a. Use the language in Figure E14.1 to describe the class diagram in Figure E14.2.
b. Discuss similarities and differences between data in storage and data in motion. For example, the description you prepared in the previous part could be used to store a class diagram in a file or to transmit a diagram from one location to another.
c. The language in this problem is used to describe the structure of class diagrams. Invent a language to describe two-dimensional polygons. Use BNF to describe your language. Describe a square and a triangle in your language.

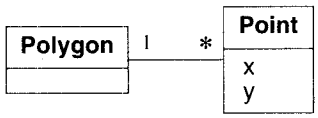

Figure E14.2 Class diagram of polygons

14.13 (6) A common problem encountered in digital systems is data corruption due to noise or hardware failure. One solution is to use a cyclic redundancy code (CRC). When data is stored or transmitted, a code is computed from the data and appended to it. When data is retrieved or received, the code is recomputed and compared with the value that was appended to the data. A match is necessary but not sufficient to indicate that the data is correct. The probability that errors will be detected depends on the sophistication of the function used to compute the CRC. Some functions can be used for error correction as well as detection. Parity is an example of a simple function that detects single-bit errors.

The function to compute a CRC can be implemented in hardware or software. The choice for a given problem is a compromise involving speed, cost, flexibility, and complexity. The hardware solution is fast, but may add unnecessary complexity and cost to the system hardware. The software solution is cheaper and more flexible, but may not be fast enough and may make the system software more complex.

For each of the following subsystems, decide whether or not a CRC is needed. If so, decide whether to implement the CRC in hardware or software. Explain your choices.

a. floppy disk controller
b. system to transmit data files from one computer to another over telephone lines
c. memory board on a computer board in the space shuttle
d. magnetic tape drive
e. validation of an account number (a CRC can be used to distinguish between valid accounts and those generated at random)

14.14 (6) Consider the scheduler software in Exercises 12.16–12.19 and 12.20–12.23.

With scheduling software it is also important to manage security—that is, the schedules that each user is permitted to read and write.

An obvious way to maintain security is to maintain a list of access permissions for each combination of user and schedule. However, this can become tedious to monitor and maintain.

Another solution is to allow permissions to be entered also for a group. A user can belong to multiple groups; each group may have multiple users and lesser groups. The users may access schedules for which they have permission or for which their groups have permission.

Extend the class models from Exercises 12.19 and 12.23 for this model of security. (Instructor's note: You should give the students our answers to Exercises 12.19 and 12.23.)

15

Class Design

The analysis phase determines what the implementation must do, and the system design phase determines the plan of attack. The purpose of class design is to complete the definitions of the classes and associations and choose algorithms for operations.

This chapter shows how to take the analysis model and flesh it out to provide a basis for implementation. The system design strategy guides your decisions, but during class design, you must now resolve the details. There is no need to change from one model to another, as the OO paradigm spans analysis, design, and implementation. The OO paradigm applies equally well in describing the real-world specification and computer-based implementation.

15.1 Overview of Class Design

The analysis model describes the information that the system must contain and the high-level operations that it must perform. You *could* prepare the design model in a completely different manner, with entirely new classes. Most of the time, however, the simplest and best approach is to carry the analysis classes directly into design. Class design then becomes a process of adding detail and making fine decisions. Moreover, if you incorporate the analysis model during design, it is easier to keep the analysis and design models consistent as they evolve.

During design, you choose among different ways to realize the analysis classes with an eye toward minimizing execution time, memory, and other cost measures. In particular, you must flesh out operations, choosing algorithms and breaking complex operations into simpler operations. This decomposition is an iterative process that is repeated at successively lower levels of abstraction. You may decide to introduce new classes to store intermediate results during program execution and avoid recomputation. However, it is important to avoid overoptimization, as ease of implementation, maintainability, and extensibility are also important concerns.

OO design is an iterative process. When you think that the class design is complete at one level of abstraction, you should consider the next lower level of abstraction. For each

level, you may need to add new operations, attributes, and classes. You may even need to revise the relationships between objects (including changes to the inheritance hierarchy). Do not be surprised if you find yourself iterating several times.

Class design involves the following steps.

- Bridge the gap from high-level requirements to low-level services. [15.2]
- Realize use cases with operations. [15.3]
- Formulate an algorithm for each operation. [15.4]
- Recurse downward to design operations that support higher-level operations. [15.5]
- Refactor the model for a cleaner design. [15.6]
- Optimize access paths to data. [15.7]
- Reify behavior that must be manipulated. [15.8]
- Adjust class structure to increase inheritance. [15.9]
- Organize classes and associations. [15.10]

15.2 Bridging the Gap

Figure 15.1 summarizes the essence of design. There is a set of features that you want your system to achieve. You have a set of available resources. Think of the distance between them as a gap. Your job is to build a bridge across the gap. There are several sources of high-level needs: use cases, application commands, and system operations and services. Resources include the operating system infrastructure, class libraries, and previous applications.

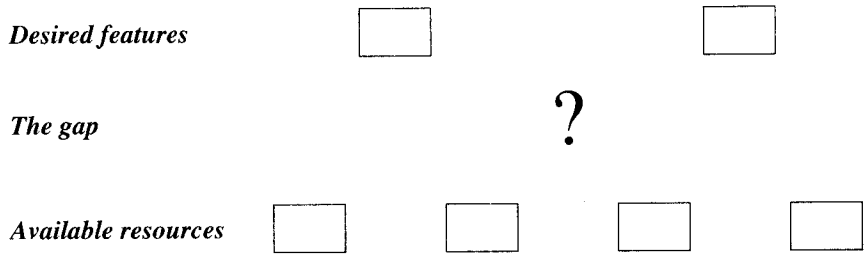

Figure 15.1 The design gap. There is often a disconnect between the desired features and the available resources.

If you can directly construct each feature from the resources, you are done. For example, a salesman can use a spreadsheet to construct a formula for his commission based on various assumptions. The resources are a good match for the task.

But usually it's not so easy. Suppose you want to build a Web-based ordering system. Now you cannot readily build the system from a spreadsheet or a programming language, because there is too big a gap between the features and the resources. You must invent some intermediate elements, so that each element can be expressed in terms of a few elements at

the next lower level (Figure 15.2). Furthermore, if the gap is large you will need to organize the intermediate elements into multiple levels. The intermediate elements may be operations, classes, or other UML constructs. Inventing good intermediate elements is the essence of successful design.

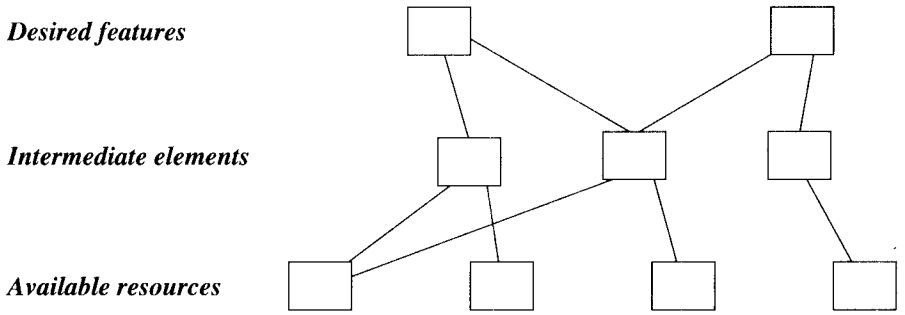

Figure 15.2 Bridging the gap. You must invent intermediate elements to bridge the gap between the desired features and the available resources.

Often the intermediate elements are not obvious. There can be many ways to decompose a high-level operation. You must guess at a likely set of intermediate operations, then try to build them. Be alert to intermediate operations that are similar but not identical. You can reduce the size of your code and increase its clarity by folding these similar operations into a smaller number of common operations. These reworked operations may be less than ideal for some of the higher-level operations. You may have to compromise, because a good design optimizes an entire system and not each separate decision.

If the intermediate elements have already been built, you can just use them, but the principle of bridging the gap is the same. You still have to find the elements in a framework or a class library, select them, and fit them together. The problem isn't making up individual elements—anybody can do that well. The problem is to fit the entire system together cleanly.

Design is difficult because it is not a pure analytic task. You cannot merely study system requirements and derive the ideal system. There are far too many choices of intermediate elements to try them all, so you must apply heuristics. Design requires synthesis: You have to invent new intermediate elements and try to fit them together. It is a creative task, like solving puzzles, proving theorems, playing chess, building bridges, or writing symphonies. You can't expect to push a button or follow a recipe and automatically get a design. A development process provides guidance, just as chess books and engineering handbooks and music theory courses help, but eventually it takes an act of creativity to produce a design.

15.3 Realizing Use Cases

In Chapter 13 we added major operations to the class model. Now during class design we elaborate the complex operations, most of which come from use cases.

Use cases define the required behavior, but they do not define its realization. That is the purpose of design—to choose among the options and prepare for implementation. Each choice has advantages and disadvantages. It is not sufficient merely to deliver the behavior, although that is a primary goal. You must also consider the consequences of each choice on performance, reliability, ease of future enhancement, and many other "ilities". Design is the process of realizing functionality while balancing conflicting needs.

Use cases define system-level behavior. During design you must invent new operations and new objects that provide this behavior. Then, in turn, you must define each of these new operations in terms of lower-level operations involving more objects. Eventually you can implement operations directly in terms of existing operations. Inventing the right intermediate operations is what we have called "bridging the gap."

To start, list the responsibilities of a use case or operation. A ***responsibility*** is something that an object knows or something it must do [Wirfs-Brock-90]. A responsibility is not a precise concept; it is meant to get the thought process going. For example, in an online theater ticket system, making a reservation has the responsibility of finding unoccupied seats to the desired show, marking the seats as occupied, obtaining payment from the customer, arranging delivery of the tickets, and crediting payment to the proper account. The theater system itself must track which seats are occupied, know the prices of various seats, and so on.

Each operation will have various responsibilities. Some of these may be shared by other operations, and others may be reused in the future. Group the responsibilities into clusters and try to make each cluster coherent. That is, each cluster should consist of related responsibilities that can be serviced by a single lower-level operation. Sometimes, if the responsibilities are broad and independent, each responsibility is in its own cluster.

Now define an operation for each responsibility cluster. Define the operation so that is not restricted to special circumstances, but don't make it so general that it is unfocused. The goal is to anticipate future uses of the new operation. If the operation can be used in several different places in the current design, you probably don't have to make it more general, except to cover the existing uses.

Finally, assign the new lower-level operations to classes. If there is no good class to hold an operation, you may need to invent a new lower-level class.

ATM example. One of the use cases from Chapter 13 is *process transaction*. Recall that a *Transaction* is a set of *Updates* and that the logic varies according to withdrawal, deposit, and transfer.

- **Withdrawal**. A withdrawal involves a number of responsibilities: get amount from customer, verify that amount is covered by the account balance, verify that amount is within the bank's policies, verify that ATM has sufficient cash, disburse funds, debit bank account, and post entry on the customer's receipt. Note that some of these responsibilities must be performed within the context of a database transaction. A database transaction ensures all-or-nothing behavior—all operations within the scope of a transaction happen or none of the operations happen. For example, the disbursement of funds and debiting of the bank account must both happen together.
- **Deposit**. A deposit involves several responsibilities: get amount from customer, accept funds envelope from customer, time-stamp envelope, credit bank account, and post en-

try on the customer's receipt. Some of these responsibilities must also be performed within the context of a database transaction.

- **Transfer**. Responsibilities include: get source account, get target account, get amount, verify that source account covers amount, verify that the amount is within the bank's policies, debit the source account, credit the target account, and post an entry on the customer's receipt. Once again some of the responsibilities must happen within a database transaction.

You can see that there is some overlap between the operations. For example, *withdrawal*, *deposit*, and *transfer* all request the amount from the customer. *Transfer* and *withdrawal* both verify that the source account has sufficient funds. A reasonable design would coalesce this behavior and build it once.

15.4 Designing Algorithms

Now formulate an ***algorithm*** for each operation. The analysis specification tells *what* the operation does for its clients, but the algorithm shows *how* it is done. Perform the following steps to design algorithms.

- Choose algorithms that minimize the cost of implementing operations.
- Select data structures appropriate to the algorithms.
- Define new internal classes and operations as necessary.
- Assign operations to appropriate classes.

15.4.1 Choosing Algorithms

Many operations are straightforward because they simply traverse the class model to retrieve or change attributes or links. The OCL (see Chapter 3) provides a convenient notation for expressing such traversals.

However, a class-model traversal cannot fully express some operations. We often use pseudocode to handle these situations. Pseudocode helps us think about the algorithm while deferring programming details. For example, many applications involve graphs and the use of transitive closure. (The transitive closure is the set of nodes that can be reached, directly or indirectly, from some starting node.) Figure 15.3 shows a simple model for an undirected graph and pseudocode for computing the transitive closure.

When efficiency is not an issue, you should use simple algorithms. In practice, only a few operations tend to be application bottlenecks. Typically, 20% of the operations consume 80% of execution time. For the remaining operations, it is better to have a design that is simple, understandable, and easy to program than to wring out minor improvements. You can focus your creativity on the algorithms for the operations that are a bottleneck. For example, scanning a set of size n for a value requires an average of $n/2$ operations, whereas a binary search takes $\log n$ operations and a hash search takes less than 2 operations on average. Here are some considerations for choosing among alternative algorithms.


```
Node::computeTransitiveClosure () returns NodeSet
   nodes:= createEmptySet;
   return self.TCloop (nodes);
Node::TCloop (nodes:NodeSet) returns NodeSet
   add self to nodes;
   for each edge in self.Edge
      for each node in edge.Node
         /* 2 nodes are associated with an edge */
         if node is not in nodes then node.TCloop(nodes);
         end if
      end for each node
   end for each edge
```

Figure 15.3 Pseudocode example. You can express difficult algorithms with pseudocode. The top method initiates computation and the bottom method recurses for nodes that are one edge away and have not been visited before.

- **Computational complexity**. How does processor time increase as a function of data structure size? Don't worry about small factors in efficiency. For example, an extra level of indirection is insignificant if it improves clarity. It is essential, however, to think about algorithm complexity—that is, how the execution time (or memory) grows with the number of input values: constant time, linear, quadratic, or exponential. For example, the infamous "bubble sort" algorithm requires time proportional to n^2, where n is the size of the list, while most alternative sort algorithms require time proportional to $n \log n$.
- **Ease of implementation and understandability**. It is worth giving up some performance on noncritical operations if you can use a simple algorithm. For precisely this reason you should try to carry the analysis class model forward to design and make minimal adjustments. Unless you have a performance problem, it is not worth doing a lot of optimizing, because you make a model harder to understand and more difficult to program against.
- **Flexibility**. You will find yourself extending most programs, sooner or later. A highly optimized algorithm often sacrifices ease of change. One possibility is to provide two versions of critical operations—a simple but inefficient algorithm that you can implement quickly and use to validate the system, and a complicated but efficient algorithm, that you can check against the simple one.

ATM example. Interactions between the consortium computer and bank computers could be complex. One issue is distributed computing; the consortium computer is at one location and the bank computers are at many other locations. Also it would be important for the consortium computer to be scalable; the ATM system cannot afford the cost of an oversized consortium computer, but the consortium computer must be able to service new banks as they

join the network. A third concern is that the bank systems are separate applications from the ATM system; there would be the inevitable conversions and compromises in coordinating the various data formats. All these issues make the choice of algorithms for coordinating the consortium and the banks important.

Many banks have sophisticated systems for reducing losses. Then a decision to approve or reject an ATM withdrawal may not be a simple formula, but could involve elaborate logic. The decision may depend on the customer's credit rating, past activity patterns, and account balance relative to the withdrawal amount. Good algorithms could reduce bank losses, much in excess of the development cost.

15.4.2 Choosing Data Structures

Algorithms require data structures on which to work. During analysis, you focused on the logical structure of system information, but during design you must devise data structures that will permit efficient algorithms. The data structures do not add information to the analysis model, but they organize it in a form convenient for algorithms. Many of these data structures are instances of ***container classes***. Such data structures include arrays, lists, queues, stacks, sets, bags, dictionaries, trees, and many variations, such as priority queues and binary trees. Most OO languages provide an assortment of generic data structures as part of their predefined class libraries.

ATM example. A *Transaction* consists of a set of *Updates*. We should revise the class model—there is a sequence of updates that occurs within a transaction. Hence a *Transaction* should have an ordered list of *Updates*. By thinking about algorithms and working through the logic of an application, you can find flaws and improve a class model.

15.4.3 Defining Internal Classes and Operations

You may need to invent new, low-level operations during the decomposition of high-level operations. Some of the low-level operations may be in the "shopping list" of operations (see Chapter 13) from analysis. But usually you will need to add new internal operations as you expand high-level operations.

The expansion of algorithms may lead you to create new classes of objects to hold intermediate results. Typically, the client's description of the problem will not mention these low-level classes because they are artifacts.

ATM example. The design details for the *process transaction* use case involves a customer receipt. The ATM should post each update to a receipt so that customers can remember what they did. The analysis class model did not include a *Receipt* class, so we will add it.

15.4.4 Assigning Operations to Classes

When a class is meaningful in the real world, the operations on it are usually clear. During design, however, you introduce internal classes that do not correspond to real-world objects but merely some aspect of them. Since the internal classes are invented, they are somewhat arbitrary, and their boundaries are more a matter of convenience than of logical necessity.

How do you decide what class owns an operation? When only one object is involved in the operation, the decision is easy: Ask (or tell) that object to perform the operation. The de-

cision is more difficult when more than one object is involved in an operation. You must decide which object plays the lead role in the operation. Ask yourself the following questions.

- **Receiver of action**. Is one object acted on while the other object performs the action? In general, it is best to associate the operation with the *target* of the operation, rather than the *initiator*.
- **Query vs. update**. Is one object modified by the operation, while other objects are only queried for their information? The object that is changed is the target of the operation.
- **Focal class**. Looking at the classes and associations that are involved in the operation, which class is the most centrally located in this subnetwork of the class model? If the classes and associations form a star about a single central class, it is the operation's target.
- **Analogy to real world**. If the objects were not software, but were the real-world objects, what real object would you push, move, activate, or otherwise manipulate to initiate the operation?

Sometimes it is difficult to assign an operation to a class within a generalization hierarchy. It is common to move operations up and down in the hierarchy during design, as their scope is adjusted. Furthermore, the definitions of the subclasses within the hierarchy are often fluid and can be adjusted during design.

ATM example. Let us consider the internal operations for *process transaction* from Section 15.3 and assign a class for each of them.

- *Customer.getAmount()*—get amount from customer. Eventually *amount* will be stored as an attribute of *Update* objects, but we presume that these objects are not available when the customer specifies the amount and are created after some checking. We will store the amount for a customer in a temporary attribute.
- *Account.verifyAmount(amount)*—verify that amount is covered by the account balance.
- *Bank.verifyAmount(amount)*—verify that amount is within the bank's policies.
- *ATM.verifyAmount(amount)*—verify that the ATM has sufficient cash. Note that there are several *verifyAmount* methods, each belonging to a different class. This is a convenient way to organize the various ways of checking an amount. These methods should all have the same signature.
- *ATM.disburseFunds(amount)*—disburse funds.
- *Account.debit(amount)*—debit bank account.
- *Receipt.postTransaction()*—add a transaction to a customer's receipt. It might seem that we should relate *Customer* to *Receipt*, but the model will be more precise if we instead relate *CashCard* to *Receipt*. By traversing the model, a *CashCard* does imply one *Customer*, but now we can also track the precise *CardAuthorization* and *CashCard* used for the ATM session.
- *ATM.receiveFunds(amount)*—accept funds envelope from the customer. The proper class for this method is not obvious. We could assign it to *ATM* or to *Customer*. We decided to assign it to *ATM* for symmetry with *ATM.disburseFunds*. We will consider time-stamping the envelope to be part of receiving the funds.

- *Account.credit(amount)*—credit bank account.
- *Customer.getAccount()*—handles both *get source account* and *get target account*. There is an implicit constraint that this method must satisfy. A customer owns many accounts and the customer can provide only an account that he or she owns. A user interface would typically satisfy this constraint by providing a list of the accounts that a customer owns and letting the customer pick one of them.

Figure 15.4 elaborates the ATM domain class model from Chapter 13 with our progress.

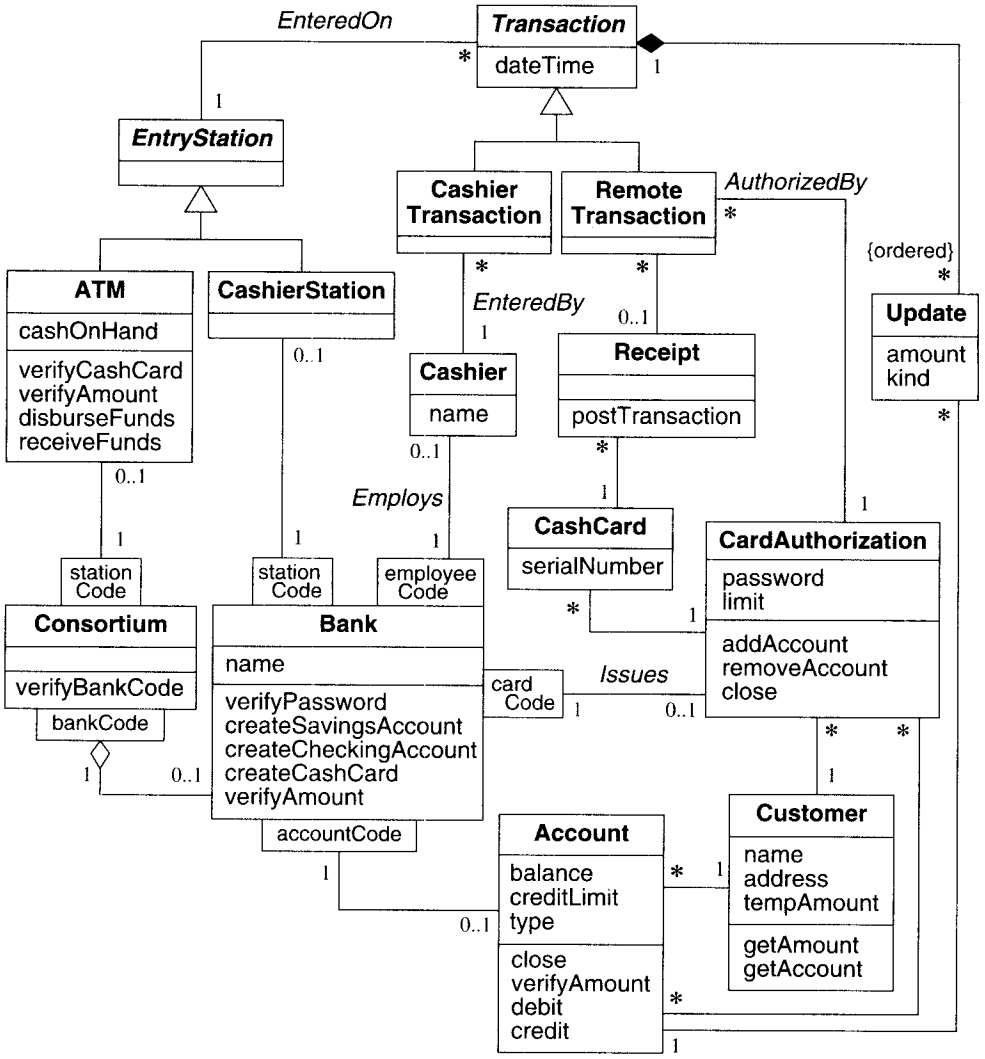

Figure 15.4 ATM domain class model with some class design elaborations

15.5 Recursing Downward

We recommend that you organize operations as layers—operations in higher layers invoke operations in lower layers. The design process generally works top down—you start with the higher-level operations and proceed to define lower-level operations. You can also work bottom up, but you risk designing operations that are never needed. Downward recursion proceeds in two main ways: by functionality and by mechanism.

15.5.1 Functionality Layers

Functionality recursion means that you take the required high-level functionality and break it into lesser operations. This is a natural way to proceed, but you can get into trouble if you perform the decomposition arbitrarily and the pieces do not relate well to classes. To avoid this, make sure you combine similar operations and attach the operations to classes.

The other danger of functionality recursion is that it may depend too much on the exact statement of top-level functionality. Then a small change can radically change the decomposition. To guard against this, you must attach operations to classes and broaden them for reuse. An operation should be coherent, meaningful, and not an arbitrary portion of code. Operations that are carefully attached to classes have much less risk than free-floating functionality. This approach to functionality makes sense, because you must implement the responsibilities of the system somewhere.

ATM example. In Section 15.3 we took a use case and decomposed it into responsibilities. In Section 15.4.4 we assigned the resulting operations to classes. We are satisfied with our operations, but would have had to rework them if they did not fit against the class model.

15.5.2 Mechanism Layers

Mechanism recursion means that you build the system out of layers of needed support mechanisms. In providing functionality, you need various mechanisms to store information, sequence control, coordinate objects, transmit information, perform computations, and provide other kinds of computing infrastructure. These mechanisms don't show up explicitly in the high-level responsibilities of a system, but they are needed to make it all work. For example, in constructing a tall building you need an infrastructure of support girders, utility conduits, and building control devices. These are not directly part of the users' needs for space, but they are needed to enable the chosen architecture. Similarly, computing architecture includes various kinds of general-purpose mechanisms, such as data structures, algorithms, and control patterns. These are not particular to a single application domain, but they may be associated with a particular software architectural style.

For example, a subject-view pattern associates many views with each subject object. A subject object contains the semantic information about some entity, and a view presents it to the user in a particular format. There are mechanisms to update subjects and broadcast the changes to all the views, and to update a view and propagate its changes into the subject. This infrastructure can serve many kinds of applications. However, as a piece of software, it is built in terms of other, more primitive mechanisms than itself.

Any large system mixes functionality layers and mechanism layers. A system designed entirely with functionality recursion is brittle and supersensitive to changes in requirements. A system designed entirely with mechanisms doesn't actually *do* anything useful. Part of the design process is to select the appropriate mix of the two approaches.

ATM example. We have already noted some important mechanisms. There is a need for both communications and distribution infrastructure. The bank and ATM computers are at different locations and must quickly and efficiently communicate with each other. Furthermore, the architecture must be resistant to errors and communications outages.

15.6 Refactoring

The initial design of a set of operations will contain inconsistencies, redundancies, and inefficiencies. This is natural, because it is impossible to get a large design correct in one pass. You must make decisions that are ultimately linked to other decisions. No matter in which order you make the decisions, some of them will be suboptimal.

Furthermore, as a design evolves, it also degrades. It is good to use an operation or class for multiple purposes. But it is inevitable that an operation or class conceived for one purpose will not fully fit additional purposes. You must revisit your design and rework the classes and operations so that they cleanly satisfy all their uses and are conceptually sound. Otherwise your application will become brittle, difficult to understand, and awkward to extend and maintain, and eventually it will collapse under its own weight.

Martin Fowler [Fowler-99] defines ***refactoring*** as changes to the internal structure of software to improve its design without altering its external functionality. It means that you step back, look across many different classes and operations, and reorganize them to support further development better. Refactoring may seem like a waste of time, but it is an essential part of any good engineering process. It is not enough to deliver functionality. If you expect to maintain a design, then you must keep the design clean, modular, and understandable. Refactoring keeps a design viable for continued development.

ATM example. In Section 15.4.4 we considered operations of the *process transaction* use case. An obvious revision is to combine *Account.credit(amount)* and *Account.debit(amount)* into a single operation *Account.post(amount)*. We would expect more opportunities for refactoring as we flesh out operations for additional use cases.

15.7 Design Optimization

A good way to design a system is to first get the logic correct and then optimize it. That is because it is difficult to optimize a design at the same time as you create it. Furthermore, a premature concern with efficiency often leads to a contorted and inferior design. Once you have the logic in place, you can run the application, measure its performance, and then fine tune it. Often a small part of the code is responsible for most of the time or space costs. It is better to focus optimization on the critical areas, than to spread effort evenly.

This does not mean that you should totally ignore efficiency during initial design. Some approaches are so obviously inefficient that you would not consider them. If there is a clean, simple, efficient way to design something, use it. But don't do something in a complicated, unnatural way just because of fears about performance. First get a clean design working. Then you can optimize it. You might find that your concern was misplaced.

The design model builds on the analysis model. The analysis model captures the logic of a system, while the design model adds development details. You can optimize the inefficient but semantically correct analysis model to improve performance, but an optimized system is more obscure and less likely to be reusable. You must strike an appropriate balance between efficiency and clarity. Design optimization involves the following tasks.

- Provide efficient access paths.
- Rearrange the computation for greater efficiency.
- Save intermediate results to avoid recomputation.

15.7.1 Adding Redundant Associations for Efficient Access

Redundant associations are undesirable during analysis because they do not add information. Design, however, has different motivations and focuses on the viability of a model for implementation. Can the associations be rearranged to optimize critical aspects of the system? Should new associations be added? Can existing associations be omitted? The associations from analysis may not form the most efficient network, when you consider access patterns and relative frequencies.

For an example, consider the design of a company's employee skills database. Figure 15.5 shows a portion of the analysis class model. The operation *Company.findSkill()* returns a set of persons in the company with a given skill. For example, an application might need all the employees who speak Japanese.

Figure 15.5 Analysis model for person skills. Derived data is undesirable during analysis because it does not add information.

For this example, suppose that the company has 1000 employees, each of whom has 10 skills on average. A simple nested loop would traverse *Employs* 1000 times and *HasSkill* 10,000 times. If only 5 employees actually speak Japanese, then the test-to-hit ratio is 2000.

Several improvements are possible. First, you could use a hashed set for *HasSkill* rather than an unordered list. An operation can perform hashing in constant time, so the cost of testing whether a person speaks Japanese is constant, provided that there is a unique *Skill* object for *speaks Japanese*. This rearrangement reduces the number of tests from 10,000 to 1000—one per employee.

In cases where the number of hits from a query is low because few objects satisfy the test, an ***index*** can improve access to frequently retrieved objects. For example, Figure 15.6

adds the derived association *SpeaksLanguage* from *Company* to *Person,* where the qualifier is the language spoken. The derived association does not add any information but permits fast access to employees who speak a particular language. Indexes incur a cost: They require additional memory and must be updated whenever the base associations are updated. As the designer, you decide when it is worthwhile to build indexes. Note that if most queries return a high fraction of the objects in the search path, then an index really does not save much.

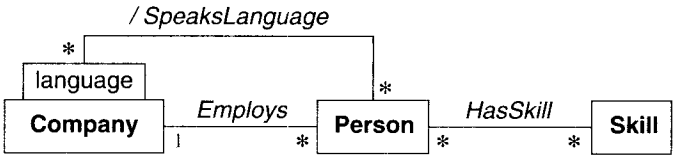

Figure 15.6 Design model for person skills. Derived data is acceptable during design for operations that are significant performance bottlenecks.

Start by examining each operation and see what associations it must traverse to obtain its information. Next, for each operation, note the following.

- **Frequency of access**. How often is the operation called?
- **Fan-out**. What is the "fan-out" along a path through the model? Estimate the average count of each "many" association encountered along the path. Multiply the individual fan-outs to obtain the fan-out of the entire path, which represents the number of accesses on the last class in the path. "One" links do not increase the fan-out, although they increase the cost of each operation slightly; don't worry about such small effects.
- **Selectivity**. What is the fraction of "hits" on the final class—that is, objects that meet selection criteria and are operated on? If the traversal rejects most objects, then a simple nested loop may be inefficient at finding target objects.

You should provide indexes for frequent operations with a low hit ratio, because such operations are inefficient when using nested loops to traverse a path in the network.

ATM example. In our discussion of the *postTransaction()* operation in Section 15.4.4 we decided to relate *Receipt* to *CashCard* for precision. However, we may still need to quickly find the customer for a receipt. Given that the tracing from *CashCard* to *CardAuthorization* to *Customer* has no fan-out, traversal will be fast and a derived association is not needed.

In the United States, banks must report cash deposits and withdrawals greater than $10,000 to the government. We could traverse from *Bank* to *Account*, then from *Account* to *Update*, and then filter out the *Updates* that are cash and greater than $10,000. Note that we would need to elaborate the class model; we could extend *kind* to distinguish between cash and noncash. A derived association from *Bank* to *Update* would speed this operation.

15.7.2 Rearranging Execution Order for Efficiency

After adjusting the structure of the class model to optimize frequent traversals, the next thing to optimize is the algorithm itself. One key to algorithm optimization is to eliminate dead

paths as early as possible. For example, suppose an application must find all employees who speak both Japanese and French. Suppose 5 employees speak Japanese and 100 speak French; it is better to test and find the Japanese speakers first, then test if they speak French. In general, it pays to narrow the search as soon as possible. Sometimes you must invert the execution order of a loop from the original specification.

ATM example. U.S. law requires that a bank not only report individual updates that are greater than $10,000 but also report "suspicious" activities that appear to be an attempt to evade the limit. For example, two withdrawals of $5000 in quick succession would be suspicious.

Suppose we not only check for large cash deposits and withdrawals, but also treat commercial and individual customers differently. We might trust individuals less and have a lower threshold for suspicious activities. We could get all suspicious *Updates* from a derived association ("suspicious" replaces "greater than $10,000" in the derived association in Section 15.7.1) and then traverse back to *Account* to distinguish between commercial and individual accounts. Special logic could then study the updates and determine the ones to report.

Alternately we could maintain two different derived associations between *Bank* and *Update*, one for individuals and the other for businesses. Then would not need to traverse back to *Account* to differentiate them.

15.7.3 Saving Derived Values to Avoid Recomputation

Sometimes it is helpful to define new classes to cache derived attributes and avoid recomputation. You must update the cache if any of the objects on which it depends are changed. There are three ways to handle updates.

- **Explicit update**. The designer inserts code into the update operation of source attributes to explicitly update the derived attributes that depend on it.
- **Periodic recomputation**. Applications often update values in bunches. You could recompute all the derived attributes periodically, instead of after each source change. Periodic recomputation is simpler than explicit update and less prone to bugs. On the other hand, if the data changes incrementally a few objects at a time, full recomputation can be inefficient.
- **Active values**. An ***active value*** is a value that is automatically kept consistent with its source values. A special registration mechanism records the dependency of derived attributes on source attributes. The mechanism monitors the values of source attributes and updates the values of the derived attributes whenever there is a change. Some programming languages provide active values.

ATM example. For convenience, we might add the class *SuspiciousUpdateGroup*. A *SuspiciousUpdateGroup* could have many *Updates* and an *Update* could belong to many *SuspiciousUpdateGroups*. This new class would store derived attributes to facilitate the study of suspicious behavior. In addition it would provide a convenient place to store comments and observations. A *SuspiciousUpdateGroup* would be a set of *Update* records that were suspected of trying to circumvent the $10,000 reporting limit.

15.8 Reification of Behavior

Behavior written in code is rigid. You can execute it, but cannot manipulate it at run time. Most of the time this is fine, because all you want to do is execute it. But if you need to store, pass, or modify the behavior at run time, you should reify it.

Reification is the promotion of something that is not an object into an object. Behavior usually meets this description. It isn't usually something that you would normally manipulate. If you reify behavior, you can store it, pass it to other operations, and transform it. Reification adds complexity but can dramatically expand the flexibility of a system.

You reify behavior by encoding it into an object and decoding it when it is run. The resulting run-time cost may or may not be significant. If the encoding invokes high-level procedures, the cost is only a few indirections. If the entire behavior is encoded in a different language, however, it must be interpreted, which can be an order of magnitude slower than direct execution of code. If the encoded behavior constitutes a small part of the run-time execution time, the performance overhead may not matter.

Exercise 4.16 in Chapter 4 illustrates reification. In one sense you can regard the tasks of a recipe as operations; in another sense they could be data in a class model.

[Gamma-95] lists a number of behavioral patterns that reify behavior. These include encoding a state machine as a table with a run-time interpreter (*State*), encoding a sequence of requests as parameterized command objects (*Command*), and parameterizing a procedure in terms of an operation that it uses (*Strategy*). These techniques have been around for a long time, but the language of patterns is convenient for describing them and weighing their benefits and costs. For example, the *Strategy* pattern was used in Fortran days for purposes such as passing a function to a mathematical integration routine. However, in Fortran there was no way to ensure correct matching of passed functions, and errors were easy to make. By encoding the passed function as an instance of a function class, you get extensibility via polymorphism, yet can enforce the signatures of an entire family of functions. In this case, OO technology permits a cleaner solution than previous techniques.

ATM example. We have already used reification in the case study. In one sense a transaction is an action—withdrawing, depositing, and transferring funds. We promoted transaction to a class so that we could describe it. The functionality that we need is routine and can readily be obtained by traversing the class model.

15.9 Adjustment of Inheritance

As class design progresses, you can often adjust the definitions of classes and operations to increase inheritance by performing the following steps.

- Rearrange classes and operations to increase inheritance.
- Abstract common behavior out of groups of classes.
- Use delegation to share behavior when inheritance is semantically invalid.

15.9.1 Rearranging Classes and Operations

Sometimes several classes define the same operation and can easily inherit it from a common ancestor, but more often operations in different classes are similar but not identical. By adjusting the definitions of the operations, you may be able to cover them with a single inherited operation.

Before using inheritance, the operations must match. All operations must have the same signature—that is, the same number and types of arguments and results. In addition, the operations must have the same semantics. You can use the following kinds of adjustments to increase the chance of inheritance.

- **Operations with optional arguments.** You may be able to align signatures by adding optional arguments that can be ignored. For example, a draw operation on a monochromatic display does not need a color parameter, but it can accept the parameter and ignore it for consistency with color displays.
- **Operations that are special cases.** Some operations may have fewer arguments because they are special cases of more general operations. Implement the special operations by calling the general operation with appropriate parameter values. For example, appending an element to a list is a special case of inserting an element into list; the insert point simply follows the last element.
- **Inconsistent names.** Similar attributes in different classes may have different names. Give the attributes the same name and move them to a common ancestor class. Then operations that access the attributes will match better. Also watch for similar operations with different names. A consistent naming strategy is important.
- **Irrelevant operations.** Several classes in a group may define an operation, but some others may not. Define the operation on the common ancestor class and declare it as a no-op on the classes that don't care about it. For example, rotation is meaningful for geometric figures, but is unimportant for circles.

ATM example. An ATM can post remote transactions on a receipt. It would seem that we should also be able to issue a receipt for cashier transactions. However, a receipt for a *RemoteTransaction* involves a *CashCard*, while a receipt for a *CashierTransaction* directly involves a *Customer*. Furthermore, the cashier software is apart from the ATM software. We will have two different kinds of receipts, a *RemoteReceipt* and a *CashierReceipt*.

15.9.2 Abstracting Out Common Behavior

You will not recognize all opportunities for inheritance during analysis, so it is worthwhile to reexamine the class model looking for commonality. In addition, you will be adding new classes and operations during design. If two classes seem to repeat several operations and attributes, it is possible that the two classes are really specializations of the same thing, when viewed at a higher level of abstraction.

When there is common behavior, you can create a common superclass for the shared features, leaving only the specialized features in the subclasses. This transformation of the class model is called ***abstracting out*** a common superclass or common behavior. We advise

you to make only abstract superclasses, meaning that there are no direct instances of it, but the behavior it defines belongs to all instances of its subclasses. (You can always do this by adding an *Other* subclass.) For example, a *draw()* operation of a geometric figure on a display screen requires setup and rendering of the geometry. The rendering varies among different figures, such as circles, lines, and splines, but the figures can inherit the setup, such as setting the color, line thickness, and other parameters, from the abstract class *Figure*.

Sometimes it is worthwhile to abstract out a superclass even when your application has only one subclass that inherits from it. Although this does not yield any immediate sharing of behavior, the abstract superclass may be reusable in future projects. It may even be a worthwhile addition to your class library. When you complete a project, you should consider the potentially reusable classes for future applications.

Abstract superclasses have benefits beyond sharing and reuse. The splitting of a class into two classes that separate the specific aspects from the more general aspects is a form of modularity. Each class is a separately maintained component with a well-documented interface.

The creation of abstract superclasses also improves the extensibility of a software product. Imagine that you are developing a temperature-sensing module for a large computerized control system. You must use a specific type of sensor (Model J55) with a particular way of reading the temperature, and a formula for converting the raw numeric reading into degrees Celsius. You could place all this behavior in a single class, with one instance for each sensor in the system. But realizing that the J55 sensor is not the only type available, you create an abstract *Sensor* superclass that defines the general behavior common to all sensors. A particular subclass called *SensorJ55* provides reading and conversion that is particular to J55.

Now, when your control system converts to a new sensor model, you need only prepare a subclass that has the specialized behavior for the new model. The superclass already has the common behavior. Perhaps best of all, you will not have to change a single line of code in the large control system that uses these sensors, because the interface is the same, as defined by the *Sensor* superclass.

There is a subtle but important way that abstract superclasses improve configuration management for software maintenance and distribution. Suppose that you must distribute your control system software to many plants throughout the country, each having a different configuration that involves (among other things) a different mix of temperature sensors. Some plants still use the old J55 model, while others have converted to the newer K99 model, and some plants may have a mixture of both types. Generating customized versions of your software to match each different configuration could be tedious.

Instead, you distribute one version of software that contains a subclass for each known sensor model. When the software starts up, it reads the customer's configuration file that tells it which sensor model is used in which location and creates instances of the particular subclasses for the relevant sensors. All the rest of the code treats the sensors as if they were all the same as defined by the *Sensor* superclass. It is even possible to change from one type of sensor to another on-the-fly (while the system is running) if the software creates a new object for the new type of sensor.

ATM example. We did pay attention to inheritance during analysis when we constructed the class model. We do not see any additional inheritance at this time. In a full-fledged

application there would be much more design detail and increased opportunities for inheritance.

15.9.3 Using Delegation to Share Behavior

Inheritance is a mechanism for implementing generalization, in which the behavior of a superclass is shared by all its subclasses. Sharing of behavior is justifiable only when a true generalization relationship occurs—that is, only when it can be said that the subclass *is* a form of the superclass. Operations of the subclass that override the corresponding operation of the superclass must provide the same services as the superclass. When class *B* inherits the specification of class *A*, you can assume that every instance of class *B is* an instance of class *A* because it behaves the same.

Sometimes programmers use inheritance as an implementation technique with no intention of guaranteeing the same behavior. It often happens that an existing class already implements some of the behavior that you want to provide in a newly defined class, although in other respects the two classes differ. The programmer is then tempted to inherit from the existing class to achieve part of the implementation of the new class. This can lead to problems if other inherited operations provide unwanted behavior. We discourage this ***inheritance of implementation*** because it can lead to incorrect behavior.

As an example of implementation inheritance, suppose that you need a *Stack* class and a *List* class is available. You may be tempted to make *Stack* inherit from *List*. You can push an element onto a stack by adding an element to the end of the list. Similarly, you can pop an element from a stack by removing an element from the end of the list. But you will also inherit unwanted operations that add or remove elements from arbitrary positions in the list. If these are ever used (by mistake or as a "short-cut"), then the *Stack* class will not behave as expected.

Often when you are tempted to use inheritance as an implementation technique, you could achieve the same goal in a safer way by making one class an associate of the other class. Then, one object can selectively invoke the desired operations of another class, using delegation rather than inheritance. ***Delegation*** consists of catching an operation on one object and sending it to a related object. You delegate only meaningful operations, so there is no danger of inheriting meaningless operations by accident.

A safer design of *Stack* would delegate to *List*, as Figure 15.7 shows. Every *Stack* instance contains a private *List* instance. (You could optimize the actual implementation of the aggregation by using an embedded object or a pointer attribute.) The *Stack.push()* operation delegates to *List* by calling its *last()* and *add()* operations to add an element at the end. The *pop()* operation has a similar implementation using the *last()* and *remove()* operations. The ability to corrupt the stack by adding or removing arbitrary elements is hidden from the client of the *Stack* class.

Some languages, such as C++ and Java, permit a subclass to inherit the form of a superclass but to selectively inherit operations from ancestors and selectively export operations to clients. This is tantamount to the use of delegation, because the subclass *is not* a form of the superclass in all respects and is not confused with it.

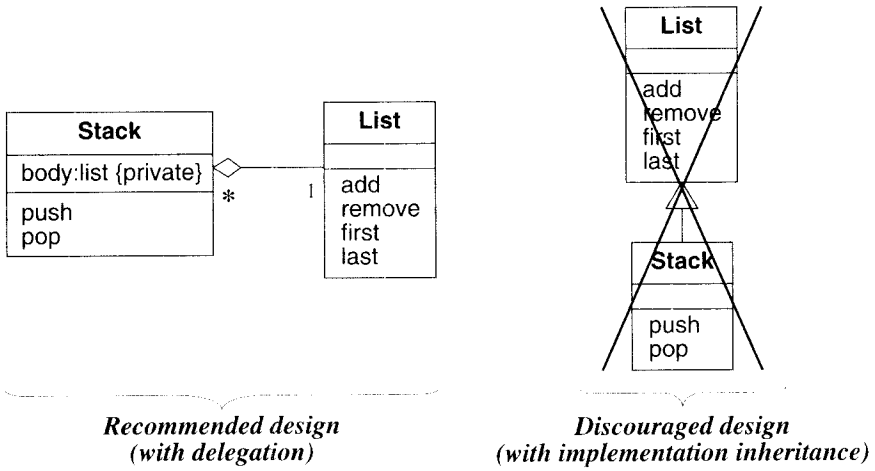

Figure 15.7 Alternative designs. Do not use implementation inheritance.

ATM example. Our use of inheritance is deep and structural. That is the only way we ever use inheritance.

15.10 Organizing a Class Design

Programs consist of discrete physical units that can be edited, compiled, imported, or otherwise manipulated. In some languages, such as C and Fortran, the units are source files. In Ada and Java, the package is an explicit language construct. [Coplien-92] shows how to use a C++ class to group static member functions, lesser classes, enumerations, and constants. You can improve the organization of a class design with the following steps.

- Hide internal information from outside view.
- Maintain coherence of entities.
- Fine-tune definition of packages.

15.10.1 Information Hiding

During analysis we did not consider information visibility—rather our focus was on understanding the application. The goals of design are different. During design we adjust the analysis model so that it is practical to implement and maintain. One way to improve the viability of a design is by carefully separating external specification from internal implementation. This is called ***information hiding***. Then you can change the implementation of a class without requiring that clients of the class modify code. In addition, the resulting "firewalls" around classes limit the effects of changes so that you can better understand them.

There are several ways to hide information.

- **Limit the scope of class-model traversals.** Taken to an extreme, a method could traverse all associations of the class model to locate and access an object. Such unconstrained visibility is appropriate during analysis, when you are trying to understand a problem, but methods that know too much about a model are fragile and easily invalidated by changes. During design you should try to limit the scope of any one method [Lieberherr-88]. An object should access only objects that are directly related (directly connected by an association). An object can access indirectly related objects via the methods of intervening objects.
- **Do not directly access foreign attributes.** Generally speaking, it is acceptable for subclasses to access the attributes of their superclasses. However, classes should not access the attributes of an associated class. Instead, call an operation to access the attributes of an associated class.
- **Define interfaces at a high a level of abstraction.** It is desirable to minimize class couplings. One way to do this is by raising the level of abstraction of interfaces. It is fine to call a method on another class for a meaningful task, but you should avoid doing so for minutia.
- **Hide external objects.** Use boundary objects to isolate the interior of a system from its external environment. A ***boundary object*** is an object whose purpose is to mediate requests and responses between the inside and the outside. It accepts external requests in a client-friendly form and transforms them into a form convenient for the internal implementation.
- **Avoid cascading method calls.** Avoid applying a method to the result of another method, unless the result class is already a supplier of methods to the caller. Instead consider writing a method to combine the two operations.

15.10.2 Coherence of Entities

Coherence is another important design principle. An entity, such as a class, an operation, or a package, is coherent if it is organized on a consistent plan and all its parts fit together toward a common goal. An entity should have a single major theme; it should not be a collection of unrelated parts.

A method should do one thing well. A single method should not contain both policy and implementation. ***Policy*** is the making of context-dependent decisions. ***Implementation*** is the execution of fully-specified algorithms. Policy involves making decisions, gathering global information, interacting with the outside world, and interpreting special cases. A policy method contains I/O statements, conditionals, and accesses data stores. A policy method does not contain complicated algorithms but instead calls the appropriate implementation methods. An implementation method encodes exactly one algorithm, without making any decisions, assumptions, defaults, or deviations. All its information is supplied as arguments, so the argument list may be long.

Separating policy and implementation greatly increases the possibility of reuse. The implementation methods do not contain any context dependencies, so they are likely to be re-

usable. Usually you must rewrite policy methods in a new application, but they are often simple and consist mostly of high-level decisions and low-level calls.

For example, consider an operation to credit interest on a checking account. Interest is compounded daily based on the balance, but all interest for a month is lost if the account is closed. The interest logic consists of two parts: an implementation method that computes the interest due between a pair of days, without regard to any forfeitures or other provisions; and a policy method that decides whether and for what interval the implementation method is called. The implementation method is complex, but likely to be reused. Policy methods are less likely to be reusable, but simpler to write.

A class should not serve too many purposes at once. If it is too complicated, you can break it up using either generalization or aggregation. Smaller pieces are more likely to be reusable than large complicated pieces. Exact numbers are somewhat risky, but as a rule of thumb consider breaking up a class if it contains more than about 10 attributes, 10 associations, or 20 operations. Always break a class if the attributes, associations, or operations sharply divide into two or more unrelated groups.

15.10.3 Fine-Tuning Packages

During analysis you partitioned the class model into packages. This initial organization may not be suitable or optimal for implementation. You should define packages so that their interfaces are minimal and well defined. The interface between two packages consists of the associations that relate classes in one package to classes in the other and operations that access classes across package boundaries.

You can use the connectivity of the class model as a guide for forming packages. As a rough rule of thumb, classes that are closely connected by associations should be in the same package, while classes that are unconnected, or loosely connected, may be in separate packages. Of course there are other aspects to consider. Packages should have some theme, functional cohesiveness, or unity of purpose.

The number of different operations that traverse a given association is a good measure of its coupling strength. We are referring to the number of different ways that the association is used, not the frequency of traversal. Try to keep strong coupling within a single package.

15.11 ATM Example

Figure 15.8 shows our final ATM domain class model after class design. (Keep in mind that the full class model also includes the application class model from Figure 13.8.)

15.12 Chapter Summary

Class design does not begin from scratch but rather elaborates the previous stages of analysis and system design. Class design adds details, such as designing algorithms, refactoring operations, optimizing classes, adjusting inheritance, and refining packages.

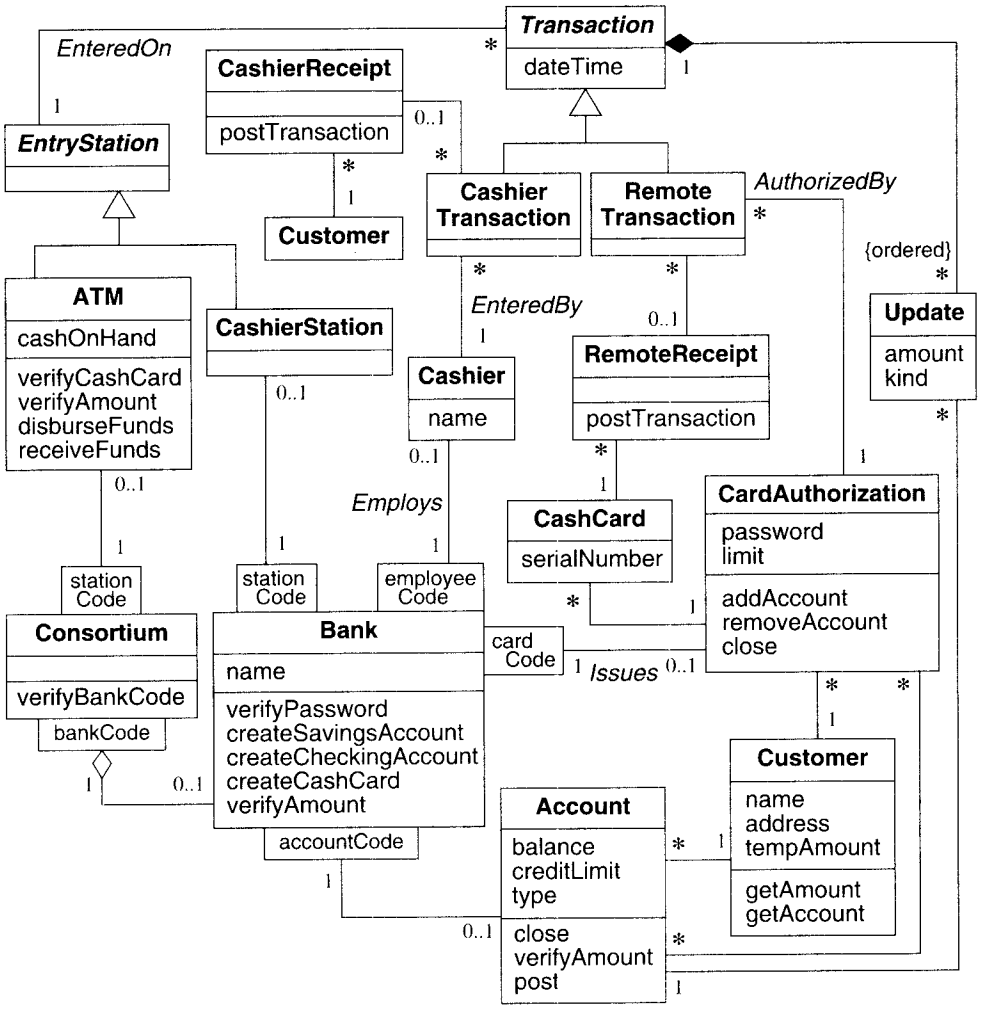

Figure 15.8 Final ATM domain class model after class design

The first step of class design is to add operations according to the use cases. Use cases define the required behavior but not its realization. The class designer invents operations that will deliver the behavior specified by the use cases.

Next, you must devise an algorithm for each operation. Class design focuses on computational complexity, but you should sacrifice small amounts of performance for greater clarity. You will need to recurse to define low-level operations to realize high-level operations. Recursion stops when you have operations that are already available or that are straightforward to implement.

The initial design of a set of operations will contain inconsistencies, redundancies, and inefficiencies. This is natural, because it is impossible to get a large design correct in one pass. Furthermore, as a design evolves, it also degrades. It is inevitable that an operation or class conceived for one purpose will not fully fit additional purposes. As you proceed with a design, you should occasionally refactor operations to improve their clarity and resilience.

During design, you may need to rework the analysis model for efficiency. Optimization does not discard the original information but adds new redundant information to speed access paths and preserve intermediate results. It can be helpful to rearrange algorithms and reduce the number of operations that need to be executed.

You should consider reification—the promotion of something that is not an object into an object. For example, a deposit, withdrawal, or transfer of funds would normally be an operation—it is something that someone *does*. However, for the ATM case study we promoted transaction to a class so that we could describe it.

As class design progresses, you can often adjust the definitions of classes and operations to increase inheritance. These adjustments include modifying the argument list of a method, moving attributes and operations from a class into a superclass, defining an abstract superclass to cover the shared behavior of several classes, and splitting an operation into an inherited part and a specific part. You should use delegation rather than inheritance when a class is similar to another class but not truly a subclass.

You must organize programs into physical units for editors and compilers as well as for the convenience of programming teams. Information hiding is a primary goal to ensure that future changes affect limited amounts of code. Packages should be coherent and organized about a common theme.

abstracting out a superclass	derived attributes	policy vs. implementation
adjusting inheritance	implementation inheritance	recursing
algorithm	index	refactoring
data structure	information hiding	reification
delegation	optimization	responsibility
derived associations	package	use case

Figure 15.9 Key concepts for Chapter 15

Bibliographic Notes

Algorithms and data structures are part of the basic computer science curriculum. [Aho-83] and [Sedgewick-95] are well-written books about algorithms.

Adding indexes and rearranging access order to improve performance is a mature technique in database optimization. See [Ullman-02] for examples.

[Lieberherr-88] is an early attempt to provide visibility guidelines (the law of Demeter) that preserve OO modularity. [Meyer-97] suggests style rules for using classes and operations.

The *class design* stage is renamed from *object design* in the first edition of this book.

References

[Aho-83] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Boston: Addison-Wesley, 1983.

[Coplien-92] James O. Coplien. *Advanced C++: Programming Styles and Idiom*. Boston: Addison-Wesley, 1992.

[Fowler-99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley, 1999.

[Gamma-95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995.

[Lieberherr-88] K. Lieberherr, I. Holland, A. Riel. Object-oriented programming: an objective sense of style. *OOPSLA'88* as *ACM SIGPLAN 23*, 11 (November 1988), 323–334.

[Meyer-97] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. Upper Saddle River, NJ: Prentice Hall, 1997.

[Sedgewick-95] Robert Sedgewick, Philippe Flajolet, and Peter Gordon. *An Introduction to the Analysis of Algorithms*. Boston: Addison-Wesley, 1995.

[Ullman-02] Jeffrey Ullman and Jennifer Widom. *A First Course in Database Systems*. Upper Saddle River, NJ: Prentice Hall, 2002.

[Wirfs-Brock-90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Upper Saddle River, NJ: Prentice Hall, 1990.

Exercises

15.1 (6) Take the use cases from Exercise 13.9 and list at least four responsibilities for each one. [Instructor's note: You may want to give the students our answer to Exercise 13.9.]

15.2 (6) Take the first three use cases from Exercise 13.16 and list at least four responsibilities for each one. [Instructor's note: You may want to give the students our answer to Exercise 13.16.]

15.3 (4) Write algorithms to draw the following figures on a graphics terminal. The figures are not filled. Assume pixel-based graphics. State any assumptions that you make.

a. circle
b. ellipse
c. square
d. rectangle

15.4 (3) Discuss whether or not the algorithm that you wrote in the previous exercise to draw an ellipse is suitable for drawing circles and whether or not the rectangle algorithm is suitable for squares.

15.5 (3) By careful ordering of multiplications and additions, the total number of arithmetic steps needed to evaluate a polynomial can be minimized. For example, one way to evaluate the polynomial $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ is to compute each term separately, adding each term to the total as it is computed, which requires 10 multiplications and 4 additions. Another way is to rearrange the order of the arithmetic to $x \cdot (x \cdot (x \cdot a_4 + a_3) + a_2) + a_1) + a_0$, which requires only 4 multiplications and 4 additions. How many multiplications and additions are required by each method for an nth-order polynomial? Discuss the relative merits of each approach.

15.6 (4) Improve the class diagram in Figure E15.1 by generalizing the classes *Ellipse* and *Rectangle* to the class *GraphicsPrimitive*, transforming the class diagram so that there is only a single one-to-one association to the object class *BoundingBox*. In effect, you will be changing the 0..1 multiplicity to exactly-one multiplicity. As it stands, the class *BoundingBox* is shared between *Ellipse* and *Rectangle*. A *BoundingBox* is the smallest rectangular region that will contain the associated *Ellipse* or *Rectangle*.

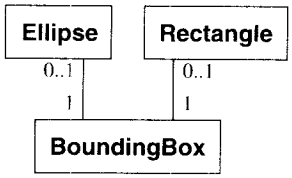

Figure E15.1 Portion of a class diagram with a shared class

15.7 (5) Which class(es) for the previous exercise must supply a delete operation visible to the outside world? To delete means to destroy an object and remove it from the application. Explain your answer.

15.8 (4) Modify the class diagram in Figure E15.2 so that a separate class provides margins. Different categories of pages may have a default margin, and specific pages may override the default.

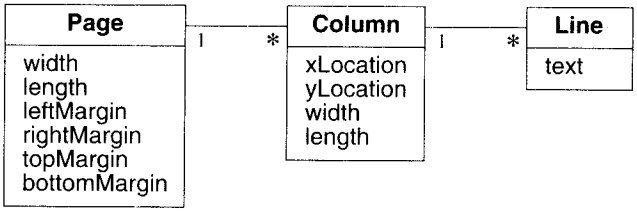

Figure E15.2 Portion of a class diagram of a newspaper

15.9 (3) Modify Figure E15.2 to make it possible to be able to determine what *Page* a *Line* is on without first determining the *Column*.

15.10 (7) Write pseudocode for each method in Figure E15.3. *Initialize* causes a deck to start with 52 cards and anything else to become empty. *Insert* and *delete* take a card as a single argument and insert or delete the card into a collection, forcing the collection to redisplay itself. *Delete* is allowed only on the top card of a pile. *Sort* is used to sort a hand by suit and rank.

Pile is an abstract class. *TopOfPile* and *bottomOfPile* are queries. *Draw* deletes the top card from a pile and inserts the card into a hand, which is passed as an argument.

Shuffle mixes a deck. *Deal* selects cards from the top of the deck one at a time, deleting them from the deck and inserting them into hands that are created and returned as an array of hands. *Display* displays a card. *Compare* determines which of two cards has the largest value. *Discard* deletes a card from the collection that contains it and places it on top of the draw pile that is passed as an argument.

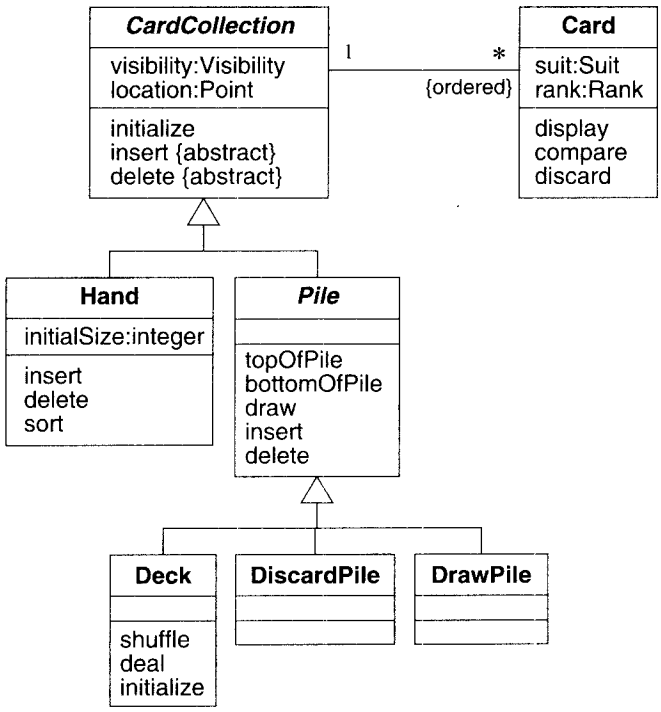

Figure E15.3 Portion of a class diagram of a card-playing program

15.11 (5) Write pseudocode for computing the net score for a trial in Figure E12.4.

Each attempt of a competitor in an event (a *Trial*) is observed by several judges. Each judge rates the attempt and holds up a score. A reader assigned to the group of judges announces the scores one at a time to a panel of scorekeepers. Three scorekeepers write the scores down, cross off the highest and the lowest scores, and total the rest. They check each other's total to detect recording and arithmetic errors. In some cases, they may ask the reader to repeat the scores. When they are satisfied, they hand their figures to three other scorekeepers, who multiply the total score by a difficulty factor for the event and take the average to determine a net score. The net scores are compared to detect and correct scoring errors.

15.12 Prepare pseudocode for the following operations to classes in Figure E12.4. You will need to add a many-to-many association *RegisteredFor* between a set of *Events* and an ordered list of *Competitors* to track who is registered for which *Events*. Use the registration order in scheduling trials.

a. (3) Find the event for a figure and meet.
b. (3) Register a competitor for an event.
c. (3) Register a competitor for all events at a meet.
d. (5) Select and schedule events for a meet.
e. (3) Schedule meets in a season.
f. (4) Assign judges and scorekeepers to stations.

15.13 (8) Figure E15.4 is a portion of a class diagram metamodel that might be used in a compiler of an OO language. Write pseudocode for the *traceInheritancePath* method that traces an inheritance hierarchy as follows. Input to the method is a pair of classes. The method returns an ordered list of classes on the path from the more general class to the more specific class. Tracing is only through generalizations; aggregations and associations are ignored. If tracing fails, an empty list is returned. You may assume that multiple inheritance is not allowed.

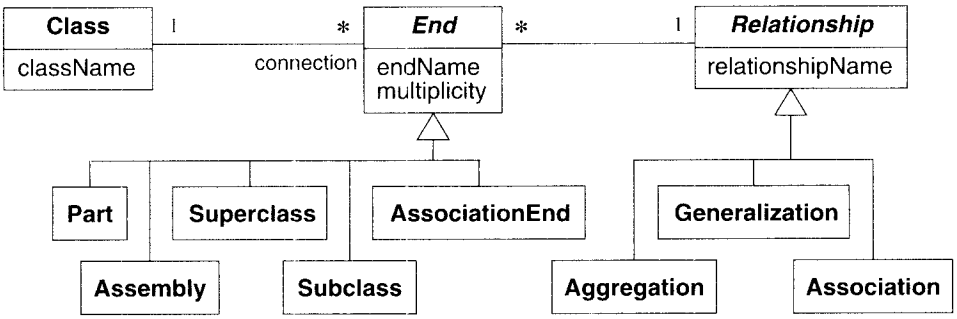

Figure E15.4 Portion of a class diagram metamodel

15.14 (8) Refine Figure E15.4 by eliminating the associations to the classes *End* and *Relationship*, replacing them with associations to the subclasses of *End* and *Relationship*. This is an example of a transformation on a class diagram. Write pseudocode for the *traceInheritancePath* method for the new diagram.

15.15 (7) Referring to Figure E15.4, prepare an algorithm for an operation that will generate a name for an association that does not already have one. Input to the operation is an instance of *Association*. The operation must return a globally unique *relationshipName*. If the association already has a name, the operation should return it. Otherwise the operation should generate a name using a strategy that you must devise.

The precise strategy is not critical, but the generated names must be unique, and anyone reading the names should be able to determine which association the name refers to. Assume all associations are binary. You may assume that a similar operation on the class *End* already has been designed that will return an *endName* unique within the context of a relationship. If the name that would be formed collides with an existing name, modify the name in some way to make it unique. If you feel you need to modify the diagram or use additional data structures, go ahead, but be sure to describe them.

15.16 (7) Improve the class diagram in Figure E15.5 by transforming it, adding the class *PoliticalParty*. Associate *Voter* with a party. Discuss why the transformation is an improvement.

15.17 (7) Sometimes an airline will substitute a smaller aircraft for a larger one for a flight with few passengers. Write an algorithm for reassigning seats so that passengers with low row numbers do not have to be reassigned. Assume both aircraft have the same number of seats per row.

15.18 (8) The need for implementation efficiency may force you to create classes that are not in the original problem statement. For example, a two-dimensional CAD system may use specialized data structures to determine which points fall within a rectangular window specified by the user.

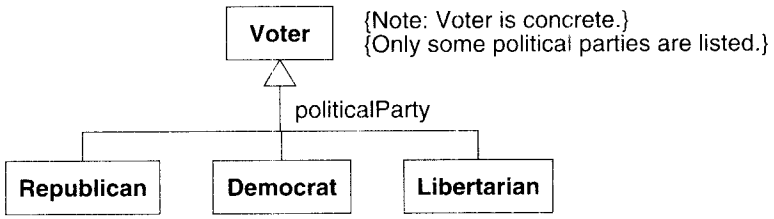

Figure E15.5 Class diagram representing voter membership in a political party

One technique is to maintain a collection of points sorted on x and then y. Points that fall within a rectangular window can usually be found without having to check all points.

Prepare a class diagram that describes collections of points sorted on x and y. Write pseudocode for the operations *search*, *add*, and *delete*. The input to *search* is a description of a rectangular region and a collection of points. The output of *search* is a set of points from the input collection that fall within the region. Inputs to both *add* and *delete* are a point and a collection of points. The input point is added or deleted from the collection.

15.19 (8) Determine how the time required by the search operation in the previous exercise depends on the number of points in a collection. Explicitly state any assumptions you make.

15.20 (3) In selecting an algorithm, it may be important to evaluate its resource requirements. How does the time required to execute the following algorithms depend on the following parameters?

a. The algorithm in Exercise 15.13 on the depth of the inheritance hierarchy.

b. The algorithm in Exercise 15.17 on the number of passengers.

16

Process Summary

As Figure 16.1 shows, a development process provides a basis for the organized production of software. We advocate that a process be based on OO concepts and the UML notation.

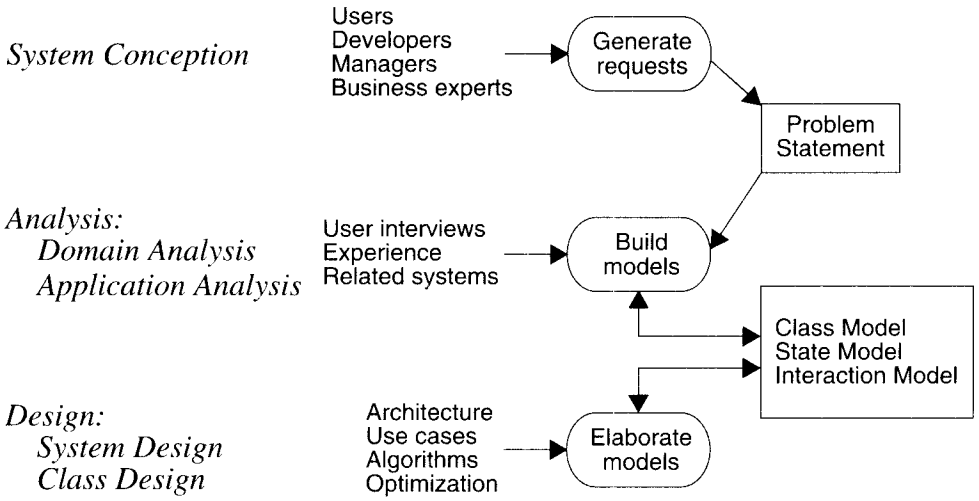

Figure 16.1 Summary of development process. A development process provides a basis for the organized production of software.

Note that there is no need to change from one model to another, as the OO paradigm spans analysis, design, and implementation. The OO paradigm applies equally well in describing the real-world specification and computer-based implementation.

This book's presentation of the stages is linear (an artifact of presentation), but in practice OO development is an iterative process, as Figure 16.1 emphasizes. When you think that

a model is complete at one level of abstraction, you should consider the next lower level of abstraction. For each level, you may need to add new operations, attributes, and classes. You may even need to revise the relationships between objects (including changes to the inheritance hierarchy). Do not be surprised if you find yourself iterating several times. Chapter 21 explains more about iterative development.

16.1 System Conception

System conception deals with the genesis of an application. Initially some person, who understands both technology and business needs, thinks of an idea for an application. The purpose of system conception is to understand the big picture—what need does the proposed system meet, can it be developed at a reasonable cost, and will the demand for the result justify the cost of building it? The input to system conception is the raw idea for a new application. The output is a problem statement that is the starting point for careful analysis.

16.2 Analysis

Analysis focuses on preparing models to get a deep understanding of the requirements. The goal of analysis is to specify what needs to be done, not how it is done. You must understand a problem before attempting a solution. It is important to consider all available inputs, including requirements statements, user interviews, real-world experience, and artifacts from related systems. The output from analysis is a set of models that specify a system in a rigorous and complete manner. There are two substages to analysis: domain analysis and application analysis.

16.2.1 Domain Analysis

Domain analysis captures general knowledge about an application—concepts and relationships known to experts in the domain. The concern is with devising a precise, concise, understandable, and correct model of the real world. Before building anything complex, the builder must understand the requirements. Domain analysis leads to class models and sometimes state models, but seldom interaction models. The job of constructing a domain model is mainly to decide which information to capture (determine the application's scope) and how to represent it (the level of abstraction).

16.2.2 Application Analysis

Application analysis follows and addresses the computer aspects of the application (application objects) that are visible to users. Application objects are not merely internal design decisions, because the users see them and must agree with them. Application classes include controllers, devices, and boundary objects. The interaction model dominates application analysis, but the class and state models are also important.

16.3 Design

Analysis addresses the *what* of an application; design addresses the *how*. Once you have a thorough understanding of an application from analysis, you are ready to deal with the details of building a practical and maintainable solution. You could prepare the design model in a completely different manner than analysis, but most of the time, the simplest and best approach is to carry the analysis classes forward into design. Design then becomes a process of adding detail and making fine decisions. There are two substages to design: system design and class design.

16.3.1 System Design

The purpose of system design is to devise a high-level strategy—the architecture—for solving the application problem. The choice of architecture is an important decision with broad consequences and is based on the requirements as well as past experience. The system designer must also establish policies to guide the subsequent class design.

16.3.2 Class Design

Class design augments and adjusts the real-world models from analysis so that they are ready for implementation. Class designers complete the definitions of the classes and associations and choose algorithms for operations.

Part 3

Implementation

Parts 1 and 2 have presented OO concepts and an analysis and design process for applying the concepts. Part 3 covers the remainder of software development and discusses the specific details for implementing a system with C++, Java, and databases. OO models are also helpful with non-OO languages, but we do not cover non-OO languages in this book, because most developers now days are using OO languages.

Chapter 17 discusses implementation issues that transcend the choice of language. The focus is on techniques for realizing associations, since few languages have intrinsic support.

Chapter 18 explains the principles of how to implement an OO design with C++ and Java. We cover C++ and Java because they are the most popular OO programming languages.

Chapter 19 shows how to implement an OO design with a database. Our focus is on relational databases, because they dominate the marketplace. As you would expect, OO designs can also be implemented with OO databases, but OO databases have but a small market share and are only used in specialty situations.

Chapter 20 concludes with style recommendations for programming in any language or database. The programming 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.

Part 3 completes our explanation of how to take OO concepts and use them to develop applications. Part 4 deals with software engineering issues that are especially important for large and complex applications.

17

Implementation Modeling

Implementation is the final development stage that addresses the specifics of programming languages. Implementation should be straightforward and almost mechanical, because you should have made all the difficult decisions during design. To a large extent, your programming code should simply translate design decisions. You must add details while writing code, but each one should affect only a small part of the program.

17.1 Overview of Implementation

It is now, during implementation, that you finally capitalize on your careful preparation from analysis and design. First you should address implementation issues that transcend languages. This is what we call ***implementation modeling*** and involves the following steps.

- Fine-tune classes. [17.2]
- Fine-tune generalizations. [17.3]
- Realize associations. [17.4]
- Prepare for testing [17.5]

The first two steps are motivated by the theory of transformations. A ***transformation*** is a mapping from the domain of models to the range of models. When modeling, it is important not only to focus on customer requirements, but to also take an abstract mathematical perspective.

17.2 Fine-tuning Classes

Sometimes it is helpful to fine-tune classes before writing code in order to simplify development or to improve performance. Keep in mind that the purpose of implementation is to realize the models from analysis and design. Do not alter the design model unless there is a compelling reason. If there is, consider the following possibilities.

- **Partition a class.** In Figure 17.1, we can represent home and office information for a person with a single class or we can split the information into two classes. Both approaches are correct. If we have much home and office data, it would be better to separate them. If we have a modest amount of data, it may be easier to combine them.

  The partitioning of a class can be complicated by generalization and association. For example, if *Person* was a superclass and we split it into home and office classes, it would be less convenient for the subclasses to obtain both kinds of information. The subclasses would have to multiply inherit, or we would have to introduce an association between the home and office classes. Furthermore, if there were associations to *Person*, you would need to decide how to associate to the partitioned classes.

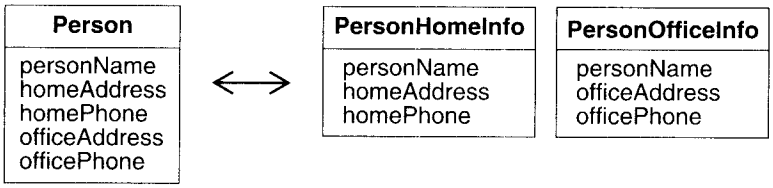

Figure 17.1 Partitioning a class. Sometimes it is helpful to fine-tune a model by partitioning or merging classes.

- **Merge classes.** The converse to partitioning a class is to merge classes. If we had started with *PersonHomeInfo* and *PersonOfficeInfo* in Figure 17.1, we could combine them. Figure 17.2 shows another example with intervening associations. Neither representation is inherently superior, because both are mathematically correct. Once again, you must consider the effects of generalization and association in your decisions.

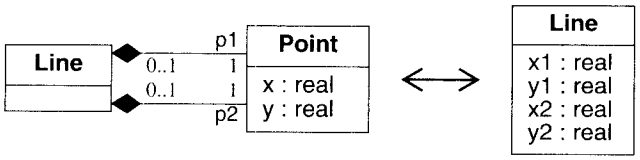

Figure 17.2 Merging classes. It is acceptable to rework your definitions of classes, but only do so for compelling development or performance reasons.

- **Partition / merge attributes.** You can also adjust attributes by partitioning and merging, as Figure 17.3 illustrates.
- **Promote an attribute / demote a class.** As Figure 17.4 shows, we can represent address as an attribute, as one class, or as several related classes. The bottom model would be helpful if we were preloading address data for an application.

ATM example. We may want to split *Customer address* into several classes if we are prepopulating address data. For example, we may preload *city*, *stateProvice*, and *postalCode*

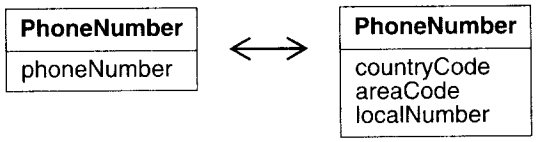

Figure 17.3 Partitioning / merging attributes

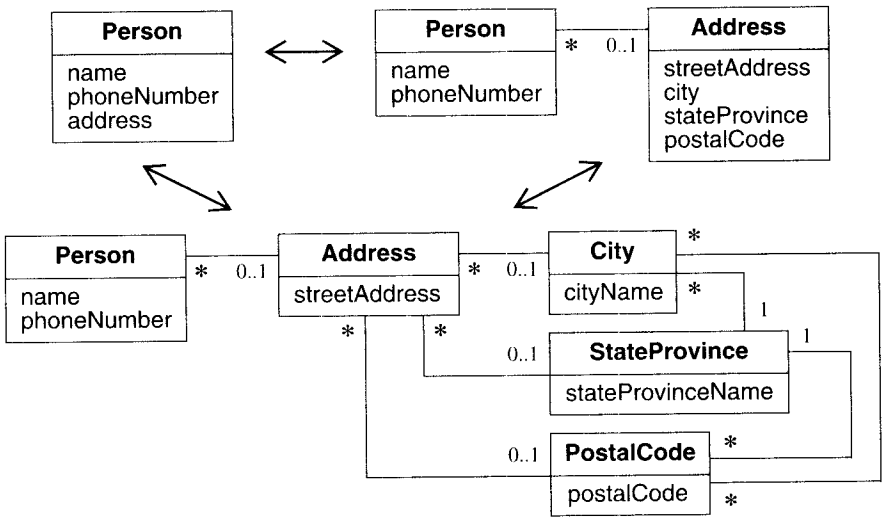

Figure 17.4 Promoting an attribute / demoting a class

data for the convenience of customer service representatives when creating a new *Customer* record.

We may also want to place *Account type* in its own class. Then it would be easier to program special behavior. For example, the screens may look different for checking accounts than for savings accounts.

All in all, the ATM model is small in size and carefully prepared, so we are not inclined to make many changes.

17.3 Fine-tuning Generalizations

As you can reconsider classes, so too you can reconsider generalizations. Sometimes it is helpful to remove a generalization or to add one prior to coding.

Figure 17.5 shows a translation model from one of our recent applications. A language translation service converts a *TranslationConcept* into a *Phrase* in the desired language. A *MajorLanguage* is a language such as English, French, or Japanese. A *MinorLanguage* is a

dialect such as American English, British English, or Australian English. All entries in the application database that must be translated store a *translationConceptID*. The translator first tries to find the phrase for a concept in the specified *MinorLanguage* and then, if that is not found, looks for the concept in the corresponding *MajorLanguage*.

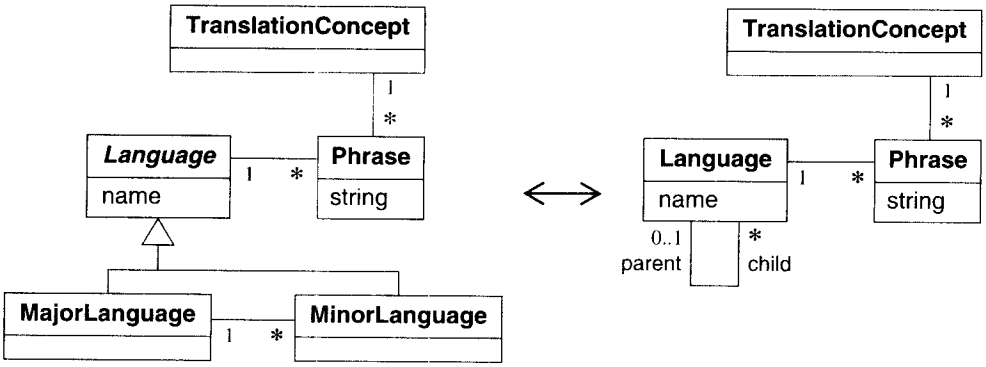

Figure 17.5 Removing / adding generalization. Sometimes it can simplify implementation to remove or add a generalization.

For implementation simplicity, we removed the generalization and used the right model. Since the translation service is separate from the application model, there were no additional generalizations or associations to consider, and it was easy to make the simplification.

ATM example. Back in Section 13.1.1 we mentioned that the ATM domain class model encompassed two applications—ATM and cashier. We did not concern ourselves with this during analysis—the purpose of analysis is to understand business requirements, and the eventual customer does not care how services are structured. Furthermore, we wanted to make sure that both applications had similar behavior. However, now that we are implementing, we must separate the applications and limit the scope to what we will actually build. Figure 17.6 deletes cashier information from the domain class model, leading to a removal of both generalizations.

Figure 17.6 is the full ATM class model. The top half (*Account* and above) presents the domain class model; the bottom half (*UserInterface*, *ConsortiumInterface*, and below) presents the application class model. The operations are representative, but only some are listed.

17.4 Realizing Associations

Associations are the "glue" of the class model, providing access paths between objects. Now we must formulate a strategy for implementing them. Either we can choose a global strategy for implementing all associations uniformly, or we can select a particular technique for each association, taking into account the way the application will use it. We will start by analyzing how associations are traversed.

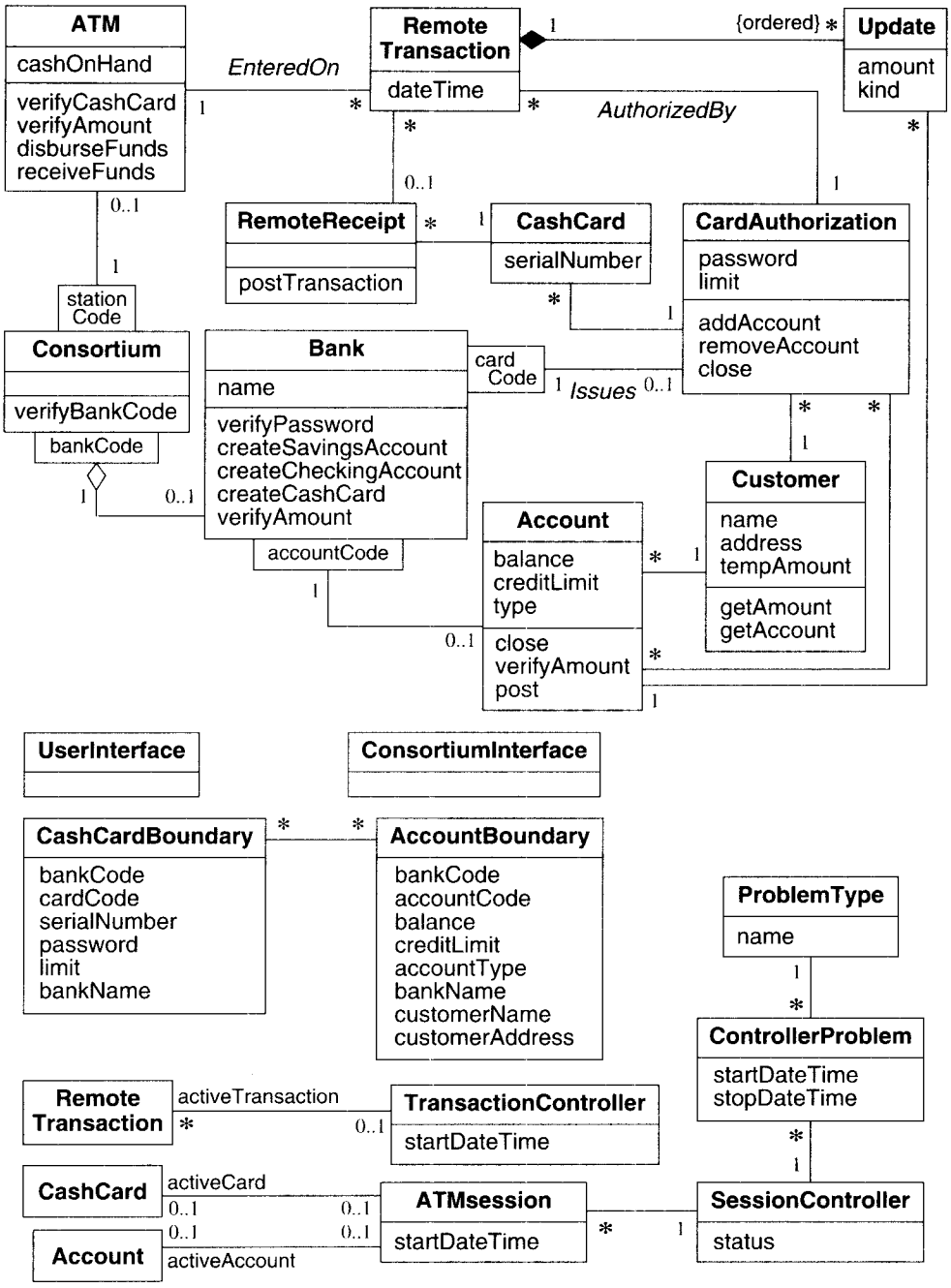

Figure 17.6 Full ATM implementation class model

17.4.1 Analyzing Association Traversal

We have assumed until now that associations are inherently bidirectional, which is certainly true in an abstract sense. But if your application has some associations that are traversed in only one direction, their implementation can be simplified. Be aware, however, that future requirements may change, and you may need to add a new operation later that traverses the association in the reverse direction.

For prototype work we always use bidirectional associations, so that we can add new behavior and modify the application quickly. For production work we optimize some associations. In any case, you should hide the implementation, using access methods to traverse and update the association. Then you can change your decision more easily.

17.4.2 One-way Associations

If an association is traversed only in one direction, you can implement it as a ***pointer***—an attribute that contains an object reference. (Note that this chapter uses the word *pointer* in the logical sense. The actual implementation could be a programming-language pointer, a programming-language reference, or even a database foreign key.) If the multiplicity is "one," as Figure 17.7 shows, then it is a simple pointer; if the multiplicity is "many," then it is a set of pointers.

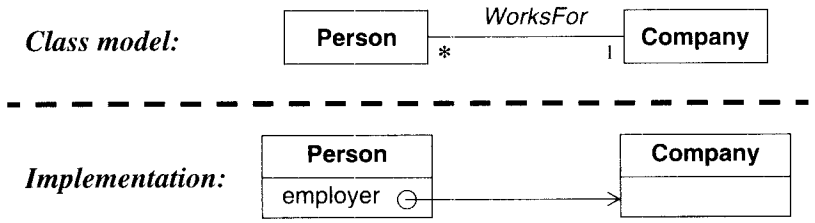

Figure 17.7 Implementing a one-way association with pointers. If an association is traversed only in one direction, you can implement it as a pointer.

17.4.3 Two-way Associations

Many associations are traversed in both directions, although not usually with equal frequency. There are three approaches to their implementation.

- **Implement one-way**. Implement as a pointer in one direction only and perform a search when backward traversal is required. This approach is useful only if there is a great disparity in traversal frequency in the two directions and minimizing both the storage and update costs is important. The rare backward traversal will be expensive.
- **Implement two-way**. Implement with pointers in both directions as Figure 17.8 shows. This approach permits fast access, but if either direction is updated, then the other must also be updated to keep the link consistent. This approach is useful if accesses outnumber updates.

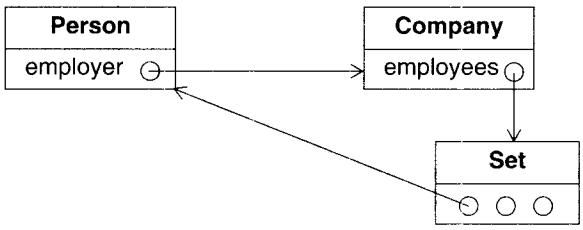

Figure 17.8 Implementing a two-way association with pointers. Dual pointers enable fast traversal of an association in either direction, but introduce redundancy, complicating maintenance.

- **Implement with an association object.** Implement with a distinct association object, independent of either class, as Figure 17.9 shows [Rumbaugh-87]. An association object is a set of pairs of associated objects (triples for qualified associations) stored in a single variable-size object. For efficiency, you can implement an association object using two dictionary objects, one for the forward direction and one for the backward direction. Access is slightly slower than with pointers, but if hashing is used, then access is still constant time. This approach is useful for extending predefined classes from a library that cannot be modified, because the association object does not add any attributes to the original classes. Distinct association objects are also useful for sparse associations, in which most objects of the classes do not participate, because space is used only for actual links.

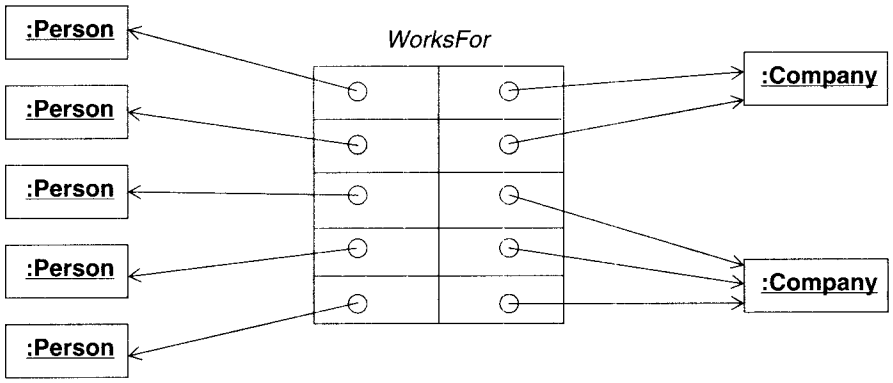

Figure 17.9 Implementing an association as an object. This is the most general approach for implementing associations but requires the most programming skill.

17.4.4 Advanced Associations

The appropriate techniques for implementing advanced associations vary.

- **Association classes**. The usual approach is to promote the association to a class. This handles any attributes of the association as well as associations of the association class. Note that promotion changes the meaning of the model; the promoted association has identity of its own, and methods must compensate to enforce the dependency of the association class on the constituent classes.

  Alternatively, if the association is one-to-one and has no further associations, you can implement the association with pointers and store any attributes for the association as attributes of either class. Similarly, if the association is one-to-many and has no further associations, you can implement the association with pointers and store attributes for the association as attributes of the "many" class, since each "many" class appears only once for the association.
- **Ordered associations**. Use an ordered set of pointers similar to Figure 17.8 or a dictionary with an ordered set of pairs similar to Figure 17.9.
- **Sequences**. Same as ordered association, but use a list of pointers.
- **Bags**. Same as ordered association, but use an array of pointers.
- **Qualified associations**. Implement a qualified association with multiplicity "one" as a dictionary object. using the techniques of Figure 17.9. Qualified associations with multiplicity "many" are rare, but you can implement them as a dictionary of object sets.
- **N-ary associations**. Promote the association to a class. Note that there is a change in identity and that you must compensate with additional programming, similar to that for association classes.
- **Aggregation**. Treat aggregation like an ordinary association.
- **Composition**. You can treat composition like an ordinary association. You will need to do some additional programming to enforce the dependency of the part on the assembly.

17.4.5 ATM Example

Exercise 17.1 addresses the implementation of associations from the ATM model.

17.5 Testing

If you have carefully modeled your application, as we advise, you will reduce errors in your software and need less testing. Nevertheless, testing is still important. Testing is a quality-assurance mechanism for catching residual errors. Furthermore, testing provides an independent measure of the quality of your software. The number of bugs found for a given testing effort is an indicator of software quality, and you should find fewer bugs as you become proficient at modeling. You should keep careful records of the bugs that you find, as well as customer complaints.

If your software is sound, the primary difficulty for developers is in finding the occasional, odd error. Fixing the errors is a much easier problem. (In contrast, if your software is haphazard, it can also be difficult to fix the errors.)

You need to test at every stage of development, not just during implementation. The nature of the testing changes, however, as you proceed. During analysis, you test the model against user expectations by asking questions and seeing if the model answers them. During design, you test the architecture and can simulate its performance. During implementation, you test the actual code—the model serves as a guide for paths to traverse.

Testing should progress from small pieces to ultimately the entire application. Developers should begin by testing their own code, their classes and methods—this is called ***unit testing***. The next step is ***integration testing***—that is, how the classes and methods fit together. You do integration testing in successive waves, putting code together in increasing chunks of scope and behavior. It is important to do integration testing early and often to ensure that the pieces of code cleanly fit together (see Chapter 21). The final step is ***system testing***, where you check the entire application.

17.5.1 Unit Testing

Developers normally check their own code and do the unit and integration testing, because they understand the detailed logic and likely sources of error. Unit testing follows the same principles as in pre-OO days: developers should try to cover all paths and cases, use special values of arguments, and try extreme and "off-by-one" values for arguments. If your methods and classes are simple and focused, it will be easier to prepare unit tests.

It is a good idea to instrument objects and methods. You can place assertions (preconditions, postconditions, invariants) in your code to trap errors. You should try to detect problems near the source (where they are easier to understand) rather than downstream (where they can be confusing).

We agree with the use of paired programmers and aggressive code inspection that is part of the agile programming movement. Along the same lines, we also recommend formal software reviews (see Chapter 22), where developers present their work to others and receive comments.

17.5.2 System Testing

Ideally, a separate team apart from the developers should carry out system testing—this is a natural role for a quality assurance (QA) organization. QA should derive their testing from the analysis model of the original requirements and prepare their test suite in parallel to other development activities. Then the system testers are not distracted by the details of development and can provide an independent assessment of an application, reducing the chance of oversights. Once alpha testing is complete, customers perform beta tests, and then if the software looks good, it is ready for general release.

The scenarios of the interaction model define system-level test cases. You can generate additional scenarios from the use cases or state machines. Pick some typical test cases, but also consider atypical situations: zero iterations, the maximum number of iterations, coincidence of events if permitted by the model, and so on. Strange paths though the state machine make good test cases, because they check default assumptions. Also pay attention to performance and stress the software with multiuser and distributed access, if that is appropriate.

As much as possible, use a test suite. The test suite is helpful for rechecking code after bug fixes and detecting errors that creep into future software releases. It can be difficult to automate testing when you have an application with an interactive user interface, but even then you can still document your test scripts for later use.

ATM example. We have carefully and methodically prepared the ATM model. Consequently we would be in a good position for testing, if we were to build a production application.

17.6 Chapter Summary

Implementation is the final development stage that addresses the specifics of programming languages. First you should address implementation issues that transcend languages—we call this implementation modeling. Sometimes it is helpful to fine-tune classes and generalizations before writing code in order to simplify development or to improve performance. Do this only if you have a compelling reason.

Associations are a key concept in UML class modeling, but are poorly supported by most programming languages. Nevertheless, you should keep your thinking clear by using associations as you study requirements and then necessarily degrade them once you reach implementation. There are two primary ways of implementing associations with programming languages—with pointers (for one or both directions) or with association objects. An association object is a pair of dictionary objects, one for the forward direction and one for the backward direction.

Even though careful modeling reduces errors, it does not eliminate the need for testing. You will need unit, integration, and system tests. For unit testing, developers check the classes and methods of their own code. Integration testing combines multiple classes and methods and subjects them to additional tests. System testing exercises the overall application and ensures that it actually delivers the requirements originally uncovered during analysis.

association object	implementation modeling	transformation
association traversal	integration testing	two-way association
dictionary object	one-way association	unit testing
fine-tuning classes	pointer	
fine-tuning generalizations	system testing	

Figure 17.10 Key concepts for Chapter 17

Bibliographic Notes

Transformations provide the motivation for Section 17.2 and Section 17.3. [Batini-92] presents a comprehensive list of transformations. [Blaha-96] and [Blaha-98] present additional transformations.

References

[Batini-92] Carlo Batini, Stefano Ceri, and Shamkant B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. Redwood City, CA: Benjamin Cummings, 1992.

[Blaha-96] Michael Blaha and William Premerlani. A catalog of object model transformations. *Third Working Conference on Reverse Engineering*, November 1996, Monterey, CA, 87–96.

[Blaha-98] Michael Blaha and William Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Upper Saddle River, NJ: Prentice Hall, 1998.

[Rumbaugh-87] James E. Rumbaugh. Relations as semantic constructs in an object-oriented language. *OOPSLA'87 as ACM SIGPLAN 22*, 12 (December1987), 466–481.

Exercises

17.1 (7) Implement each association in Figure 17.6. Use one-way or two-way pointers as the semantics of the problem dictates. Explain your answers.

17.2 (5) Implement each association in Figure E12.3. Use one-way pointers wherever possible. Should any of the association ends be ordered? Explain your answers.

17.3 (4) Implement each association in Figure E15.2. Use one-way or two-way pointers as the semantics of the problem dictates. Should any of the association ends be ordered? Explain your answers.

17.4 (3) Implement the association in Figure E15.3. Use one-way or two-way pointers as the semantics of the problem dictates. Explain your answer.

17.5 (7) Implement each association in Figure E12.4. Use one-way or two-way pointers as the semantics of the problem dictates. Should any of the association ends be ordered? Explain your answers.

18

OO Languages

This chapter discusses how to take a generic design and make the final implementation decisions that are required to realize the design in C++ or Java. These are the two dominant languages used in OO implementation. The goal of this chapter is to produce code for a program. We do not intend to give a C++ or Java tutorial, but to highlight language features and explain their use with models.

Chris Kelsey was the primary author of this chapter and we thank her for her help.

18.1 Introduction

It is relatively easy to implement an OO design with an OO language, since language constructs are similar to design constructs. In this book we will focus on C++ and Java, since they are the dominant OO languages. Even if you are using another language, many of the principles will be the same and the discussion here will be relevant.

C++ and Java have much in common. Java is younger than C++ and borrows heavily from C++ syntax. Both are strongly typed languages, where variables and values must be known to belong to a particular native or user-defined type. Strong typing can improve reliability by detecting mismatched method arguments and assignments, and it increases opportunities for optimization.

18.1.1 Introduction to C++

C++ was developed by Bjarne Stroustrup at AT&T Bell Laboratories in the 1980s, with the intent of extending the widely used, procedural C language to include OO capabilities. It retains the "close to the machine" characteristics that are the hallmark of its C heritage, while adding the "close to the problem" capabilities of more direct expression of OO concepts.

Bell Labs originally implemented C++ as a preprocessor that translated C++ into standard C. As C++ came into mainstream use in the 1990s, direct compilers with symbolic de-

buggers and other development tools appeared. ISO/ANSI standardized the language and its libraries in 1998. C++ compilers for various operating systems are widely available from several major vendors, as well as from the Free Software Foundation.

C++ language syntax is a superset of C. Thus, C++ is a hybrid language, in which some entities are object types and some are traditional primitive types. Because of its origins, it retains features that are inconsistent with "pure" OO programming, such as free-standing functions that are not methods of a class. However, syntax and semantics remain consistent across native and object data types.

C++ supports generalization with inheritance and run-time method resolution (***polymorphism***). Classes may include a mix of polymorphic and nonpolymorphic methods. To enable run-time resolution for a method, a superclass must explicitly declare that method as ***virtual***. The implementation is efficient, typically achieved by having each object contain a pointer to a table of methods for its class. There is no shared basic *Object* type as characterizes other OO languages. C++ permits multiple inheritance.

In addition to overriding methods via inheritance, C++ allows ***overloading***—methods and functions may share the same name but have parameters that vary in number or type. Upon invocation, the language chooses the method or function with the appropriate parameters. Similarly, C++ operators may be overloaded, allowing a method to be expressed intuitively [such as $a + b$ instead of *a.add(b)*, where a and b are of type *ComplexNumber*].

Access specifiers promote encapsulation by restricting the availability of class members (methods or data) to methods of the class itself (***private***), the class and its subclasses (***protected***), or any class/method/function (***public***). A class may grant selective access to otherwise private members with a ***friend*** declaration. C++ performs all access restriction on a class, not object, basis—which means there is no access restriction among same-class objects. C++ ***namespaces*** provide a semantic scope for symbols but do not affect accessibility of visible entities. Namespaces were introduced in part to alleviate name conflicts among external libraries.

C++ exposes its memory management to the programmer, who may customize memory allocation strategies. The compiler statically allocates storage for objects (primitive or class types) declared at compile time. An application may also dynamically obtain storage from the heap at run time by using the *new* operation. A dynamically created object persists in memory until a *delete* operation explicitly destroys it.

Memory addresses remain fixed for the lifetime of an object and identify it. The programmer can get the address of a statically allocated object (using operator &), as well as obtain an address as a handle for a dynamically allocated one. To access the target of a pointer, the pointer is *dereferenced* with the operator *, such that for any object O at address A, it is true that $\&O == A$ and $*A == O$. As with C, the programmer is generally not protected from memory-based errors at compile time or run time. C++ offers no run-time error-detection facilities; errors generally result in undefined program behavior.

In addition to the C-style pointers used as operands in memory operations, C++ includes ***references*** that syntactically appear as object aliases. References must be bound to an existing object at their creation. They are constant in their binding and so cannot be null. Otherwise, references behave much as permanently dereferenced pointers.

Classes have ***constructors*** and ***destructors***, methods that C++ automatically invokes upon creation and destruction of objects. These are typically used for initialization and any operations that must occur upon termination (often deallocation), respectively.

In summary, C++ is a flexible language characterized by a concern for run-time efficiency, the ability to form broad type hierarchies, and uniform semantics. It provides a platform for fine granularity of expression and control at the expense of some simplicity.

18.1.2 Introduction to Java

Java came into being as a by-product of an early 1990s Sun Microsystems, Inc. exploration into programming consumer devices, a project that required a portable and device-independent language. It made its public debut in 1995 as a download on the then-infant World Wide Web, and the ability to run restricted Java programs was soon thereafter incorporated into Web browsers to allow dynamic and interactive content on Web pages. Within a few years, with the explosion of the Internet and distributed application architectures, Java became a staple of commercial development. Commercial, shareware, and free Java tools are widely available, as the language continues to evolve and its libraries grow. Sun maintains control over language releases and makes basic tools available at no charge for most computing platforms.

Java's popularity comes not so much from the core language itself as from its portability and its huge library of associated classes. Java source code compiles to an intermediate bytecode, which in turn runs on a platform-specific Java Virtual Machine. JVMs are available for nearly all operating systems and are readily substitutable with JVMs from other vendors. Native Java compilers are also available.

Just as C++ leveraged the experience of large numbers of C programmers, so Java leveraged the large base of C/C++ speakers. Java syntax is quite similar to C++, although its object and memory models differ significantly.

Java is a strongly typed language, with distinct primitive and object types. The usage of the two differs significantly: Java statically allocates primitives and treats them as value-based variables ($i = j$ assigns j's value to variable i), but dynamically allocates object types at run time and manipulates them only through reference variables ($i = j$ assigns the physical object referred to by j to an alternative reference i). Java has no reference syntax for primitives but does provide tools for conversion between primitives and corresponding object types, such as *int* and *Integer*. Java version 1.5 adds ***autoboxing***, the automatic conversion between primitives and their corresponding wrappers in common circumstances such as parameter passing and use in collections.

All Java object types share a single *Object* ancestor and so are, at the most abstract level, type compatible. Run-time type checking throws exceptions if invalid downcasts (treating an object as a subtype) are used. Polymorphism is automatic, and a programmer can prevent it only by explicitly prohibiting the definition of overriding methods. Java supports only single inheritance, although ***interfaces*** (uninstantiable class specifications containing only constants and method declarations) can emulate multiple superclasses.

Packages organize classes and provide a scope for identifiers. Each source-code compilation unit (file) in Java can contain at most one public class and must bear the name of that class, although the file may define other supporting classes. A declaration at the top of the

file identifies the package. Files using public symbols from a different package use an *import* directive to access the public elements of that package (such as *import java.io.** to use classes in Java's i/o package/library).

Similar to C++'s access specifiers, Java's access modifiers restrict the availability of attributes and methods. Without explicit specification, an attribute or method has by default ***package*** accessibility and may be used by all other methods defined within the package. Explicit options are ***private*** (intraclass access), ***public*** (universal access) and ***protected*** (extends access to subclasses defined outside their parent class's package). Thus the Java meaning of *protected* is a bit different than C++.

Memory management is the province of the JVM. All Java code—data and methods—exists within the context of a class. The system loads and unloads classes as needed and relocates code within memory during run time. The programmer need not, and cannot, know the location of objects—they must be addressed through their corresponding references that syntactically appear as object variables.

Object deallocation is done through garbage collection—when all references to an object have been retired, the system will return object memory to its pool. Although the programmer can suggest object destruction, the system cannot be forced to collect garbage on demand. Java also offers run-time memory error detection (such as array out of bounds and attempted use of null references) and throws appropriate exceptions on detection.

In summary, Java emphasizes run-time portability and code safety at the cost of some efficiency and flexibility. Java provides abstract interfaces to emulate multiple inheritance, distinguishes between object and primitive type semantics, and offers a rich library of objects for both general implementation and for system-level integration on distributed platforms.

18.1.3 Comparison of C++ and Java

Table 18.1 compares C++ and Java.

18.2 Abbreviated ATM Model

Figure 18.1 shows a portion of the ATM model that we will use as an example. We have added *CheckingAccount* and *SavingsAccount* so that we can discuss generalization.

18.3 Implementing Structure

The first step in implementing an OO design is to implement the structure specified by the class model. You should perform the following tasks.

- Implement data types. [18.3.1]
- Implement classes. [18.3.2]
- Implement access control. [18.3.3]
- Implement generalizations. [18.3.4]
- Implement associations [18.3.5]

	C++	Java
Memory management	Accessible to programmer. Objects at fixed address.	System controlled. Objects relocatable in memory.
Inheritance model	Single and multiple inheritance. Polymorphism explicit per method. No universal base class. Encourages mix-in hierarchies.	Single inheritance with abstract interfaces. Polymorphism automatic. Universal *Object* ancestor.
Access control and object protection	Thorough and flexible model with *const* protection available.	Cumbersome model encourages weak encapsulation.
Type semantics	Consistent between primitive and object types	Differs for primitive and object types.
Program organization	Functions and data may exist external to any class. Global (file) and namespace scopes available.	All functions and data exist within classes. Package scope available.
Libraries	Predominantly low-level functionality. Rich generic (template) container (data structures) and algorithm library.	Massive. Classes for high-level services and system integration included.
Run-time error detection	Programmer responsibility. Results in undefined behavior at run time.	System responsibility. Results in compile-time or run-time termination.
Portability	Source must be recompiled for platform. Native code runs on CPU.	Bytecode classes portable to platform-specific JVMs. JVM must be available.
Efficiency	Excellent.	Good. Can vary with JVM implementation.

Table 18.1 C++ vs. Java. Both are powerful languages with different trade-offs.

18.3.1 Data Types

If you have not already assigned data types to attributes, you must do so now. Certain data types merit special consideration.

Primitives

Floating-point numbers, integer types, characters, and booleans can express simple values. Where possible, use numeric values instead of strings. Numeric types typically allow better storage and processing efficiency, as well as easier maintenance of attribute integrity. (See subsequent section on enumerations.)

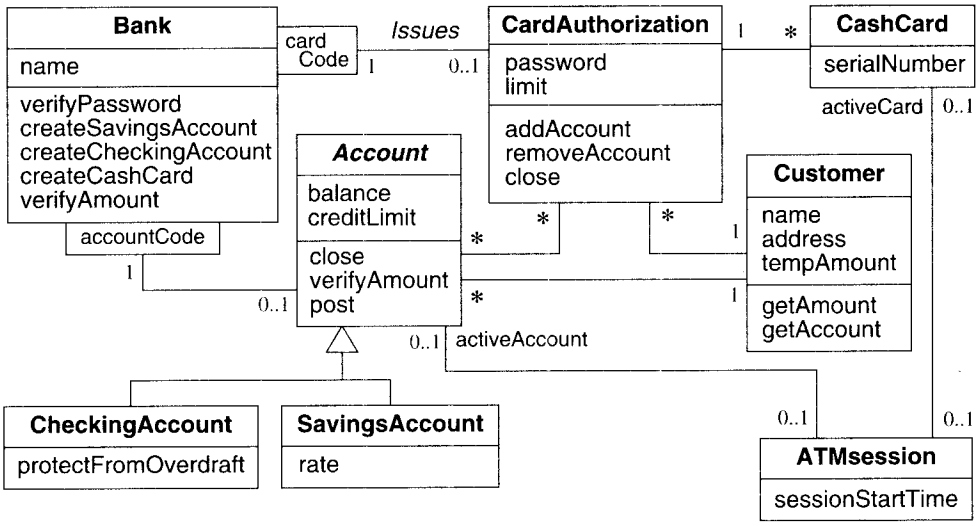

Figure 18.1 Abbreviated ATM implementation class model used in this chapter

Object Types

You can use objects to collect and organize attribute values into richer types. C++ supports the physical nesting of objects, where an instance of one object is physically created within the memory space of another during object construction. Java supports only the referencing of objects as members, so a Java object within another object is analogous to a C++ pointer or reference member that must be explicitly bound to another object.

C++ structs are technically no different than classes except that all members are public by default. They are conventionally used as "POD"—that is, "Plain Old Data"—to bind values together in a logical grouping with no methods. A constructor may be added to initialize data members. Although all members are public, the host object encapsulates access to the struct members.

```
struct address {
    string street;
    string city;
    string state;
    address() : street(""), city(""),state("") {}
};

class Customer {
    string name;
    address addr; // object attribute
    float tempAmount;
public :
    // ...
```

```
    string City() { return addr.city; } // use
};
```

You can follow a similar strategy with Java, although the host object's initialization or constructor must explicitly create the object-attribute instance. Default package access yields access to object-attribute members within the using class.

```
class Address {
    String street = "";
    String city = "";
    String state = "";
}

public class Customer {
    private String name;
       // note explicit construction of attribute
    private address addr = new Address(); // object attribute
    private float tempAmount;
    // ...
    String city() { return addr.city; } // use
}
```

Reference Types

You can use Java object references, and C++ pointers and references, to implement associations. In Java, all class-type variables represent references to objects, while in C++ a variable may directly represent an object. C++ requires extra care that objects are not mistakenly used where a reference type is intended.

Object Identifiers

OO languages have built-in mechanisms for identifying objects, and ways to test object identity. There is typically no need to create explicit object identifier types. If you need a unique object identifier, you can obtain it at run time from the system.

In C++ an object's actual memory address serves as a unique identifier and can be obtained by applying the & (address of) operator to an object or object reference. Object identity can be tested by pointer (address) comparison. Java's == operator provides run-time identity comparison. If a run-time identifier is needed, the universal *Object* superclass contains *hashCode*() and *toString*() methods that, if not indiscriminately overridden, yield unique integer and string identifiers. Although memory addresses are not available in Java, most systems implement these methods based on internal object location.

Do not confuse a unique domain identifier, such as a bank account code or taxpayer number, with an object identifier. A domain identifier describes a domain-dependent property, while an object identifier describes a system-based attribute.

Enumerations

Enumerations provide two advantages—a value domain constraint and symbolic representation of values. For example, CLUB, DIAMOND, HEART, SPADE can express the range of playing-card suits.

C++ can directly implement enumerations. Each enumeration is a discrete type, for which you can define methods and operators. Members that are not explicitly assigned a value take on sequential integral values, starting at 0 if not otherwise specified.

```
enum Card = { CLUB, DIAMOND, HEART, SPADE };
```

There is an implicit conversion from an enumerated type value to integral type, but not vice versa. C++ guarantees that the sizes of objects of enumerated type are large enough to hold the domain's largest value. For example, a variable of the enum {FALSE,TRUE} could be sized as a single bit. Enumerations may appear at global scope or may be class members.

In practice, it can be cumbersome to use C++ enumerations, owing to conversion issues and the burden of redefining operations that are natively available for *ints*. Enums are best used to clarify code by providing symbolic constants. Encapsulated attribute values may be internally stored and manipulated as *ints*, while the public interface uses the constants to restrict parameter values, and method implementations may use the constants as bounds.

```
class Car {
public: // make enum public so clients can use it
    enum direction {N,E,S,W};
private:
    int mph;     // car speed
    int nesw;    // car direction
public:
       // require enum type parameter
    Car(int speed,direction dir) : mph(speed), nesw(dir) {}
       // int allows ++, -- without overloading for enum type
    Car& TurnRight() {if (++nesw >W) nesw = N; return *this;}
    Car& TurnLeft()  {if (--nesw <N) nesw = W; return *this;}
       // ...
};

int main()
{
    Car (15,Car::E): // client uses public enum value
    // ...
}
```

Java versions prior to 1.5 do not include the ability to create enumerated types. Java enums are similar to those in C++. In the absence of enumerations, you can use interfaces (see Section 18.3.4) to group and share constants. The explicit field modifiers *public*, *static*, and *final* are optional here, as they apply by default to all *int* fields specified within an interface. (Note that the "Enumeration" found in the Java class library is a deprecated interface providing rudimentary iteration operations and is unrelated to enumerated types.)

```
public interface Card {
    public static final int CLUB = 0;
    public static final int DIAMOND = 1;
    public static final int HEART = 2;
    public static final int SPADE = 3;
```

```
    ...
}
```

18.3.2 Classes

OO programming languages provide direct support for implementing objects. You must declare each attribute and method in a class model as part of its corresponding C++ or Java class. You will also need to add attributes and methods for implementing associations (see Section 18.3.5). It is good practice to carry forward the names from the design model.

You can regard objects as entities that provide service to client objects that make requests. Services appear as public methods of a class. Methods provide the protocol for obtaining services. In general, method parameters represent information that an object needs to perform a service, and the return value of the method represents the object's response to the client requesting the service.

Because the public interface describes the class's services, it is best to work "outside in" when defining classes. Start with the public interface—methods intended to provide services to others—and add internal methods and attributes as needed to support the public interface methods.

Because the public methods document class behavior, they typically appear at the beginning of a class declaration for easy visibility, although there is no requirement that they do so. Both C++ and Java resolve symbols only after reading an entire class definition, so members can refer to each other in any order.

18.3.3 Access Control

A class should define public methods for services that clients can request or invoke. A class may also specify data that is immutable as public. All other members of the class—attributes and methods used internally to implement public functionality—should be rendered invisible and inaccessible to other objects and functions.

Both Java and C++ rely on ***access specifiers*** (called ***access modifiers*** in Java) to control clients' access to methods and data. The most basic specifiers, applied to attributes and methods, are the same for both languages. Only methods of the class can access ***private*** attributes or methods. Any client can access ***public*** members of the class.

Access Control in Java

Strong encapsulation in Java requires detailed attention to packages as well as access specifiers. A Java package provides a scope that extends access privileges to other entities within that scope. Packages are indicated by a package declaration at the top of a source file; multiple files can thus be grouped into the same package. Unless the *private* access specifier explicitly qualifies an attribute or method, other methods of classes defined in the same package can freely access it. If no package is specified, a class is considered to be in a global default package where all but its explicitly private members are available to all other clients residing in any other source file without package specification.

We recommend that you avoid the default package and instead name packages to manage access control. Furthermore, you should declare all attributes and nonpublic methods as

private—package access alone provides only weak encapsulation. You should guard access to all attributes by wrapping them with appropriate methods.

In addition to attributes and methods, Java classes have explicit access control. A Java class itself must be declared *public* to allow its public methods to be accessed by clients outside its package. See *Visibility and access to classes* below.

Access Control in C++

C++ access specifiers apply to sections of the class declaration. All members are assumed private until the compiler encounters an access specifier. Specifiers can appear anywhere in the declaration and apply until another specifier is encountered. Again, you should guard all attributes and methods other than the known public interface as private and relax access only with sound justification. Access specification within C++ structs is the same, with the exception that members are by default *public* until a more restrictive specifier is encountered (see Section 18.3.1).

C++ allows selective access to private members through a ***friend*** declaration. The class containing the declaration grants access to a named function, method, or class and in doing so allows the named entity full access to its private members. Friendship is best used sparingly, as it provides additional paths to encapsulated members.

Visibility and Access to Classes

For a class type to be recognized and its public members available, it must be visible to its potential clients. For both C++ and Java, the basic compilation unit is the file. Classes in a single file are visible to each other, but in practice it is best to place no more than a single class, along with incidental support classes, in a single file. Thus, class types from outside the file must explicitly be made visible to the compilation unit.

In Java, the package structure governs visibility and access. Classes in files declared to be in the same package are always visible to one another. Packages are dependent on disk file structure—all files declared to be in a package must reside in a directory named with the package name in order to be located by the Java system. For both compilation and execution, Java uses the environmental variable CLASSPATH, which gives a relative starting point for the system to search for packages and classes.

Although a package can contain many classes, each Java source file can have at most one *public* class, and the file must have the same name as the sole public class within. Classes outside the package can use only public methods of public classes. For a public class to be visible to a class in a different package, the client class's source file must include an import directive: *import packagename.classname* or *import packagename.**. The system locates classes by prepending the classpath to the package and optional file name.

For the ATM case study, we would place each of the classes in a separate file. For example, the source file Bank.java would contain:

```
package bankInfo
public class Bank { ... }
```

and the file Customer.java would contain:

```
package bankInfo
public class Customer { ... }
```

The files Bank.java, Customer.java, and other classes in the *bankInfo* package must reside in a disk directory named *bankInfo*. For *Bank* and *Customer* to be visible to the *ATMsession* class, which is not part of the *bankInfo* package, the ATMsession.java source file must first declare:

```
import bankInfo.*
public class ATMsession { ... }
```

The import statement lets the ATMsession implementation see and use the public methods of the public classes found in the *bankInfo* package.

The rules and naming requirements of packages are somewhat onerous, but do not be tempted to simplify by using a global package. Without controlling the visibility and access of the classes themselves, you abandon much of access control. Although private attributes are still respected as private, without packages, methods designed for internal access to those private attributes effectively become public, and nonpublic classes intended for restricted use also become publicly available.

C++ has no language-specified dependence on the location of source code or compiled code. C++ programs divide code into header files, which contain declarations (including classes), and implementation files, which contain the actual code for all but the most simple methods (very small methods are typically implemented within class declarations). To introduce symbols from one file that are needed in another, the *#include<filename>* (when the source resides on a system-specified search path) or *#include "filename"* (when the full path is specified within the quotes) directive is placed at the top of the client file. The directive incorporates the header file containing the declarations of the desired class into the client's compilation unit by substitution, so that the symbols contained in the header are effectively placed into the file at the directive's location.

C++ has a *namespace* feature that serves as a scoping mechanism for program names but does not affect access control. A class is placed into a namespace if it is declared within a namespace declaration in a file and has the fully qualified name of namespace::classname. A namespace can span source files, and a source file can contain multiple namespaces. To avoid cumbersome syntax, the symbols of one namespace can be introduced into another with a *using* directive. C++ standard library names exist within the namespace *std*.

When no namespace is specified, symbols are in the global namespace. Because namespaces impact visibility but not access control, they are used primarily to prevent or disambiguate coincident symbol names as might be found when utilizing multiple libraries.

Access Control in Inheritance Hierarchies

Both languages include the *protected* access specifier, but it works differently in C++ and Java. In C++, a protected member is accessible only by methods of the class and its subclasses.

In Java, protected attributes and methods have package access, which is further extended to methods of subclasses that are defined outside the package where the superclass is defined. Java does not have the equivalent of C++'s more restrictive protected access.

Applying Access Control

A UML model may have annotations—{public}, {private}, {protected}, {package} or +, –, #, ~—indicating access specifiers for class members (called *visibility* in the UML). The lack

of an explicit indication implies that a method is part of the public interface. A thorough implementation should include access specifiers and make attributes private.

18.3.4 Generalization

OO languages provide robust support for generalization through ***inheritance***. A class can serve as the parent ***superclass*** to one or more child ***subclasses***. A subclass inherits all the members of its parent and may add attributes and methods of its own. The subclass may also ***override*** superclass methods to let the child express a behavior of the same name and signature (parameters and return) differently. Subclasses in turn may act as superclasses to successive generations of subclasses, allowing the creation of hierarchies of classes, with each level expressing extended and/or more specific behaviors.

Beyond the convenience of making subclasses easy to specify, the inheritance mechanism lets objects be utilized more abstractly. Inheritance enables ***polymorphism*** (from the Greek "many faces"), where a child type can be addressed as if it were any of its ancestral types. At run time, the system resolves which kind of child is in use, and invokes that subclass's variation of a behavior specified more generally by the ancestor. In a sound hierarchy, each subclass should be able to sensibly perform all superclass behaviors, such that anywhere the parent type might appear, the child type can fulfill the parent's behavioral expectations. [Liskov-88]

You need not fully specify all classes in an inheritance hierarchy. You can simply declare higher-level abstractions, deferring implementation detail to subclasses. These are ***abstract classes***—classes that are described but remain partially or wholly unimplemented. Abstract classes describe types at a level where there is insufficient knowledge to implement a concrete behavior, but the general specification is known. For example, any *Shape* may be drawn, but until the type of *Shape* is specified, it is impossible to know *how* it is drawn.

Because they are incompletely implemented, you cannot create objects of abstract types. Instead, you create ***concrete*** objects of their subclass types—ones that fully implement the parent's specified behavior—and assign a superclass-type reference. The subclass objects are referred to as if they were of the more generic type. For example, a *Square* or *Circle*—concrete subclasses of an abstract *Shape*—might be assigned to a *Shape*-type reference variable. Although both would appear as *Shapes*, when their *draw()* method is invoked, each will polymorphically manifest its proper behavior. In Java,

```
Shape s1 = new Circle();
Shape s2 = new Square();
s1.draw(); s2.draw();
```

will result in drawings of different shapes.

C++ and Java implement somewhat different models of inheritance.

Inheritance in Java

All Java objects share a common ancestor, the *Object* class. *Object* contains minimal methods and fields to support programming concerns such as object identity, equivalence, and concurrency. The common basis also allows the manipulation of objects at an entirely generic level, such as forming collections of otherwise unrelated types. Using objects at an "un-

typed" level of abstraction avoids some of the constraints imposed by a strongly typed language.

Inheritance is implemented with the *extends* keyword.

```
class Account {
    private float balance;
    public void Post(float amount) { ... }
    public float Balance() { return balance; }
}
class SavingsAccount extends Account {
    private float rate; // add interest rate attribute
    float CalcInterest() {
       // calculate unpaid interest due
    }
}
```

A Java class can extend at most one class. However, Java provides some of the benefits of multiple inheritance through interfaces. A Java *interface* is a class specification with no implementation. It consists of method declarations without implementations and may also contain constant fields that are allocated per class, not per object. Like classes, interfaces can be *extended* to create subinterfaces.

To use an interface, a class must declare that it implements the interface, and provide code for all of the interface's methods. A class may extend at most one class and in addition may implement any number of interfaces, allowing a simulation of multiple inheritance. When multiple interfaces are implemented, an object may be referred to through the type of any of its interfaces. Here, methods required for interest-bearing accounts are abstracted into a separate interface, which in turn is implemented by a *SavingsAccount*. This can ensure a uniformity of interest-bearing methods across all classes that implement the interface, be they *SavingsAccount*, *InterestBearingCheckingAccount*, or some other type.

```
interface InterestBearingAcct {
    float CalcInterest();
}
class SavingsAccount extends Account
    implements InterestBearingAcct {
    private float rate;
    public float CalcInterest() {
       // implement interest calculation
    }
}
```

A *SavingsAccount* can now participate in programs as an *Account*, a *SavingsAccount*, or an *InterestBearingAcct*.

Besides concrete classes and interfaces, Java allows abstract classes—incompletely implemented classes that cannot be instantiated, but serve as parents to concrete subtypes. Both the class itself and any unimplemented methods must be specified as *abstract*.

```
public abstract class AbstractExample {
    void method1() { /* ... */ }
```

```
    abstract void method2 ();
}
```

A *public* class, because it is accessible to any client that has imported the package in which it resides, can serve as a superclass to a subclass defined in a different package. Subclasses defined outside their superclass's package can access protected, as well as public, members of their parent. Private members of a superclass are not visible to subclasses. (See Section 18.3.3.)

To prevent further subclassing of a type, a class can be declared ***final***. Similarly, a ***final*** qualifier applied to a method prevents that method from being overridden.

Inheritance in C++

C++ classes share no common parent; class hierarchies can start arbitrarily with any class. A superclass is called a ***base***, and a subclass is called a ***derived*** class. A ***direct base*** is an immediate parent of a derived class, while an ***indirect base*** is a more remote ancestor.

In C++, unlike Java, polymorphism is not automatic. A class may mix methods that are automatically resolved by type at run time with those that are not. To activate polymorphism, a method must be declared as ***virtual***. Those that are not declared as *virtual* will be invoked according to the type by which the object is referenced, not the actual type of the derived class, even if the derived class overrides the method.

```
class Hello { //...
    public:
       void method1() { cout << "hello\n"; }
       virtual void method2() { cout << "hello\n"; }
};

class Goodbye : public Hello { //...
    public:
       void method1() { cout << "goodbye\n" ; }
       void method2() { cout << "goodbye\n"; }
};

int main() {
    Goodbye g;
    Hello& h = g; // same object through base type

    g.method1(); g.method2(); h.method1(); h.method2();
}
```

The output is:

```
goodbye  // method1 not virtual; called via derived
goodbye  // method2 is virtual; called via derived
hello    // method1 -- not virtual! -- via base
goodbye  // method2 via base
```

C++ does not provide a way to prohibit overriding of methods, but in a hierarchy where some methods are overridden and others are not, a nonvirtual method may indicate the authors' intent that the base implementation should be inherited intact and left unmodified.

C++ incorporates access specification in inheritance syntax. A derived class may be specified as *public*, *protected*, or *private*. Public inheritance specifies that all public methods inherited from the base remain public in the derived class.

```
class SavingsAccount : public Account { ...}
```

In private inheritance, all methods become private to the derived class. Private inheritance is used to indicate that a derived class is implemented in terms of its base, but it is not intended to describe a logical *is-a* relationship where the base should be considered an abstraction of the derived class. In practice, encapsulating the base type as a member of the (would-be) derived class (using delegation, see Section 15.9.3) is often the better strategy. Protected inheritance dictates that public methods in the base become accessible only to further derivations. In practice, this is rarely used.

Abstract classes are created by the inclusion of at least one pure virtual method, which is a virtual method that uses "initialization to 0" syntax at declaration: *void fn() = 0*. Such classes cannot be instantiated, and their derived classes are inherently abstract unless they implement all pure virtual functions inherited from their base.

C++ supports multiple inheritance, although in practice the best combinations of parent classes are mixes of concrete types and predominantly abstract classes that represent behavior specifications. The latter perform somewhat like Java interfaces, with the significant convenience that default implementations for methods may be written where appropriate and be maintained at a base level instead of requiring maintenance across derivations.

```
class Account { // nonabstract
    public:
          // assume Post implementation may be
          // specific to derived account types;
          // base implementation is default way.
       virtual void Post(float amount) { ... }
       float Balance() { return balance; }
    private:
       float balance;
};

class InterestBearingAcct { // abstract class
    public:
       virtual float CalcInterest() = 0; // "pure virtual"
       float Rate() { return rate; }
    private:
       float rate;
};

class SavingsAccount : public Account,
       public InterestBearingAcct {
    public:
       virtual float CalcInterest() {
       // calculate unpaid interest
```

```
        }
};
```

18.3.5 Associations

OO languages lack direct support for associations. However, you can readily implement links with object references or distinct association objects. (See Section 17.4.) You should promote the association to a class if there are attributes describing the association itself. We also recommend promotion for n-ary and qualified associations.

One-Way Associations

One-way associations reduce interdependencies among classes. When one class references another, the referenced class must be visible and accessible to the hosting class. The referencing class must enforce and maintain the association, and typically utilizes the interface of the referenced class. To the extent these dependencies can be reduced to one side of an association, maintenance is reduced and reusability may be enhanced.

For example, in Figure 17.7, if we do not need to retrieve the collection of employees for a company, a pointer from *Person* to *Company* will suffice. In Java, the *Person* class can simply contain a *Company* field.

```
public class Company { ... }
public class Person {
    private Company employer;
    ...
}
```

C++ offers two options for implementing referencing attributes. Most commonly, a pointer may be used, which allows a link to be changed. Assuming a *Person* may change employers leads to the following C++ code. (This C++ code also permits a *null* employer, which is inconsistent with the multiplicity in Figure 17.7. Update methods would have to prevent a *null* employer to enforce the multiplicity.)

```
class Company { ... }
class Person {
    Company* employer;
    ...
}
```

A C++ reference member implies a permanent link in which the containing object has a dependency on the attribute object. References must be bound at initialization. They cannot be null, nor can they be assigned (changed), so the language enforces the preexistence of the link target. Assuming that an *Account* is issued for a particular *Bank* and is not transferable, and that no *Account* may be issued without a *Bank*,

```
class Bank { ... }
class Account {
    Bank& bank;
    ...
}
```

This reference-binding requirement of C++ necessitates the existence of a *Bank* prior to the construction of an *Account* (see Section 18.4.1). You can achieve a partial constraint in Java by declaring an object-type attribute as *final*, which ensures that a reference variable cannot be reassigned to a different object, although it does not entail the preexistence requirement.

When implementing associations through referencing attributes, take care not to subject the objects involved to inadvertent changes. An object that hosts a referencing attribute can potentially open a back door by inappropriately exposing the referenced object itself or its attributes. It is particularly important that attributes representing links should be well encapsulated and be modified or reported only through intentional and safe methods. C++ offers an added level of security with the ability to apply the *const* qualifier to referenced objects in situations where they may be intentionally exposed, but are not intended to be modified in the context of the association. Java does not distinguish between constant and mutable objects.

Two-Way Associations

Two-way associations entail link maintenance for both association ends. You can implement a one-to-one association with either a single reference on each end or an association object. Similarly, a one-to-many association requires a single reference on one end and a collection of pointers on the other end or an association object. For example, a customer may have several accounts and we want to be able to navigate this association in both directions. Both Java and C++ have collection object types available from their libraries that can represent the "many" side of the link. In Java,

```
public class Account {
    private Customer customer; // the 1 side of one-to-many
...
}

import java.util.* // to access HashSet class
public class Customer {
       // list of account ref's
    private HashSet acccunts = new HashSet();
...
}
```

Association Classes

Association classes can increase independence of the objects involved by removing direct references to related classes, but they do so at some loss of efficiency and increase in complexity of implementing operations. Independent association classes decouple linked objects but require the overhead of navigating through the link.

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. Both Java and C++ have library support for map objects. To encapsulate links and make link maintenance more intuitive, the maps may be wrapped into an application class that serves as a manager for construction or use of the objects involved.

Choosing an Implementation

If the model has not already prescribed implementation of an association, your choice may depend more on the scale, architecture, or enterprise environment of an application than on the nature of the association itself. If you are constrained from modifying existing classes, an association class may be preferable. Association classes often best implement one-to-many associations where the "many" side can be quite large, or is sparse, and can provide independent and extensible management operations where required.

For simple associations, referencing attributes provide navigation across links. Where possible, one-way associations provide more efficient and safer implementations. Care should be taken to encapsulate links and to prevent inadvertent exposure of a referent's information or interface through a host object.

18.4 Implementing Functionality

Once you have the structure in place, you can start implementing methods. For each class, specify methods by signature (method name, parameters, return type). The class model implies many methods. Obviously, you can create and destroy objects and links as well as access attribute values. More subtly, you can traverse a class model leading to additional methods. Methods also arise from derived attributes, the state model, and the interaction model.

- Object creation. [18.4.1]
- Object lifetime. [18.4.2]
- Object destruction. [18.4.3]
- Link creation. [18.4.4]
- Link destruction. [18.4.5]
- Derived attributes. [18.4.6]

Objects have state, behavior, and identity. This is reflected in the object life cycle. Objects are created by the system and should be initialized in a valid state. C++ has facilities for destruction of objects on demand, while Java can only suggest destruction to its garbage collector. Both have special methods that run at object creation, and both can specify behavior at the termination of an object's lifetime.

Object can request services or information from one another. They do so by invoking other objects' methods. In both Java and C++, object behaviors are invoked using a *membership operator* '.' to indicate that the right-hand operand—a method of the object's class—should be invoked on the named target object. The same membership syntax accesses attributes—*X.y* names the *y* attribute of the object *X*, though normally attributes are encapsulated and so not reachable from another object.

Within the context of a class, objects have an implicit reference to self, called ***this*** in both Java and C++. You need not explicitly qualify member names within class definitions: In Java, *fn()* means *this.fn()* and *y=10* means *this.y=10*. In C++, *this* is a pointer, and so the pointer membership operator "->" would be used, *this->fn()*.

For an object to call upon another object for services or information, it must have a name by which it can access its target. This handle may be provided in the form of a link, where the requesting object has a pointer or reference to another, or through a parameter, where one object receives a handle to another at method call. Any invocation of the target objects' methods (or, rarely, the use of target objects' attributes) takes place through the target handle and is governed by the target's members' access specifications.

Both C++ and Java allow *static* class members that are shared by all objects of a class type. Although they are subject to access specification, as any other data or method of a class might be, these static elements have entirely different life cycles than conventional objects and can be accessed through the class itself, apart from any particular instances that may exist. (See Section 18.4.2.)

18.4.1 Object Creation

In OO languages, objects are typically created dynamically (during run time), through a request to the system to create an object of a particular type. In Java, this is a requirement for nonprimitive types, while primitive types are statically (at compile time) allocated simply by declaration. In C++, both object and primitive types can be either statically or dynamically allocated, and the results are considered as objects in either case. Both Java and C++ use the ***new*** operator to create objects. In C++,

```
// static allocation -- creates a single account
Account acct1;

    // statically allocate an array of 10 accounts;
Account accounts1[10];

    // dynamic allocation: define a pointer,
    // initialize with result of new operation
Account* acct2 = new Account;

    // dynamically create array of 10 accounts
Account* accounts2 = new Account[10];
```

In Java,

```
    // create a reference variable,
    // initialize with result of new operation
Account acct = new Account();

    // create array of 10 references -- no accounts are made!
Account accounts = new Account[10];

    // now create the accounts for the array:
for (int i = 0; i < 10; i++) accounts[i] = new Account();
```

When a new object is created, the system allocates storage for its attribute values and performs other chores involved with the start of the object life cycle. OO languages free the programmer from having to understand technical details of object implementation such as

object layout and internal identifiers. Java and C++ also let the programmer specify operations to occur at the time of object creation, so that you can ensure that objects come into existence in a valid logical state. Once the system has completed its creation chores, a special method, called a ***constructor***, is automatically invoked. The constructor takes the form of a method, with no return value, that shares the class name. It may have any number of parameters and may be overloaded. For example, in Java, the following code specifies that *Account* has two constructors, one requiring an opening balance and one that takes no arguments.

```
public class Account {
...
    public Account(float OpeningBalance) {
       balance = OpeningBalance;
    }
    public Account() { balance = 0; }
    ...
}
```

Constructors may be used to assign values to members, to create member objects, or perform other start-of-life processing on an object. Constructors run after the object is fully formed, so they may call other methods and in general have the same capabilities, privileges, and constraints of other methods. If no constructor is defined for an object, a constructor of the form *X(){}* is presumed to exist. Though it performs nothing, it allows creation expressions matching its imaginary declaration. Once you create a parameterized constructor, the system will no longer provide the no-argument form for you. If you need one, you will have to overload *X(...)* with *X()*, as done above for *Account*.

Understanding constructors in subclasses is not difficult, if you consider object construction. Subclass-type objects inherit their attributes and methods because they contain their parent type, plus any members they add at the subclass level. You can think of it as if their parent type is created, and then their own piece is added on, and so while constructors are not inherited, they run sequentially from the most general through the most specific level. In the case of C++, where objects can be statically allocated, member objects get constructed recursively—including the invocation of their respective constructors—before the host object can be fully constructed. Only after the host object is complete does its own constructor run. You can see this in action (here in C++).

```
class X {
    public:
       X() { cout << "X!"; }
};

class Y : public X {
    public:
       Y() { cout << "Y!"; }
};

class Z : public Y {
    public:
```

```
        Z() { cout << "Z!"; }
};

int main() {
    Z z; // simply make a Z...
    return 0;
}
```

Running the code produces the result: X!Y!Z!

It is very poor practice to define no constructors for a class. By default, C++ performs no default initialization of object members, while Java initializes members to 0 or null according to type. Neither provides a valid-state object for most classes. The purpose of the constructor is to have a way to perform initialization and operations on an object so that, at the time of its entry into the program domain, it is a fully formed, logically intact, safe-to-operate-on instance of its type.

If there are no start-of-life operations to be performed by a newly constructed object, Java allows nonzero default initializations to be performed within the class definition. In C++, initialization take place within a *member initialization list* that specifies initial values for members before entry into the constructor code block. Both languages allow assignment to members within the body of the constructor, although initialization should be preferred over assignment wherever possible.

In Java,

```
public class Account {
    private float balance = 0; // initialization
    ...
    public Account() {} // constructor need not assign
}
```

In C++,

```
class Account {
    float balance; // values not allowed
    ...
public:
    Account() : balance(0) {} // initialization via list
}
```

In C++, the system also supplies a copy constructor, that specifies the semantics of copying by assignment. Like the default (no-argument) constructor, if one is not written for the class, a constructor of the form *X(const X& x)* is provided, with the default value-based meaning of shallow (memberwise) copying. Java discourages object copying by requiring the programmer to specify that a class implements the *Cloneable* interface and override *Object's clone()* method.

18.4.2 Object Lifetime

Statically allocated (created at compile time) objects, which may be any type in C++ and primitive types and reference variables in Java, exist within the scope of a code block, indicated by { }. They are automatically destroyed when program control passes out of their

scope. Dynamically allocated (*new*'d) objects persist in memory from the time of their creation until they are explicitly destroyed in C++, or no longer in use by the program in Java (see object destruction below).

Both C++ and Java allow class members to be qualified as ***static***. Static members belong not to an instance of a class, but to the class itself, and (if public) can be accessed in the absence of any instance of the class: (C++) *X::StaticMethod()* or (Java) *X.StaticMethod()*. As a convenience, both static data and methods can be addressed through a particular object, but neither are instance members of that object.

Static data members come into existence before any particular object of the class type is created—at program entry in C++ or at class loading in Java—and remain in existence throughout the program. In C++, static members exist as independent objects (memory allocations), while in Java each class itself has an independent instantiation (the ***class instance***), of which statics are considered members. Although statics may also be accessed within the scope of their class during method definition or through particular object instances, there is only one instance of a static data member per class, and its life cycle is entirely independent of any object through which it is accessed. Because static methods are not members of a particular object, the notion of *this* (and *super* in Java) has no meaning within the context of a static method, and regular instance data cannot be referenced without qualification with a specific object reference.

In C++, variables defined at global scope (not within a function, including *main*, method, or as a member of another object) have a life cycle much like class-based statics. They are created before the program starts, and are destroyed at program termination. While global data is discouraged as poor practice, global functions are commonly used in C++. Although Java does not allow free-standing functions to be defined at global scope, global functions are regularly emulated in utility library classes that contain groups of static methods. Several of Java's library classes, such as *Integer* or *Collections*, are predominantly or wholly composed of static methods.

Because static members and class instances are created apart from object instances, the initialization of statics occurs independently. Statics in C++ are declared within, but defined and initialized outside, the class, while Java static fields may be initialized conventionally or within a static initialization block. Both C++ and Java statics are guaranteed to be initialized only at some point before use in a program, but neither language specifies a relative instantiation or initialization order.

18.4.3 Object Destruction

When an object is no longer needed, it may be destroyed and its storage returned to the memory pool. With statically allocated (compile-time) objects, this happens as they fall out of scope. If, for some reason, you retain a handle to an object that has fallen out of scope, using it would be an error, somewhat analogous to calling the telephone number for someone who has moved and is no longer available at the number you called.

For dynamically allocated objects, with their potentially unconstrained lifetimes, the problem becomes one of eliminating objects that are no longer of interest, or whose existence is no longer logically valid, and recycling their space in memory. For this, C++ and Java

each pursue one of two basic strategies: Either the programmer takes responsibility for explicitly removing objects that are no longer needed or sensible (C++), or the system keeps an eye out for whether an object is actually in use or not (Java). The latter approach, called ***garbage collection***, is typically implemented in a way that the system tracks references to the object. When there are no remaining references (no valid handles to the object exist any more), the system tags the object as eligible for recycling. The system periodically runs a *garbage collector* that reclaims tagged objects' space for the memory pool.

Manual disposal of objects, as C++ practices, places a burden on the programmer to be aware of allocations and responsible for object cleanup. However, it also allows tight control of the object life cycle and the ability to fully model object events, including exit or termination of an object from its domain. Garbage collection frees the programmer from memory management chores, but restricts the ability to exercise end-of-life-cycle control.

C++ classes can have ***destructors***, analogous to constructors, that run automatically at the destruction of an object. Destructors take the form of a no-argument constructor, with the ~ before the class name: *X::~X() { ... }*. They never take arguments and have no return value.

Because C++ object destruction can be explicitly invoked and predictably executed, it is common for the destructor to undo things a constructor has done. Most common is the allocation and deallocation of dynamically created object members, but this pairing of functionality may include increment/decrement of counts, acquisition and release of resources, and so forth. Destructors can also be used to predictably invoke any logically necessary (often cleanup) behaviors that are required at end of object life.

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

The process of destruction is the inverse of construction: First, the destructor code runs, then any members that themselves require destruction are in turn destructed (starting with their destructors), and finally the whole of the memory is returned to the heap.

To explicitly destroy a dynamically allocated object, the ***delete*** operator is used. Empty array brackets [ ] are used to differentiate the deletion of individual objects from the deletion of a dynamically allocated array.

```
class X { ... }
...
X x1 = new X;
X* *x2 = new X[20];
...
delete x1;
delete [] x2;
```

After deleting a C++ object, it is advisable to immediately assign 0 to any pointers that were pointing to it. *Delete* is implemented to have no effect when (accidentally) called on a null pointer, whereas "double deleting" an object results in undefined—and often disastrous—behavior.

Java objects are not explicitly destroyed by the programmer. Java discerns when there are no more references to an object in use, at which time the object becomes eligible to be destroyed at the garbage collector's convenience. The programmer can attempt to induce destruction, by explicitly setting reference variables to *null* to detach them from objects, and may even suggest garbage collection, but cannot guarantee timely destruction of an object.

Java does allow classes to override *Object's finalize()* method, which acts in a manner similar to the C++ destructor. However, because the exact time of object destruction cannot be triggered, you cannot rely on any operations within *finalize()* to occur at a predictable time.

18.4.4 Link Creation

Links are forged and destroyed (valued or set to null) as objects interact with one another. Referential links should be created in the course of whatever behavior (method) initiates an association between objects, and destroyed in the course of whatever behavior terminates the association. In general, you should avoid direct *set* methods in favor of operations that encapsulate data and forge links only under validated conditions.

To create a link in a one-way association, an object retains a handle to a parameter object. For bidirectional links, handles can be exchanged, although it is usually preferable to initiate the exchange from one side and encapsulate the other to ensure consistent logic and full link updates. Which class governs the exchange is a matter of application logic.

In the following simplified Java example, *Students* maintain a list of current *Courses*, and *Courses* maintain class lists of current *Students*. Because the decision to take a course is generated by a student and conditioned on the student's status, the creation of links is governed by the *Student.addClass(Course)* method. The *addClass* method in turn calls the *Course's enroll(Student)* method, which is accessible to *Student* due to the class's shared package. However, package access prevents public access to clients (in this case, the class *EnrollmentApplication*), thereby restricting link creation to the orderly process prescribed by *Student's addClass* method.

```
// file Course.java
package school;
public class Course {
    private String title;
    private HashSet students = new HashSet();

    public Course (String nm) { title = nm; }
    public String courseName() { return title; }

    boolean enroll(Student stu) {
       // check that course isn't already full, etc.
       // if ok, return result of HashSet.add method
       return students.add(stu);
    }
```

```
    public void printClassList() {
        System.out.println("Class List for " + courseName() +
            ":");
        Iterator it = students.iterator();
        while (it.hasNext())
            System.out.println(
                ((Student)it.next()).studentName());
    }
}

//file Student.java
package school;
public class Student {
    private String name;
    private HashSet classes = new HashSet();

    public Student(String nm) { name = nm; }
    public String studentName() { return name; }
    public boolean addClass(Course crs) {
        // validate student's ability to take this course;
        // if ok, request course to enroll student;
        // if course returns true, add course to list
        return ( crs.enroll(this)) ?
            classes.add(crs) : false ;
    }

    public void printCourses() {
        System.out.println("Courses for " + studentName() +
            ":");

        Iterator it = classes.iterator();
        while (it.hasNext())
            System.out.println(
                ((Course)it.next()).courseName());
    }
}

// file EnrollmentApplication.java
import local.school.*
public class EnrollmentApplication {
    public static void main(String [] args) {

        Student mike = new Student("mike");
        Student bill = new Student("bill");
        ...
        Course tt = new Course("Type Theory");
```

```
        mike.AddClass(tt);
        bill.AddClass(tt);

        tt.PrintClassList();
        mike.PrintCourses()
    }
}
```

Objects can be constructed with knowledge of their collaborators, and so establish links at the time of object creation. Often, it is desirable to condition object creation on establishment of links. In such cases, it is necessary to delegate construction to "factory" techniques that examine criteria before invoking construction, as neither Java nor C++ constructors can return an error, nor can they abort the construction process. Factories may be classes, or simply static methods (see Section 18.4.2).

Here (in C++), a *Transaction* will not be constructed until an account number is validated and a handle to the associated account obtained.

```
class Transaction {
    ...
    protected:
        // private or protected constructor to prevent
        // public construction
        Transaction(Account& acct) { ... }

    public:
        ...
        static Transaction* MakeTransaction
            ( const char* acctNumber) {
            Account* acct;
            // null pointer return indicates no transaction
            if (  /* account is not ok */ )  return 0;
            // return a  transaction for valid account
            return new Transaction(*acct);
            ...
        }
};
```

18.4.5 Link Destruction

Link destruction is generally the inverse of link creation. The activity that dictates the dissociation of objects is the method where a referential link is typically broken. In most cases, this simply means the handle attribute is set to a null value, or an entry is removed from a collection of links. A *Student* might drop a *Course*.

```
// Student.dropClass(Course):
    public boolean dropClass(Course crs) {
        if (!classes.contains(crs)) return false;
        if (!crs.drop(this)) return false;
        classes.remove(classes.indexOf(crs));
```

```
        return true;
    }

// Course.drop(Student):
    boolean drop(Student stu) {
        if (!students.contains(stu)) return false;
        students.remove(students.indexOf(stu));
        return true;
    }
```

When links involving independently existing objects have been created, typically each object is intended to continue existence beyond the life of the link. Care must be taken that the referenced object is accessible through other references, or that a handle to a referenced object is captured from a link-destroying method. Failure to do so can result in a memory leak (C++) or an inaccessible object that may be garbage-collected in Java.

In C++, a common idiom is to pair constructor-destructor activity to implement the creation and destruction of links. Those links often reflect the acquisition and release of resources.

Links involving objects that are existentially dependent on one another may require destruction of one of the linked objects. This, in turn, may imply the updating or destruction of other objects that may be linked to the dependent object, or suggest that the link sustaining the dependent object not be destroyed. This is analogous to database operations, where record deletion may be propagated (cascaded) to related records or prohibited in order to preserve the integrity of existing records.

18.4.6 Derived Attributes

For accuracy and currency, it is always desirable to calculate fresh values from independent data at the time of use. Usually the time required to compute fresh values is negligible when weighed against the costs of managing updates and redundancy for stored derived data.

It is a common mistake to include redundant state data in objects. States can often be determined by examining other attributes (in Java).

```
public class Account {
        // wrong: redundant state data for overdrawn
    private boolean overdrawn;
    private float balance;
    ...
        // compute state upon request instead
    public boolean isOverdrawn() { return balance < 0; }
    ...
}
```

States can also be determined via links (here C++):

```
class Customer {
    ...
    List<Account> accts;
public:
```

```
    ...
    bool hasOverdrawnAccount() {
        // iterate through all accounts;
        // return true if any are overdrawn
    }
}
```

or may be implied by the presence of links (again, in C++):

```
class Telephone {
    Telephone* connectedTo;
    ...
    public:
        // are we connected to another phone?
        // if our link isn't null, we are...
    bool isBusy() { return connectedTo != 0; }
    ...
}
```

18.5 Practical Tips

Here are tips for using C++ and Java to implement an OO design.

- **Enumerations**. Use enumerated types for clarity and enforcement of domain values. (Section 18.3.1)
- **Java packages**. Avoid the default package and instead use named packages to manage access control. (Section 18.3.3)
- **Access control**. Declare all attributes and nonpublic methods as *private*. Relax encapsulation only if it is essential to do so. (Section 18.3.3)
- **C++ friend**. Use the friend declaration sparingly, because it can compromise encapsulation. (Section 18.3.3)
- **Java interfaces**. Consider using Java interfaces as a workaround for multiple inheritance in a model. An interface declares methods and constant fields and also provides a type for accessing objects. (Section 18.3.4)
- **C++ private inheritance**. Avoid private inheritance of classes in C++. Delegation is a better strategy. (Section 18.3.4)
- **One-way associations**. Use a one-way association when two-way association traversal is not needed. One-way associations are easier to maintain and reduce object interdependencies. (Section 18.3.5)
- **C++ reference and Java final**. Note association ends that must be bound at initialization and cannot be changed. C++ references can fully enforce these semantics, and the Java *final* property can partially enforce them. (Section 18.3.5)
- **Constructors**. It is a very poor practice to define no constructors for a class. A constructor should always initialize an object to a valid initial state. (Section 18.4.1)

- **C++ deletion**. When deleting a C++ object, it is advisable to immediately set its pointer(s) to 0. Delete has no effect for a null pointer, but accidental deletion of an already deleted object can be disastrous. (Section 18.4.3)
- **Link destruction**. When a link is destroyed, make sure the associated objects are accessible through other handles or intentionally destroyed. Otherwise you have a memory leak (C++) or an object that is inadvertently garbage collected (Java). (Section 18.4.5)

18.6 Chapter Summary

It is relatively easy to implement an OO design with an OO language, since language constructs are similar to design constructs. C++ and Java are the two dominant OO languages, and hence the subject of this chapter.

The first step in implementing an OO design is to implement the structure specified by the class model. Begin by assigning data types to attributes. Where possible, use numeric values instead of strings. Numeric types typically allow better storage and processing efficiency, as well as easier maintenance of attribute integrity.

Next define classes. It is best to work "outside in" by starting with the public interface. Then add internal methods, attributes, and lesser classes as needed to support the interface methods.

You should pay careful attention to access control—it provides the means to encapsulate attributes and methods and limit access to them. The most basic specifiers are the same for both C++ and Java. Only methods of the class can access private members. Any client can access public members. The *protected* specifier in C++ limits member access to the class and its subclasses. Java is more permissive, letting attributes and methods in the same package also access protected attributes and methods. Java packages are important both for organizing code and controlling access. The C++ *friend* declaration allows selective access to private members.

OO languages provide robust support for generalization through inheritance. C++ supports multiple inheritance, but Java is limited to single inheritance. However, Java does provide some of the benefits of multiple inheritance through the use of interfaces. A Java interface is a class specification with no implementation.

In Java polymorphism is automatic, though you can prevent further subclassing by declaring a class as *final*. Similarly, a *final* qualifier applied to a method prevents the method from being overridden. In contrast, polymorphism is not automatic in C++. To activate polymorphism, a method must be declared as *virtual*.

OO languages lack direct support for associations. However, you can readily implement links with pointers/references or distinct association objects. One-way associations reduce interdependencies among classes, so you should use them when traversal is needed in only one direction. C++ offers two options for implementing association ends. A pointer lets a link be changed or set to null. In contrast, a reference must be bound at initialization and can-

not be changed. Java lacks the full equivalent to the C++ reference—you can define an object-type attribute as final, which ensures that a reference variable cannot be changed but does not require binding at initialization.

When you need to traverse an association in both directions, you should use a two-way implementation. You can bury a pointer/reference (or set of pointers/references) in each related class or use an association object. With two-way pointers/references, you have redundancy and must be careful to keep both directions mutually consistent.

Once you have the structure in place, you can start implementing methods. You should carefully define constructors for new objects and be sure to initialize them to a valid state. Similarly, you should pay attention to destructors, being sure to release any system or program resources that are no longer needed upon the end of the object.

You should seldom include redundant data in objects; for accuracy and reliability it is usually better to calculate fresh values from independent data at the time of use.

abstract	derived data	interface	private
access modifier	destructor	namespace	protected
access specifier	enumeration	new	public
association	final	overloading	reference
concrete	friend	package	static
constructor	garbage collection	pointer	virtual
data type	generalization	polymorphism	

Figure 18.2 Key concepts for Chapter 18

Bibliographic Notes

[Stroustrup-94] provides an interesting discussion of language-design issues.

References

[Arnold-00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Third Edition*. Boston: Addison-Wesley, 2000.

[Gosling-00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Third Edition*. Boston: Addison-Wesley, 2000. (Available online: http://java.sun.com/docs/books/jls/)

[Liskov-88] Barbara Liskov. Data abstraction and hierarchy. *SIGPLAN Notices, 23*, 5 (May 1988).

[Stroustrup-94] Bjarne Stroustrup. *The Design and Evolution of C++*. Boston: Addison-Wesley, 1994.

[Stroustrup-97] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Boston: Addison-Wesley, 1997.

Exercises

18.1 (1) Assign a data type to each attribute in Figure E18.1.

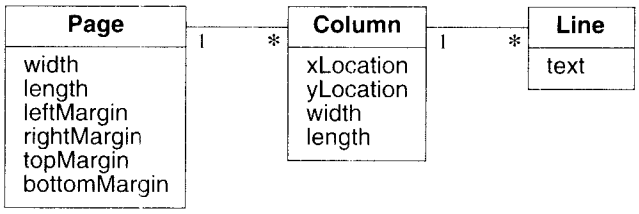

Figure E18.1 Portion of a class diagram of a newspaper

18.2 (4) Consider the model in Figure E18.2. For now, ignore the attributes, methods, and association. See Exercise 15.10 for an explanation of the model.

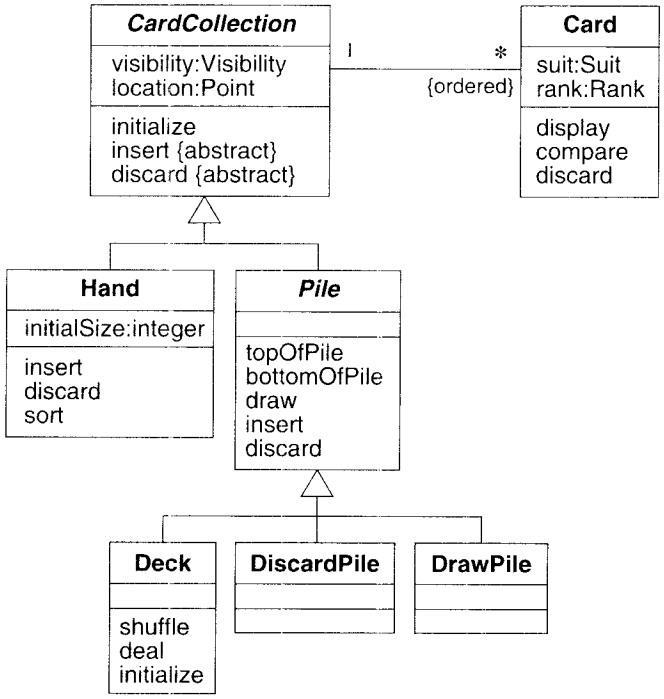

Figure E18.2 Portion of a class diagram of a card playing program

a. Prepare C++ declarations for the classes and the inheritance. Pay careful attention to access control.

b. Prepare Java declarations for the classes and the inheritance. Pay careful attention to access control and packages.

18.3 (6) In Figure E18.2 *visibility* controls whether the front or the back of a card is displayed. Add to your answer for Exercise 18.2 but, for now, continue to ignore the methods and association. [Instructor's note: You may want to give the students our answer to Exercise 18.2.]
a. In C++, add declarations for all attributes, except location. Also define the enumerations.
b. In Java, add declarations for all attributes, except location. Also define the enumerations.

18.4 (7) Add the *CardCollection—Card* association to your C++ and Java answers from Exercise 18.3. [Hint: You can use a template which is not explained in this book. You can answer the exercise by considering the implementation of associations and reading about the standard library and templates on the Web or in a C++ language book.]

18.5 (6) Declare methods for your answer for Figure E18.2 for both C++ and Java. See Exercise 15.10 for an explanation of the methods.

18.6 Write C++ or Java code, including class declarations and methods, to implement the following using pointers.
a. (9) One-to-one association that is traversed in both directions.
b. (6) One-to-many association that is traversed in the direction from one to many. The association is unordered.
c. (6) One-to-many association that is traversed in the direction from one to many. The association is ordered.
d. (8) Many-to-many association that is traversed in both directions. The association is ordered in one direction, and unordered in the other direction.

18.7 (7) Describe strategies for managing memory, assuming that automatic garbage collection is not available. Your answer should provide guidelines that a programmer could use during coding.
a. **A system for text manipulation.** The system often creates one large string in contiguous memory from several smaller strings. You cannot waste memory and cannot set an upper bound on string length or the number of strings to combine. Write pseudocode for a method that combines strings, recovering memory that is no longer used.
b. **A multipass 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 methods 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 and let the operating system allocate a large amount of virtual memory. (2) Recover deallocated objects with garbage collection.
c. **Software that runs for a long time, such as banking software or an air traffic control system.** You have the same computer system and run-time library as described in Exercise 18.7(b). Discuss the relative merits of the two approaches.
d. **A method which may create and return an object that uses a large block of memory.** Discuss the relative merits of the following two approaches: (1) Each time the method is called, it destroys the object created the last time it was called, if any. (2) Each time the method is called, it may create a new object. It is up to the calling method to destroy the object when it is no longer needed. Comment on these two approaches.

18.8 (6) How might you organize Figure E12.4 into Java packages?

18.9 (7) Write C++ and Java declarations for the model in Figure E18.3. Implement the entire model. Be sure to consider access control and Java packages. Pay attention to constructors and destructors. Implement all associations as bidirectional. See the exercises in Chapter 12 for an explanation of the model.

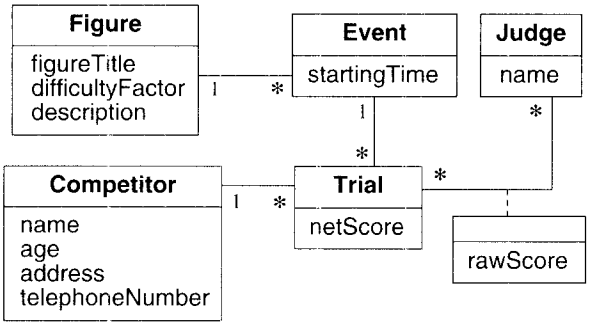

Figure E18.3 Partially completed class diagram for a scoring system

18.10 (6) Write C++ or Java code that corresponds to the pseudocode for Exercise 15.11. Make sure that your code respects encapsulation. [Instructor's note: You may want to give the students our answer to Exercise 15.11.]

18.11 (8) Write C++ or Java code for each of the OCL expressions in Section 3.5.3. Assume that there is a bidirectional implementation of associations. A specification, such as an OCL expression, need not respect encapsulation, but the code you write should.

18.12 (6) Using C++ or Java, implement all associations involving the classes *Box, Link, LineSegment,* or *Point* in Figure E18.4. Note that the editor allows links only between pairs of boxes.

18.13 (7) Implement the *cut* operation on the class *Box* in Figure E18.4 using C++ or Java. For simplicity, *cut* simply deletes the things that are cut and does not store them in a buffer. 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 (for C++ only). You may assume another method will update the display.

18.14 (7) Write a method using C++ or Java that will create a link between two boxes. Inputs to the method are two boxes and a list of points. The method should update associations and create object instances as needed. You may assume another method will update the display. Also write a method to destroy a link.

18.15 (8) Using C++ or Java, implement the following queries on Figure E18.4:

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.

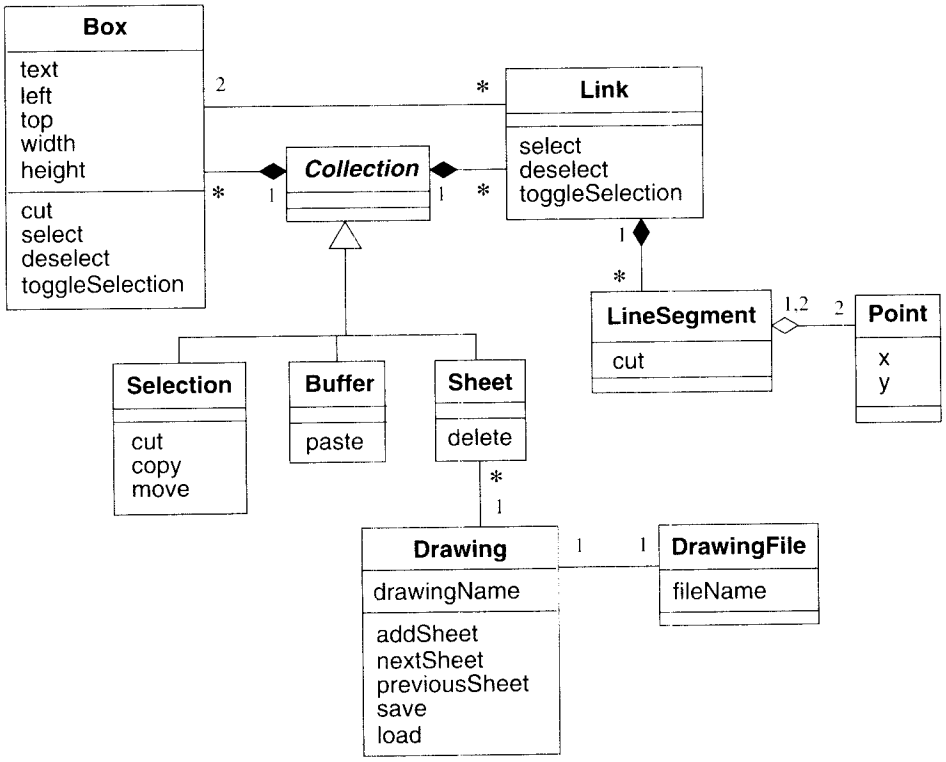

Figure E18.4 Class diagram for a diagram editor

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.

19

Databases

The OO paradigm is versatile and applies to databases as well as programming code. It might surprise you, but you can implement UML models not only with OO databases but also with relational databases. The resulting databases are efficient, coherent, and extensible.

You prepare a database by first performing the analysis steps described in Chapter 12 and constructing a domain model. The remaining methodology chapters in Part 2 apply mostly to design of programming code and to a lesser extent to databases. This chapter resumes where Chapter 12 ends. How can we map a model to database structures and tune the result for fast performance? How can we couple the resulting database to programming code? This chapter also includes a brief introduction to databases for new readers.

The chapter emphasizes relational databases because they dominate the marketplace. OO databases are practical only for niche applications and are discussed at the end of the chapter.

19.1 Introduction

19.1.1 Database Concepts

A ***database*** is a permanent, self-descriptive store of data that is contained in one or more files. Self-description is what sets a database apart from ordinary files. A database contains the data structure or ***schema***—description of data— as well as the data.

A ***database management system (DBMS)*** is the software for managing access to a database. One major objective of OO technology is to promote software reuse; for data-intensive applications DBMSs can replace much application code. You are achieving reuse when you use generic DBMS code, rather than custom-written application code. There are additional reasons for using a DBMS.