a major component of a system organized around some coherent theme. A system may be divided into subsystems using either *partitions* or *layers*.

superclass (in generalization) a more abstract version of another class, the subclass.

supplier a system component that provides a service to another component. The component requesting the service is a *client*.

system an organized collection of components that interact.

system design the first stage of design, during which high-level decisions are made about the overall structure of the system, its architecture, and the strategies used to implement the system.

system development life cycle the process of creating a hardware/software system from its conception, through analysis, design, implementation, testing, and maintenance.

table (in a relational database) an organization of data that has a specific number of columns and an arbitrary number of rows. Often called a relation.

ternary association an association between three classes.

third normal form (in a relational database) property of a schema such that it is in second normal form and each nonprimary-key field depends on the primary key.

this (in C++) the default name of the target object of a method. (Corresponds to *self* in Smalltalk and *Current* in Eiffel.)

thread of control a single path of execution through a program, a dynamic model, or some other representation of control flow.

transaction manager a database system whose main function is to store and access information.

transition a change of state caused by an event.

triggering (see *propagation*)

tuple an ordered list of data values.

type a set of objects or values with similar behavior, usually expressed by the operations defined on the type, without regard for the potential implementation of the type. A type is a semantic property.

variable aggregate an aggregate with a finite number of levels but a varying number of parts. (Contrast with *fixed aggregate*.)

view (in a relational database) a virtual table that is derived from one or more underlying tables.

virtual something that has conceptual but not actual existence; (in C++) an operation that can be overridden by a descendent class.

weak typing a property of a programming language such that the type of each variable need not be declared. (Contrast with *strong typing*.)

wrapper a class or operation that encapsulates a call to library routines or some other code that is being reused.

Answers to Selected Exercises

Criteria for selecting exercises to be answered include: exercises with short answers in the core chapters; exercises that extend chapters by introducing new material; key exercises in a series of questions; answers that clarify subtle or difficult points; and prototypes for real problems that are commonly encountered. Most exercises have multiple correct answers so the answers given here should be used as a guide and not as a test of correctness.

1.5b. Criminal investigations use combinations of photographs, fingerprinting, bloodtyping, DNA analysis, and dental records to identify people, living and/or deceased, who are involved in, or the subject of, a criminal investigation.

d. Telephone numbers are adequate for identifying almost any telephone in the world. In general a telephone number consists of a country code plus a province, city, or area code, plus a local number plus an optional extension number. Businesses may have their own telephone systems with other conventions. Depending on the relative location of the telephone that you are calling, parts of the number may be implied and can be left out, but extra access digits may be required to call outside the local region. In North America local calls require 7 digits. Long distance calls use an access digit (0 or 1) + area code (3 digits) + local number (7 digits). Dialing Paris requires an access code (011) + country code (33) + city code (1) + local number (8 digits). The access code is not part of the identifier. Dialing is an example of navigation, not identity.

g. One way that employees are given restricted, after-hours access to a company is through the use of a special, electronically-readable card. Of course, if an employee loses a card and does not report it, someone who finds it could use it for unauthorized entry. Another approach is a picture ID which requires inspection by a guard.

1.8a. These are all devices that enhance vision in some way. With the exception of the scanning electron microscope, all the devices listed work by reflecting or refracting light. Eyeglasses and binoculars are designed for use with two eyes, the rest of the objects on the list are designed for use with one eye. Telescopes, bomb sights, and binoculars are used to view things far away. A microscope is used to magnify something that is very small. Eyeglasses may enlarge or reduce, depending on whether the prescription is for a near-sighted or a far-sighted person. Some other classes that could be included in this list are optical microscopes, cameras, and magnifying glasses.

b. Pipes, check valves, faucets, filters, and pressure gauges are all plumbing supplies with certain temperature and pressure ratings. Compatibility with various types of fluids is also a consideration. Check valves and faucets may be used to control flow. With the exception of the pressure gauge, all of the items listed have two ends and have a pressure-flow characteristic for a given fluid. All of the items are passive. Some other classes include pumps, tanks, and connectors.

f. Square root, exponential, sine, and cosine are all functions of a single variable. Both real and complex definitions are commonly used. As real functions, each maps a real input to a real output. As a real function, square root is defined only for nonnegative numbers. As complex functions, each maps a complex input to a complex output.

2.3a. For a Trans-Atlantic cable, resistance to saltwater is the main consideration. The cable must lie unmaintained at the bottom of the ocean for a long time. The ratio of strength/weight is important to avoid breakage while the cable is being installed. Cost is an important economic factor.

c. Weight is important for wire that is to be used in the electrical system of an airplane, because it affects the total weight of the plane. Toughness of the insulation is important to resist chafing due to vibration. Resistance of the insulation to fire is also important to avoid starting or feeding electrical fires in flight.

3.2 A class diagram is shown in Figure A3.2. The smallest number of points required to construct a polygon is three. One way to express the fact that points are in a sequence is to indicate that the association is ordered. The multiplicity of the association depends on how points are identified. If a point is identified by its location, then points are shared and the association is many-to-many. On the other hand, if each point belongs to exactly one polygon, as shown in the diagram, then several points may have the same coordinates. The difference between the two situations is clarified in the next answer.

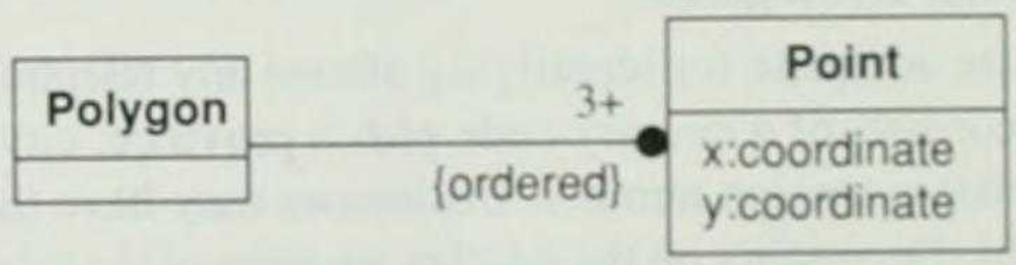


Figure A3.2 Class diagram for polygon and points

3.3a. An instance diagram for two triangles with a common side in which a point belongs to exactly one polygon is shown in Figure A3.3a.

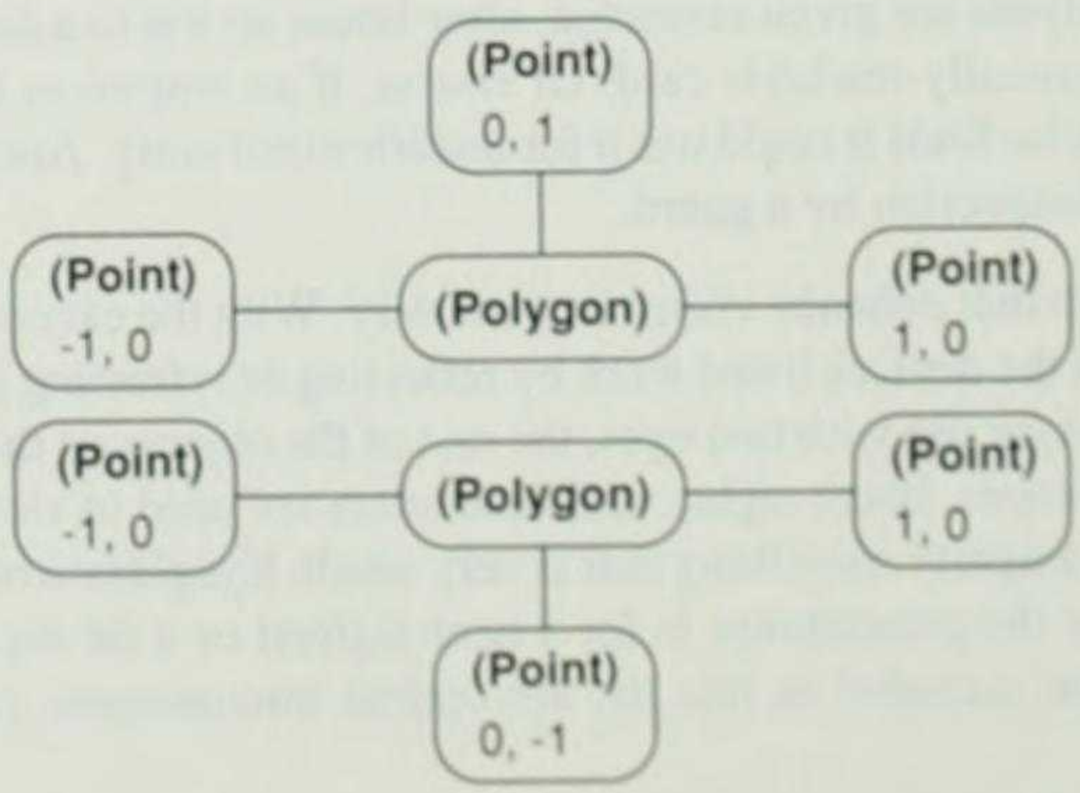


Figure A3.3a Instance diagram where each point belongs to exactly one polygon

b. An instance diagram for two triangles with a common side in which points may be shared is shown in Figure A3.3b.

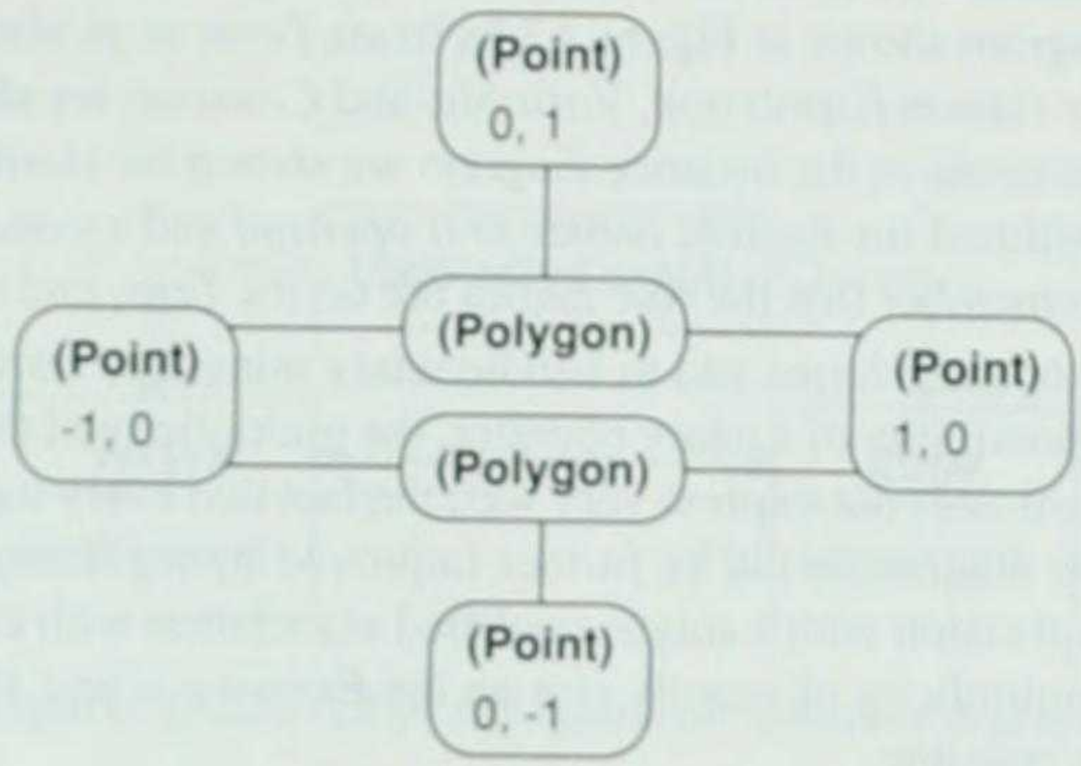


Figure A3.3b Instance diagram where each point can belong to multiple polygons

3.8a. An instance diagram for the expression *(X+Y/2)/(X/3+Y)* is shown in Figure A3.8a. The object diagram in the exercise is actually a metamodel for binary expressions. Parentheses are required for an infix representation but are not needed in the metamodel. There are other representations, such as postfix, in which parentheses also are not needed. For example, the same expression becomes *X Y 2 / + X 3 / Y + /* in a postfix representation.

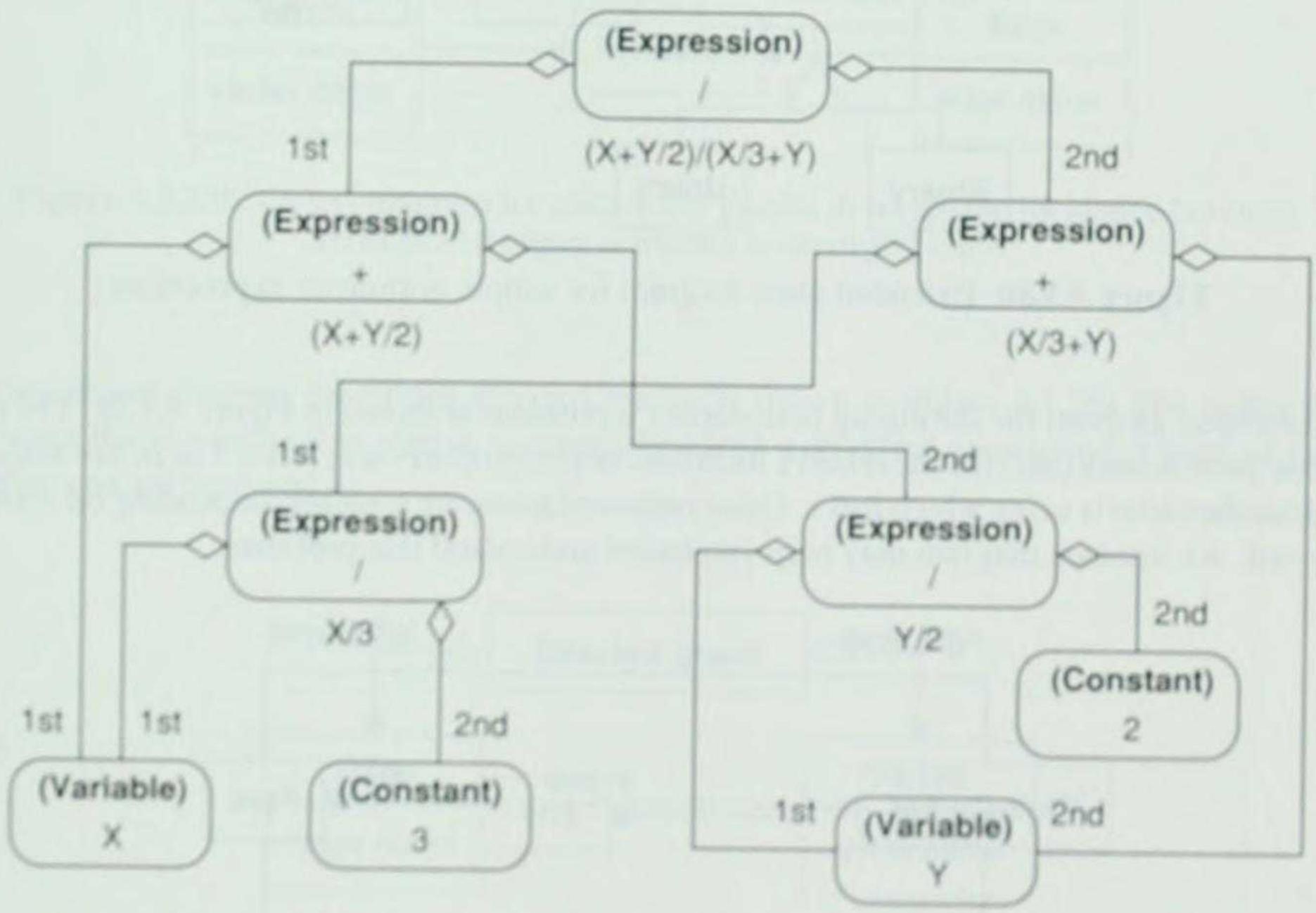


Figure A3.8a Instance diagram for the expression (X+Y/2)/(X/3+Y)

The diagram in this exercise contains recursion. Expressions are formed from terms which themselves may be expressions. Very complex expressions may be represented leading to complex instance diagrams.

Figure E3.6 indicates that terms may be shared by expressions. The situation is analogous to sharing of points discussed in exercise 3.3. If the direction of the links in the corresponding instance diagrams are taken into account, then instance diagrams are directed acyclic graphs.

The instance diagram shown in Figure A3.8a treats *Term* as an abstract class, since only direct instances of the classes *Expression*, *Variable*, and *Constant* are shown.

The partial expressions in the instance diagram are shown for clarity. To save space, *1st* and *2nd* have been substituted for the role names *first operand* and *second operand*. In reading the instance diagram, remember that the role names are on the *Term* end of the associations.

b. The extensions to not share terms and to handle unary minus are shown in Figure A3.8b. Note that because of the possibility of a unary operator, the multiplicity of the second operand is zero-one. Also the diagram does not express very well the fact that every term must belong to exactly one expression. The diagram could be further improved by replacing the two associations between *Term* and *Expression* with a single, qualified association with the qualifier *operand*. This would result in a multiplicity of exactly one on the *Expression* end of the association. Several other variations are possible.

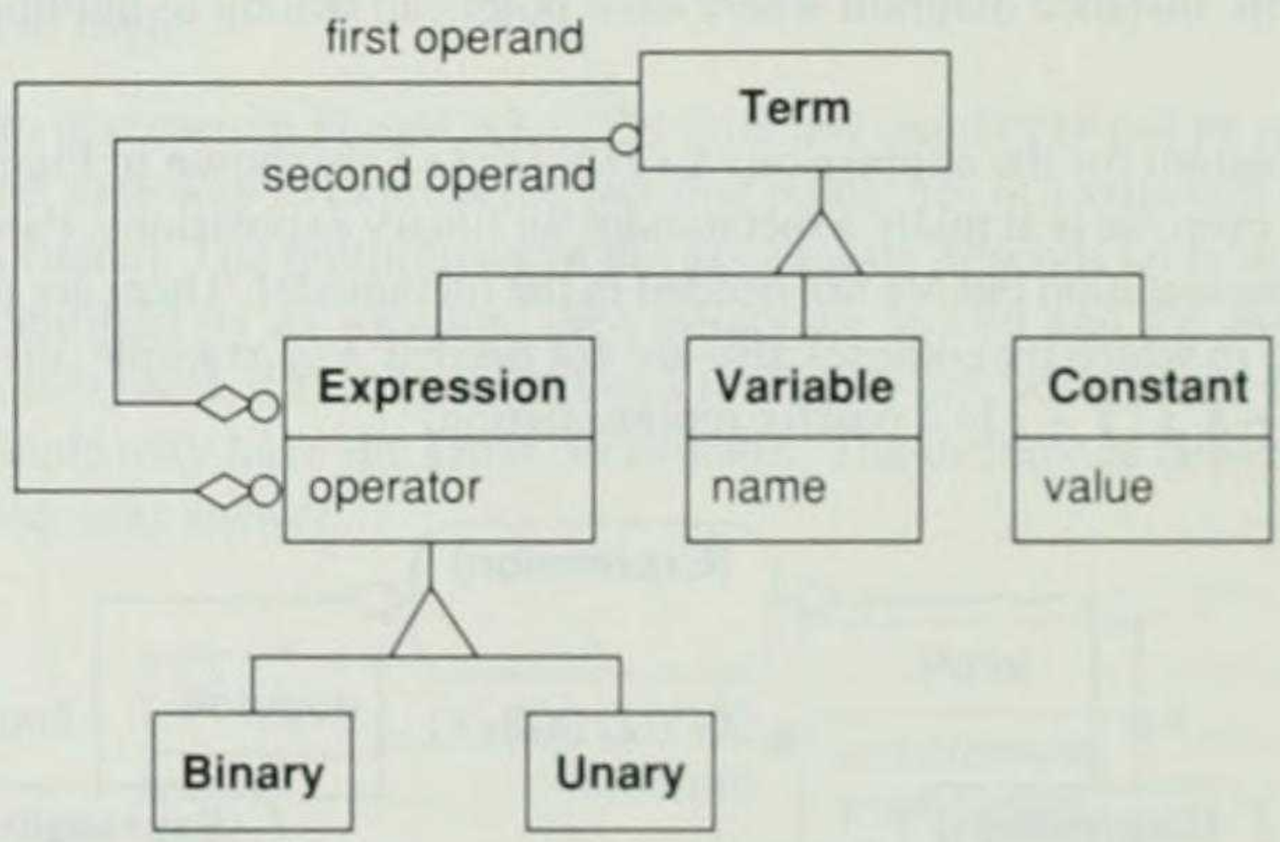


Figure A3.8b Extended class diagram for simple arithmetic expressions

3.22 An object diagram for the dining philosopher's problem is shown in Figure A3.22. The one-to-one associations describe the relative locations of philosophers and forks. The *In use* association describes who is using which forks. Other representations are possible, depending on your viewpoint. An instance diagram may help you better understand this problem.

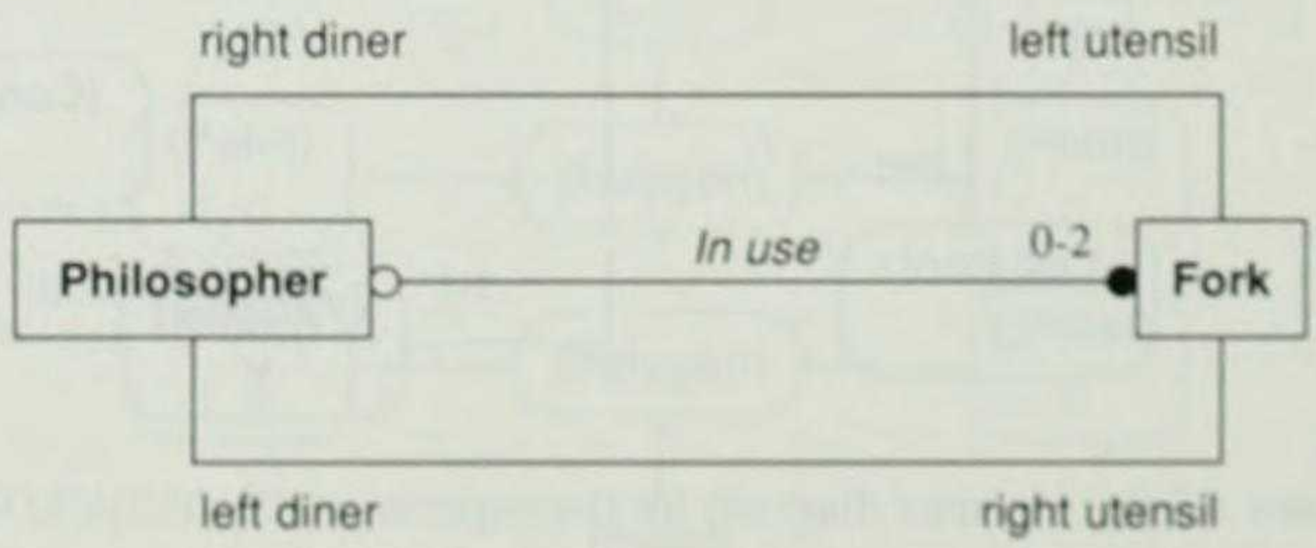


Figure A3.22 Object diagram for the dining philosopher problem

3.23 This is an important exercise, because graphs occur in many applications. Several variations are possible, depending on your viewpoint. Figure A3.23a accurately represents undirected graphs as described in the exercise. Although not quite as accurate, your answer could omit the class *Undirected graph*.

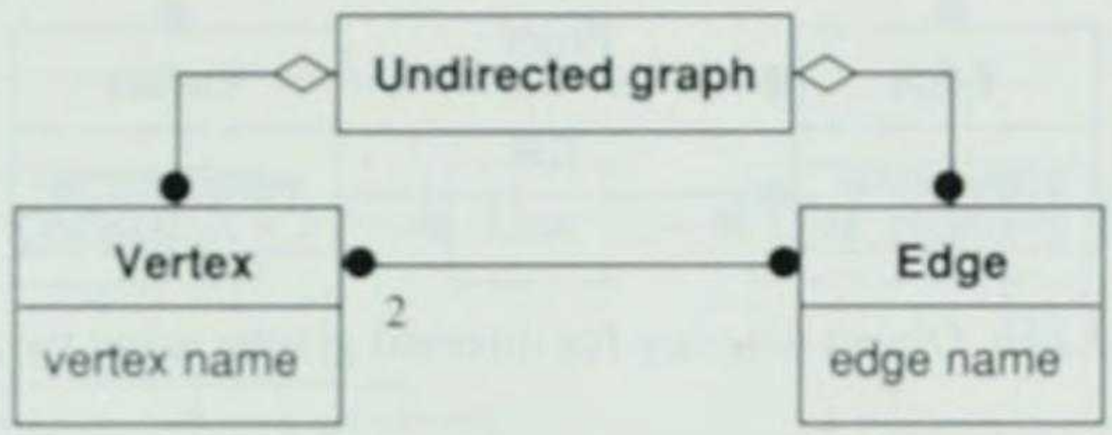


Figure A3.23a Object diagram for undirected graphs

We have found it useful for some graph related queries to elevate the association between vertices and edges to the status of an object class as shown in Figure A3.23b.

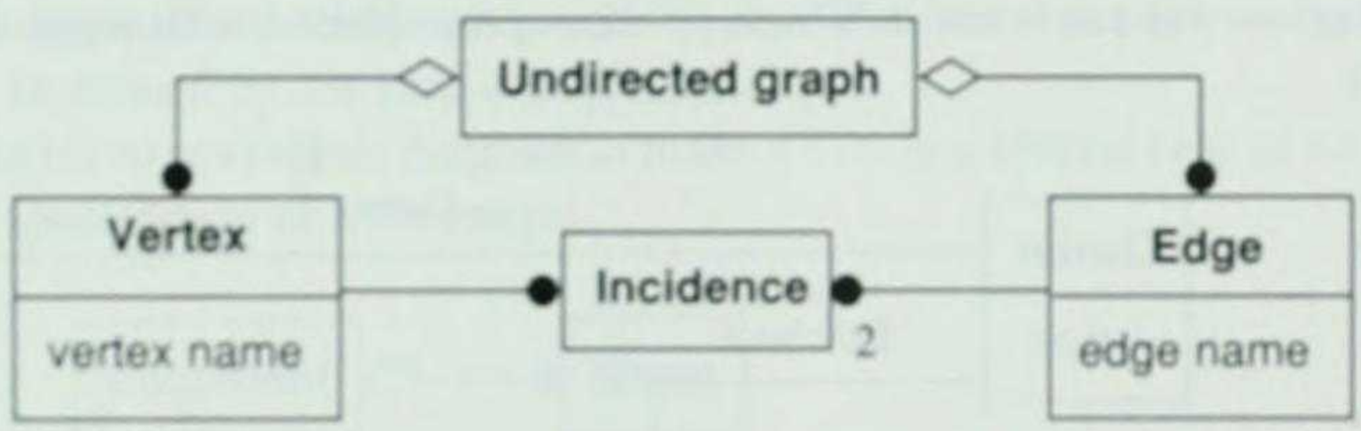


Figure A3.23b Object diagram for undirected graphs in which the incidence between vertices and edges is treated as an object class

3.26 One object diagram describing directed graphs is shown in Figure A3.26a.The distinction between the two ends of an edge is accomplished with a qualified association. Values of the qualifier *end* are *from* and *to*.

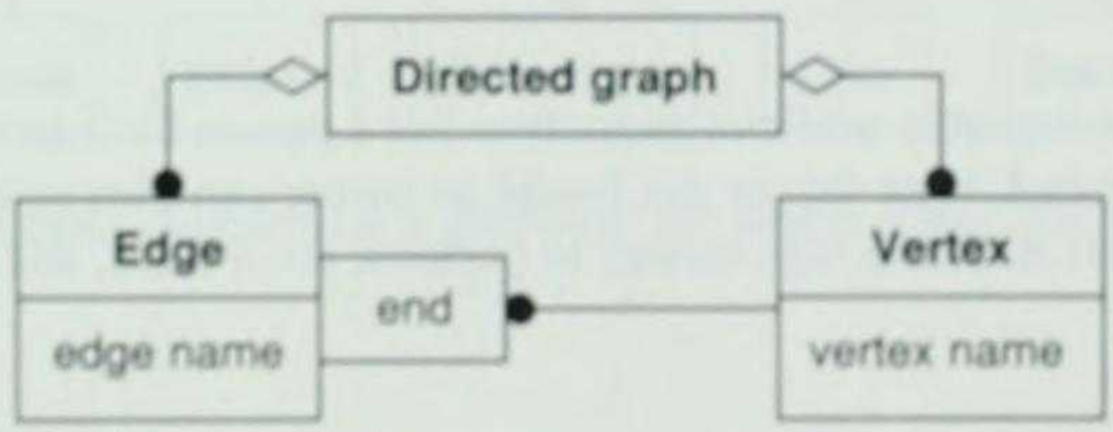


Figure A3.26a Object diagram for directed graphs using a qualified association

Figure A3.26b shows another representation of directed graphs. The distinction between the two ends of an edge is accomplished with separate associations for the two ends of an edge.

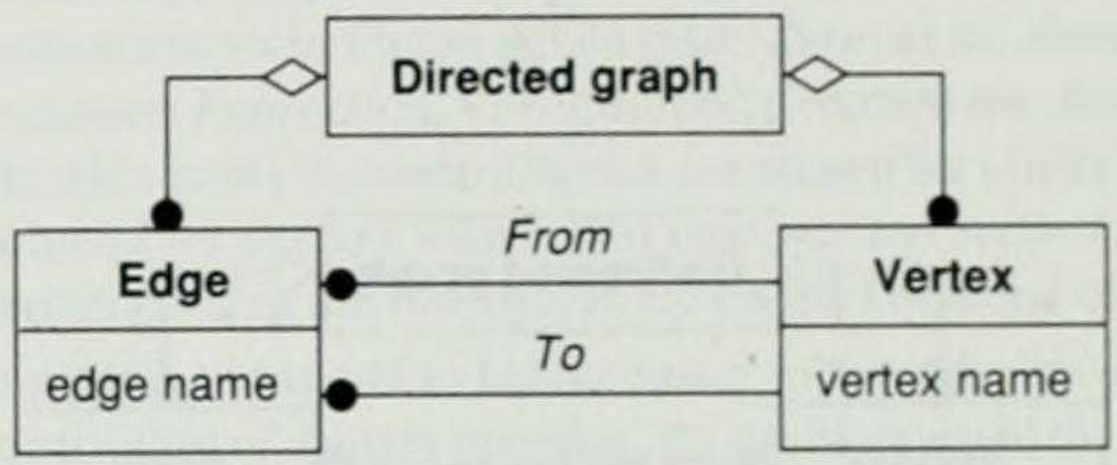


Figure A3.26b Object diagram for directed graphs using two associations

The advantage of the qualified association is that only one association must be queried to find one or both vertices that a given edge is connected to. If the qualifier is not specified, both vertices can be found. By specifying *from* or *to* for the *end* qualifier, the vertex that is connected to an edge at the given *end* can be found.

The advantage of using two separate associations is that the need to manage enumerated values for the qualifier *end* is eliminated.

3.28 An object diagram for car loans in which pointers are replaced with relationships is shown in Figure A3.28.

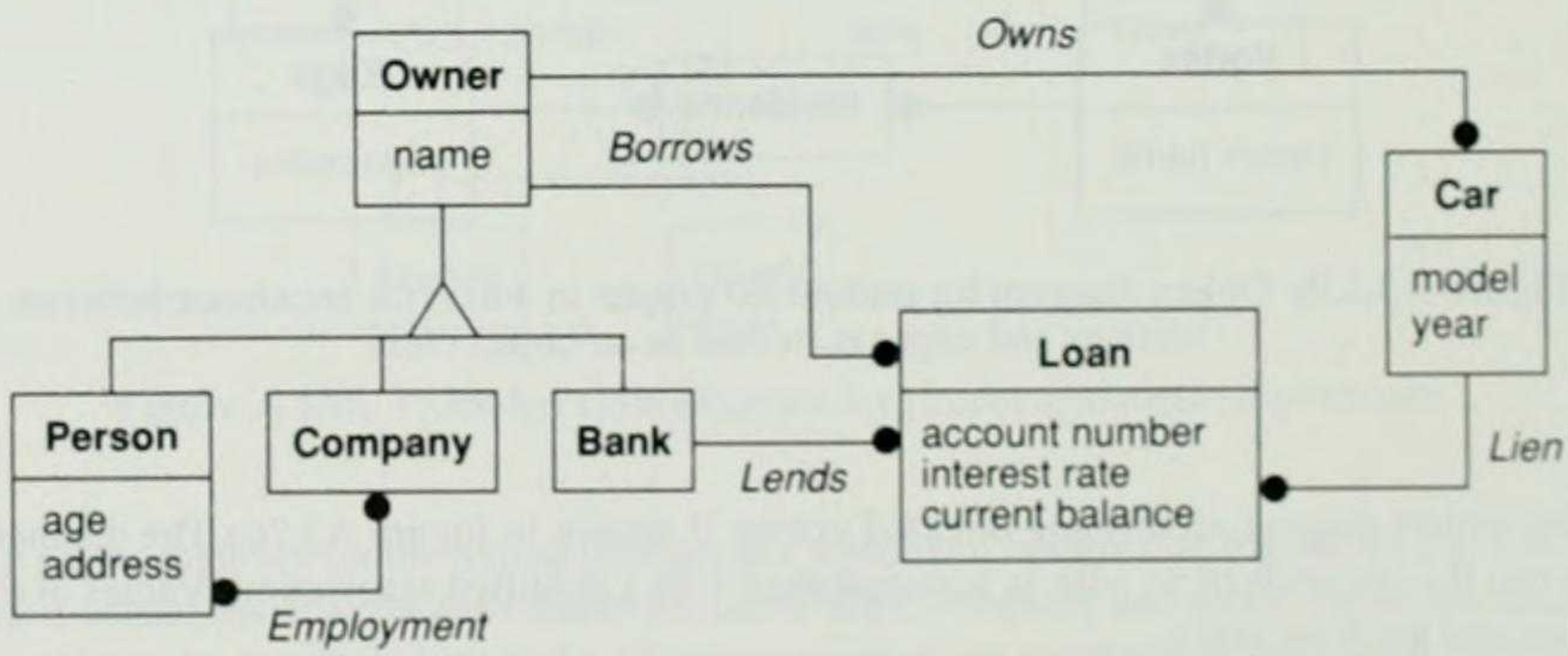


Figure A3.28 Proper object model for car loans

In this form, the arguably artificial restriction that a person have no more than three employers has been eliminated. Note that in this model an owner can own several cars. A car can have several loans against it. Banks loan money to persons, companies, and other banks.

4.2 The object diagram in Figure A4.2 abstracts the classes *Buffer*, *Selection*, and *Sheet* into the superclass *Collection*. Overall, this revision is recommended. Using the generalization relationship promotes code reuse because many operations apply equally well to the subclasses. Six aggregation relationships in the original diagram, which shared similar characteristics, have been reduced to two. Finally, the constraint that each *Box* and *Link* should belong to exactly one *Buffer*, *Selection*, or *Sheet* has been captured by the structure of the diagram.

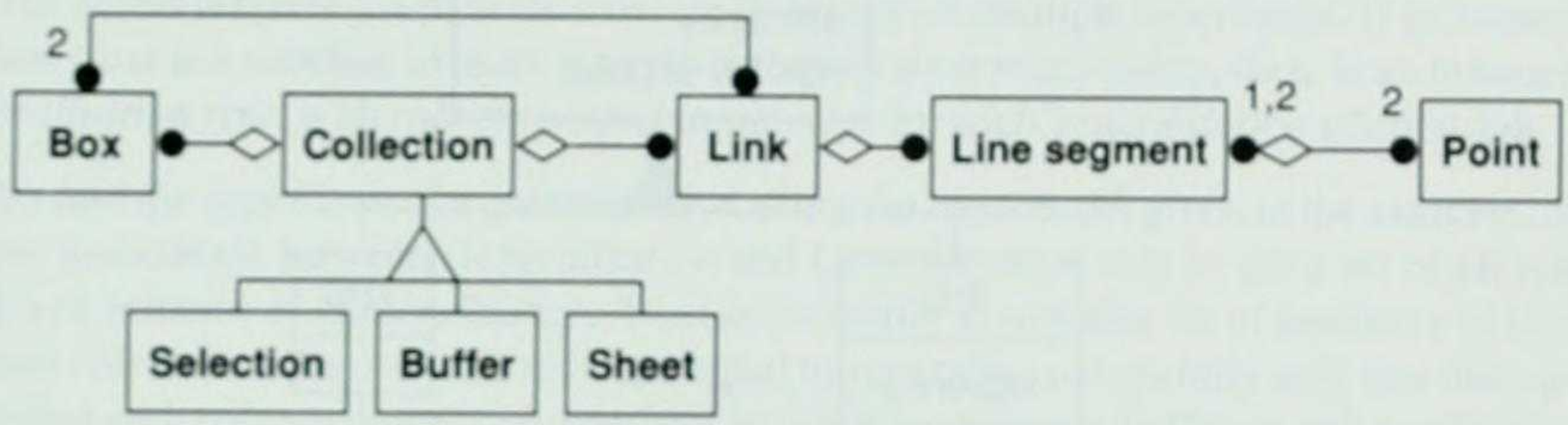


Figure A4.2 Abstraction of the classes *Selection*, *Buffer*, and *Sheet* into the class *Collection*

4.4 An object diagram for a graphical document editor is shown in Figure A4.4. The requirement that a *Group* contains 2 or more *Drawing objects* is expressed as a multiplicity of 2+ on *Drawing object* in its aggregation relationship with *Group*. The fact that a *Drawing object* need not be in a *Group* is expressed by the zero-one multiplicity.

It is possible to revise this diagram to make a *Circle* a special case of an *Ellipse* and to make a *Square* a special case of a *Rectangle*.

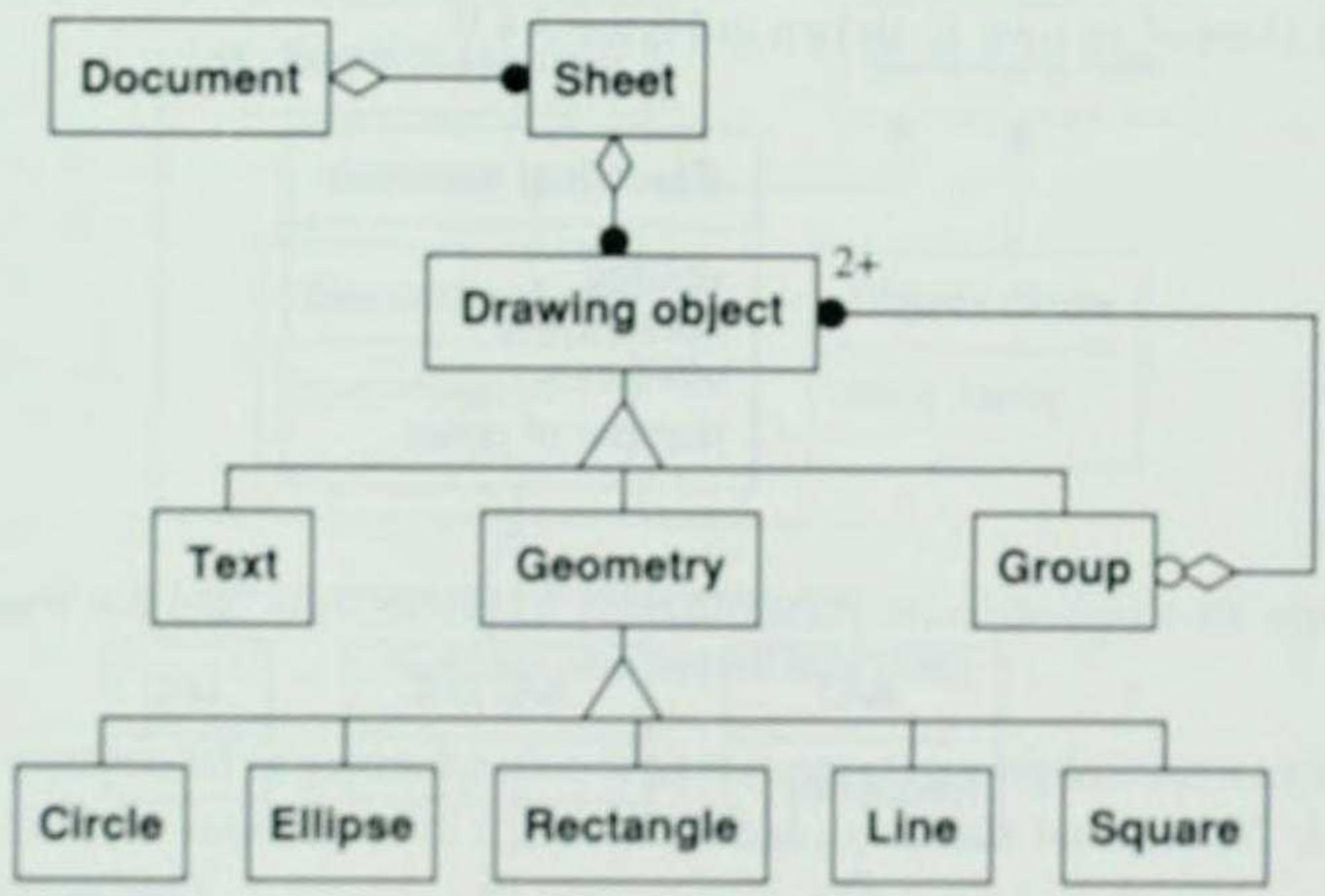


Figure A4.4 Object diagram for a graphical document editor that supports grouping

4.8 An object diagram showing the relationships among several classes of electrical machines is shown in Figure A4.8. We have included attributes that were not requested.

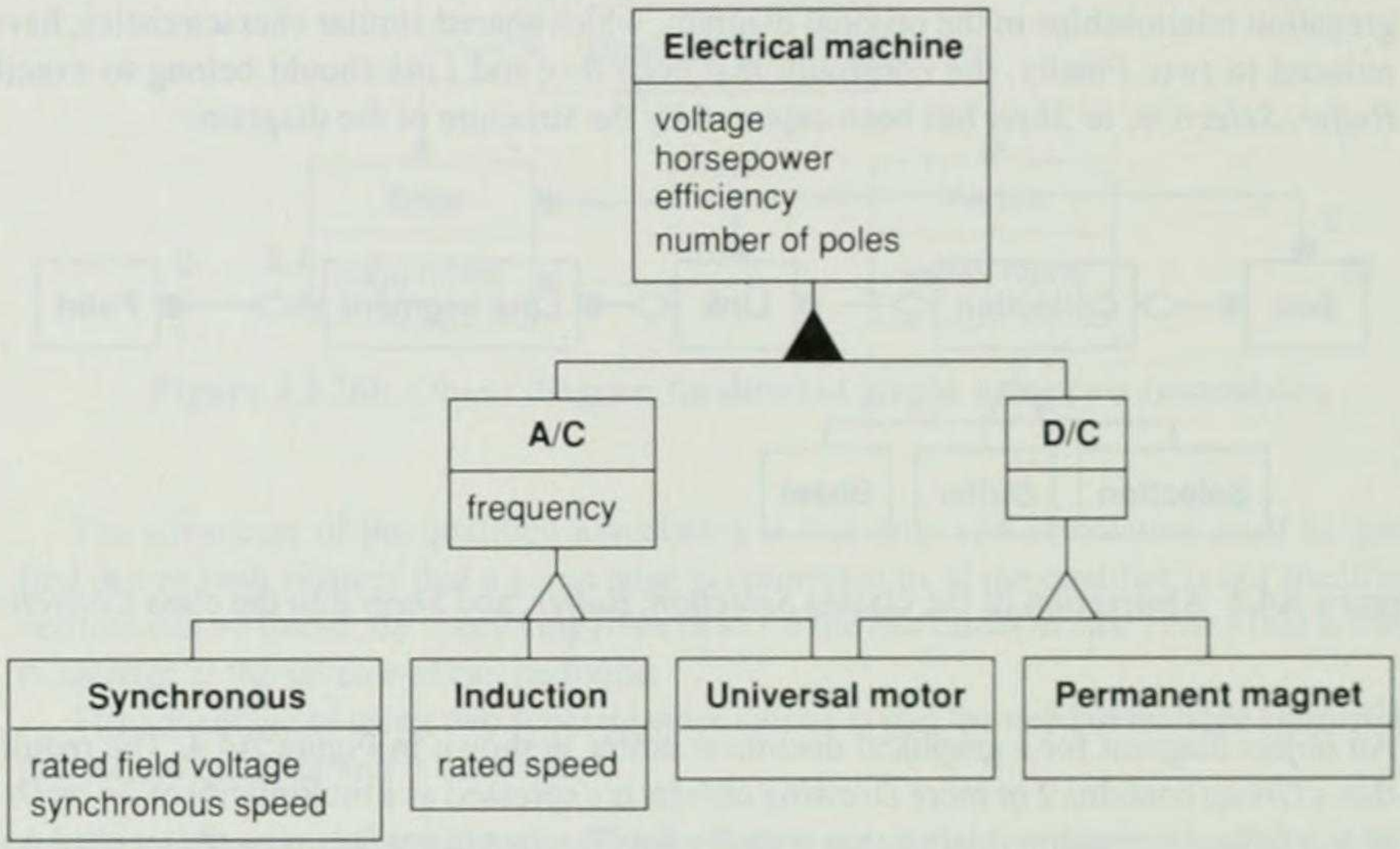


Figure A4.8 Partial taxonomy for electrical machines

4.9 One way to eliminate the multiple inheritance in to convert the overlapping combination of classes into a class of its own as shown in Figure A4.9.

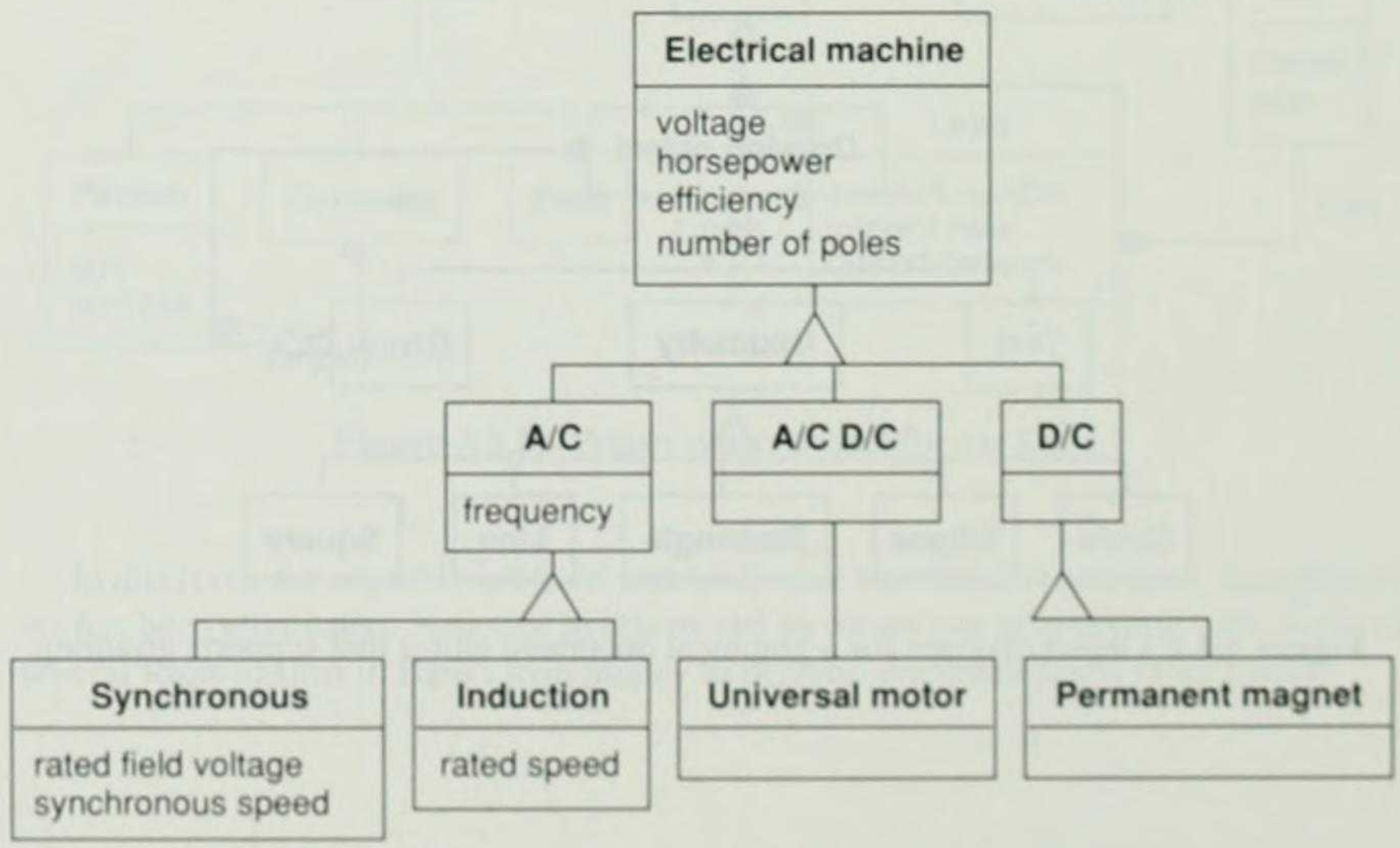


Figure A4.9 Elimination of multiple inheritance

4.10 The object diagram in Figure A4.10 is a metamodel of the following subset of the OMT notation: object classes, attributes, and binary associations.

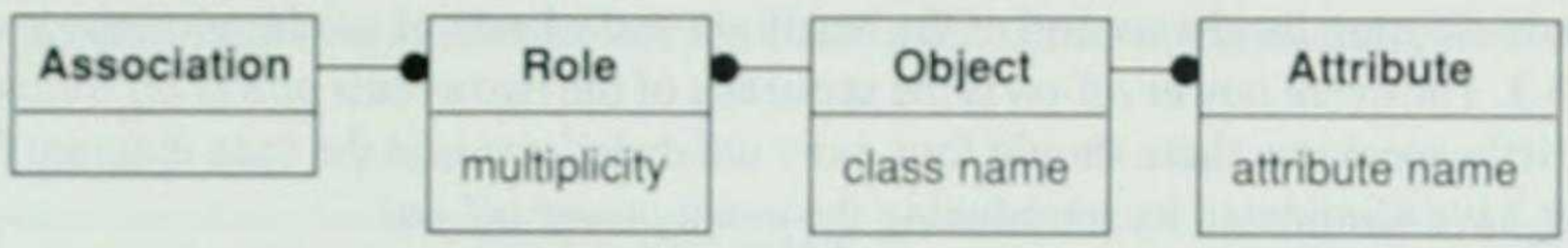


Figure A4.10 Metamodel for a subset of the OMT notation

4.13 The object diagram given in the exercise does support multiple inheritance. If an instance of *Object class* is a subclass in more than one generalization relationship, there is an instance of *Generalization role*, with *role type* equal to *subclass* for each generalization relationship.

4.14 To find the superclass of a generalization using the metamodel given in the exercise, first query the association between *Generalization* and *Generalization role* to get a set of all roles of the given instance of *Generalization*. Then sequentially search this set of instances of *Generalization role* to find the one with *role type* equal to *superclass*. (Hopefully only one instance will be found with *role type* equal to *superclass*, which is a constraint that is not enforced by the model.) Finally, scan the association between *Generalization role* and *Object class* to get the superclass.

One possible revision which simplifies superclass lookup is shown in Figure A4.14a. To find the superclass of a generalization, first query the association between *Generalization* and *Superclass role*. Then query the association between *Superclass role* and *Object class* to find the corresponding instance of *Object class*.

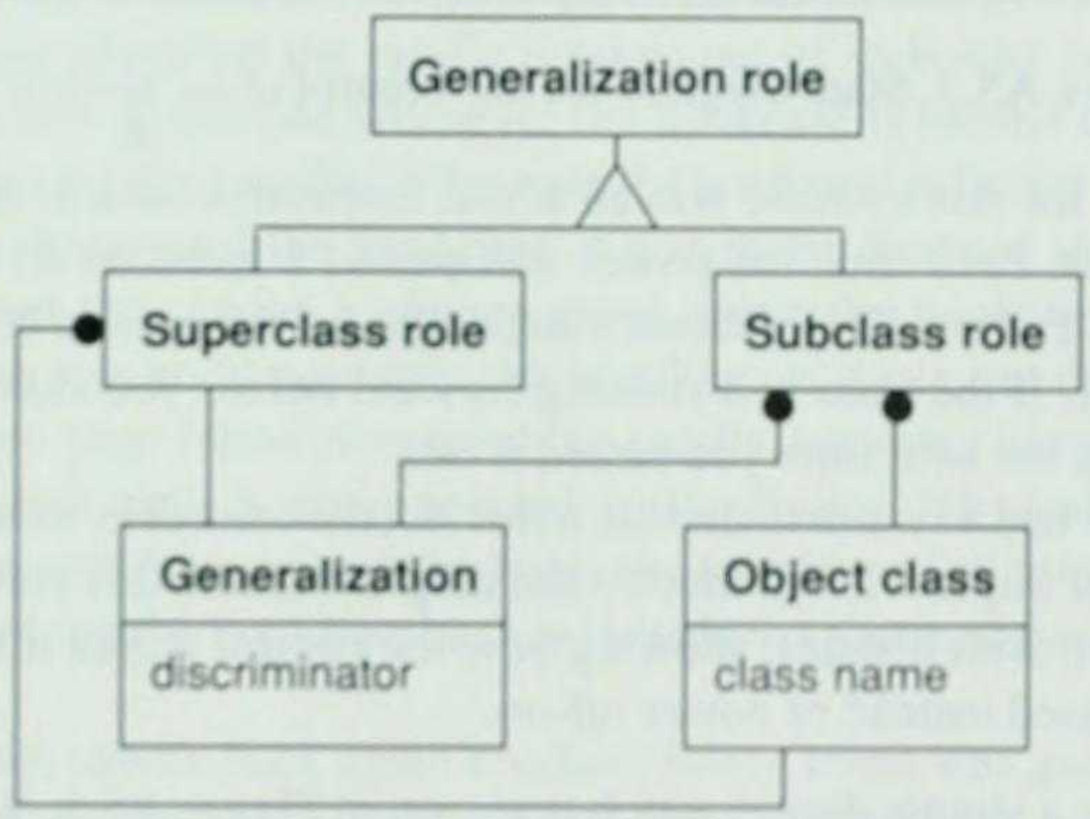


Figure A4.14a Metamodel of generalization relationships with separate subclass and superclass roles

Another metamodel of generalization which supports multiple inheritance is shown in Figure A4.14b. To find the superclass of a generalization using this metamodel, simply query the *Superclass* association.

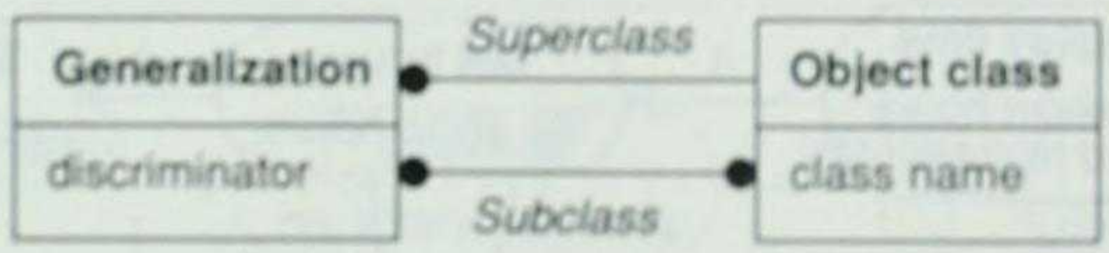


Figure A4.14b Simplified metamodel of generalization relationships

We do not imply that the metamodel in Figure A4.14b is the best model of generalization, only that it simplifies the query given in the exercise. The choice of which model to use will depend on other factors as well.

5.3 The state diagram for the control of the headlight and wheels of an electric train is shown in Figure A5.3. The event *power off-on* is the sequence of the two events *power off* followed by *power on*. Strictly speaking, there should four more unlabeled states in the state diagram for the wheels that we have eliminated by introducing the event *power off-on*.

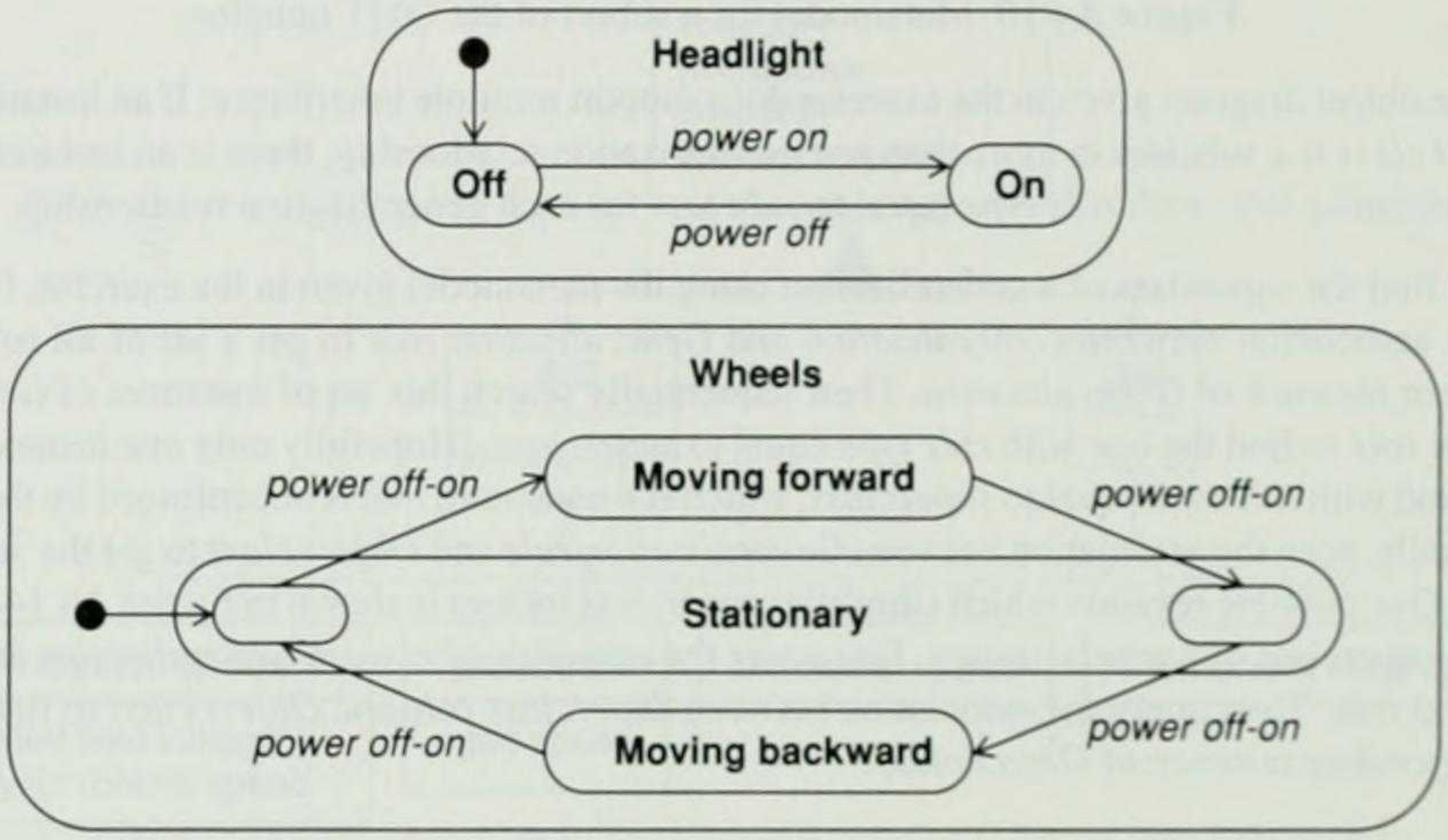


Figure A5.3 State diagram for the control of an electric train

The inspiration for this exercise was an actual electromechanical control which contained a wheel with contacts. Each time the power was pulsed a solenoid driven ratchet advanced the wheel one quarter of a turn. The stationary state allowed the train to be stopped with power still applied to the track. If the train was running forward before you shut it off and put it away, it would be stationary the next time you turned it on.

The control also had a switch to disable it that left the control in whatever state it was in when it was shut off. You might want to modify the diagram to add this feature. You also might prepare a state diagram with 8 states, showing how the control works if separate events *power on* and *power off* are used instead of *power off-on*.

5.5 A state diagram for a simple digital watch is shown in Figure A5.5. We assume that pressing a button is an event and that we may ignore the release of a button. We use *A* in the diagram to refer to pressing the A button and *B* to refer to pressing the B button.

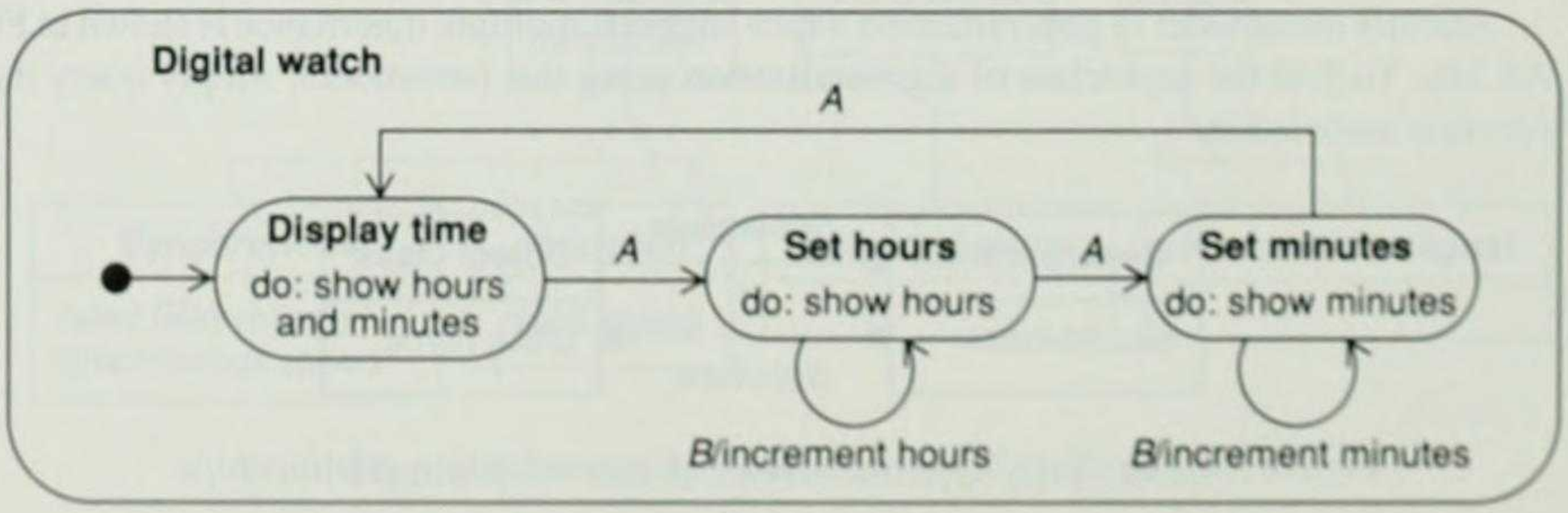


Figure A5.5 State diagram for a simple digital watch

5.10 The completed state diagram for the motor control described in the exercise is shown in Figure A5.10.

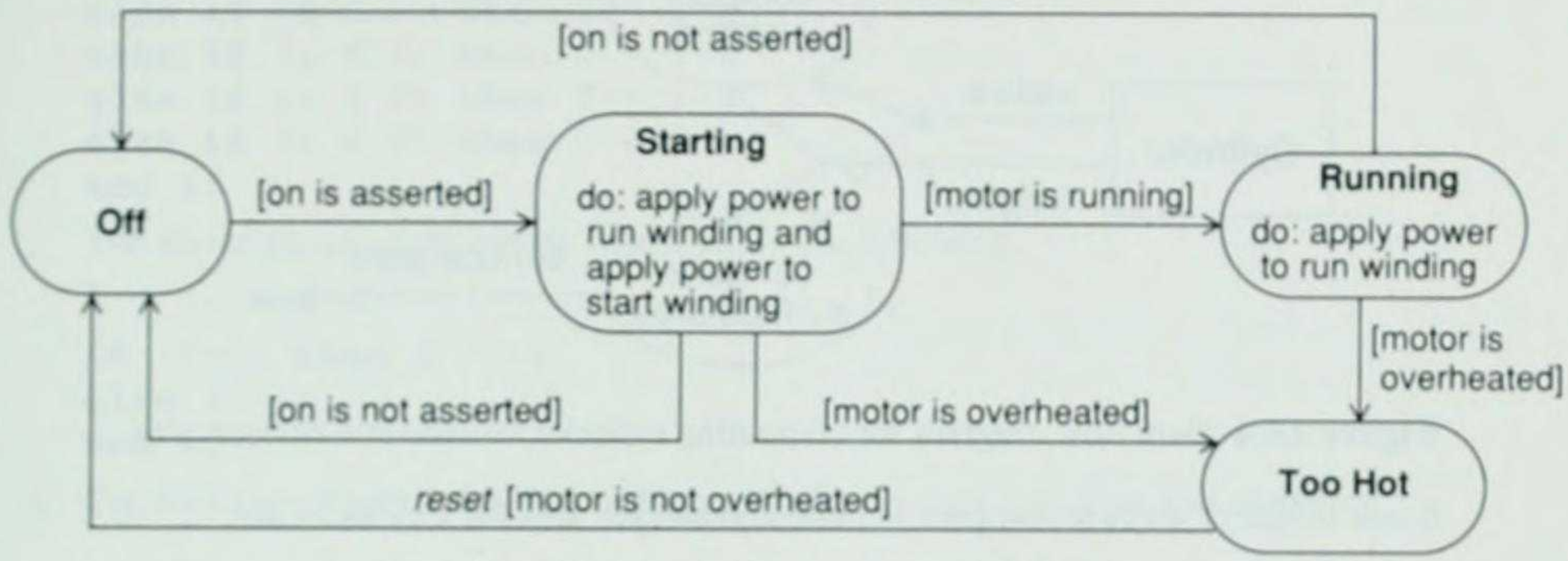


Figure A5.10 State diagram for a simple motor control

5.20 This exercise demonstrates that even simple state diagrams can lead to complex behavior. A state diagram that will explain the scenario given in the exercise is shown in Figure A5.20. A *change* event occurs whenever the candle is taken out of its holder or whenever it is put back. The condition *at north* is satisfied whenever the bookcase is behind the wall. The condition *at north, east, south, or west* is satisfied whenever the bookcase is facing front, back, or to the side.

When you first discovered the bookcase, it was in the *Stopped* state pointing south. When your friend removed the candle, a *change* event drove the bookcase into the *Rotating* state. When the bookcase was pointing north, the condition *at north* put the bookcase back into the *Stopped* state. When your friend reinserted the candle, another *change* event put the bookcase into the *Rotating* state until it again pointed north. Pulling the candle out generated another *change* event and would have caused the bookcase to rotate a full turn if you had not blocked it with your body. Forcing the bookcase back is outside the scope of the control and does not have to be explained.

When you put the candle back again another *change* event was generated, putting the bookcase into the *Rotating* state once again. Taking the candle back out resulted in yet another *change* event, putting the bookcase into the *Stopping* state. After 1/4 turn, the condition *at north, east, south or west* was satisfied, putting the bookcase into the *Stopped* state.

What you should have done at first to gain entry was to take the candle out and quickly put it back before the bookcase completed 1/4 turn.

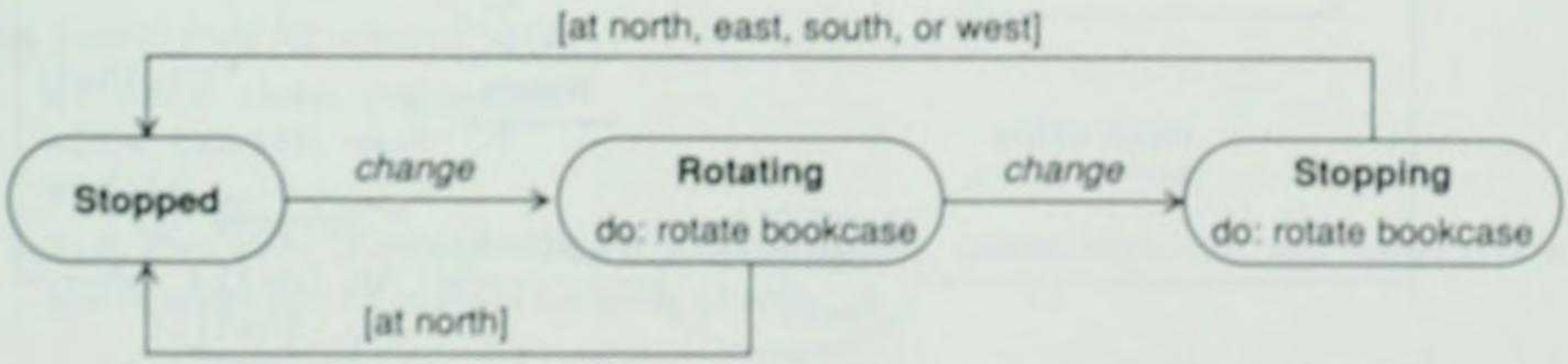


Figure A5.20 State diagram for the control of the entrance to a secret passageway

6.4 Figure A6.4 contains the data flow diagram for computing cylinder geometry.

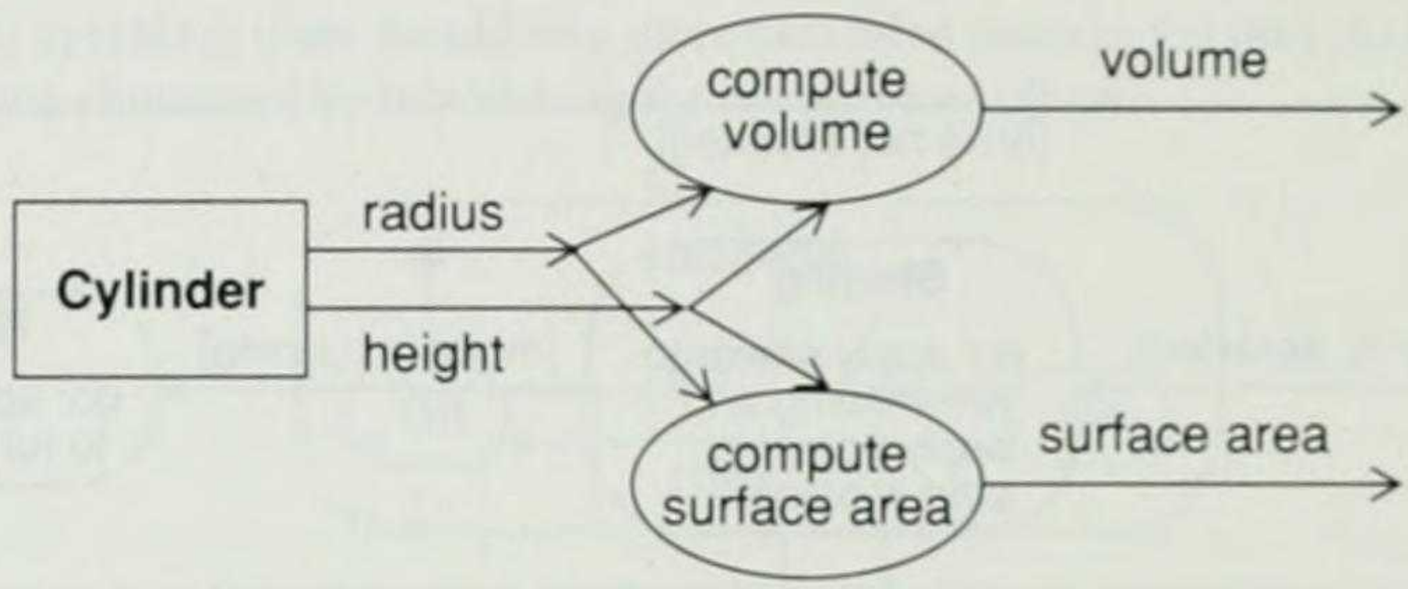


Figure A6.4 Data flow diagram for computing cylinder volume and surface area

Some different ways to compute volume and surface area for a cylinder are:

- *A formula.* Volume = $\pi r^2 h$. Surface area = $2\pi rh$. (r=radius, h=height)
- *A lookup table.* Volume and surface area are listed for standard values of radius and height.
- *Numerical methods.* Calculate volume and surface area from differential equations.

6.5 Figure A6.5 shows the data flow diagram for computing the mean for a sequence of input values.

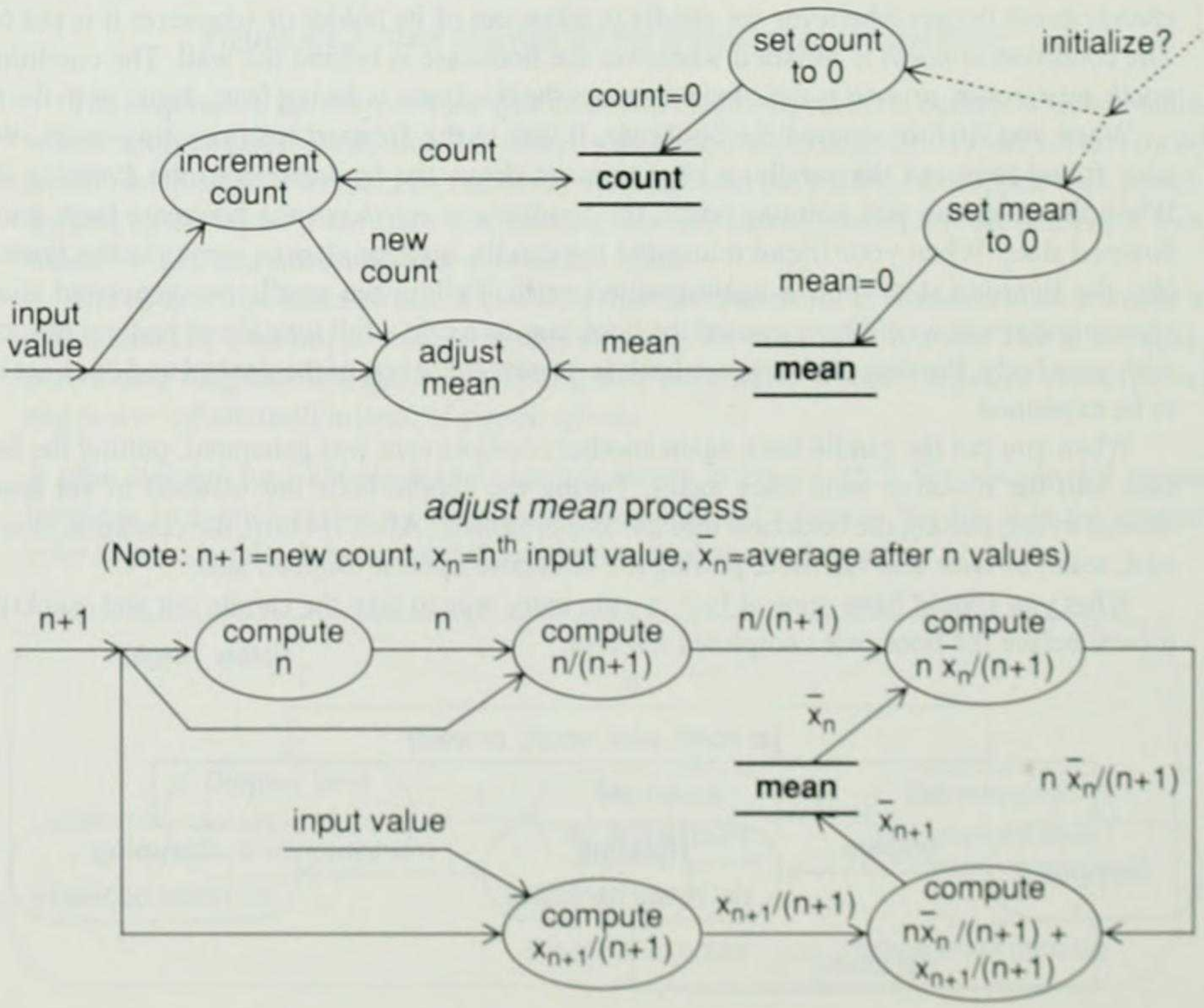


Figure A6.5 Data flow diagram for computing mean of a sequence of values

6.7a. Using conditional statements, the algorithm for T(x) can be expressed as:

```
if (x > -3) and (x ≤ -2) then T = 3+x
else if (x ≤ -1) then T = -x-1
else if (x ≤ 0) then T = 1+x
else if (x ≤ 1) then T = 1-x
else if (x ≤ 2) then T = x-1
else if (x ≤ 3) then T = 3-x
end if
```

The above pseudo code can be simplified to the following.

```
t = (x mod 2) - 1
if (t<0) then T = -t
else T = t
end if
```

c. The data flow diagram for T(x) in Figure A6.7 only uses functions and arithmetic.

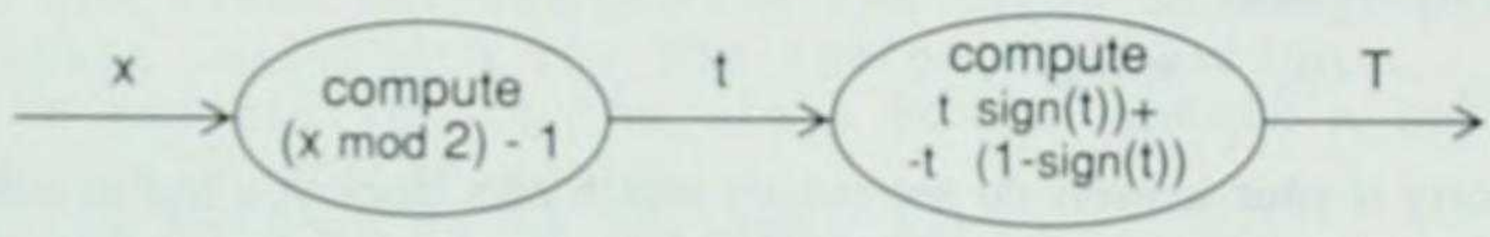


Figure A6.7 Data flow diagram for *T(x)*

6.10a. absolute value: if $x \geq 0$ then $|x| = x$, else $|x| = -x$

b. trigonometric sine: $\sin \theta = y/r$

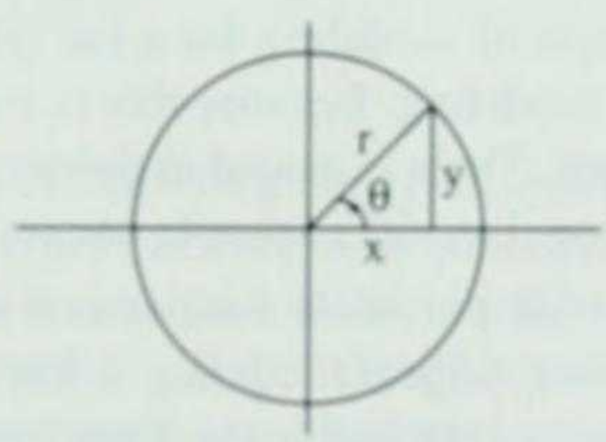


Figure A6.10 Definition of trigonometric sine

c. natural logarithm: $\ln(x) = \int_1^x (1/t)\, dt$

d. square root: y = sqrt (x) such that x = y * y

6.11a. pseudo code for absolute value

```
if x ≥ 0 then return x
else return -x
end if
```

b, c, d. Two ways of implementing trigonometric sine, natural logarithm, and square root are by power series and fixed point iteration.

Mathematics can be used to derive an infinite power series ($a_0x^0 + a_1x^1 + a_2x^2 + \ldots$) that is equivalent to the desired function. The function can then be approximated by evaluating the first

portion of the power series and truncating the remainder. The sine, logarithm, and square root functions have an infinite number of power series; a desirable power series would converge quickly for the arguments of interest and bound the maximum error. Before computing a power series, the argument value is usually normalized to lie within a standard range. Thus for instance, the following property can be used to transform the sine argument to an equivalent argument that lies in the range of 0° to 360°

```
sin (θ) = sin (θ mod 360°)
```

One could normalize the argument of the sin function even further by recognizing that

```
sin (θ) = - sin (θ - 180°)
```

and thus transform the sin argument to lie in the range of 0° to 180°.

Another way of implementing these functions is by fixed point iteration: Guess an initial value and then repeatedly apply an identity formula, until you are close enough to the correct answer. For example, an initial estimate of y = sqrt(x) is y = (1+x)/2; in fact this is a very good estimate of the square root for values of x close to 1. A new value of x can be computed with the following formula:

$$y_{n+1} = (y_n + x/y_n) / 2$$

8.1 Do not worry if your answers do not exactly match ours since you had to make assumptions about the functional specifications. The point of the exercise is to make you think about the examples in terms of the three aspects of modeling.

a. Functional modeling, object modeling, and dynamic modeling, in that order, are important for a bridge playing program because good algorithms are needed to yield intelligent playing. The game involves a great deal of strategy. Close attention to inheritance and method design can result in significant code reuse. The interface is not complicated, so the dynamic model is simple and could be omitted.

c. The order of importance of aspects of modeling for a car cruise control is dynamic modeling, object modeling, and functional modeling. Because this is a control application you can expect the dynamic model to be important. There is a need to thoroughly understand scenarios and protocols for interaction. The functional model is simple because there are not many calculations.

e. For a spelling checker the order of importance of aspects of modeling is object modeling, functional modeling, dynamic modeling. Object modeling is important because of the need to store a great deal of data and to able to access it quickly. Functional modeling is important because an efficient algorithm is needed to check spelling quickly. The dynamic model is simple because the user interface is simple. All that is needed is to give the user a chance to correct each misspelled word that is found.

8.8a. We assume there is exactly one instance of the class *Selection*. Declare an error if there is not exactly one instance. Ascend the class hierarchy to the class *Collection*. Scan the association between the classes *Collection* and *Box* to obtain a set of boxes. Scan the association between the classes *Collection* and *Link* to obtain a set of links.

c. We assume that *boxes* is a global variable that contains a set. Initialize *boxes* to be empty. To determine all boxes that are linked directly or indirectly to a given box, call the following recursive procedure with the given box as the argument:

```
Retrieve_boxes(given_box)
      Retrieve set of links associated with given_box
      for each link
        for each box associated with the link
          if box is not in boxes then
            Add box to boxes
            Retrieve_boxes(box)
          end if
        end for each box
      end for each link
```

e. Take advantage of the fact that a link is associated with exactly 2 boxes. If *given_box* and *given_link* are not associated then declare an error. Otherwise place the 2 boxes associated with *given_link* into a set and delete *given_box* from the set.

g. We assume that *selection* is a global variable that contains an instance of *Selection*. We assume that *bridges* is a global variable in which the set of bridging links is to be placed.

```
Ascend the class hierarchy from the class Selection to the
      class Collection for the instance selection.
Scan the association to the class Box to obtain a set of all
      boxes in the selection.
Initialize bridges to be an empty set.
for each box in the set.
      for each link associated with the box.
        Get the box on the other end of the link,
          using the answer to part e).
        if the box on the other end is not in the selection.
          Add the link to bridges.
        end if
      end for each link
end for each box
```

8.9 There is more than one way of satisfying the given queries. Some readers may find Figure E8.3 easier to understand than Figure E8.2 and will consequently arrive at simpler queries by using it. Others may find the reverse to be true. However, the two are nearly equivalent from the point of view of the given queries. For some of the queries Figure E8.3 requires an extra step. The reader may find that one of the approaches has advantages during implementation in a particular language (or database) depending on the features of the language.

8.16 The following candidate classes are really attributes or redundant attributes:

address, age, average score, child's name, date, difficulty factor, net score, team name

The following are redundant:

child, contestant, individual, person, registrant

These are vague or irrelevant:

back, card, conclusion, corner, individual prize, leg, pool, prize, team prize, try, water ballet

The following are implementation details:

file of team member data, list of scheduled meets, number, group

These candidates are really operations:

compute average, register

The astute reader will notice some of the candidate classes that are not eliminated here do not appear in Figure E8.4. That is because Figure E8.4 is only a partially completed object diagram.

8.19a. Scan the association between the classes *Team* and *Competitor* using the given team to get the set of its members.

c. There are several ways to find the net score of a competitor for a given figure at a given meet. One way is the following:

Find the set of all events held at the given meet from the association between *Meet* and *Event*. Find the set of all events associated with the given figure. Intersect the two sets. Given the way that meets are conducted, there should be exactly one event in the intersection. Use a similar process using the event and competitor as input to find the trial for the competitor in that event. The net score is contained in the trial.

e. Use the association between *Meet* and *Event* to find all events held at the given meet. Use the association between *Event* and *Trial* to find all trials of all events at the given meet. Use the association between *Competitor* and *Trial* to find all trials by the given competitor. Intersect the set of all trials of all events at the given meet with the set of all trials by the given competitor. The result is the set of all trials by the given competitor at the given meet. Compute the average score from net scores of the trials by the given competitor at the given meet.

g. The object diagram in Figure E8.4 does not support this query very well. Find all meets in the given season. From all meets find all events in the given season. From all events find all trials in the given season. For each trial merge the associated competitor into the answer set.

8.23 The 2 teams were the Dolphins and the Whales. The 4 competitors were Heather Martin, Elissa Martin, Cathy Lewis, and Christine Brown. Stations were set up on the northwest and southeast corners of the pool with three judges each. The 4 events were the Ballet Leg, the Dolphin, the Front Pike Somersault, and the Back Pike Somersault. Heather was number 3. In this scenario we assume that there is a personal computer at each station. There are other possible scenarios, depending on the system architecture. When Heather approached the station to try the Ballet Leg the computer operator called her number as it appeared on the display. When Heather verified her number the operator confirmed it with the computer which was then ready to accept scores. After Heather performed the Ballet Leg the 3 judges held up the scores 3.8, 3.6, and 3.7. A scorekeeper read the scores. As they were read the computer operator entered them into the computer. Another scorekeeper wrote them down and checked the numbers on the computer display after they were entered. In this case the operator made a mistake and the judges were asked to repeat the scores. When everyone was satisfied the operator verified the scores and the computer stored them in a database.

When all of the events were over a batch program was run to merge the data from the two stations into a single database. Then another batch program was run to convert the raw scores into net scores for each trial, to compute the overall scores for each competitor, to sort the competitors by overall score, and to print out the results. The print outs included an ordered listing of the winners and summary sheets for each competitor. The summary sheet for each competitor included raw scores, degree of difficulty, and net scores for each event.

8.26 A partial shopping list of operations is shown in Figure A8.26.

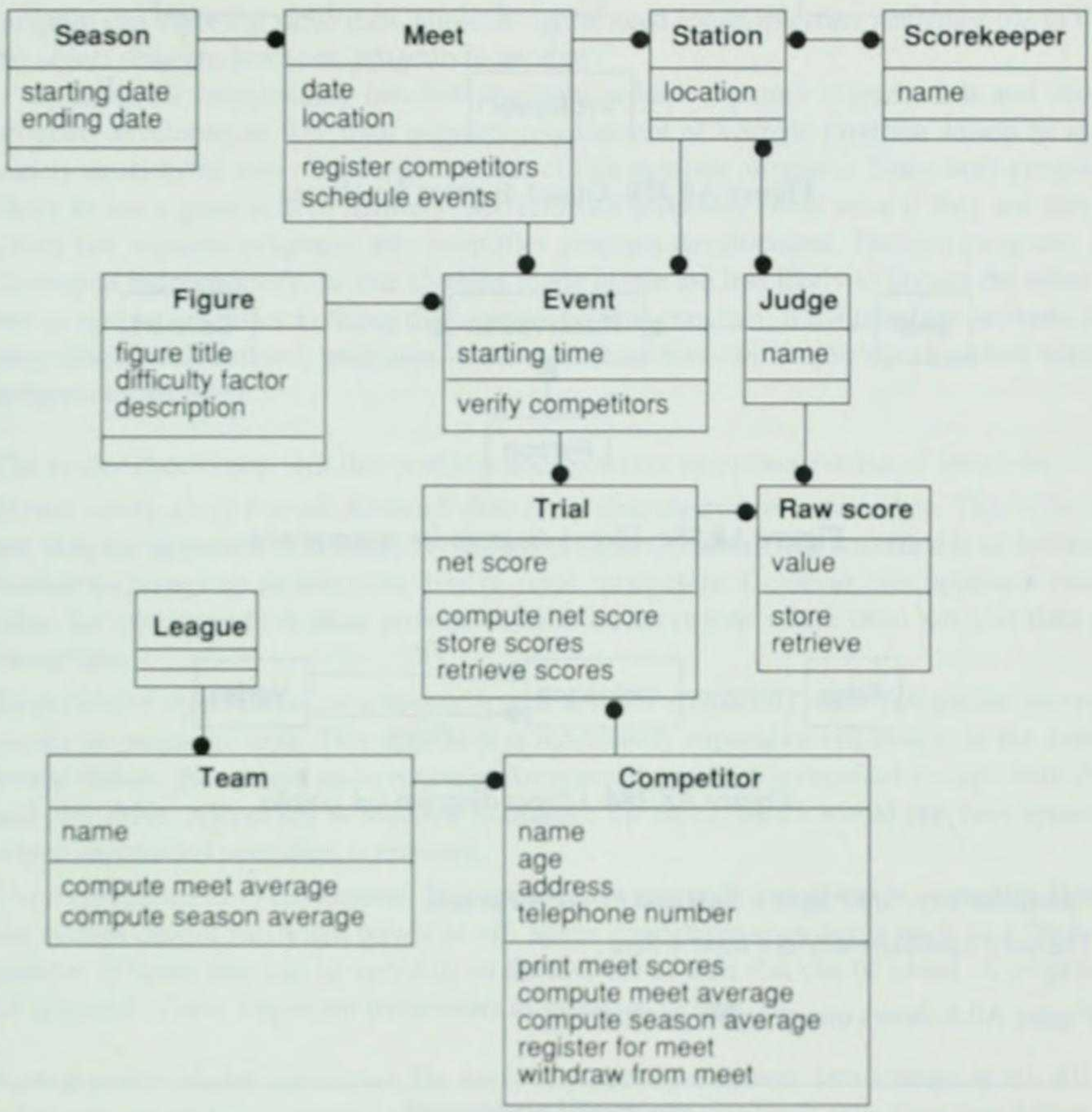


Figure A8.26 Partial object diagram for a scoring system including operations

8.28 The revised diagrams are shown in Figures A8.28a-A8.28d. Note that Figure A8.28a could be simplified by combining *Appointment* and *Date-time* and treating *Date-time* as an attribute. Figure A8.28d could be simplified by combining *Edge* and *Incidence*. In general a ternary association can always be converted into a class. Some thought may be required to get the multiplicities right.

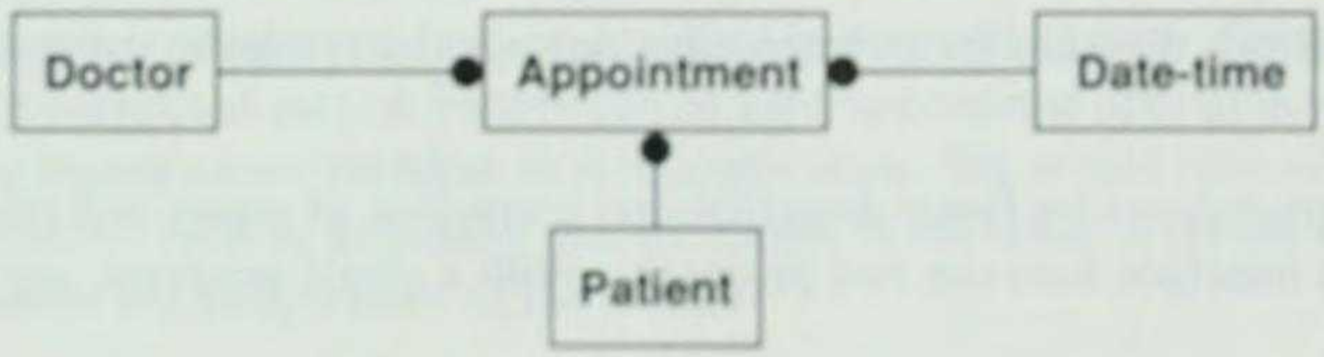


Figure A8.28a Object diagram for appointments

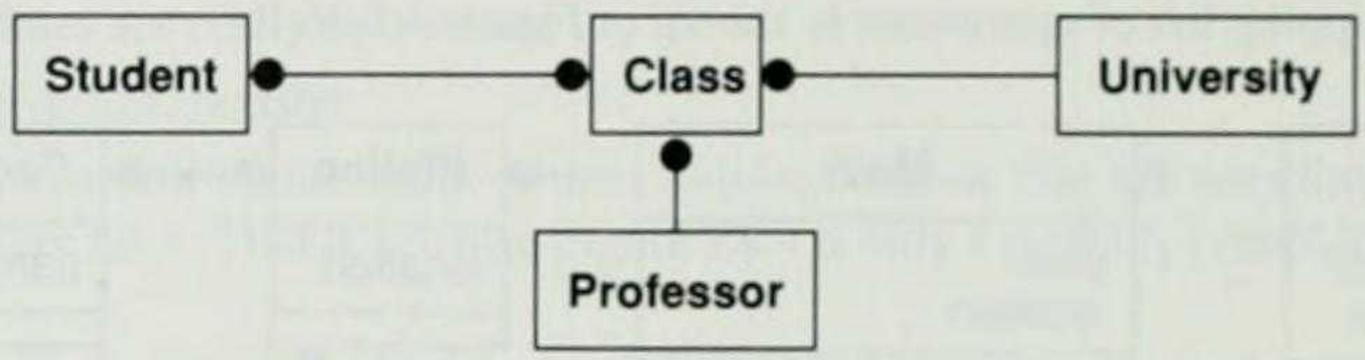


Figure A8.28b Object diagram for classes

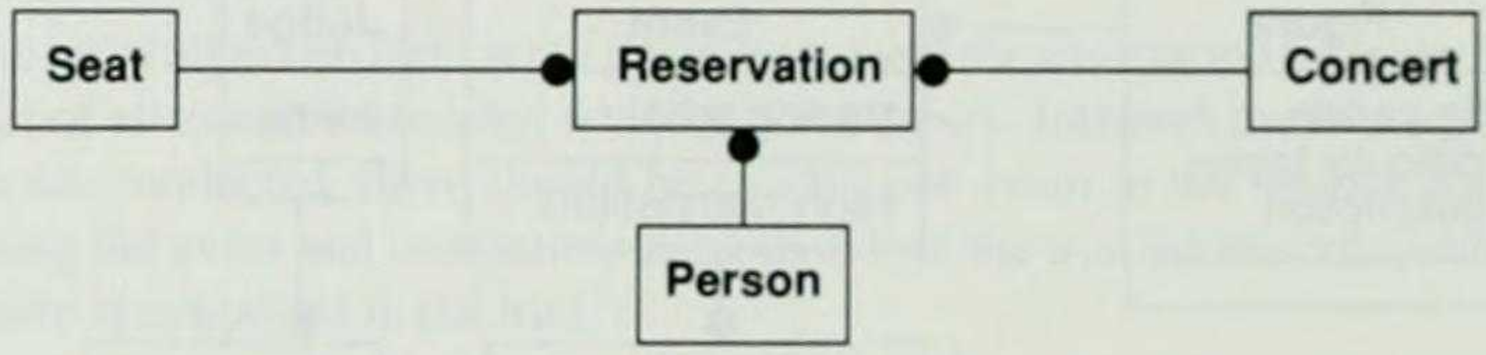


Figure A8.28c Object diagram for reservations

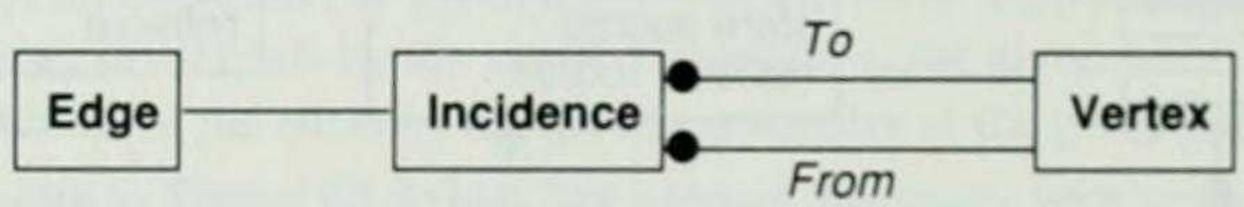


Figure A8.28d Object diagram for graphs

8.33a. Candidate keys are *Flight + Seat* and *Flight + Person*.

b. The only candidate key is *Flight + Seat*.

9.6 Figure A9.6 shows one possible partitioning.

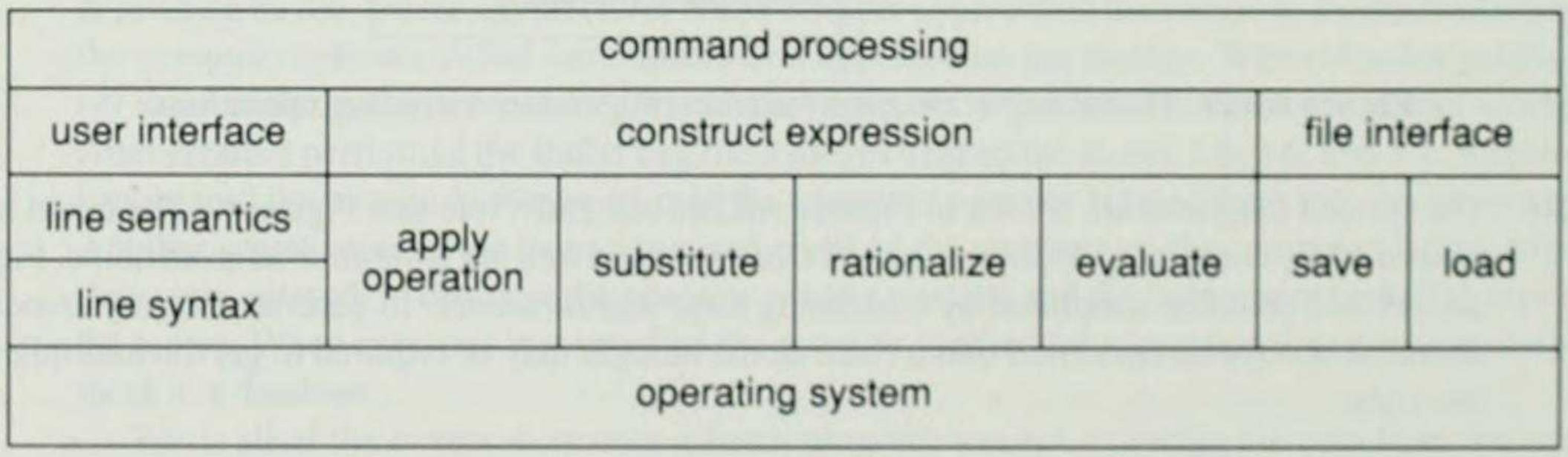


Figure A9.6 Block diagram for an interactive polynomial symbolic manipulation system

9.7 A single program provides faster detection and correction of errors and eliminates the need to implement an interface between two programs. With a single program, any errors that the sys-

tem detects in the process of converting the object diagram to a database schema can be quickly communicated to the user for correction. Also, the editing and the conversion portions of the program can share the same data, eliminating the need for an interface such as a file to transfer the object diagram from one program to another.

Splitting the functionality into two programs reduces memory requirements and decouples program development. The total memory requirement of a single program would be approximately equal to the sum of the requirements of two separate programs. Since both programs are likely to use a great deal of memory, performance problems could arise if they are combined. Using two separate programs also simplifies program development. The two programs can be developed independently, so that changes made in one are less likely to impact the other. Also, two programs are easier to debug than one monolithic program. If the interface between the two programs is well defined, problems in the overall system can be quickly identified within one program or the other.

9.11 The reader should note that this problem does not give an exhaustive list of solutions.

a. *Do not worry about it at all. Reset all data every time the system is turned on.* This is the cheapest, simplest approach. It is relatively easy to program, since all that is needed is an initialization routine on power up to allow the user to enter parameters. However, this approach cannot be taken for systems which must provide continuous service or which must not lose data during power loss.

c. *Keep critical information on a magnetic disk drive. Periodically make full and/or incremental copies on magnetic tape.* This approach is moderately expensive and bulky. In the event of a power failure, the system stops running. An operating system is required to cope with the disk and tape drive. A operator is required to manage the tapes, which would preclude applications where unattended operation is required.

e. *Use a special memory component.* This approach is relatively cheap and is automatic. However, the system cannot run when power is off. Some restrictions may apply such as a limit on the number of times data can be saved or on the amount of data that can be saved. A program may be required to save important parameters as power is failing.

9.12a. *Four function pocket calculator.* Do not worry about permanent data storage at all. All of the other options are too expensive to consider. This type of calculator sells for a few dollars and is typically used to balance checkbooks. Memory requirements are on the order of 10 bytes.

c. *System clock for a personal computer.* Only a few bytes are required, but the clock must continue to run with the main power off. Battery backup is an inexpensive solution. Clock circuits can be designed that will run for 5 years from a battery.

e. *Digital control and thermal protection unit for a motor.* On the order of 10 to 100 bytes are needed. This application is sensitive to price. An uninterruptable power supply is too expensive to consider. Tape and disk drives are too fragile for the harsh environment of the application. Use a combination of switches, special memory components, and battery backup. Switches are a good way to enter parameters, since an interface is required anyway. Special memory components can store computed data. A battery can be used to continue operation with power removed but presents a maintenance problem in this application. We would question this requirement, seeking alternatives such as assuming that the motor is hot when it is first turned on or using a sensor to measure the temperature of the motor.

9.13a. A description of the diagram, assuming that tabs, spaces, and line feeds are ignored, is:

```
(DIAGRAM
        (CLASS
           (NAME "Polygon"))
        (CLASS
           (NAME "Point")
           (ATTRIBUTE "x")
           (ATTRIBUTE "y"))
        (ASSOCIATION
           (ROLE (NAME "Polygon") ONE)
           (ROLE (NAME "Point") MANY)))
```

9.14 The hardware approach is fastest, but incurs the cost of the hardware. The software approach is cheapest and most flexible, but may not be fast enough. Use the software approach whenever it is fast enough. General purpose systems favor the software approach, because of its flexibility. Special purpose systems can usually integrate the added circuitry with other hardware.

Actually, there is another approach, firmware, that may be used in hardware architectures. Typically, in this approach a hardware controller calculates the CRC under the direction of a microcoded program that is stored in a permanent memory that is totally invisible externally. We will count this approach as hardware.

a. Use a hardware approach for a floppy disk controller. Flexibility is not needed, since a floppy disk controller is a special purpose system. Speed is needed because of the high data rate.

c. Use hardware to check memory. This is an example of a very specific application, where the function can probably be integrated with the circuitry in the memory chips. The data rate is very high.

e. Use a software approach to validate an account number. The data rate is very low. The system handling the account number is probably running on a general purpose computer.

10.5 Figure A10.5 enforces a constraint that is missing in Figure E10.1: Each boundary corresponds to exactly one ellipse or rectangle. One measure of the quality of an object diagram is how well its structure captures constraints.

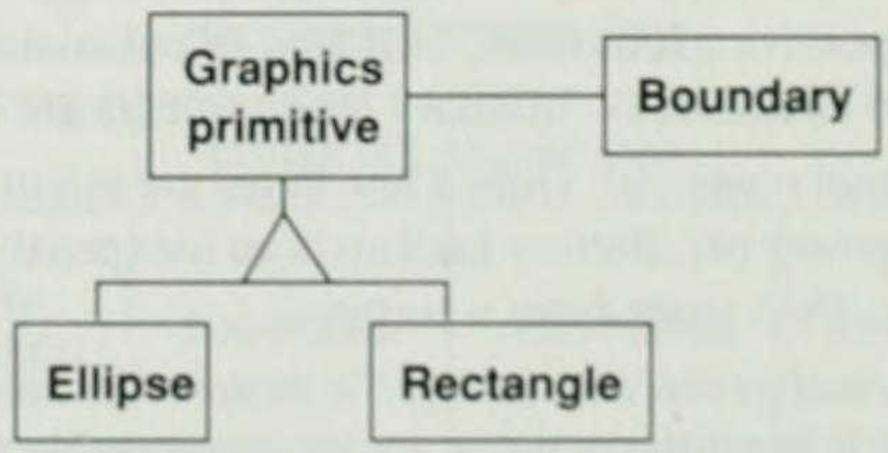


Figure A10.5 Improved object diagram for part of a CAD system

10.11 A derived association supports direct traversal from *Page* to *Line*. In general derived entities present a trade-off: They speed execution of certain queries but incur an update cost to keep the redundant data consistent with the base data.

The line-page association is derived by composing the line-column and column-page associations. Since it is present for optimization, it would probably be traversed only from lines to pages and could therefore be implemented as a pointer attribute within class *Line*.

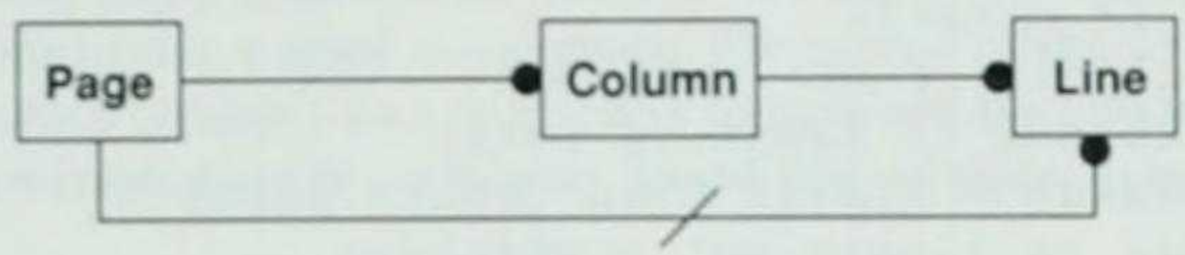


Figure A10.11 Object diagram for a newspaper with a derived association

10.12 Type *Rank* is an enumeration of {Ace, King, Queen, 10, 9, 8, 7, 6, 5, 4, 3, 2}. The ranks are ordered but the order varies from game to game (some games rank the ace highest, some lowest). Depending on the games to be supported, the enumeration could be ordered or unordered. If the enumeration is unordered, then an ordering function must be provided for rank values in a particular game. It would be possible to implement a rank value as an integer. Type *Suit* is an enumeration of {Spade, Club, Heart, Diamond}. For many games, suits are unordered; the order of suits varies in other games, so an ordering function on suit values is probably best. Suit and rank could also be implemented as objects rather than pure values, given the complexity of the ordering functions and other peculiarities of real card games.

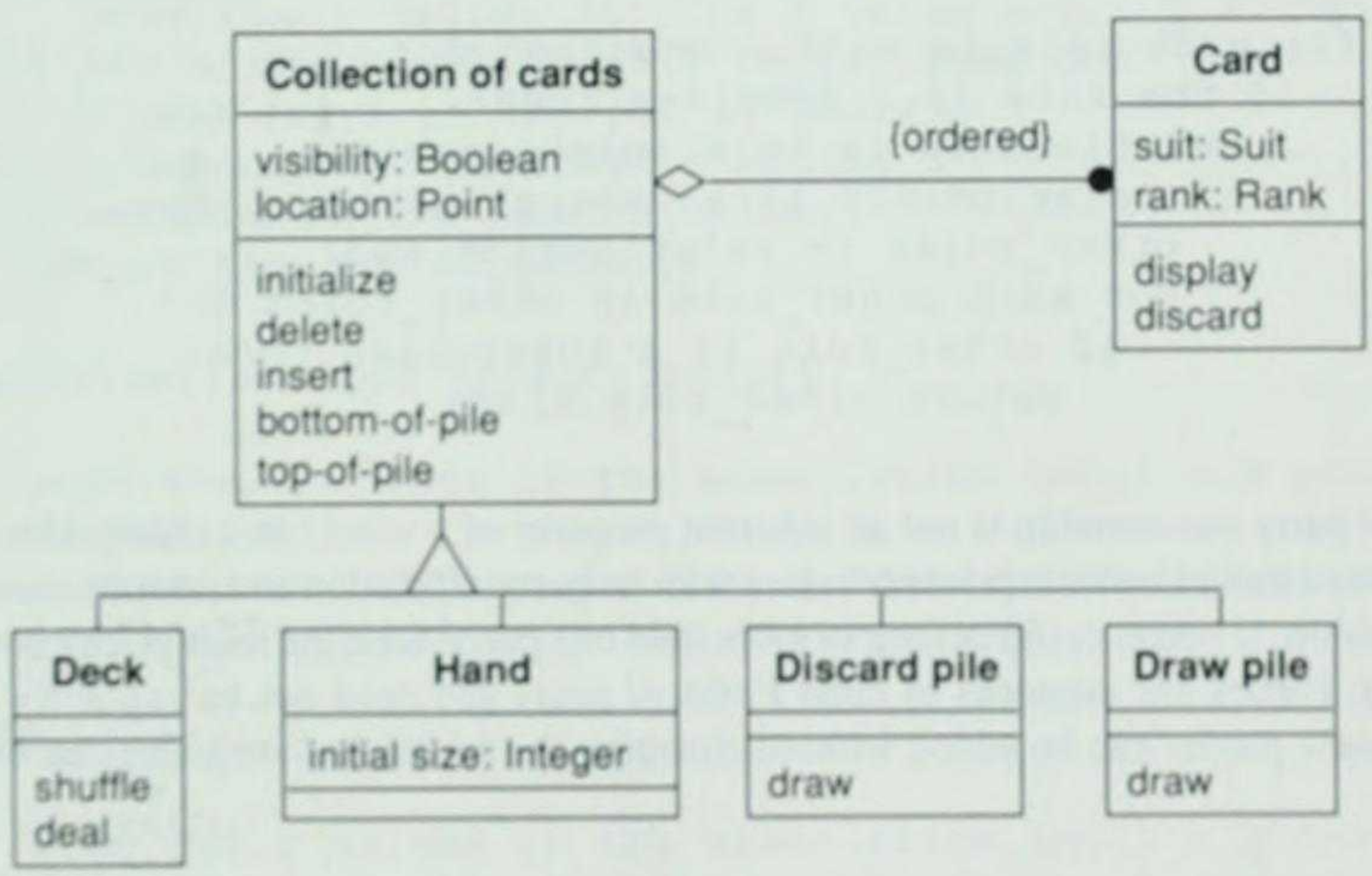


Figure A10.12 Portion of an object diagram for a card playing program

10.19 The code listed below sketches out a solution. This code lacks internal assertions that would normally be included to check the correctness of the metadata. For example, error code should be included to handle the case where the role is a subclass and the relationship is not Generalization. In code that interacts with users or external data sources, it is usually a good idea to add an error check as an else clause for conditionals that "must be true."

```
trace_inheritance_path (class1, class2): Path
{
      path := new Path of Class;
      // try to find a path from class1 as descendent of class 2
      classx := class1;
      while classx is not null do:
        add classx to front of path;
        if classx = class2 then return path;
        classx := classx.get_superclass
      // try to find a path from class2 as descendent of class 1
      path.clear;
      classx := class2;
      while classx is not null do:
        add classx to front of path;
        if classx = class1 then return path;
        classx := classx.get_superclass
      // the two classes are not directly related
      // return an empty path
      path.clear
      return path
}
Class::get_superclass : Class
{
      for each role in self.connection do:
        if the role is a Subclass then:
          relationship := role.relationship;
          if relationship is a Generalization then:
            other_roles := relationship.end;
            for each other_role in other_roles do:
              if other_role is a Superclass then:
                return other_role.class
}
```

10.22 Political party membership is not an inherent property of a voter but a changeable association. The revised model better represents voters with no party affiliation and permits changes in party membership. If voters could belong to more than one party, then the multiplicity could easily be changed. Parties are instances of class *Political party* and need not be explicitly listed in the model; new parties can be added without changing the model and attributes can be attached to parties.

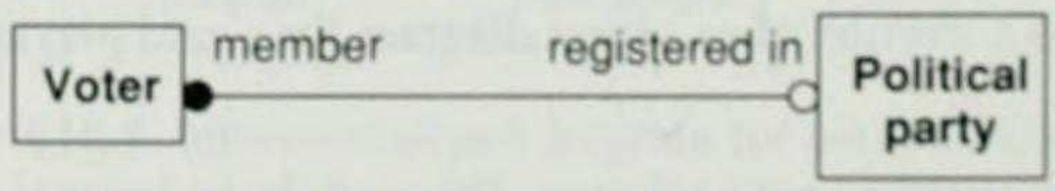


Figure A10.22 Improved object diagram for representing voter membership in a political party

10.24 The left figure shows an index on points using a doubly-qualified association. The association is sorted first on the *x* qualifier and then on the *y* qualifier. Because the index is an optimization, it contains redundant information also stored in the point objects. The right figure shows the same diagram using singly-qualified associations. We had to introduce a dummy class *Strip* to represent all points having a given x-coordinate. The second model would be easier to implement on most systems because a data structure for a single sort key is more likely to be available. The actual implementation could use B-trees, linked lists, or arrays to represent the association.

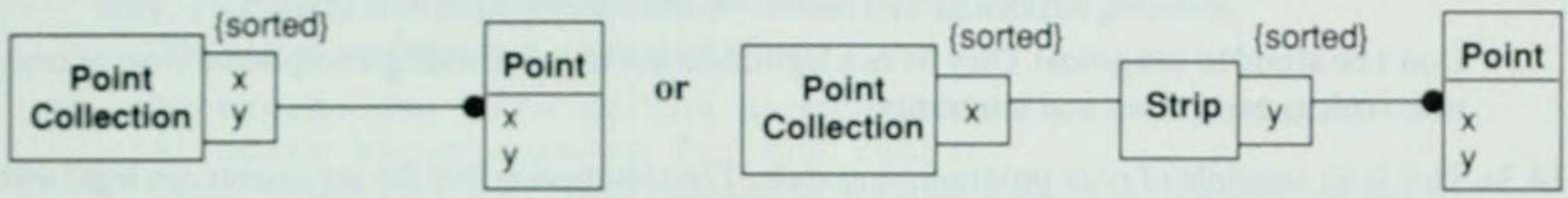


Figure A10.24 Object diagram for fast two-dimensional searching

The code listed below specifies search, add, and delete methods.

```
PointCollection::search (region: Rectangle): Set of Point
{
      make a new empty set of points;
      scan the x values in the association until x ≥ region.xmin;
      while the x qualifier ≤ region.xmax do:
        scan the y values for the x value until y ≥ region.ymin;
        while the y qualifier ≤ region.ymax do:
          add (x,y) to the set of points;
          advance to the next y value;
        advance to the next x value;
      return the set of points;
}
PointCollection::add (point: Point)
{
      scan the x values in the association until x ≥ point.x;
      if x = point.x then
        scan the y values for the x value until y ≥ point.y
      insert the point into the association;
}
PointCollection::delete (point: Point)
{
      scan the x values in the association until x ≥ point.x;
      if x = point.x then
        scan the y values for the x value until y ≥ point.y
        if y = point.y then
          for each collection point with the current x,y value:
            if collection point = point, delete it and return
      return error, point not found
}
```

Note that the scan operation should be implemented by a binary search to achieve logarithmic rather than linear times. A scan falls through to the next statement if it runs out of values.

10.27 Pseudo code for a garage door opener is listed below.

```
<closed>      wait for depress event
<opening>     start opening door
              wait for door open event
<open>        wait for depress event
<closing>     start closing door
              wait for either depress or door closed event:
                if depress event then goto opening
                if door closed event then goto closed
```

Don't be afraid to use gotos! They have a legitimate use in representing exceptional flow of control, such as exceptions and interrupts.

14.3a. This is an example of poor programming style. The assumption that the arguments are legal and the functions called are well behaved will cause trouble during program test and integration. The following statements will cause the program to crash if the argument to *strlen* is zero:

```
root_length = strlen(root_name);
suffix_length = strlen(suffix);
```

The following statement will assign zero to *sheet_name* if the program runs out of memory, causing a program crash during the call to *strcpy* later in the function:

```
sheet_name = malloc(root_length + suffix_length + 1);
```

The following statements will cause the program to crash if any of the arguments are zero:

```
sheet_name = strcpy(sheet_name, root_name);
sheet_name = strcat(sheet_name, suffix);
```

If *sheet_type* is invalid the switch statement will fall through leaving *sheet* without an assigned value. Also, it is possible that the call to *vert_sheet_new* or the call to *horiz_sheet_new* could return zero for some reason. Either condition would make it possible for the following statement to crash:

```
sheet->name = sheet_name;
```

15.5a. An interpreter provides a convenient way to quickly find out how a subroutine in a library behaves, bypassing the edit, compile, link, and execute cycle encountered in compiled languages. Some languages have both a compiler and an interpreter. The interpreter is used for rapid program development and the compiler is used to produce an efficient final version of the program.

b. A system builders is recommended to avoid this kind of error. It assures that all of the required steps and only the required steps are executed to rebuild an application after you modify one or more source files. Otherwise you are faced with the choice of rebuilding everything, which is time consuming, or trying to remember which steps are affected by your changes, which is error prone.

c. A symbolic debugger is a convenient tool for diagnosing this type of problem. It allows you to run your program until the fault condition occurs and then see the line where the fault is occurring as well as the values of program variables. This particular problem is likely to show up in the function described in Exercise 14.3. It appears that the program is running out of memory.

d. A change control system is an excellent way of coordinating a team software project.

e. A system builder solves this problem.

f. Some languages provide a solution to this problem. In C, for example, there is a #line construct that a preprocessor could insert into its output to point to a line in another file.

15.6a. Because of tight memory requirements a text segment should be deallocated when it is no longer needed. The problem is to develop a uniform policy that determines when segments are no longer needed. Classes and methods provide a convenient framework for a solution. One set of guidelines is to place the responsibility for memory reclamation with the methods which modify text segments. Access to text segments outside of the object which owns them should be read only. Temporary text segments should be deallocated as soon as possible.

One way to combine text segments follows:

```
Determine the size of the segments that are to be combined.
Allocate enough memory for the result.
Copy the original text segments into the allocated memory.
Deallocate the memory assigned to the original text segments.
```

b. For a multipass compiler it is a good strategy to simply let the operating system allocate a large amount of virtual memory. The amount of memory needed depends on the size of the source code being compiled. Because programs are partitioned into modest sized modules it is possible to place a reasonable upper bound on the amount of memory needed.

c. You cannot let the operating system allocate a large amount of virtual memory and forget about garbage collection in systems that run indefinitely. Eventually all memory will be consumed. Memory must be deallocated when objects are no longer referenced.

d. The first approach, deallocating memory within the subroutine that allocated it in the first place, guarantees that memory is reclaimed without burdening other routines with the responsibility of determining when a block of memory is no longer needed. However, doing so complicates recursive procedures.

17.14 Object diagram for Figure E17.9.

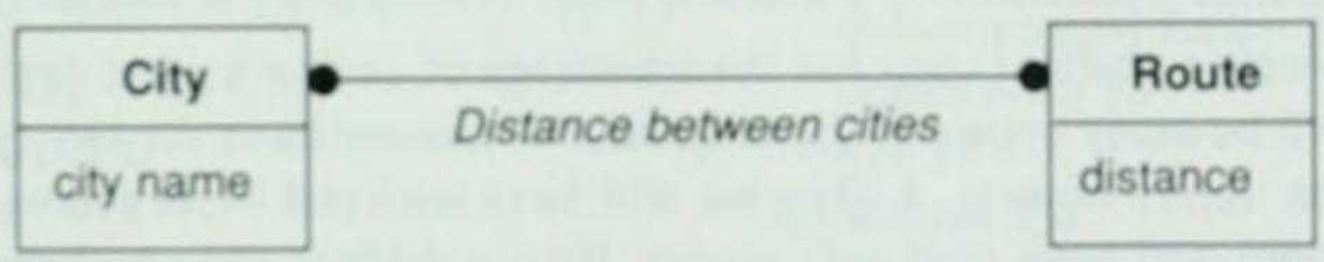


Figure A17.14 Object diagram for distance between cities

17.15 The following SQL code determines the distance between two cities for Figure E17.9.

```
select distance
from   Route, City C1, City C2,
       Distance-between-cities D1, Distance-between-cities D2
where  D1.route-ID = D2.route-ID and
       D1.route-ID = Route.route-ID and
       D1.city-ID = C1.city-ID and
       D2.city-ID = C2.city-ID and
       C1.city-name = :given-name1 and
       C2.city-name = :given-name2;
```

17.16 Object diagram for Figure E17.10.

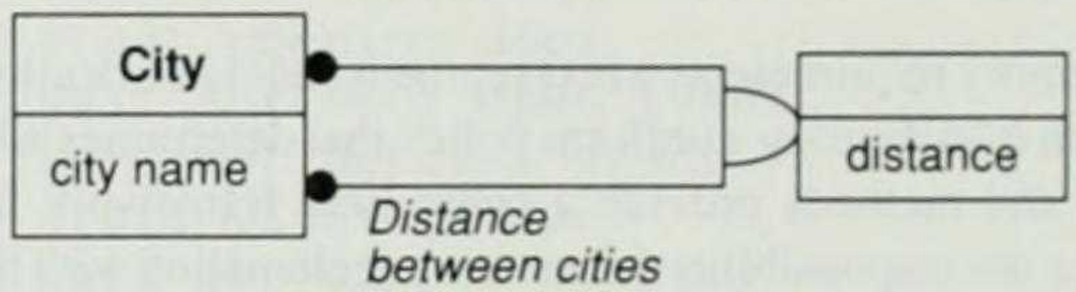


Figure A17.16 Object diagram for distance between cities

17.17 The following SQL code determines the distance between two cities for Figure E17.10.

```
select distance
from   City C1, City C2, Distance-between-cities D
where  D.city1-ID = C1.city-ID and
       D.city2-ID = C2.city-ID and
       C1.city-name = :given-name1 and
       C2.city-name = :given-name2;
```

17.18 We make the following observations about Figure A17.14 and Figure A17.16.

- Figure A17.16 is awkward because of the symmetry between city1 and city2. Either data must be stored twice with waste of storage, update time, and possible integrity problems, or special application logic must enforce an arbitrary constraint.
- Figure A17.14 has an additional table.

All in all, Figure A17.14 is a much better approach. With a self association it is frequently helpful to promote the association to a class in order to break the symmetry.

20.3 Shallow copy methods use less space than deep copy methods. Subsequent changes to the original are automatically inherited by a shallow copy. A deep copy is independent of the original.

a. For library objects a shallow copy has the advantage of saving a great deal of space since it is likely there will be many copies of the original. We assume there are two types of applications that will access library objects. A librarian will have read and write access to the original. All other applications will have read only access. With a shallow copy a change in the original is automatically reflected in the copies. This may be desirable in some applications and undesirable in others. A danger of the shallow copy is that a mistake while using the librarian program could have disastrous results.

b. Because changes in the new sheet should not affect the old sheet a deep copy must be used.

c. Either a shallow copy or a deep copy is suitable for copying selected items from one place on a sheet to another, depending on the desired behavior. Use a shallow copy if the copies should inherit changes made to the original. Use a deep copy if it is desired to change the copies independently.

Index

A

abstract class 61–62
  in animation case study 427
abstract operation 62
abstracting out a common superclass 243
abstraction 7, **16**
access operation 131
access to attributes 321. *See also* private, protected, public
action **92–93**, 131–132
  entry and exit 101–102
  internal 102
  notation for 93
  on a state 101
  on a transition 93
active values 238
activity **92**, 132
  notation for 92
actor object (in a DFD) 126
Ada language 324
  associations 357
  class definitions 343
  encapsulation 322, 360
  implementation in 341
  inheritance 349
  instantiation 346
  methods 345
  method resolution 354
aggregate:
  fixed 59
  propagation of an operation upon 60
  recursive 59
  variable 59
aggregation 36–38, **57–61**
  versus association 58
  and concurrency 99
  in database schema 380
  versus generalization 58
  notation for 38
  tree 38
algorithms 254
  design of 230–235
allocation:
  of operations to classes 234–235
  of subsystems to processors 203–204
analysis 5, 145, **148–188**, 260–262
  of access paths 236
  in animation case study 420–423
  of association traversal 245
  in compiler case study 401–406
  document 262
  iteration of 185
  model 149
  in OLIE case study 436–443
  summary of 261–262
  transformation into design 228
ancestor class 39

animation case study 416–431
architectural frameworks 211–216
architecture 198, **211–216**
in the ATM example 217
multiple processor 203
result of system design 198
three schema 367–368
assertion **324**, 328
association **27–38**, 156–161
in Ada 357
analyzing traversal of 245
in C 356
in C++ 313
as a class 33
in database schema 376–382
derived 159
design of 245–248
directionality 27
in Eiffel 315
in Fortran 358
identifying 156–161
importance of 31
modeled as a class 33
multiplicity of **30**, 160
in non-OO language 355–358
notation for 28
in an OO language 312–313
ordered 35
qualified **35**, 160, 162
redundancy to improve speed 235
role names 34–35
in Smalltalk 317
ternary **28**, 159
traversal of 34, 246
ATM example 151–185
architecture 217
implementation of control 240
problem statement 151
attribute **23–25**, 161–163
access to 321
class 71
derived **26**, 34, 35, 75
event attribute 85
identifying 161–163
link attribute **31–32**, 163
notation for 24
private 249
automatic transition 103

B

base attribute **26**, 131
batch transformation 212–213
in object diagram compiler 407
boundary conditions 210
browser 323
building tools 323

C

C language:
in animation case study 428
association 356
class definitions 342
encapsulation 359
implementation in 341
inheritance 348
instantiation 346
method 344
method resolution 351
C++ language 324, **326**
association 313
calling operations 305
class definitions 299
creating objects 301
encapsulation 322
inheritance 308–309
candidate key 71–72
case statements, avoidance of 286
case studies:
computer animation 416–431
electrical distribution design system 433–448
object diagram compiler 397–413
change control tools 323
class 2, **22**, 153–156
abstract 61–62
in Ada 343
ancestor 39
in C 342
in C++ 299
concrete 61
in database schema 375–376
descendent 39
in Eiffel 300
in Fortran 343
identifying 153–156

class:
in non-OO language 342–344
notation for 23
as an object (metaobject) 71
in Smalltalk 300
class attribute 71
class descriptors 71
class diagram 23
class feature 71
class library 318–319
Eiffel 315, **327**
ET++ (C++) 326
Interviews (C++) 326
NIH (C++) 314, **326**
Smalltalk 317, **325**
class method (Smalltalk). *See* class operation
class operation 71, 304
class variable (Smalltalk). *See* class attribute
classification 2
client **138**, 230
client-supplier relationship 200
CLOS 25, 318, 320, 321, 322, **328**, 332
clustering 79
coherence of entities 250
Common Lisp Object System. *See* CLOS
complexity, analysis of 254
concrete class 61
concurrency **202**, 241
and aggregation 99
implementation of 358
among objects 99
within an object 99
synchronization 104
concurrent system 209
condition 91–92
notation for 91
configuration problems, solution of 243
constraint **73–75**, 132, 324
identifying 182
on links 74
notation for 74
container class **233**, 319
contour 97
contractual model of programming 328
control 18, **84**
implementation of 207–210, **239–241**
in ATM example 240
control flow 129

D

data dictionary 156
data flow 126
data flow diagram (DFD) 18, **124–129**
construction of 180–182
nesting of 129
data management 205–206
data model 367
data store 127–128
data structures 254
selection of 233
database (DBMS) 366–389
advanced relational 387–388
design of schema 373–387
versus files 205–206
integrity of information 370–371
object-oriented 329–330
relational 368–371
debugger 323
debugging code 288
declaration of types for efficient code 320
decomposition 168, **199–201**
delegation 284–285
in animation case study 426
to avoid improper inheritance 244
as a substitute for multiple inheritance 67
Demeter, Law of 286
derived association 159
derived attribute **26**, 34, 35, 75, 131
in OMTool example 237
derived entity 75
derived link 75
derived object 75
in OMTool example 237
derived values, update of 238
descendent class 39
design 227–254
conversion into OO language 296–313
of database schema 373–387
of database systems 329
derivation from analysis model 228
document 263
documentation of 251–252
implementation of 278–280
object design phase 227–254
optimization of 235–238
trade-offs, prioritizing 210

DFD. *See* data flow diagram
diagram:
class 23
data flow 180–182
event flow 173
instance 23
object 23
state 85, **89–91**, 173–179
directionality of association 27
discriminator 39
documentation:
of analysis **186**, 262
of code 289
of design **251–252**, 263
of system design 262
DSM language 313, 318, 320, 323, 332
in compiler case study 412
dynamic binding. *See* method resolution
dynamic model 6, 17, 18, **84–113**, 261
in animation case study 422
completeness and consistency of 179
and concurrency 202
construction of 169–179
implementation of 207–210, **239–241**
in OLIE case study 441
relationship to other models 110

E

edge triggered event 103
efficiency:
of design 235–238
of OO languages **319**, 327
periodic recomputation of derived attributes 238
rearranging execution order 237
saving attributes to avoid recomputation 237
setting trade-off priorities 210
trade-offs versus flexibility 232
using an index 236–237
Eiffel language 318, 324, **327**
association 315
calling operations 306
class definitions 300
class library 315, **327**
creating objects 303
encapsulation 321
encapsulation **7**, 31, 285. *See also* information hiding
in Ada 360
in C 359
in Fortran 361
language support for 321
entity integrity 370
Entity-Relationship (ER) diagram 267, **271**
environment for program development 322
ET++ class library 326
event **85–86**, 169–179
identifying 173
notation for 89
sending 101, 103
event flow diagram 173
event generalization 98
notation for 98
event trace 86
construction and use of 173
event-driven system 208
EXODUS (DBMS) 388
extensibility:
improved by use of inheritance 243
programming tips for 285–286
extension 63–65

F

factoring out common code 283–284
feature 26
class feature 71
in Eiffel 300
finite state machine. *See* state diagram
firing of transition 89
fixed aggregate 59
flight simulator example 133–137
foreign key 370
Fortran:
association 358
class definitions 343
encapsulation 361
implementation in 341
inheritance 350
instantiation 347
method 345
method resolution 355
frameworks, architectural 211–216
friend function (C++) 312, 322

function:
describing during analysis 182
as an operation on an object 184
functional model 6, 17, 18, **123–140**, 261
in animation case study 423
in compiler case study 402
construction of 179–183
in OLIE case study 442
relationship to other models 137

G

garbage collection 320–321
GemStone 330
generalization 38–43. *See also* inheritance
in database schema 382–385
event generalization 98
as extension and restriction 63–65
notation for 39
state generalization 96–98
uses of 41–42
generics. *See* parameterized class
guarded transition 91
notation for 91
guardian object 207
guidelines for programming 281–292

H

hardware requirements, estimation of 203
higraph 114
homomorphism 76–77

I

identifying:
associations 156–161
attributes 161–163
classes 153–156
identity 2, **22**, 48
impact of OO methodology 146
implementation 5, **278–280**
in animation case study 428–430
in compiler case study 412
in a database system 279–280
in OLIE case study 448
implementation method 250
index, improving efficiency by use of 236–237
information hiding 7, **249–250**, 321. *See also* encapsulation
information modeling notations 271
inherent concurrency 202
inheritance 3, **38–43**. *See also* generalization
abstracting out common behavior 242–244
in Ada 349
during analysis 163–165
avoiding improper use of 244, **284–285**
in C 348
in C++ 308
limitations on 326
in database schema 382–385
in Eiffel 310
of events 98
in Fortran 350
increasing during design 242–244
multiple inheritance **65–69**, 164
notation for 39
rearranging classes and operations 242
in Smalltalk 311
of states 97
instance 2, **22**
notation for 70
instance diagram 23
instance variable (Smalltalk). *See* attribute
instantiation 69
in Ada 346
in C 346
in Fortran 347
in non-OO language 345
notation for (in a DFD) 128
integrity of data in DBMS 370–371
interactive interface 214–215
interface class 319
Interviews 326
invariant **133**, 324, 328
is-a. *See* generalization
iteration during modeling 166

J

Jackson Structured Development (JSD) 268–271
join class 65

K

key, foreign or primary 370
a-kind-of. *See* generalization

L

lambda transition 103
language:
  Ada 341
  C 341
  C++ 326
  CLOS 328
  comparison of OO 330
  database 329–330, **366–389**
  DSM 409
  Eiffel 327
  Fortran 341
  Lisp 342
  non-object-oriented 340–363
  object-oriented **296–333**
  Pascal 342
  Smalltalk 325
  SQL **369**, 400
  Trellis-Owl 332
layer 200
leveling 129
life cycle 6, **144**
  analysis 145, **148–188**, 260–262
  for database applications 367
  impact of OO approach 146
  maintenance 144
  object design 145, **227–254**, 263
  system design 145, **198–219**, 263
  testing 144
link 27
  constraints on 74
  derived 75
  notation for 28
link attribute **31–32**, 163
  implementation of 248
  notation for 32
Lisp, implementation in 342. *See also* CLOS
lock 207

M

maintenance 144
make (system-building tool) 323
member (C++) 299. *See also* feature
member function (C++). *See* method
memory management **320–321**, 327
metaclass 71
metadata 69–71
meta-information 323
metamodel in compiler case study 406
method 3, **25**
  in Ada 345
  in C 344
  in Fortran 345
  in non-OO language 344–345
method caching 319
method resolution 230, **319**
  in Ada 354
  in C 351
  in C++, limitation of 326
  in Fortran 355
  in non-OO language 351–355
  optimization of 319
methodology 144
  comparison of 266–274
model 15–19
  dynamic model 18, **84–113**, 169–179
  functional model 18, **123–140**, 179–183
  object model 17, **21–48**, 152–169
  relationship between 18, **137–139**
Model/View/Controller (Smalltalk) 325
modes, avoidance of 283
module **43**, 200, 251
  notation for 43
  partitioning during analysis 168
multiple inheritance **65–69**, 164
  accidental 66
  in database schema 384
  language support for 318
  substitutes for 67–69
  workarounds 67
multiplicity **30–31**, 160
  notation for 30

N

name
  importance of good 46, **288–289**
  as an object attribute 162
nested state diagram 94–98
NIH class library (C++) 314
normal forms 371

notation for:
action 93
entry and exit 101
internal 102
activity 92
actor object (in a DFD) 127
aggregation 38
argument list 25
association 28
as a class 33
attribute 24
class 23, 26
condition 91
constraint 74
control flow 129
data flow 126
data store 127
default value 24
derived entity 75
event 89
sending 103
event generalization 98
generalization 39
guarded transition 91
inheritance 39
instance 70
link 28
link attribute 32
module 43
multiplicity 30
no arguments 25
object instance 23
operation 25
in pseudocode 184
process (in a DFD) 125
qualified association 36
in pseudocode 184
qualifier 36
result type 25
role name 34
selection or creation of a DFD object 128
sheet 43
state 89, 93
concurrency in 99, 104
initial or final 91
nested 95-97
notation for:
state diagram 93
state generalization 97
synchronization 104
ternary association 28
transition 89
automatic 103
type 24
visibility of attributes 249

O

object 21–22
allocation in non-OO language 345
in database schema 375–376
derived 75
design of 248
identifying classes of 153–156
in non-OO language 342–344
object design 5, 145, **227–254**, 263
in animation case study 426–428
in compiler case study 408–412
in OLIE case study 445–448
summary of 263
object diagram **17**, 23
object diagram compiler 397–413
object ID:
in database design 374
unnecessary in object model 24
object instance:
notation for 23
object model 6, 17, **21–48**, 261
advanced concepts 57–79
in animation case study 421
avoiding assumptions about **249**, 285
in compiler case study 402, 404
completeness and consistency of 166
concepts and notation 21–48
construction of 152–169
conversion to database schema **373–387**, 397–413
in OLIE case study 436, 437
relationship to entity-relationship (ER) 48
relationship to other models 110, 136
object modeling:
practical tips 46
Object Modeling Technique. *See* OMT

Objective-C language 323, 332
object-oriented, meaning of 1
object-oriented development 4
  impact on life cycle 146
object-oriented language 296–333
  comparison chart 331
  features of 318–325
object-oriented methodologies 273
OMT (Object Modeling Technique) 1, 5, **16–18**
OMT methodology:
  comparison with others 266–274
  preview of 144–147
  summary of 260–264
OMTool example 229–234
OO. *See* object-oriented
operation 2, **25–26**
  abstract 62
  in C++ 305
  in a DFD 130–132
  in Eiffel 306
  identifying 183–185
  notation for 25
  overriding 64–65
  query 26
  shopping list 185
  in Smalltalk 307
  specification of 130–132
  in a state diagram 92
optimization:
  avoiding excessive 287
  of design 235–238
  specifying criteria for 183
ordering one-to-many associations 35
origin class 62
ORION 330
OSCAR case study 416–431
overriding an operation 42–43, 64–65

P

package 322
parameterized class 324
partition 201
partitioning 168, **199–201**, 251
a-part-of. *See* aggregation
part-whole relationship. *See* aggregation
Pascal language, implementation in 342
peer-to-peer relationship 200
persistence of data 324
Petri nets 114
pointer 27, 31
policy method 250
policy versus implementation 250, **282–283**
polymorphism 2, **25**, 328
POSTGRES (DBMS) 388
preconditions and postconditions **324**, 328
primary key 370
priorities for design trade-offs 210
private access 297–301
private operations **286**, 322
problem statement 150–151
procedure-driven system 208
process:
  in a DFD 124
  implementation of 138
program structure chart 230
programming language:
  non-object-oriented 340–363
  object-oriented 296–333
programming style 281–292
programming-in-the-large 288–291
propagation of operations 60
protected access (C++) 309
protocol 62
  in animation case study 427–428
public access 297–301
public operations 286

Q

qualification 35–36
qualified association **35**, 160, 162
  in database schema 380
  notation for 36
qualifier, notation for 36
query operation **26**, 131
query to a database system 329

R

race condition 103
real-time system 216
recursive aggregate 59
redundancy to improve speed
  associations 235
  data 237

referential integrity 370
requirements **150–151**, 186
restriction 42, **63–65**
reusability:
  during analysis 169
  programming tips for 282–285
robustness, programming tips for 286–288
role 34–35
role name **34–35**, 160
  notation for 34
  uniqueness of 35

S

scenario 86, **170–172**
schema:
  for a database 367–368
  generation from object model **373–387**, 397–413
selection:
  as a data flow 128
  notation for (in a DFD) 128
self (Smalltalk) 307
self reference 307
service 199
sheet 43
  notation for 43
  partitioning during analysis 168
shopping list operations 185
signature **25**, 42, 130
simulation, dynamic 215
Smalltalk 318, 323, **325**, 332
  association 317
  calling operations 307
  class definitions 300
  class library 317
  creating objects 304
  encapsulation 321
specialization 42
specification:
  of interfaces 289
  OMT used for 280
state 84-112
  concurrency in 99–100, 104–105
  generalization 96–98
  implementation of 239–241
  initial and final 91
  notation for 89, 93, 97
state:
  substate 97
  superstate 97
state diagram 18, 85, **89–91**, 267
  construction of 173–179
  conversion to code 239
  level of abstraction 88
  nested 94–98
  one shot versus continuous 91
state machine engine 241
state tree 114
  in OLIE case study 445–446
statechart 113
static binding 319
strong typing 320
structure chart 230, 267
Structured Analysis/Structured Design (SA/SD) 266–268
subclass 3, **39**
substate 97
subsystem 199–201
superclass 3, **39**
superstate 97
supplier **138**, 230
system architecture 198, **211–216**
system design 5, 145, **198–219**, 262
  in animation case study 424–425
  in compiler case study 407–408
  document 262
  in OLIE case study 444–445
  summary of 262
system topology **201**, 204, 217

T

table (in RDBMS) 369
terminator 127
ternary association **28**, 159
  in database schema 380–382
  notation for 28
testing for completeness and consistency:
  of dynamic model 179
  of object model 166
testing phase of development 144
this (C++) 305
thread of control **202**, 230
three models, combining 229
three schema architecture **367–368**, 373–374

trade-offs:
database versus flat files 205–206
efficiency versus flexibility 203, 232
hardware versus software 203
objects versus primitive types 248
setting priorities for 210
transaction manager 216
transformation:
continuous 213–214
transition 89
automatic 103
guarded 91
lambda 103
notation for 89
Trellis-Owl language 318, 321, 332
triggering operations 60

U

user interface:
design of 214
specification of 172

V

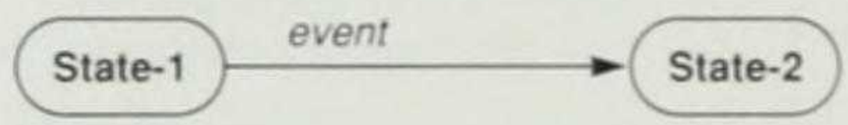
variable aggregate 59
view in DBMS 371
virtual function (C++) 326
visibility:
of associations **249**, 285
of attributes 321
of classes 322
of operations **286**, 322

W

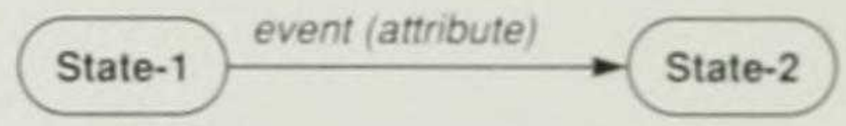
weak typing 320

Dynamic Model Notation

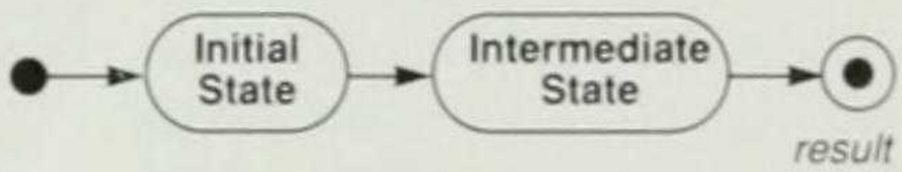
Event causes Transition between States:



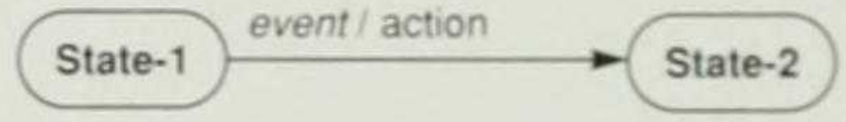
Event with Attribute:



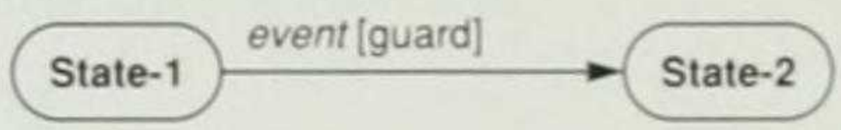
Initial and Final States:



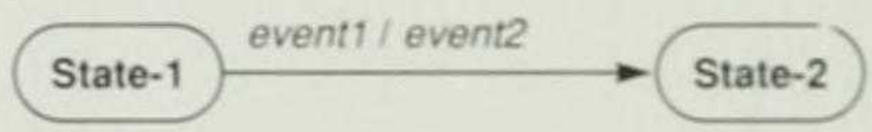
Action on a Transition:



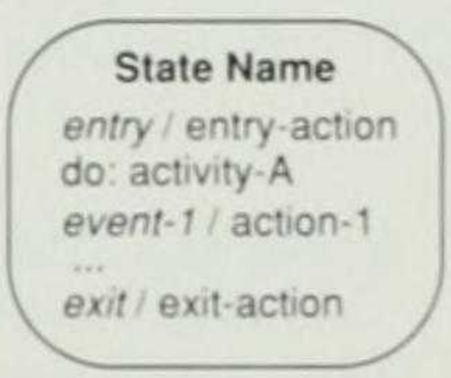
Guarded Transition:



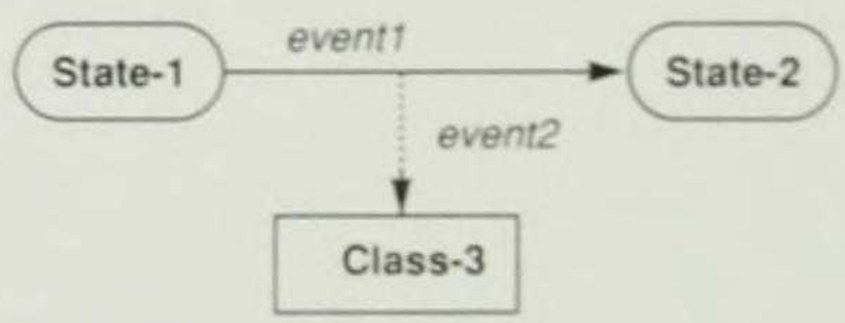
Output Event on a Transition:



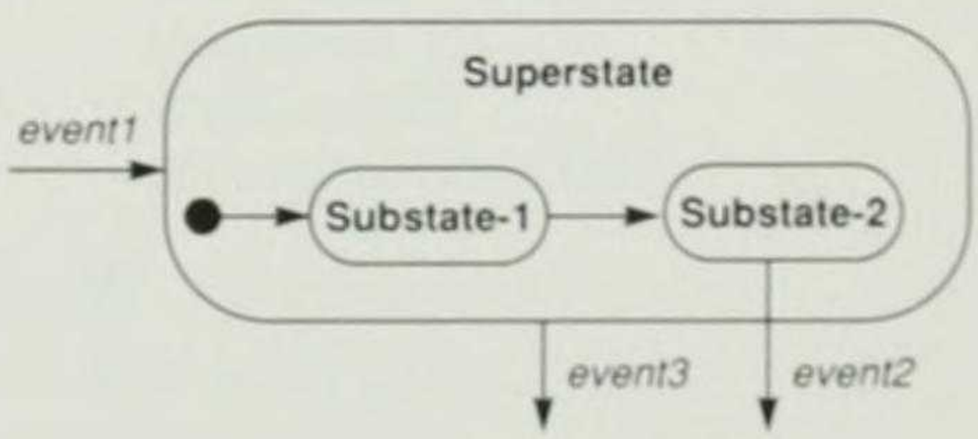
Actions and Activity while in a State:



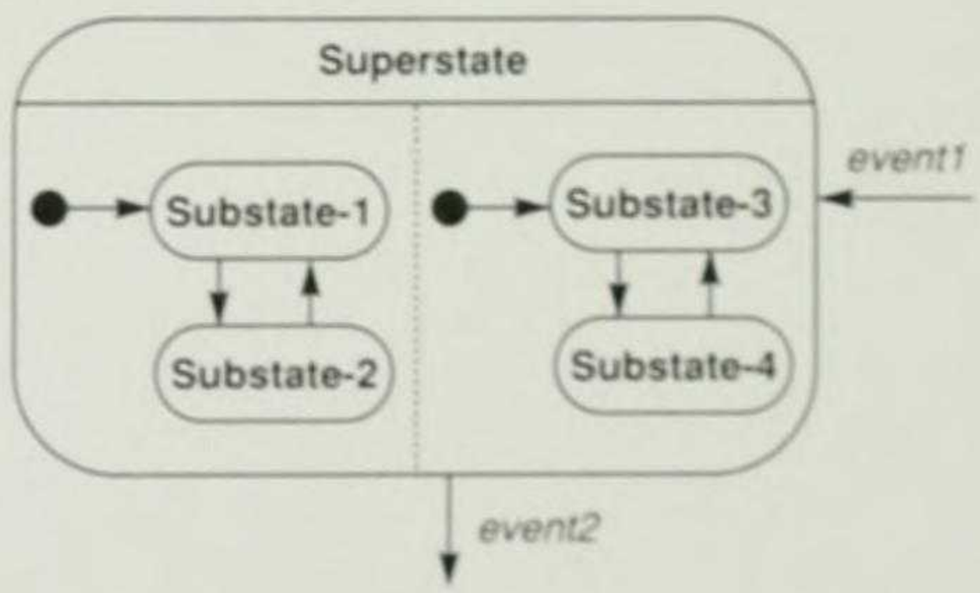
Sending an event to another object:



State Generalization (Nesting):

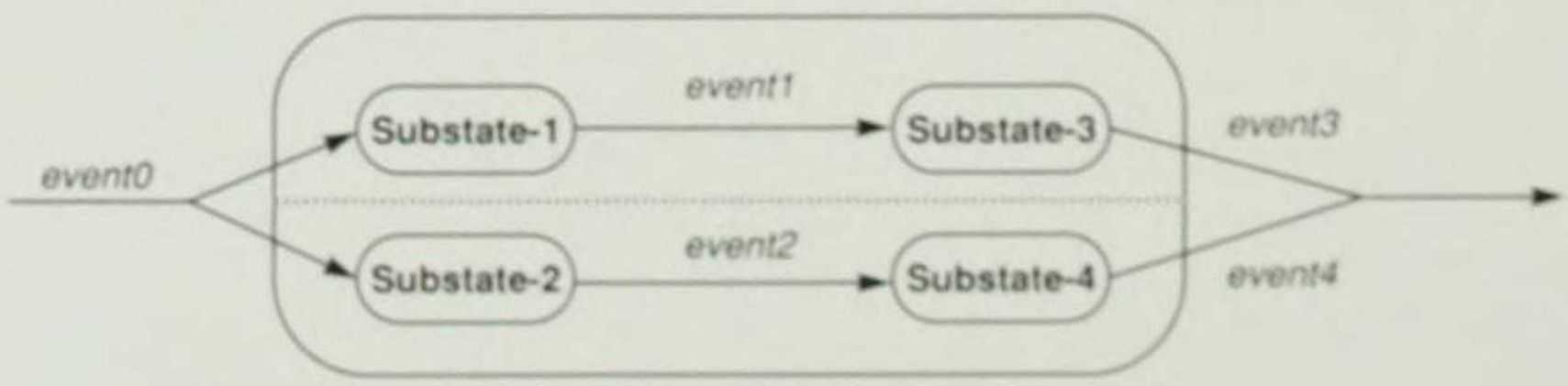


Concurrent Subdiagrams:



Splitting of control:

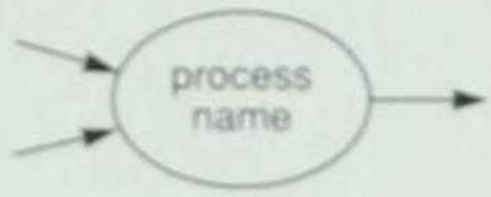
Synchronization of control:



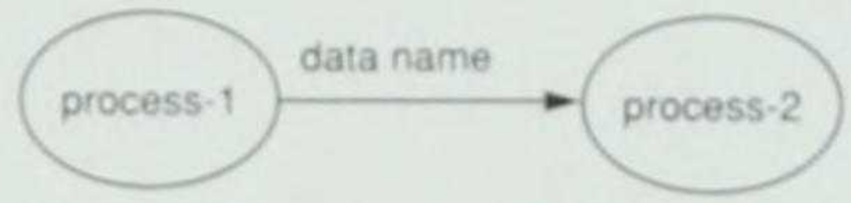
This page may be freely copied without obtaining permission from the publisher.

Functional Model Notation

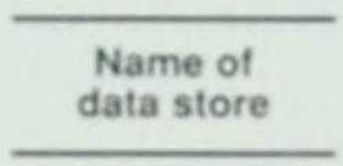
Process:



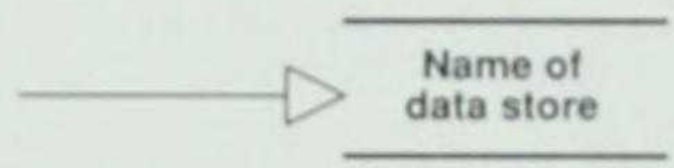
Data Flow between Processes:



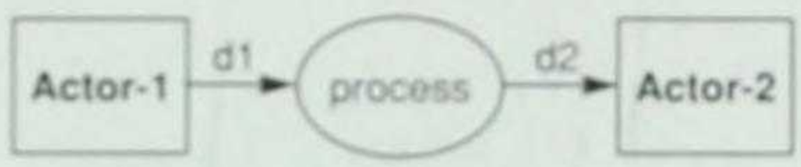
Data Store or File Object:



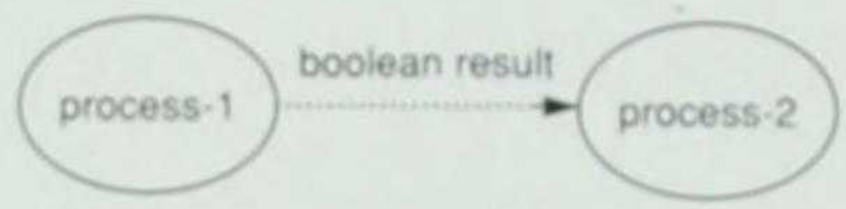
Data Flow that Results in a Data Store:



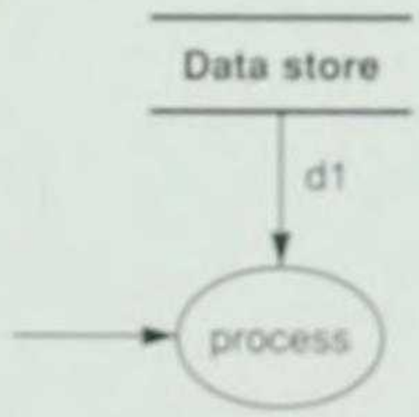
Actor Objects (as Source or Sink of Data):



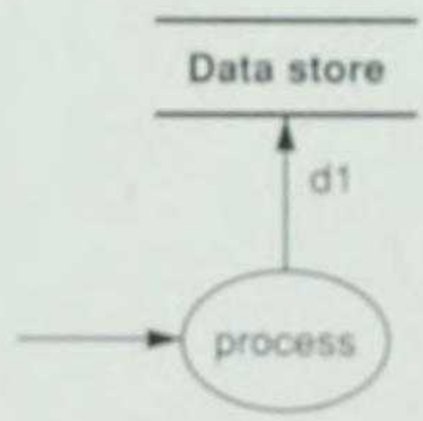
Control Flow:



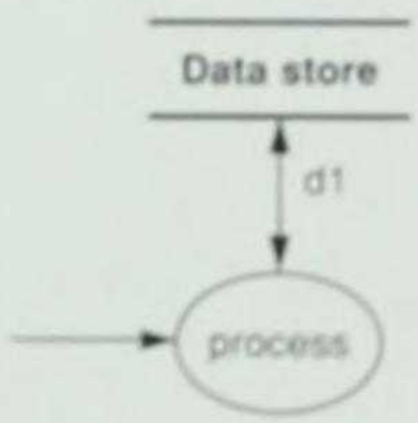
Access of Data Store Value:



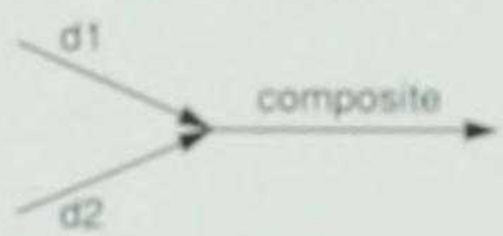
Update of Data Store Value:



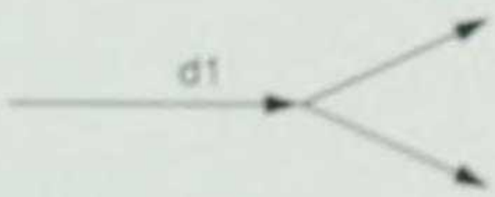
Access and Update of Data Store Value:



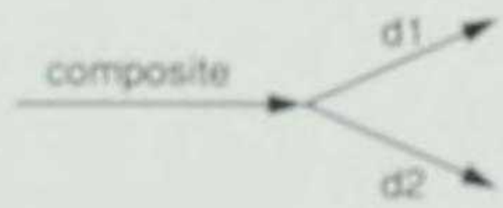
Composition of Data Value:



Duplication of Data Value:



Decomposition of Data Value:



This page may be freely copied without obtaining permission from the publisher.

OBJECT-ORIENTED MODELING AND DESIGN

JAMES RUMBAUGH / MICHAEL BLAHA / WILLIAM PREMERLANI
FREDERICK EDDY / WILLIAM LORENSEN

Object-oriented modeling and design promote better understanding of requirements, cleaner designs, and more maintainable systems. Often, books on related subjects rely too heavily on programming and coding—forcing readers to think in terms of the computer, and not the application.

Object-Oriented Modeling and Design emphasizes that object-oriented technology is *more* than just a way of programming. It applies techniques to the entire software development cycle. This volume presents a new object-oriented software development methodology—from analysis, through design, to implementation.

Significant features:

- focuses on high-level, front-end conceptual processes of analysis and design, rather than just on the low-level, back-end implementation steps of programming.
- covers the entire development life cycle—analysis, design, implementation—without a change of notation at each stage.
- presents graphical notation and methodology independent of any particular programming language.
- includes case studies of industrial object-oriented applications developed by the authors.
- offers extensive examples and exercises that bring out fine points, summary lists of concepts and methodology steps, and almost 300 diagrams.

Object-Oriented Modeling and Design offers the best in practical advice and examples for industrial applications.

$ 55.33

ISBN 0-13-629841-9
90000
9 780136 298410

PRENTICE HALL
Englewood Cliffs, N.J. 07632