12

Domain Analysis

Domain analysis, the next stage of development, is concerned with devising a precise, concise, understandable, and correct model of the real world. Before building anything complex, the builder must understand the requirements. Requirements can be stated in words, but these are often imprecise and ambiguous. During analysis, we build models and begin to understand the requirements deeply.

To build a domain model, you must interview business experts, examine requirements statements, and scrutinize related artifacts. You must analyze the implications of the requirements and restate them rigorously. It is important to abstract important features first and defer small details until later. The successful analysis model states what must be done, without restricting how it is done, and avoids implementation decisions.

In this chapter you will learn how to take OO concepts and apply them to construct a domain model. The model serves several purposes: It clarifies the requirements, it provides a basis for agreement between the stakeholders and the developers, and it becomes the starting point for design and implementation.

12.1 Overview of Analysis

As Figure 12.1 shows, analysis begins with a problem statement generated during system conception. The statement may be incomplete or informal; analysis makes it more precise and exposes ambiguities and inconsistencies. The problem statement should not be taken as immutable but should serve as a basis for refining the real requirements.

Next, you must understand the real-world system described by the problem statement, and abstract its essential features into a model. Statements in natural language are often ambiguous, incomplete, and inconsistent. The analysis model is a precise, concise representation of the problem that permits answering questions and building a solution. Subsequent design steps refer to the analysis model, rather than the original vague problem statement.

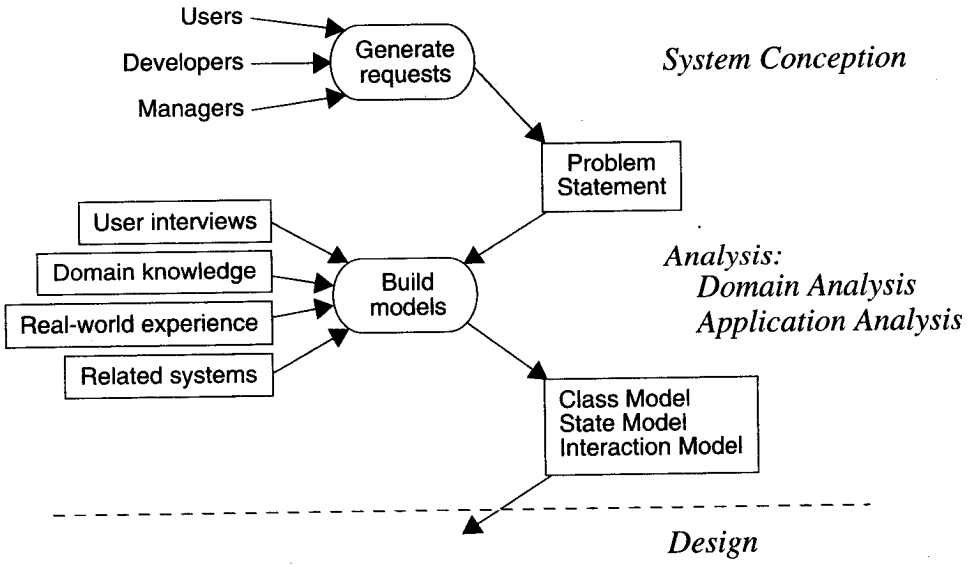

Figure 12.1 Overview of analysis. The problem statement should not be taken as immutable, but rather as a basis for refining the requirements.

Perhaps even more important, the process of constructing a rigorous model of the problem domain forces the developer to confront misunderstandings early in the development process while they are still easy to correct.

The analysis model addresses the three aspects of objects: static structure of objects (class model), interactions among objects (interaction model), and life-cycle histories of objects (state model). All three submodels are not equally important in every problem. Almost all problems have useful class models derived from real-world entities. Problems concerning reactive control and timing, such as user interfaces and process control, have important state models. Problems containing significant computation as well as systems that interact with other systems and different kinds of users have important interaction models.

Analysis is not a mechanical process. The exact representations involve judgment and in many regards are a matter of art. Most problem statements lack essential information, which must be obtained from the requestor or from the analyst's knowledge of the real-world problem domain. Also there is a choice in the level of abstraction for the model. The analyst must communicate with the requestor to clarify ambiguities and misconceptions. The analysis models enable precise communication.

We have divided analysis into two substages. The first, ***domain analysis***, is covered in this chapter and focuses on understanding the real-world essence of a problem. The second, ***application analysis***, is covered in the next chapter and builds on the domain model—incorporating major application artifacts that are seen by users and must be approved by them.

12.2 Domain Class Model

The first step in analyzing the requirements is to construct a domain model. The domain model shows the static structure of the real-world system and organizes it into workable pieces. The domain model describes real-world classes and their relationships to each other. During analysis, the class model precedes the state and interaction models because static structure tends to be better defined, less dependent on application details, and more stable as the solution evolves. Information for the domain model comes from the problem statement, artifacts from related systems, expert knowledge of the application domain, and general knowledge of the real world. Make sure you consider all information that is available and do not rely on a single source.

Find classes and associations first, as they provide the overall structure and approach to the problem. Next add attributes to describe the basic network of classes and associations. Then combine and organize classes using inheritance. Attempts to specify inheritance directly without first understanding classes and their attributes can distort the class structure to match preconceived notions. Operations are usually unimportant in a domain model. The main purpose of a domain model is to capture the information content of a domain.

It is best to get ideas down on paper before trying to organize them too much, even though they may be redundant and inconsistent, so as not to lose important details. An initial analysis model is likely to contain flaws that must be corrected by later iterations. The entire model need not be constructed uniformly. Some aspects of the problem can be analyzed in depth through several iterations while other aspects are still sketchy.

You must perform the following steps to construct a domain class model.

- Find classes. [12.2.1–12.2.2]
- Prepare a data dictionary. [12.2.3]
- Find associations. [12.2.4–12.2.5]
- Find attributes of objects and links. [12.2.6–12.2.7]
- Organize and simplify classes using inheritance. [12.2.8]
- Verify that access paths exist for likely queries. [12.2.9]
- Iterate and refine the model. [12.2.10]
- Reconsider the level of abstraction. [12.2.11]
- Group classes into packages. [12.2.12]

12.2.1 Finding Classes

The first step in constructing a class model is to find relevant classes for objects from the application domain. Objects include physical entities, such as houses, persons, and machines, as well as concepts, such as trajectories, seating assignments, and payment schedules. All classes must make sense in the application domain; avoid computer implementation constructs, such as linked lists and subroutines. Not all classes are explicit in the problem statement; some are implicit in the application domain or general knowledge.

As Figure 12.2 shows, begin by listing candidate classes found in the written description of the problem. Don't be too selective; write down every class that comes to mind. Classes often correspond to nouns. For example, in the statement "a reservation system to sell tickets to performances at various theaters" tentative classes would be *Reservation, System, Ticket, Performance,* and *Theater.* Don't operate blindly, however. The idea to is capture concepts; not all nouns are concepts, and concepts are also expressed in other parts of speech.

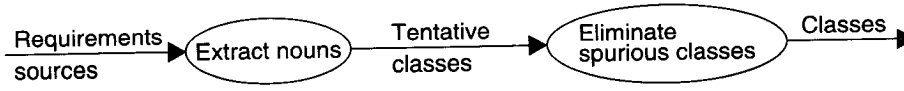

Figure 12.2 Finding classes. You can find many classes by considering nouns.

Don't worry much about inheritance or high-level classes; first get specific classes right so that you don't subconsciously suppress detail in an attempt to fit a preconceived structure. For example, if you are building a cataloging and checkout system for a library, identify different kinds of materials, such as books, magazines, newspapers, records, videos, and so on. You can organize them into broad categories later, by looking for similarities and differences.

ATM example. Examination of the concepts in the ATM problem statement from Chapter 11 yields the tentative classes shown in Figure 12.3. Figure 12.4 shows additional classes that do not appear directly in the statement but can be identified from our knowledge of the problem domain.

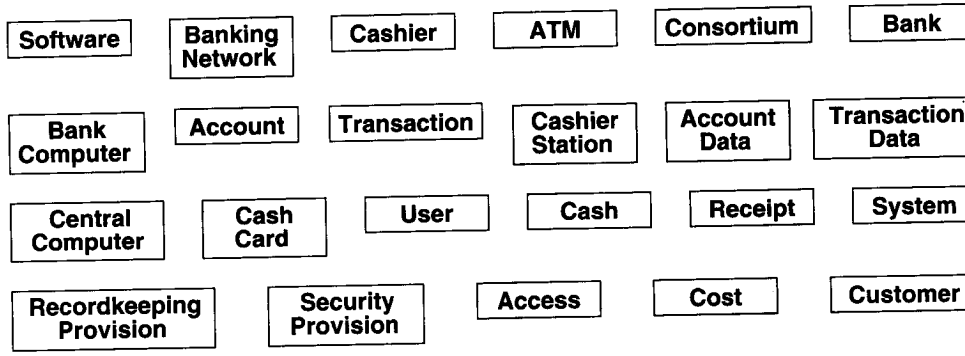

Figure 12.3 ATM classes extracted from problem statement nouns

Figure 12.4 ATM classes identified from knowledge of problem domain

12.2.2 Keeping the Right Classes

Now discard unnecessary and incorrect classes according to the following criteria. Figure 12.5 shows the classes eliminated from the ATM example.

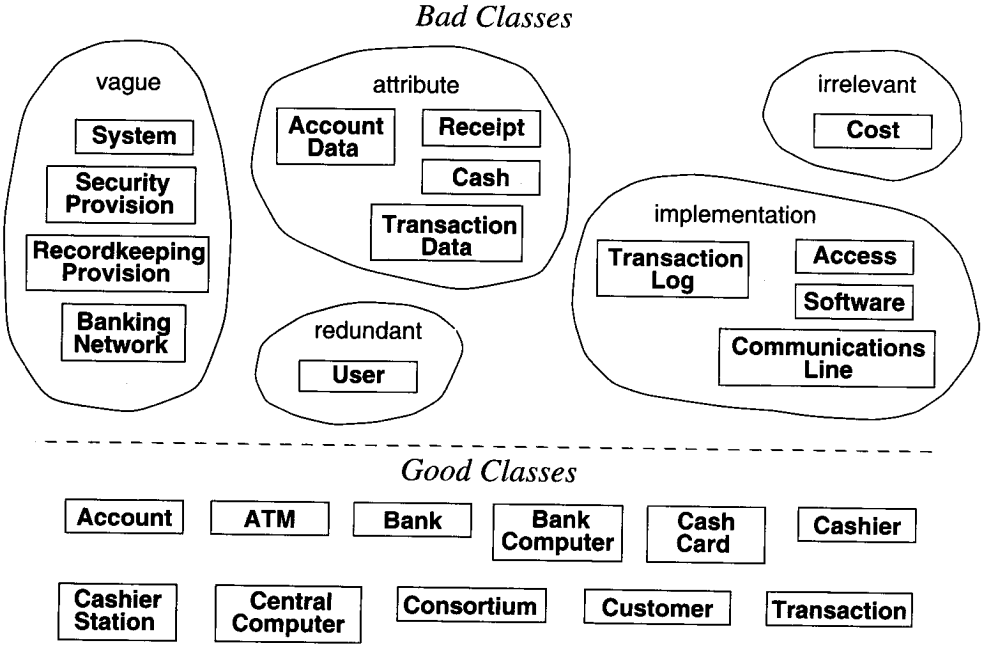

Figure 12.5 Eliminating unnecessary classes from ATM problem

- **Redundant classes.** If two classes express the same concept, you should keep the most descriptive name. For example, although *Customer* might describe a person taking an airline flight, *Passenger* is more descriptive. On the other hand, if the problem concerns contracts for a charter airline, *Customer* is also an appropriate word, since a contract might involve several passengers.

  ATM example. *Customer* and *User* are redundant; we retain *Customer* because it is more descriptive.

- **Irrelevant classes.** If a class has little or nothing to do with the problem, eliminate it. This involves judgment, because in another context the class could be important. For example, in a theater ticket reservation system, the occupations of the ticket holders are irrelevant, but the occupations of the theater personnel may be important.

  ATM example. Apportioning *Cost* is outside the scope of the ATM software.

- **Vague classes.** A class should be specific. Some tentative classes may have ill-defined boundaries or be too broad in scope.

ATM example. *RecordkeepingProvision* is vague and is handled by *Transaction.* In other applications, this might be included in other classes, such as *StockSales, TelephoneCalls,* or *MachineFailures.*

- **Attributes.** Names that primarily describe individual objects should be restated as attributes. For example, *name*, *birthdate*, and *weight* are usually attributes. If the independent existence of a property is important, then make it a class and not an attribute. For example, an employee's office would be a class in an application to reassign offices after a reorganization.

  ATM example. *AccountData* is underspecified but in any case probably describes an account. An ATM dispenses cash and receipts, but beyond that cash and receipts are peripheral to the problem, so they should be treated as attributes.

- **Operations.** If a name describes an operation that is applied to objects and not manipulated in its own right, then it is not a class. For example, a telephone call is a sequence of actions involving a caller and the telephone network. If we are simply building telephones, then *Call* is part of the state model and not a class.

  An operation that has features of its own should be modeled as a class, however. For example, in a billing system for telephone calls a *Call* would be an important class with attributes such as *date*, *time*, *origin*, and *destination.*

- **Roles.** The name of a class should reflect its intrinsic nature and not a role that it plays in an association. For example, *Owner* would be a poor name for a class in a car manufacturer's database. What if a list of drivers is added later? What about persons who lease cars? The proper class is *Person* (or possibly *Customer*), which assumes various different roles, such as *owner, driver,* and *lessee.*

  One physical entity sometimes corresponds to several classes. For example, *Person* and *Employee* may be distinct classes in some circumstances and redundant in others. From the viewpoint of a company database of employees, the two may be identical. In a government tax database, a person may hold more than one job, so it is important to distinguish *Person* from *Employee;* each person can correspond to zero or more instances of employee information.

- **Implementation constructs.** Eliminate constructs from the analysis model that are extraneous to the real world. You may need them later during design, but not now. For example, CPU, subroutine, process, algorithm, and interrupt are implementation constructs for most applications, although they are legitimate classes for an operating system. Data structures, such as linked lists, trees, arrays, and tables, are almost always implementation constructs.

  ATM example. Some tentative classes are really implementation constructs. *TransactionLog* is simply the set of transactions; its exact representation is a design issue. Communication links can be shown as associations; *CommunicationsLine* is simply the physical implementation of such a link.

- **Derived classes.** As a general rule, omit classes that can be derived from other classes. If a derived class is especially important, you can include it, but do so only sparingly. Mark all derived classes with a preceding slash ('/') in the class name.

12.2.3 Preparing a Data Dictionary

Isolated words have too many interpretations, so prepare a data dictionary for all modeling elements. Write a paragraph precisely describing each class. Describe the scope of the class within the current problem, including any assumptions or restrictions on its use. The data dictionary also describes associations, attributes, operations, and enumeration values. Figure 12.6 shows a data dictionary for the classes in the ATM problem.

12.2.4 Finding Associations

Next, find associations between classes. A structural relationship between two or more classes is an association. A reference from one class to another is an association. As we discussed in Chapter 3, attributes should not refer to classes; use an association instead. For example, class *Person* should not have an attribute *employer*; relate class *Person* and class *Company* with association *WorksFor*. Associations show relationships between classes at the same level of abstraction as the classes themselves, while object-valued attributes hide dependencies and obscure their two-way nature. Associations can be implemented in various ways, but such implementation decisions should be kept out of the analysis model to preserve design freedom.

Associations often correspond to stative verbs or verb phrases. These include physical location (*NextTo*, *PartOf*, *ContainedIn*), directed actions (*Drives*), communication (*TalksTo*), ownership (*Has*, *PartOf*), or satisfaction of some condition (*WorksFor*, *MarriedTo*, *Manages*). Extract all the candidates from the problem statement and get them down on paper first; don't try to refine things too early. Again, don't treat grammatical forms blindly; the idea is to capture relationships, however they are expressed in natural language.

ATM example. Figure 12.7 shows associations. The majority are taken directly from verb phrases in the problem statement. For some associations the verb phrase is implicit in the statement. Finally, some associations depend on real-world knowledge or assumptions. These must be verified with the requestor, as they are not in the problem statement.

12.2.5 Keeping the Right Associations

Now discard unnecessary and incorrect associations, using the following criteria.

- **Associations between eliminated classes**. If you have eliminated one of the classes in the association, you must eliminate the association or restate it in terms of other classes.

  ATM example. We can eliminate *Banking network includes cashier stations and ATMs, ATM dispenses cash, ATM prints receipts, Banks provide software, Cost apportioned to banks, System provides recordkeeping,* and *System provides security.*

- **Irrelevant or implementation associations**. Eliminate any associations that are outside the problem domain or deal with implementation constructs.

  ATM example. For example, *System handles concurrent access* is an implementation concept. Real-world objects are inherently concurrent; it is the implementation of the access algorithm that must be concurrent.

Account—a single account at a bank against which transactions can be applied. Accounts may be of various types, such as checking or savings. A customer can hold more than one account.

ATM—a station that allows customers to enter their own transactions using cash cards as identification. The ATM interacts with the customer to gather transaction information, sends the transaction information to the central computer for validation and processing, and dispenses cash to the user. We assume that an ATM need not operate independently of the network.

Bank—a financial institution that holds accounts for customers and issues cash cards authorizing access to accounts over the ATM network.

BankComputer—the computer owned by a bank that interfaces with the ATM network and the bank's own cashier stations. A bank may have its own internal computers to process accounts, but we are concerned only with the one that talks to the ATM network.

CashCard—a card assigned to a bank customer that authorizes access of accounts using an ATM machine. Each card contains a bank code and a card number. The bank code uniquely identifies the bank within the consortium. The card number determines the accounts that the card can access. A card does not necessarily access all of a customer's accounts. Each cash card is owned by a single customer, but multiple copies of it may exist, so the possibility of simultaneous use of the same card from different machines must be considered.

Cashier—an employee of a bank who is authorized to enter transactions into cashier stations and accept and dispense cash and checks to customers. Transactions, cash, and checks handled by each cashier must be logged and properly accounted for.

CashierStation—a station on which cashiers enter transactions for customers. Cashiers dispense and accept cash and checks; the station prints receipts. The cashier station communicates with the bank computer to validate and process the transactions.

CentralComputer—a computer operated by the consortium that dispatches transactions between the ATMs and the bank computers. The central computer validates bank codes but does not process transactions directly.

Consortium—an organization of banks that commissions and operates the ATM network. The network handles transactions only for banks in the consortium.

Customer—the holder of one or more accounts in a bank. A customer can consist of one or more persons or corporations; the correspondence is not relevant to this problem. The same person holding an account at a different bank is considered a different customer.

Transaction—a single integral request for operations on the accounts of a single customer. We specified only that ATMs must dispense cash, but we should not preclude the possibility of printing checks or accepting cash or checks. We may also want to provide the flexibility to operate on accounts of different customers, although it is not required yet.

Figure 12.6 Data dictionary for ATM classes. Prepare a data dictionary for all modeling elements.

Verb phrases

Banking network includes cashier stations and ATMs
Consortium shares ATMs
Bank provides bank computer
Bank computer maintains accounts
Bank computer processes transaction against account
Bank owns cashier station
Cashier station communicates with bank computer
Cashier enters transaction for account
ATMs communicate with central computer about transaction
Central computer clears transaction with bank
ATM accepts cash card
ATM interacts with user
ATM dispenses cash
ATM prints receipts
System handles concurrent access
Banks provide software
Cost apportioned to banks

Implicit verb phrases

Consortium consists of banks
Bank holds account
Consortium owns central computer
System provides recordkeeping
System provides security
Customers have cash cards

Knowledge of problem domain

Cash card accesses accounts
Bank employs cashiers

Figure 12.7 Associations from ATM problem statement

- **Actions.** An association should describe a structural property of the application domain, not a transient event. Sometimes, a requirement expressed as an action implies an underlying structural relationship and you should rephrase it accordingly.

  ATM example. *ATM accepts cash card* describes part of the interaction cycle between an ATM and a customer, not a permanent relationship between ATMs and cash cards. We can also eliminate *ATM interacts with user. Central computer clears transaction with bank* describes an action that implies the structural relationship *Central computer communicates with bank.*

- **Ternary associations.** You can decompose most associations among three or more classes into binary associations or phrase them as qualified associations. If a term in a ternary association is purely descriptive and has no identity of its own, then the term is an attribute on a binary association. Association *Company pays salary to person* can be rephrased as binary association *Company employs person* with a *salary* value for each *Company-Person* link.

  Occasionally, an application will require a general ternary association. *Professor teaches course in room* cannot be decomposed without losing information. We have not encountered associations with four or more classes in our work.

ATM example. *Bank computer processes transaction against account* can be broken into *Bank computer processes transaction* and *Transaction concerns account. Cashier enters transaction for account* can be broken similarly. *ATMs communicate with central computer about transaction* is really the binary associations *ATMs communicate with central computer* and *Transaction entered on ATM.*

- **Derived associations.** Omit associations that can be defined in terms of other associations, because they are redundant. For example, *GrandparentOf* can be defined in terms of a pair of *ParentOf* associations. Also omit associations defined by conditions on attributes. For example, *youngerThan* expresses a condition on the birth dates of two persons, not additional information.

  As much as possible, classes, attributes, and associations in the class model should represent independent information. Multiple paths between classes sometimes indicate derived associations that are compositions of primitive associations. *Consortium shares ATMs* is a composition of the associations *Consortium owns central computer* and *Central computer communicates with ATMs.*

  Be careful, because not all associations that form multiple paths between classes indicate redundancy. Sometimes the existence of an association can be derived from two or more primitive associations and the multiplicity can not. Keep the extra association if the additional multiplicity constraint is important. For example, in Figure 12.8 a company employs many persons and owns many computers. Each employee is assigned zero or more computers for the employee's personal use; some computers are for public use and are not assigned to anyone. The multiplicity of the *AssignedTo* association cannot be deduced from the *Employs* and *Owns* associations.

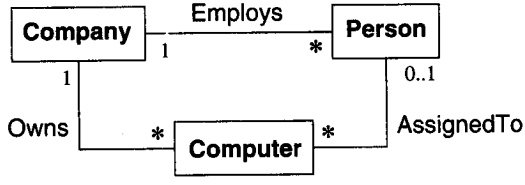

Figure 12.8 Nonredundant associations. Not all associations that form multiple paths between classes indicate redundancy.

Although derived associations do not add information, they are useful in the real world and in design. For example, kinship relationships such as *Uncle, MotherInLaw,* and *Cousin* have names because they describe common relationships considered important within our society. If they are especially important, you may show derived associations in class diagrams, but put a slash in front of their names to indicate their dependent status and to distinguish them from fundamental associations.

Further specify the semantics of associations as follows:

- **Misnamed associations.** Don't say how or why a situation came about, say what it is. Names are important to understanding and should be chosen with great care.

ATM example. *Bank computer maintains accounts* is a statement of action; rephrase as *Bank holds account.*

- **Association end names.** Add association end names where appropriate. For example, in the *WorksFor* association a *Company* is an *employer* with respect to a *Person* and a *Person* is an *employee* with respect to a *Company.* If there is only one association between a pair of classes and the meaning of the association is clear, you may omit association end names. For example, the meaning of *ATMs communicate with central computer* is clear from the class names. An association between two instances of the same class requires association end names to distinguish the instances. For example, the association *Person manages person* would have the end names *boss* and *worker.*

- **Qualified associations.** Usually a name identifies an object within some context; most names are not globally unique. The context combines with the name to uniquely identify the object. For example, the name of a company must be unique within the chartering state but may be duplicated in other states (there once was a Standard Oil Company in Ohio, Indiana, California, and New Jersey). The name of a company qualifies the association *State charters company*; *State* and *company name* uniquely identify *Company.* A qualifier distinguishes objects on the "many" side of an association.

  ATM example. The qualifier *bankCode* distinguishes the different banks in a consortium. Each cash card needs a bank code so that transactions can be directed to the appropriate bank.

- **Multiplicity.** Specify multiplicity, but don't put too much effort into getting it right, as multiplicity often changes during analysis. Challenge multiplicity values of "one." For example, the association *one Manager manages many employees* precludes matrix management or an employee with divided responsibilities. For multiplicity values of "many" consider whether a qualifier is needed; also ask if the objects need to be ordered in some way.

- **Missing associations.** Add any missing associations that are discovered.

  ATM example. We overlooked *Transaction entered on cashier station, Customers have accounts,* and *Transaction authorized by cash card.* If cashiers are restricted to specific stations, then the association *Cashier authorized on cashier station* would be needed.

- **Aggregation.** Aggregation is important for certain kinds of applications, especially for those involving mechanical parts and bills of material. For other applications aggregation is relatively minor and it can be unclear whether to use aggregation or ordinary association. For these other applications, don't spend much time trying to distinguish between association and aggregation. Aggregation is just an association with extra connotations. Use whichever seems more natural at the time and move on.

  ATM example. We decide that a *Bank* is a part of a *Consortium* and indicate the relationship with aggregation.

ATM example. Figure 12.9 shows a class diagram with the remaining associations. We have included only significant association names. Note that we have split *Transaction* into *Re-*

moteTransaction and *CashierTransaction* to accommodate different associations. The diagram shows multiplicity values. We could have made some analysis decisions differently. Don't worry; there are many possible correct models of a problem. We have shown the analysis process in small steps; with practice, you can elide several steps together in your mind.

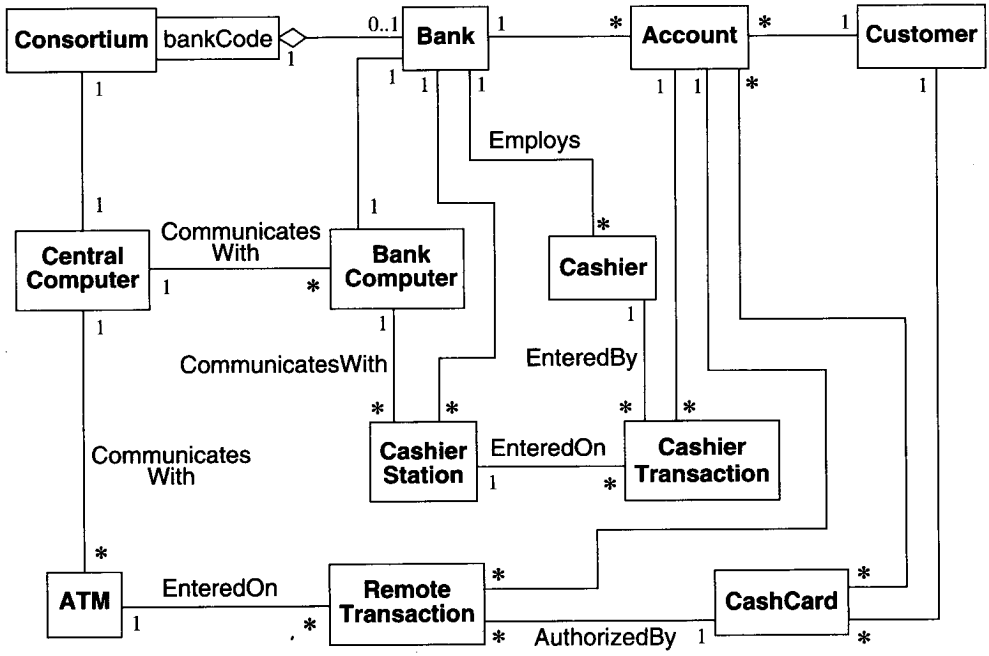

Figure 12.9 Initial class diagram for ATM system

12.2.6 Finding Attributes

Next find attributes. Attributes are data properties of individual objects, such as weight, velocity, or color. Attribute values should not be objects; use an association to show any relationship between two objects.

Attributes usually correspond to nouns followed by possessive phrases, such as "the color of the car" or "the position of the cursor." Adjectives often represent specific enumerated attribute values, such as *red, on,* or *expired.* Unlike classes and associations, attributes are less likely to be fully described in the problem statement. You must draw on your knowledge of the application domain and the real world to find them. You can also find attributes in the artifacts of related systems. Fortunately, attributes seldom affect the basic structure of the problem.

Do not carry discovery of attributes to excess. Only consider attributes directly relevant to the application. Get the most important attributes first; you can add fine details later. Dur-

ing analysis, avoid attributes that are solely for implementation. Be sure to give each attribute a meaningful name.

Normally, you should omit derived attributes. For example, *age* is derived from *birthdate* and *currentTime* (which is a property of the environment). Do not express derived attributes as operations, such as *getAge*, although you may eventually implement them that way.

Also look for attributes on associations. Such an attribute is a property of the link between two objects, rather than being a property of an individual object. For example, the many-to-many association between *Stockholder* and *Company* has an attribute of *numberOfShares*.

12.2.7 Keeping the Right Attributes

Eliminate unnecessary and incorrect attributes with the following criteria.

- **Objects**. If the independent existence of an element is important, rather than just its value, then it is an object. For example, *boss* refers to a class and *salary* is an attribute. The distinction often depends on the application. For example, in a mailing list *city* might be considered as an attribute, while in a census *City* would be a class with many attributes and relationships of its own. An element that has features of its own within the given application is a class.
- **Qualifiers**. If the value of an attribute depends on a particular context, then consider restating the attribute as a qualifier. For example, *employeeNumber* is not a unique property of a person with two jobs; it qualifies the association *Company employs person.*
- **Names**. Names are often better modeled as qualifiers rather than attributes. Test: Does the name select unique objects from a set? Can an object in the set have more than one name? If so, the name qualifies a qualified association. If a name appears to be unique in the world, you may have missed the class that is being qualified. For example, *departmentName* may be unique within a company, but eventually the program may need to deal with more than one company. It is better to use a qualified association immediately.

  A name is an attribute when its use does not depend on context, especially when it need not be unique within some set. Names of persons, unlike names of companies, may be duplicated and are therefore attributes.
- **Identifiers**. OO languages incorporate the notion of an object identifier for unambiguously referencing an object. Do not include an attribute whose only purpose is to identify an object, as object identifiers are implicit in class models. Only list attributes that exist in the application domain. For example, *accountCode* is a genuine attribute; *Banks* assign *accountCodes* and customers see them. In contrast, you should not list an internal *transactionID* as an attribute, although it may be convenient to generate one during implementation.
- **Attributes on associations**. If a value requires the presence of a link, then the property is an attribute of the association and not of a related class. Attributes are usually obvious on many-to-many associations; they cannot be attached to either class because of their

multiplicity. For example, in an association between *Person* and *Club* the attribute *membershipDate* belongs to the association, because a person can belong to many clubs and a club can have many members. Attributes are more subtle on one-to-many associations because they could be attached to the "many" class without losing information. Resist the urge to attach them to classes, as they would be invalid if multiplicity changed. Attributes are also subtle on one-to-one associations.

- **Internal values.** If an attribute describes the internal state of an object that is invisible outside the object, then eliminate it from the analysis.
- **Fine detail.** Omit minor attributes that are unlikely to affect most operations.
- **Discordant attributes.** An attribute that seems completely different from and unrelated to all other attributes may indicate a class that should be split into two distinct classes. A class should be simple and coherent. Mixing together distinct classes is one of the major causes of troublesome models. Unfocused classes frequently result from premature consideration of implementation decisions during analysis.
- **Boolean attributes.** Reconsider all boolean attributes. Often you can broaden a boolean attribute and restate it as an enumeration [Coad-95].

ATM example. We apply these criteria to obtain attributes for each class (Figure 12.10). Some tentative attributes are actually qualifiers on associations. We consider several aspects of the model.

- *BankCode* and *cardCode* are present on the card. Their format is an implementation detail, but we must add a new association *Bank issues CashCard. CardCode* is a qualifier on this association; *bankCode* is the qualifier of *Bank* with respect to *Consortium.*
- The computers do not have state relevant to this problem. Whether the machine is up or down is a transient attribute that is part of implementation.
- Avoid the temptation to omit *Consortium*, even though it is currently unique. It provides the context for the *bankCode* qualifier and may be useful for future expansion.

Keep in mind that the ATM problem is just an example. Real applications, when fleshed out, tend to have many more attributes per class than Figure 12.10 shows.

12.2.8 Refining with Inheritance

The next step is to organize classes by using inheritance to share common structure. Inheritance can be added in two directions: by generalizing common aspects of existing classes into a superclass (bottom up) or by specializing existing classes into multiple subclasses (top down).

- **Bottom-up generalization.** You can discover inheritance from the bottom up by searching for classes with similar attributes, associations, and operations. For each generalization, define a superclass to share common features. You may have to slightly redefine some attributes or classes to fit in. This is acceptable, but don't push too hard if it doesn't fit; you may have the wrong generalization. Some generalizations will suggest themselves based on an existing taxonomy in the real world; use existing concepts whenever possible. Symmetry will suggest missing classes.

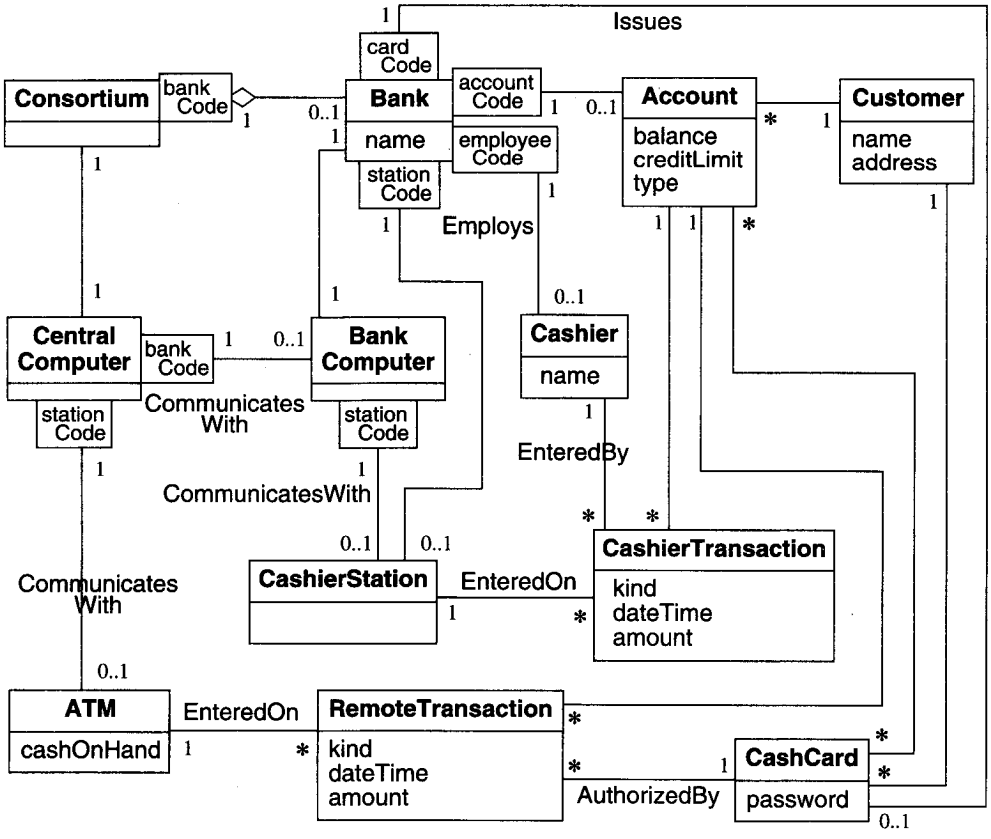

Figure 12.10 ATM class model with attributes

ATM example. *RemoteTransaction* and *CashierTransaction* are similar, except in their initiation, and can be generalized by *Transaction*. On the other hand, *CentralComputer* and *BankComputer* have little in common for purposes of the ATM example.

- **Top-down specialization.** Top-down specializations are often apparent from the application domain. Look for noun phrases composed of various adjectives on the class name: *fluorescent* lamp, *incandescent* lamp; *fixed* menu, *pop-up* menu, *sliding* menu. Avoid excessive refinement. If proposed specializations are incompatible with an existing class, the existing class may be improperly formulated.
- **Generalization vs. enumeration.** Enumerated subcases in the application domain are the most frequent source of specializations. Often, it is sufficient to note that a set of enumerated subcases exists, without actually listing them. For example, an ATM account could be refined into *CheckingAccount* and *SavingsAccount*. While undoubtedly useful in some banking applications, this distinction does not affect behavior within the ATM application; *type* can be made a simple attribute of *Account*.

- **Multiple inheritance**. You can use multiple inheritance to increase sharing, but only if necessary, because it increases both conceptual and implementation complexity.
- **Similar associations**. When the same association name appears more than once with substantially the same meaning, try to generalize the associated classes. Sometimes the classes have nothing in common but the association, but more often you will uncover an underlying generality that you have overlooked.

  ATM example. *Transaction* is entered on both *CashierStation* and *ATM; EntryStation* generalizes *CashierStation* and *ATM*.
- **Adjusting the inheritance level**. You must assign attributes and associations to specific classes in the class hierarchy. Assign each one to the most general class for which it is appropriate. You may need some adjustment to get everything right. Symmetry may suggest additional attributes to distinguish among subclasses more clearly.

Figure 12.11 shows the ATM class model after adding inheritance.

12.2.9 Testing Access Paths

Trace access paths through the class model to see if they yield sensible results. Where a unique value is expected, is there a path yielding a unique result? For multiplicity "many" is there a way to pick out unique values when needed? Think of questions you might like to ask. Are there useful questions that cannot be answered? They indicate missing information. If something that seems simple in the real world appears complex in the model, you may have missed something (but make sure that the complexity is not inherent in the real world).

It can be acceptable to have classes that are "disconnected" from other classes. This usually occurs when the relationship between a disconnected class and the remainder of the model is diffuse. However, check disconnected classes to make sure you have not overlooked any associations.

ATM example. A cash card itself does not uniquely identify an account, so the user must choose an account somehow. If the user supplies an account type (savings or checking), each card can access at most one savings and one checking account. This is probably reasonable, and many cash cards actually work this way, but it limits the system. The alternative is to require customers to remember account numbers. If a cash card accesses a single account, then transfers between accounts are impossible.

We have assumed that the ATM network serves a single consortium of banks. Real cash machines today often serve overlapping networks of banks and accept credit cards as well as cash cards. The model would have to be extended to handle that situation. We will assume that the customer is satisfied with this limitation on the system.

12.2.10 Iterating a Class Model

A class model is rarely correct after a single pass. The entire software development process is one of continual iteration; different parts of a model are often at different stages of completion. If you find a deficiency, go back to an earlier stage if necessary to correct it. Some refinements can come only after completing the state and interaction models.

There are several signs of missing classes.

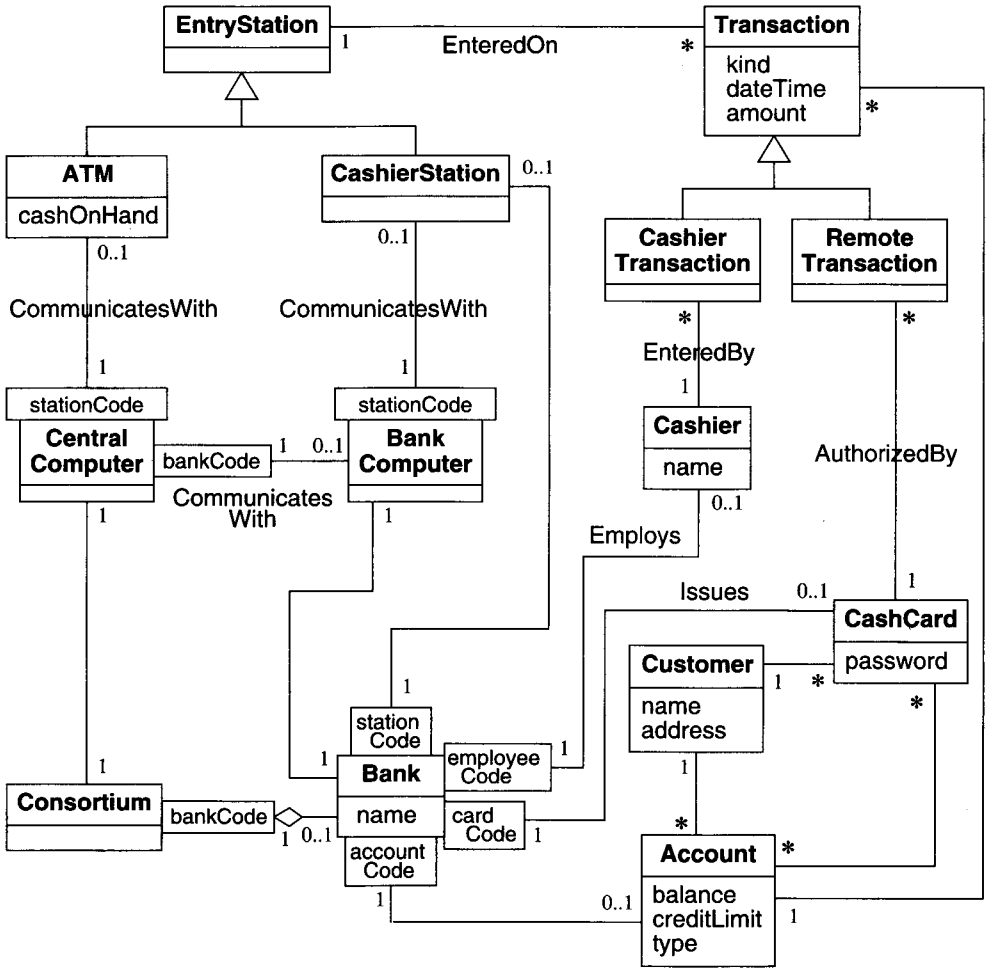

Figure 12.11 ATM class model with attributes and inheritance

- **Asymmetries in associations and generalizations**. Add new classes by analogy.
- **Disparate attributes and operations on a class**. Split a class so that each part is coherent.
- **Difficulty in generalizing cleanly**. One class may be playing two roles. Split it up and one part may then fit in cleanly.
- **Duplicate associations with the same name and purpose**. Generalize to create the missing superclass that unites them.
- **A role that substantially shapes the semantics of a class**. Maybe it should be a separate class. This often means converting an association into a class. For example, a person

can be employed by several companies with different conditions of employment at each; *Employee* is then a class denoting a person working for a particular company, in addition to class *Person* and *Company.*

Also look out for missing associations.

- **Missing access paths for operations**. Add new associations so that you can answer queries.

Another concern is superfluous model elements.

- **Lack of attributes, operations, and associations on a class**. Why is the class needed? Avoid inventing subclasses merely to indicate an enumeration. If proposed subclasses are otherwise identical, mark the distinction using an attribute.
- **Redundant information**. Remove associations that do not add new information or mark them as derived.

And finally you may adjust the placement of attributes and associations.

- **Association end names that are too broad or too narrow for their classes**. Move the association up or down in the class hierarchy.
- **Need to access an object by one of its attribute values**. Consider a qualified association.

In practice, model building is not as rigidly ordered as we have shown. You can combine several steps, once you are experienced. For example, you can find candidate classes, reject the incorrect ones without writing them down, and add them to the class diagram together with their associations. You can take some parts of the model through several steps and develop them in some detail, while other parts are still sketchy. You can interchange the order of steps when appropriate. If you are just learning class modeling, however, we recommend that you follow the steps in full detail the first few times.

ATM example. *CashCard* really has a split personality—it is both an authorization unit within the bank allowing access to the customer's accounts and also a piece of plastic data that the ATM reads to obtain coded IDs. In this case, the codes are actually part of the real world, not just computer artifacts; the codes, not the cash card, are communicated to the central computer. We should split cash card into two classes: *CardAuthorization*, an access right to one or more customer accounts; and *CashCard*, a piece of plastic that contains a bank code and a cash card number meaningful to the bank. Each card authorization may have several cash cards, each containing a serial number for security reasons. The card code, present on the physical card, identifies the card authorization within the bank. Each card authorization identifies one or more accounts—for example, one checking account and one savings account.

Transaction is not general enough to permit transfers between accounts because it concerns only a single account. In general, a *Transaction* consists of one or more *updates* on individual accounts. An *update* is a single action (withdrawal, deposit, or query) on a single account. All updates in a single transaction must be processed together as an atomic unit; if any one fails, then they all are canceled.

The distinction between *Bank* and *BankComputer* and between *Consortium* and *CentralComputer* doesn't seem to affect the analysis. The fact that communications are pro-

cessed by computers is actually an implementation artifact. Merge *BankComputer* into *Bank* and *CentralComputer* into *Consortium*.

Customer doesn't seem to enter into the analysis so far. However, when we consider operations to open new accounts, it may be an important concept, so leave it alone for now.

Figure 12.12 shows a revised class diagram that is simpler and cleaner.

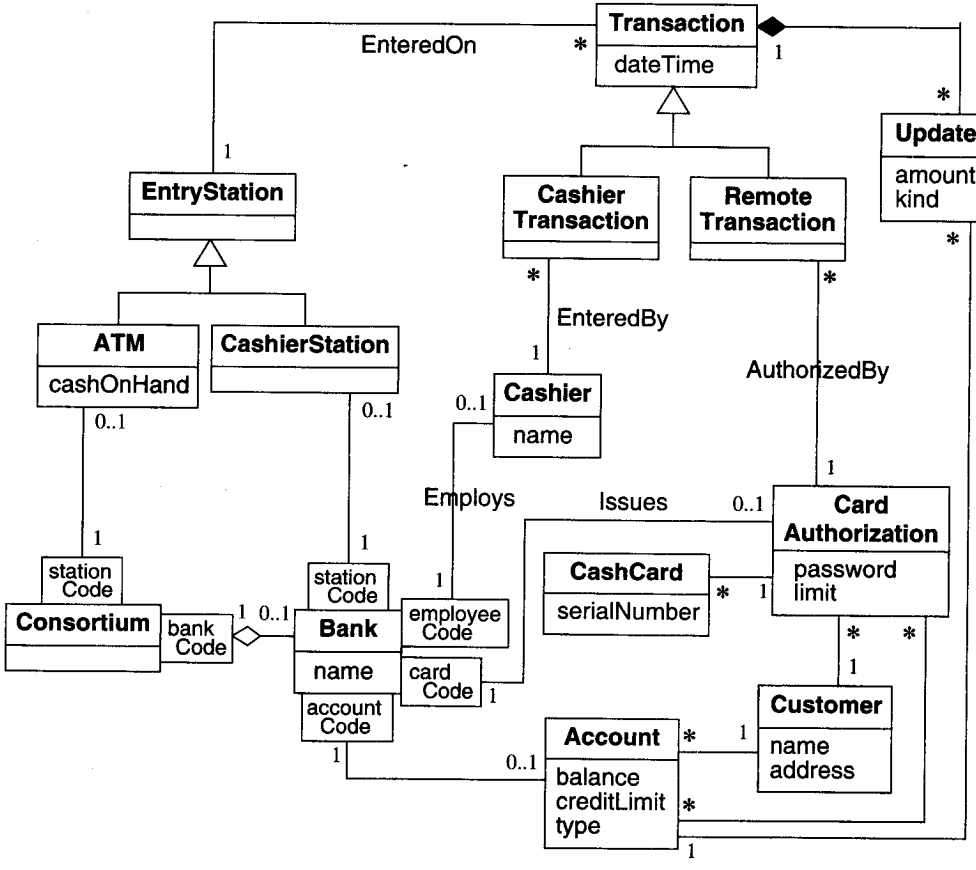

Figure 12.12 ATM class model after further revision

12.2.11 *Shifting the Level of Abstraction*

So far in analysis, we have taken the problem statement quite literally. We have regarded nouns and verbs in the problem description as direct analogs of classes and associations. This is a good way to begin analysis, but it does not always suffice. Sometimes you must raise the level of abstraction to solve a problem. You should be doing this throughout as you build a model, but we put in an explicit step to make sure you do not overlook abstraction.

For example, we encountered one application in which the developers had separate classes for *IndividualContributor*, *Supervisor*, and *Manager*. *IndividualContributors* report to *Supervisors* and *Supervisors* report to *Managers*. This model certainly is correct, but it suffers from some problems. There is much commonality between the three classes—the only difference is the reporting hierarchy. For example, they all have phone numbers and addresses. We could handle the commonality with a superclass, but that only makes the model larger. An additional problem arose when we talked to the developers and they said they wanted to add another class for the persons to whom managers reported.

Figure 12.13 shows the original model and an improved model that is more abstract. Instead of "hard coding" the management hierarchy in the model, we can "soft code" it with an association between boss and worker. A person who has an *employeeType* of "individualContributor" is a worker who reports to another person with an *employeeType* of "supervisor." Similarly, a person who is a supervisor reports to a person who is a manager. In the improved model a worker has an optional boss, because the reporting hierarchy eventually stops. The improved model is smaller and more flexible. An additional reporting level does not change the model's structure; it merely alters the data that is stored.

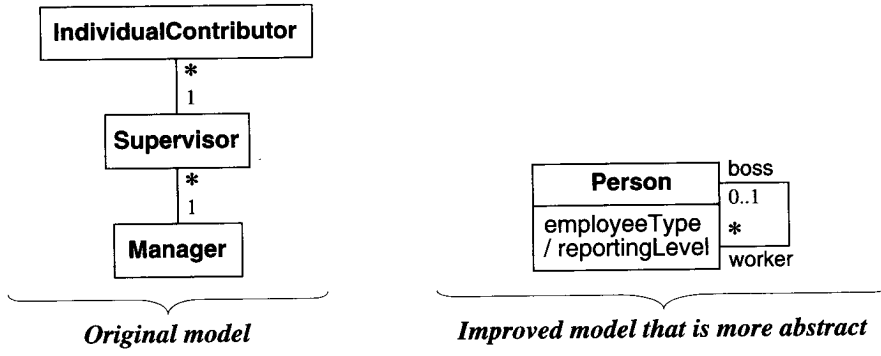

Figure 12.13 Shifting the level of abstraction. Abstraction makes a model more complex but can increase flexibility and reduce the number of classes.

One way that you can take advantage of abstraction is by thinking in terms of patterns. Different kinds of patterns apply to the different development stages, but here we are interested in patterns for analysis. A ***pattern*** distills the knowledge of experts and provides a proven solution to a general problem. For example, the right side of Figure 12.13 is a pattern for modeling a management hierarchy. Whenever we encounter the need for a management hierarchy, we immediately think in terms of the pattern and place it in our application model. The use of tried and tested patterns gives us the confidence of a sound approach and boosts our productivity in building models.

ATM example. We have already included some abstractions in the ATM model. We distinguished between a *CashCard* and a *CardAuthorization*. Furthermore, we included the notion of transactions rather than trying to list each possible kind of interaction.

12.2.12 Grouping Classes into Packages

The last step of class modeling is to group classes into packages. A ***package*** is a group of elements (classes, associations, generalizations, and lesser packages) with a common theme. Packages organize a model for convenience in drawing, printing, and viewing. Furthermore, when you place classes and associations in a package, you are making a semantic statement. Generally speaking, classes in the same package are more closely related than classes in different packages.

Normally you should restrict each association to a single package, but you can repeat some classes in different packages. To assign classes to packages, look for cut points— a cut point is a class that is the sole connection between two otherwise disconnected parts of a model. Such a class forms the bridge between two packages. For example, in a file management system, a *File* is the cut point between the directory structure and the file contents. Try to choose packages to reduce the number of crossovers in the class diagrams. With a little care, you can draw most class diagrams as planar graphs, without crossing lines.

Reuse a package from a previous design if possible, but avoid forcing a fit. Reuse is easiest when part of the problem domain matches a previous problem. If the new problem is similar to a previous problem but different, you may have to extend the original model to encompass both problems. Use your judgment about whether this is better than building a new model.

ATM example. The current model is small and would not require breakdown into packages, but it could serve as a core for a more detailed model. The packages might be:

- tellers—cashier, entry station, cashier station, ATM
- accounts—account, cash card, card authorization, customer, transaction, update, cashier transaction, remote transaction
- banks—consortium, bank

Each package could add details. The account package could contain varieties of transactions, information about customers, interest payments, and fees. The bank package could contain information about branches, addresses, and cost allocations.

12.3 Domain State Model

Some domain objects pass through qualitatively distinct states during their lifetime. There may be different constraints on attribute values, different associations or multiplicities in the various states, different operations that may be invoked, different behavior of the operations, and so on. It is often useful to construct a state diagram of such a domain class. The state diagram describes the various states the object can assume, the properties and constraints of the object in various states, and the events that take an object from one state to another.

Most domain classes do not require state diagrams and can be adequately described by a list of operations. For the minority of classes that do exhibit distinct states, however, a state model can help in understanding their behavior.

First identify the domain classes with significant states and note the states of each class. Then determine the events that take an object from one state to another. Given the states and the events, you can build state diagrams for the affected objects. Finally, evaluate the state diagrams to make sure they are complete and correct.

The following steps are performed in constructing a domain state model.

- Identify domain classes with states. [12.3.1]
- Find states. [12.3.2]
- Find events. [12.3.3]
- Build state diagrams. [12.3.4]
- Evaluate state diagrams. [12.3.5]

12.3.1 Identifying Classes with States

Examine the list of domain classes for those that have a distinct life cycle. Look for classes that can be characterized by a progressive history or that exhibit cyclic behavior. Identify the significant states in the life cycle of an object. For example, a scientific paper for a journal goes from *Being written* to *Under consideration* to *Accepted* or *Rejected*. There can be some cycles, for example, if the reviewers ask for revisions, but basically the life of this object is progressive. On the other hand, an airplane owned by an airline cycles through the states of *Maintenance, Loading, Flying,* and *Unloading*. Not every state occurs in every cycle, and there are probably other states, but the life of this object is cyclic. There are also classes whose life cycle is chaotic, but most classes with states are either progressive or cyclic.

ATM example. *Account* is an important business concept, and the appropriate behavior for an ATM depends on the state of an *Account*. The life cycle for *Account* is a mix of progressive and cycling to and from problem states. No other ATM classes have a significant domain state model.

12.3.2 Finding States

List the states for each class. Characterize the objects in each class—the attribute values that an object may have, the associations that it may participate in and their multiplicities, attributes and associations that are meaningful only in certain states, and so on. Give each state a meaningful name. Avoid names that indicate how the state came about; try to directly describe the state.

Don't focus on fine distinctions among states, particularly quantitative differences, such as small, medium, or large. States should be based on qualitative differences in behavior, attributes, or associations.

It is unnecessary to determine all the states before examining events. By looking at events and considering transitions among states, missing states will become clear.

ATM example. Here are some states for an *Account*: *Normal* (ready for normal access), *Closed* (closed by the customer but still on file in the bank records), *Overdrawn* (customer withdrawals exceed the balance in the account), and *Suspended* (access to the account is blocked for some reason).

12.3.3 Finding Events

Once you have a preliminary set of states, find the events that cause transitions among states. Think about the stimuli that cause a state to change. In many cases, you can regard an event as completing a do-activity. For example, if a technical paper is in the state *Under consideration*, then the state terminates when a decision on the paper is reached. In this case, the decision can be positive (*Accept paper*) or negative (*Reject paper*). In cases of completing a do-activity, other possibilities are often possible and may be added in the future—for example, *Conditionally accept with revisions*.

You can find other events by thinking about taking the object into a specific state. For example, if you lift the receiver on a telephone, it enters the *Dialing* state. Many telephones have pushbuttons that invoke specific functions. If you press the *redial* button, the phone transmits the number and enters the *Calling* state. If you press the *program* button, it enters the *Programming* state.

There are additional events that occur within a state and do not cause a transition. For the domain state model you should focus on events that cause transitions among states. When you discover an event, capture any information that it conveys as a list of parameters.

ATM example. Important events include: *close account*, *withdraw excess funds*, *repeated incorrect PIN*, *suspected fraud*, and *administrative action*.

12.3.4 Building State Diagrams

Note the states to which each event applies. Add transitions to show the change in state caused by the occurrence of an event when an object is in a particular state. If an event terminates a state, it will usually have a single transition from that state to another state. If an event initiates a target state, then consider where it can occur, and add transitions from those states to the target state. Consider the possibility of using a transition on an enclosing state rather than adding a transition from each substate to the target state. If an event has different effects in different states, add a transition for each state.

Once you have specified the transitions, consider the meaning of an event in states for which there is no transition on the event. Is it ignored? Then everything is fine. Does it represent an error? Then add a transition to an error state. Does it have some effect that you forgot? Then add another transition. Sometimes you will discover new states.

It is usually not important to consider effects when building a state diagram for a domain class. If the objects in the class perform activities on transitions, however, add them to the state diagram.

ATM example. Figure 12.14 shows the domain state model for the *Account* class.

12.3.5 Evaluating State Diagrams

Examine each state model. Are all the states connected? Pay particular attention to paths through it. If it represents a progressive class, is there a path from the initial state to the final state? Are the expected variations present? If it represents a cyclic class, is the main loop present? Are there any dead states that terminate the cycle?

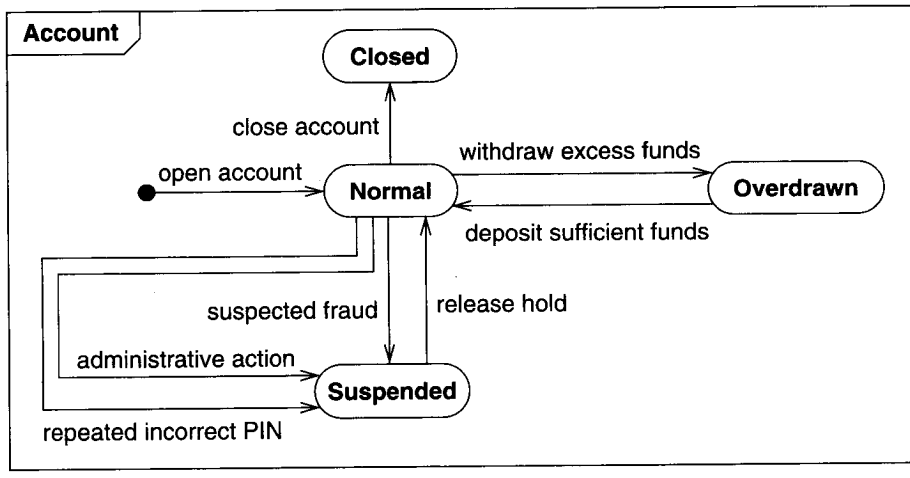

Figure 12.14 Domain state model. The domain state model documents important classes that change state in the real world.

Use your knowledge of the domain to look for missing paths. Sometimes missing paths indicate missing states. When a state model is complete, it should accurately represent the life cycle of the class.

ATM example. Our state model for *Account* is simplistic but we are satisfied with it. We would require substantial banking knowledge to construct a deeper model.

12.4 Domain Interaction Model

The interaction model is seldom important for domain analysis. During domain analysis the emphasis is on key concepts and deep structural relationships and not the users' view of them. The interaction model, however, is an important aspect of application modeling and we will cover it in the next chapter.

12.5 Iterating the Analysis

Most analysis models require more than one pass to complete. Problem statements often contain circularities, and most applications cannot be approached in a completely linear way, because different parts of the problem interact. To understand a problem with all its implications, you must attack the analysis iteratively, preparing a first approximation to the model and then iterating the analysis as your understanding increases. There is no firm line between analysis and design, so don't overdo it. Verify the final analysis with the requestor and application domain experts.

12.5.1 Refining the Analysis Model

The overall analysis model may show inconsistencies and imbalances within and across models. Iterate the different portions to produce a cleaner, more coherent model. Try to refine classes to increase sharing and improve structure. Add details that you glossed over during the first pass.

Some constructs will feel awkward and won't seem to fit in right. Reexamine them carefully; you may have the wrong concepts. Sometimes major restructuring in the model is needed as your understanding increases. It is easier to do now than it will ever be, so don't avoid changes just because you already have a model in place. When there are many constructs that appear similar but don't quite fit together, you have probably missed or miscast a more general concept. Watch out for generalizations factored on the wrong aspects.

A common difficulty is a physical object that has two logically distinct aspects. Each aspect should be modeled with a distinct object. An indication of this problem is a class that doesn't fit in cleanly and seems to have two sets of unrelated attributes, associations, and operations.

Other indications to watch for include exceptions, many special cases, and lack of expected symmetry. Consider restructuring your model to capture constraints better within its structure.

Be wary of codifying arbitrary business practices in your model. Software should facilitate operation of the business and not inhibit reasonable changes. Often you can introduce abstractions that increase business flexibility without substantially complicating a model.

Remove classes or associations that seemed useful at first but now appear extraneous. Often two classes in the analysis can be combined, because the distinction between them doesn't affect the rest of the model in any meaningful way. There is a tendency for models to grow as analysis proceeds. This is a concern, since the amount of development work escalates as a model becomes larger in size. Take a close look at your model for minor concepts to cut or abstractions that can simplify the model.

A good model feels right and does not appear to have extraneous detail. Don't worry if it doesn't seem perfect; even a good model will often have a few small areas where the design is adequate but never feels quite right.

12.5.2 Restating the Requirements

When the analysis is complete, the model serves as the basis for the requirements and defines the scope of future discourse. Most of the real requirements will be part of the model. In addition you may have some performance constraints; these should be stated clearly, together with optimization criteria. Other requirements specify the method of solution and should be separated and challenged, if possible.

You should verify the final model with the requestor. During analysis some requirements may appear to be incorrect or impractical; confirm corrections to the requirements. Also business experts should verify the analysis model to make sure that it correctly models the real world. We have found analysis models to be an effective means of communication with business experts who are not computer experts.

The final verified analysis model serves as the basis for system architecture, design, and implementation. You should revise the original problem statement to incorporate corrections and understanding discovered during analysis.

12.5.3 Analysis and Design

The goal of analysis is to specify the problem fully without introducing a bias to any particular implementation, but it is impossible in practice to avoid all taints of implementation. There is no absolute line between the various development stages, nor is there any such thing as a perfect analysis. Don't treat the rules we have given too rigidly. The purpose of the rules is to preserve flexibility and permit changes later, but remember that the goal of modeling is to accomplish the total job, and flexibility is just a means to an end.

ATM example. We have no further changes to the ATM model at this time. A true application is more likely to incur revision than a textbook example, because you have reviewers who are passionate about the application and have a vested interest in it.

12.6 Chapter Summary

The domain model captures general knowledge about an application—concepts and relationships known to experts in the domain. The domain model has class models and sometimes state models, but seldom has an interaction model. The purpose of analysis is to understand the problem and the application so that a correct design can be constructed. A good analysis captures the essential features of the problem without introducing implementation artifacts that prematurely restrict design decisions.

The domain class model shows the static structure of the real world. First find classes. Then find associations between classes. Note attributes, though you can defer minor ones. You can use generalization to organize and simplify the class structure. Group tightly coupled classes and associations into packages. Supplement the class models with a data dictionary—brief textual descriptions, including the purpose and scope of each element.

If a domain class has several qualitatively different states during its life cycle, make a state diagram for it, but most domain classes will not require state diagrams.

Methodologies are never as linear as they appear in books. This one is no exception. Any complex analysis is constructed by iteration on multiple levels. You need not prepare all parts of the model at the same pace. The result of analysis replaces the original problem statement and serves as the basis for design.

Bibliographic Notes

Abbott explains how to use nouns and verbs in the problem statement to seed thinking about an application [Abbott-83]. [Coad-95] is a good book with some examples of analysis patterns.