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.

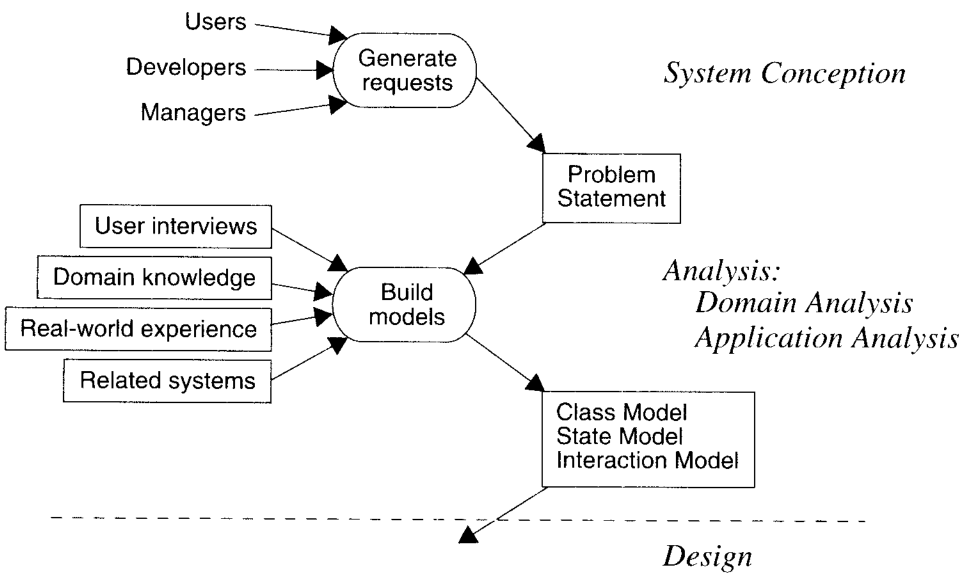


Figure 12.1 Overview of analysis. The problem statement should not be taken as immutable, but rather as a basis for refining the requirements.

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.

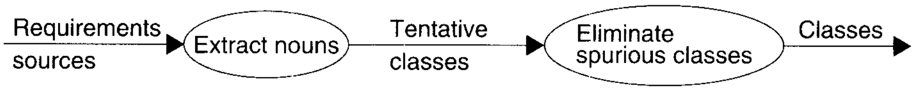


Figure 12.2 Finding classes. You can find many classes by considering nouns.

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.

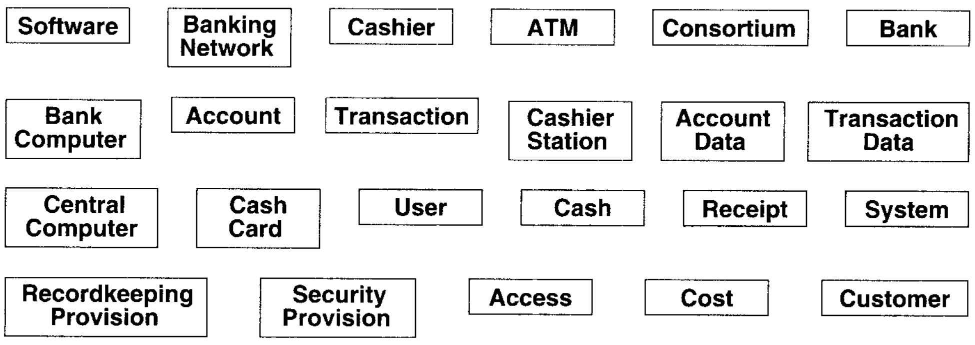


Figure 12.3 ATM classes extracted from problem statement nouns

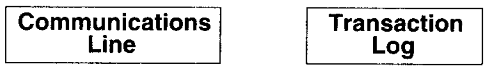


Figure 12.4 ATM classes identified from knowledge of problem domain

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.

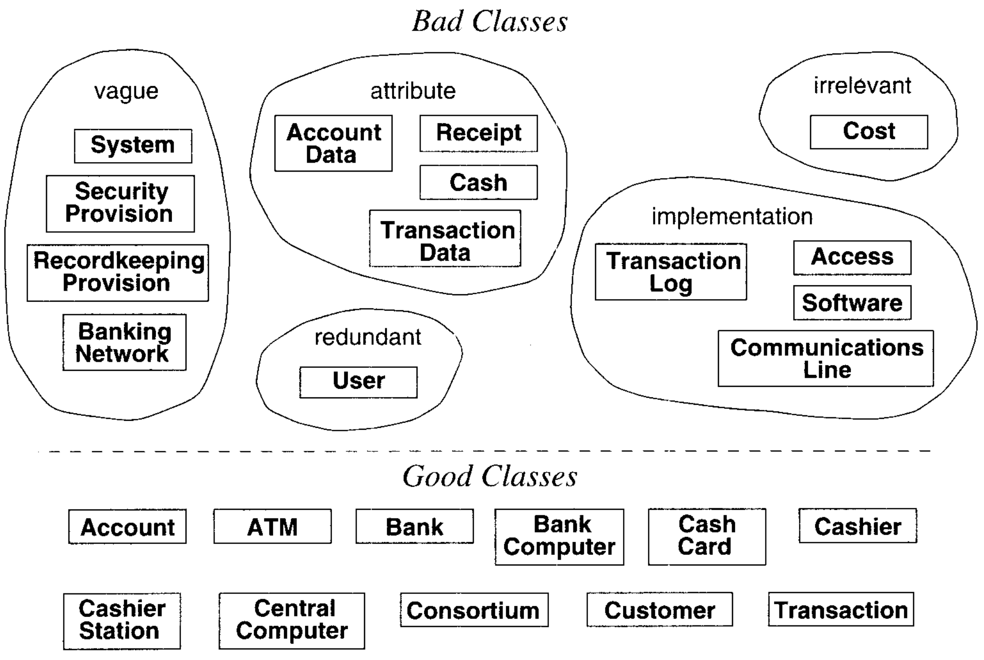


Figure 12.5 Eliminating unnecessary classes from ATM problem

- **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.

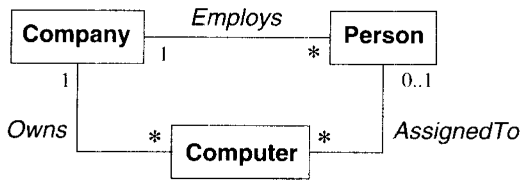


Figure 12.8 Nonredundant associations. Not all associations that form multiple paths between classes indicate redundancy.

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.

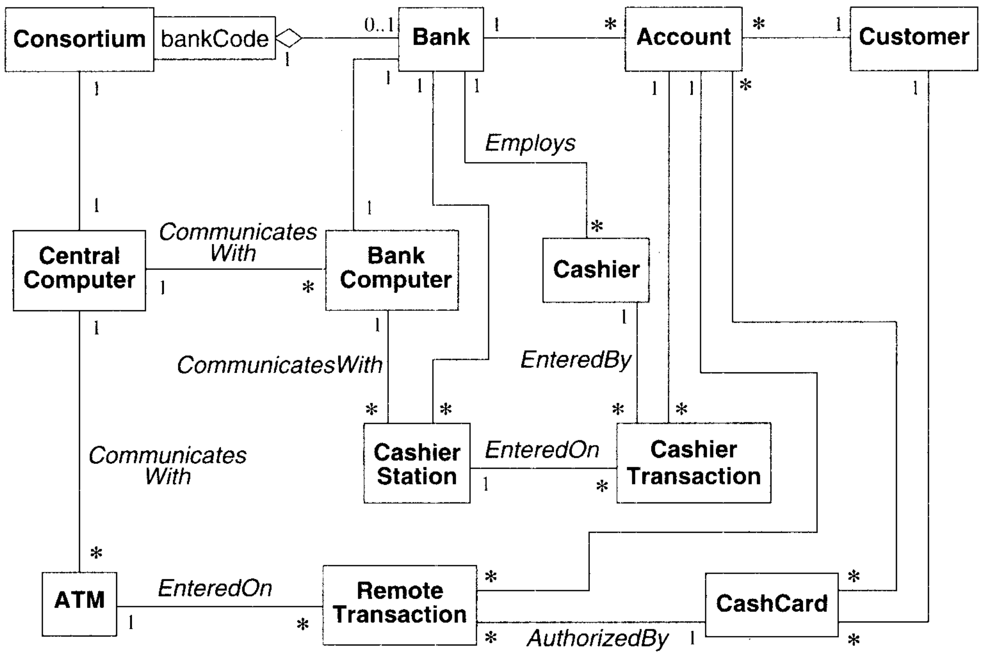


Figure 12.9 Initial class diagram for ATM system

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. During 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.

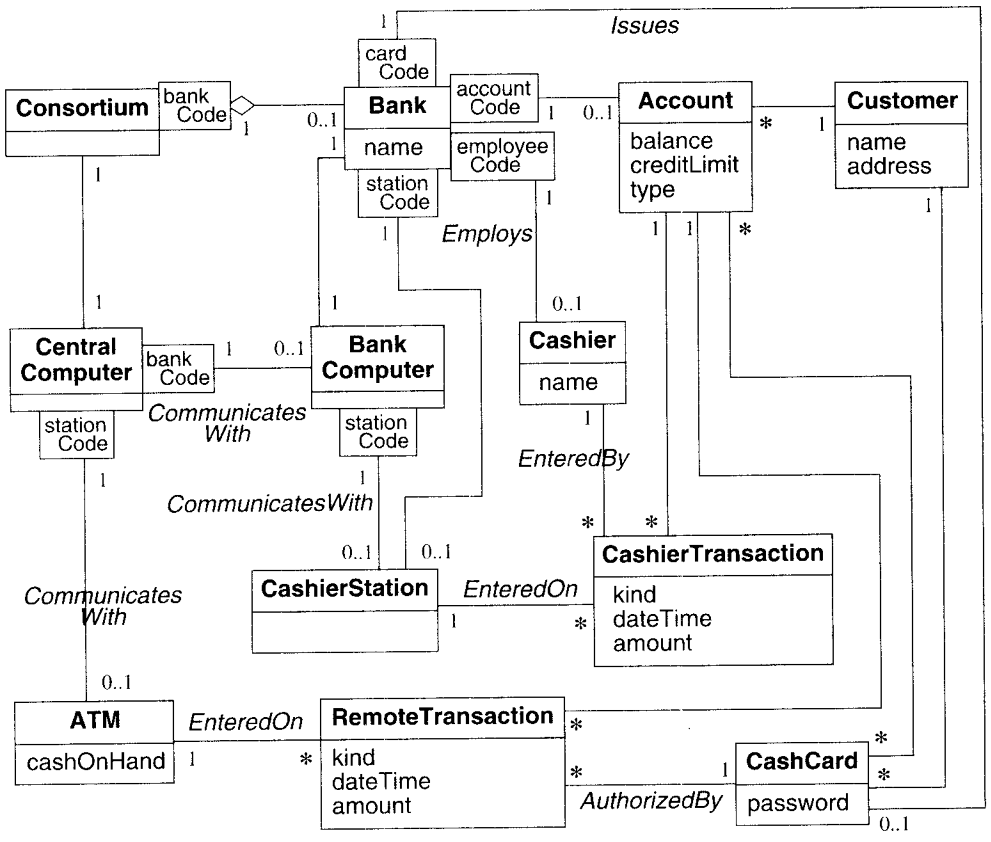


Figure 12.10 ATM class model with attributes

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.

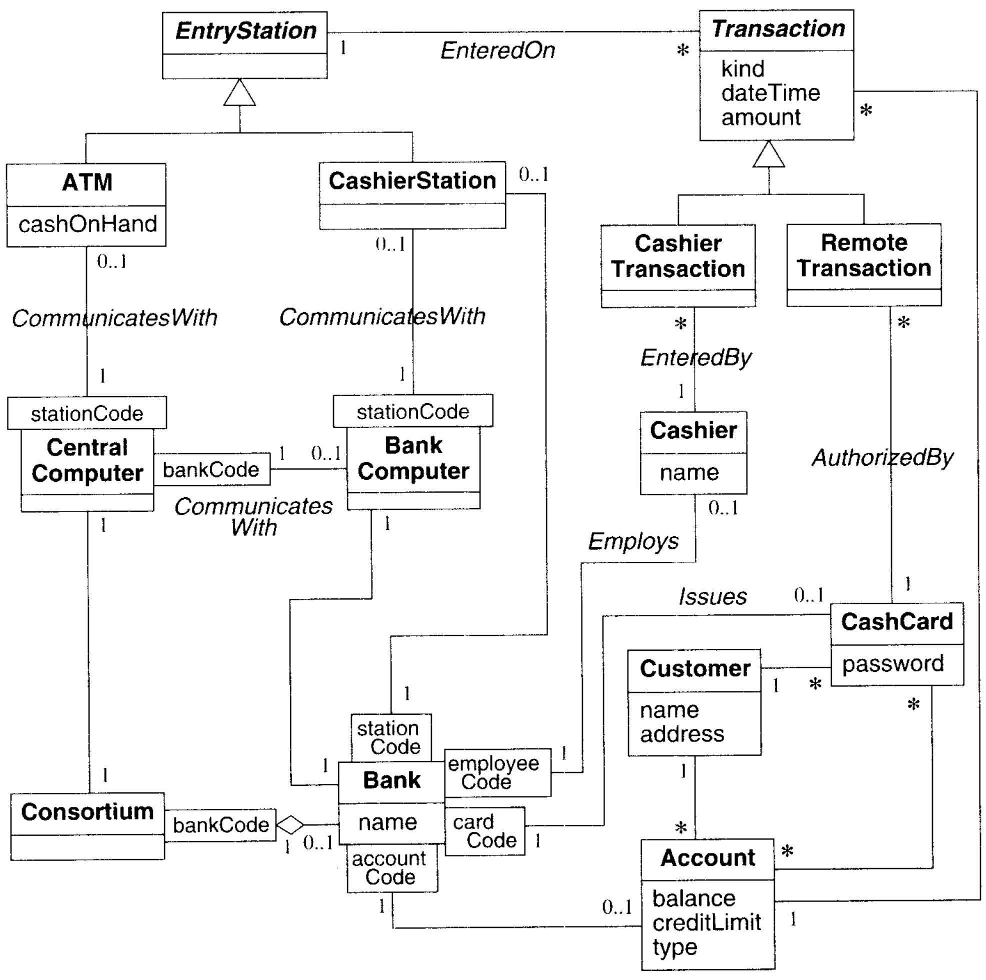


Figure 12.11 ATM class model with attributes and inheritance

- **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 processed 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.

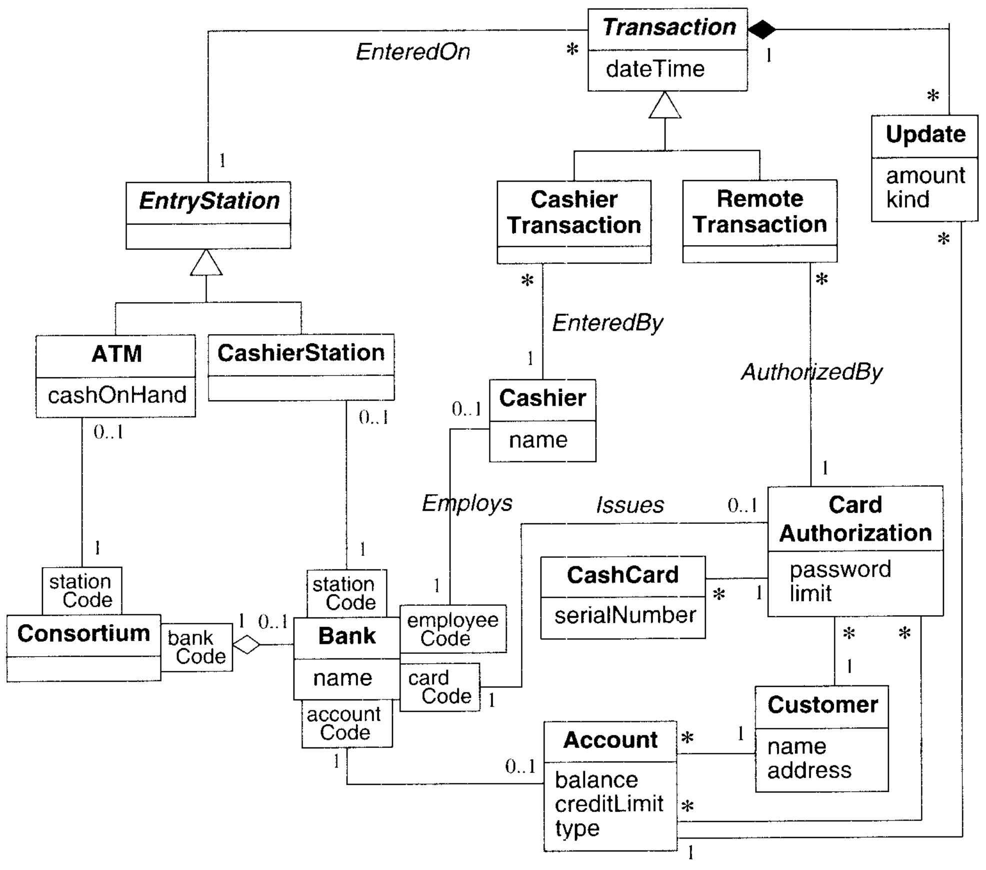


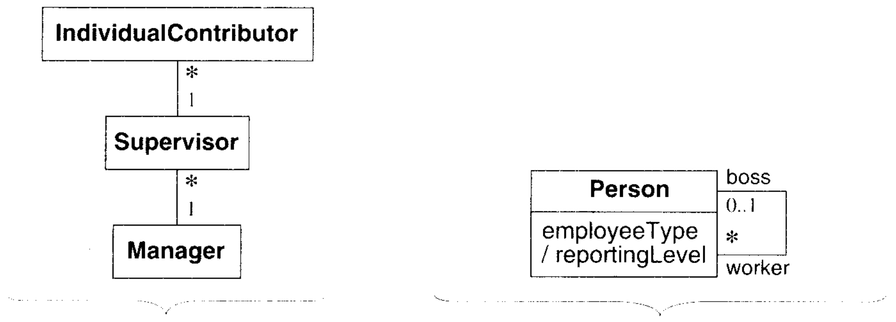
Figure 12.12 ATM class model after further revision

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.



Original model ***Improved model that is more abstract***

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?

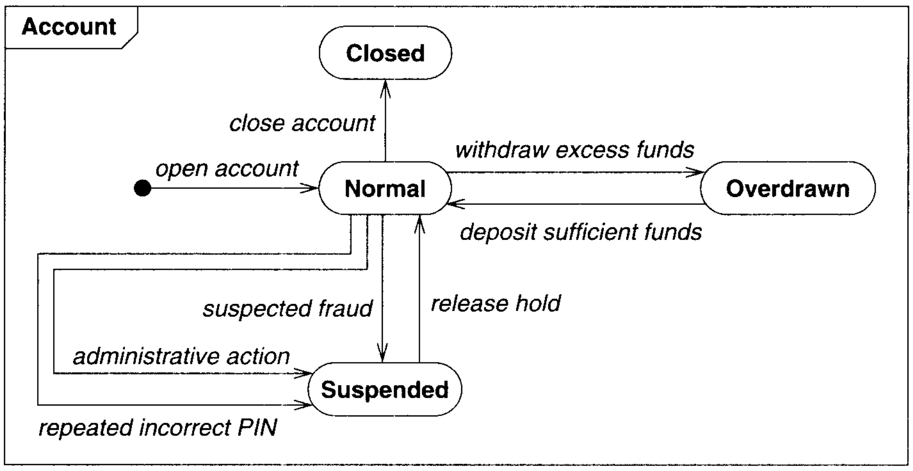


Figure 12.14 Domain state model. The domain state model documents important classes that change state in the real world.

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.

building the domain class model	finding classes
building the domain state model	finding events
data dictionary	finding states
domain analysis	refining a model with inheritance
finding associations	shifting the level of abstraction
finding attributes	testing the model

Figure 12.15 Key concepts for Chapter 12

References

[Abbott-83] Russell J. Abbott. Program Design by Informal English Descriptions. *Communications of the ACM 26*, 11 (November 1983), 882–894.

[Coad-95] Peter Coad, David North, and Mark Mayfield. *Object Models: Strategies, Patterns, and Applications*. Upper Saddle River, NJ: Yourdon Press, 1995.

Exercises

12.1 (3) For each of the following systems, identify the relative importance of the three aspects of modeling: 1) class modeling, 2) state modeling, 3) interaction modeling. Explain your answers. For example, for a compiler, the answer might be 3, 1, and 2. Interaction modeling is most important for a compiler because it is dominated by data transformation concerns.

a. bridge player
b. change-making machine
c. car cruise control
d. electronic typewriter
e. spelling checker
f. telephone answering machine

12.2 (7) Create a class diagram for each system from Exercise 11.6. Note that the requirements are incomplete, so your class models will also be incomplete.

Exercises 12.3–12.8 are related. Do the exercises in sequence. The following are tentative specifications for a simple diagram editor that could be used as the core of a variety of applications.

The editor will be used interactively to create and modify drawings. A drawing contains several sheets. Drawings are saved to and loaded from named ASCII files. Sheets contain boxes and links. Each box may optionally contain a single line of text. Text is allowed only in boxes. The editor must automatically adjust the size of a box to fit any enclosed text. The font size of the text is not adjustable. Any pair of boxes on the same sheet may be linked by a series of alternating horizontal and vertical lines. Figure E12.1 shows a simple, one sheet drawing.

The editor will be menu driven, with pop-up menus. A three-button mouse will be used for menu, object, and link selections. The following are some operations the editor should provide: create sheet, delete sheet, next sheet, previous sheet, create box, link boxes, enter text, group selection, cut selections, move selections, copy selections, paste, edit text, save drawing, and load drawing. Copy, cut,

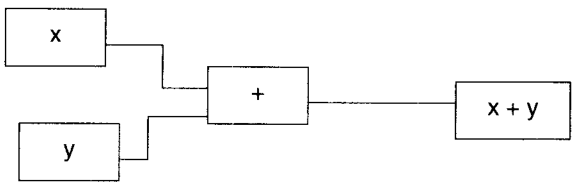


Figure E12.1 A sample drawing

and paste will work through a buffer. Copy will create a copy of selections from a sheet to the buffer. Cut will remove selections to the buffer. Paste will copy the contents of the buffer to the sheet. Each copy and cut operation overwrites the previous contents of the buffer. Pan and zoom will not be allowed; sheets will have fixed size. When boxes are moved, enclosed text should move with them and links should be stretched.

12.3 (3) The following is a list of candidate classes. Prepare a list of classes that should be eliminated for any of the reasons given in this chapter. Give a reason for each elimination. If there is more than one reason, give the main one.

character, line, *x* coordinate, *y* coordinate, link, position, length, width, collection, selection, menu, mouse, button, computer, drawing, drawing file, sheet, pop-up, point, menu item, selected object, selected line, selected box, selected text, file name, box, buffer, line segment coordinate, connection, text, name, origin, scale factor, corner point, end point, graphics object.

12.4 (3) Prepare a data dictionary for proper classes from the previous exercise.

12.5 (3) The following is a list of candidate associations and generalizations for the diagram editor. Prepare a list of associations and generalizations that should be eliminated or renamed for any of the reasons given in this chapter. Give a reason for each elimination or renaming. If there is more than one reason, give the main one.

a box has text, a box has a position, a link logically associates two boxes, a box is moved, a link has points, a link is defined by a sequence of points, a selection or a buffer or a sheet is a collection, a character string has a location, a box has a character string, a character string has characters, a line has length, a collection is composed of links and boxes, a link is deleted, a line is moved, a line is a graphical object, a point is a graphical object, a line has two points, a point has an *x* coordinate, a point has a *y* coordinate

12.6 Figure E12.2 is a partially completed class diagram for the diagram editor. Show how could it be used for each of the following queries. Use a combination of the OCL (see Chapter 3) and pseudocode to express your queries.

a. (2) Find all selected boxes and links.
b. (4) Given a box, determine all other boxes that are directly linked to it.
c. (8) Given a box, find all other boxes that are directly or indirectly linked to it.
d. (2) Given a box and a link, determine if the link involves the box.
e. (3) Given a box and a link, find the other box logically connected to the given box through the other end of the link.
f. (4) Given two boxes, determine all links between them.
g. (6) Given a selection, determine which links are "bridging" links. If a selection does not include all boxes on a sheet, "bridging" links may result. A "bridging" link is a link that con-

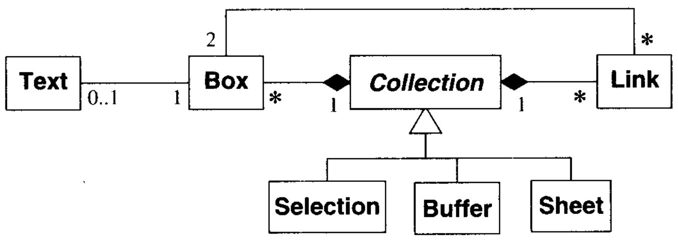


Figure E12.2 Partially completed class diagram for a diagram editor

nects a box that has been selected to a box that has not. A link that connects two boxes that are selected or two boxes that are not selected is not a "bridging" link. "Bridging" links require special handling during a *cut* or a *move* operation on a selection.

12.7 (6) Figure E12.3 is a variation of the class diagram in which the class *Connection* explicitly represents the connection of a link to a box. Redo the queries from the previous exercise using this representation.

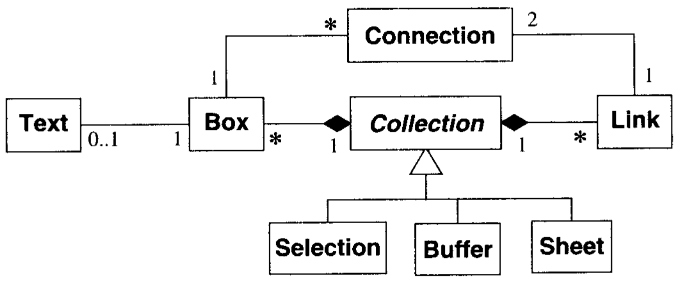


Figure E12.3 Alternative partially completed class diagram for a diagram editor

12.8 (5) What classes require state diagrams? Describe some relevant states and events.

Exercises 12.9–12.13 are related. Do the exercises in sequence. These exercises concern a computerized scoring system that you have volunteered to create for the benefit of a local children's synchronized swimming league. Teams get together for competitions called meets during which the children perform in two types of events: figures and routines. Figure events, which are performed individually, are particular water ballet maneuvers such as swimming on your back with one leg raised straight up. Routines, which are performed by the entire team, are water ballets. Both figures and routines are scored, but your system need only address figures.

Each child must provide his or her name, age, address, and team name to register prior to the meet. To simplify scoring, each contestant is assigned a number.

During a meet, figure events are held simultaneously at several stations that are set up around a swimming pool, usually one at each corner. There are volunteer judges and scorekeepers. Scorekeepers tend to tire, so there is often turnover in their ranks. Several judges and scorekeepers are assigned to each station during a meet. Over the course of a season each judge and scorekeeper may serve several stations. For scoring consistency, each figure is held at exactly one station with the same judges. A station may process several figure events in the course of a meet.

Contestants are organized into groups, with each group starting at a different station. When a child is finished at one station, he or she proceeds to another station for another event. When everyone has been processed at a station for a given event, the station switches to the next event assigned to it.

Each competitor gets one try at each event, called a trial. Just before a trial, the child's number is announced to the child and to the scorekeepers. Sometimes the children get out of order or the scorekeepers become confused and the station stops while the problem is fixed. Each judge indicates a raw score for each observed trial by holding up numbered cards. The raw scores are read to the scorekeepers, who record them and compute a net score for the trial. The highest and lowest raw scores are discarded, and the average of the remaining scores is multiplied by a difficulty factor for the figure.

Individual and team prizes are awarded at the conclusion of a meet based on top individual and team scores. There are several age categories, with separate prizes for each category. Individual prizes are based on figures only. Team prizes are based on figures and routines.

Your system will be used to store all information needed for scheduling, registering, and scoring. At the beginning of a season, all swimmers will be entered into the system and a season schedule will be prepared, including deciding which figures will be judged at which meets. Prior to a meet, the system will be used to process registrations. During a meet, it will record scores and determine winners.

12.9 (3) The following is a list of candidate classes for the scoring system. Prepare a list of classes that should be eliminated for any of the reasons given in this chapter. Give a reason for each elimination. If there is more than one reason, give the main one.

address, age, age category, average score, back, card, child, child's name, competitor, compute average, conclusion, contestant, corner, date, difficulty factor, event, figure, file of team member data, group, individual, individual prize, judge, league, leg, list of scheduled meets, meet, net score, number, person, pool, prize, register, registrant, raw score, routine, score, scorekeeper, season, station, team, team prize, team name, trial, try, water ballet.

12.10 (3) Prepare a data dictionary for proper classes from the previous exercise.

12.11 (4) The following is a list of candidate associations and generalizations for the scoring system. Prepare a list of associations and generalizations that should be eliminated or renamed for any of the reasons given in this chapter. Give a reason for each elimination or renaming. If there is more than one reason, give the main one.

a season consists of several meets, a competitor registers, a competitor is assigned a number, a number is announced, competitors are split into groups, a meet consists of several events, several stations are set up at a meet, several events are processed at a station, several judges are assigned to a station, routines and figures are events, raw scores are read, highest score is discarded, lowest score is discarded, figures are processed, a league consists of several teams, a team consists of several competitors, a trial of a figure is made by a competitor, a trial receives several scores from the judges, prizes are based on scores.

12.12 Figure E12.4 is a partially completed class diagram for the scoring system. The association between meet and event is not derived, because an event may be determined for a meet before a station is assigned to it. Show how it could be used for each of the following queries. Use a combination of the OCL (see Chapter 3) and pseudocode to express your queries.

a. (2) Find all the members of a given team.

b. (6) Find which figures were held more than once in a given season.

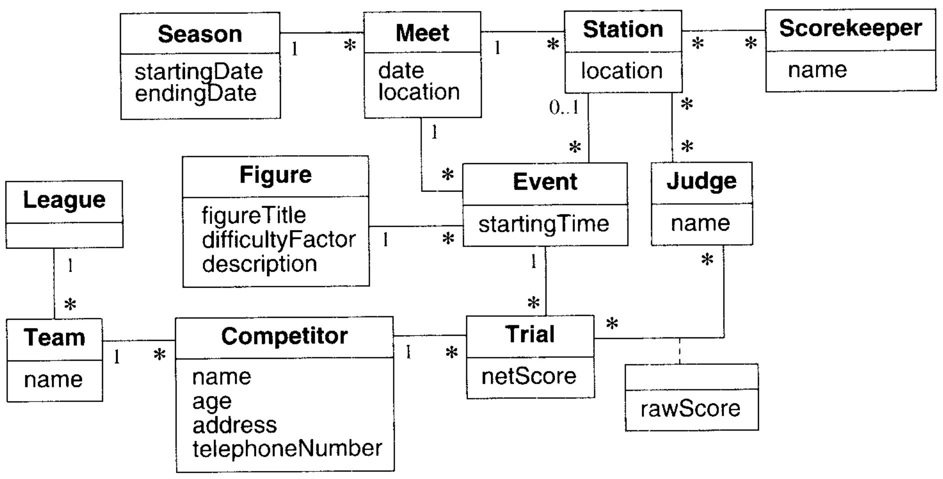


Figure E12.4 Partially completed class diagram for a scoring system

c. (6) Find the net score of a competitor for a given figure at a given meet.
d. (6) Find the team average over all figures in a given season.
e. (6) Find the average score of a competitor over all figures in a given meet.
f. (6) Find the team average in a given figure at a given meet.
g. (4) Find the set of all individuals who competed in any events in a given season.
h. (7) Find the set of all individuals who competed in all of the events held in a given season.
i. (6) Find all the judges who judged a given figure in a given season.
j. (6) Find the judge who awarded the lowest score during a given event.
k. (6) Find the judge who awarded the lowest score for a given figure.
l. (7) Modify the diagram so that the competitors registered for an event can be determined.

12.13 (5) What classes require state diagrams? Describe some relevant states and events.

12.14 (7) Revise the diagrams in Figure E12.5, Figure E12.6, Figure E12.7, and Figure E12.8 to eliminate ternary associations. In some cases you will have to promote the association to a class.

Figure E12.5 is a relationship between *Doctor*, *Patient*, and *DateTime* that might be encountered in a system used by a clinic with several doctors on the staff. The combination of *DateTime* + *Patient* is unique as well as *DateTime* + *Doctor*.

Figure E12.6 is a relationship between *Student*, *Professor*, and *University* that might be used to express the contacts between students attending and professors teaching at several universities. There is one link in the relationship for a student that takes one or more classes from a professor at a university. The combination of *Student* + *Professor* + *University* is unique.

Figure E12.7 shows the relationship expressing the seating arrangement at a concert. *Concert* + *Seat* is unique.

Figure E12.8 expresses the connectivity of a directed graph. Each edge of a directed graph is connected in a specific order to exactly two vertices. More than one edge can be connected between a given pair of vertices. The attribute *Edge* is unique for the relationship.

In each case, try to come as close as possible to the original intent and compare the merits of the original and the revised models.

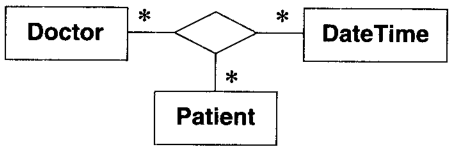


Figure E12.5 Ternary association for ***Doctor, Patient,*** **and** ***DateTime***

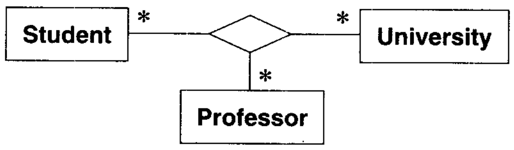


Figure E12.6 Ternary association for ***Student, Professor,*** **and** ***University***

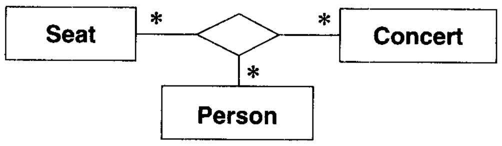


Figure E12.7 Ternary association for ***Seat, Person,*** **and** ***Concert***

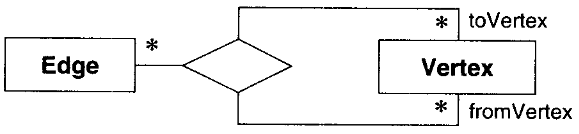


Figure E12.8 Ternary association for directed graphs

12.15 (9) Figure E12.9 lists requirements for a document manager. We then prepared the initial model in Figure E12.10. Note some flaws in the model.

Develop software for managing professional records of papers, books, journals, notes, and computer files. The system must be able to record authors of published works in the appropriate order, name of work, date of publication, publisher, publisher city, an abstract, as well as a comment. The software must be able to group published works into various categories that are defined by the user to facilitate searching. The user must be able to assign a quality indicator of the perceived value of each work.

Only some of the papers in each issue of a journal may be of interest. It would also be helpful to be able to attach comments to sections or even individual pages of a work.

Figure E12.9 Requirements for a document manager

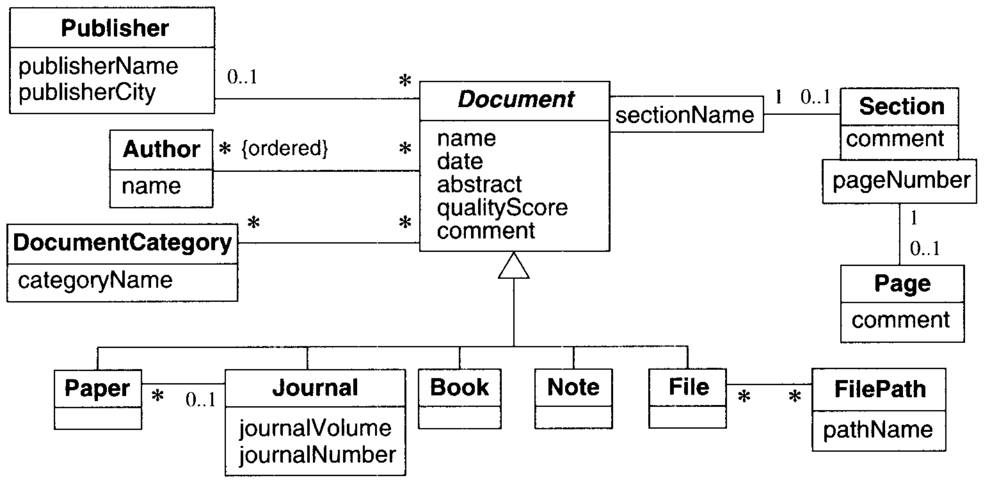


Figure E12.10 Initial model for a document manager

- There is little difference between subclasses. Is an outline of a paper a "paper" or a "note"? How should we handle a paper that is in both an electronic file and a binder? How should we represent information about slides for talks?
- We would like to handle both standard comments (applicable to many documents and chosen by point and click in a user interface) and custom comments (applicable to one document and specifically typed by the user).
- We should be able to comment on a numbered page without having sections.

Improve the model by making it more abstract. (Hint: You should have generic classes for location, document properties, and comments. It is adequate to represent document composition with a hierarchy.)

Exercises 12.16–12.19 are related. Do the exercises in sequence. The following are tentative specifications for scheduling software.

The scheduling software must support the following functions: arranges meetings, schedules appointments, plans tasks, and tracks holidays (including vacations).

The scheduler runs on a network that many users share. Each user may have a schedule. A schedule contains multiple entries. Most entries belong to a single schedule; however, a meeting entry may appear in many schedules.

There are four kinds of entries: meetings, appointments, tasks, and holidays. Meetings and appointments both occur within a single day and have a start time and end time. In contrast, tasks and holidays may extend over several days and just have a start date and end date. Any entry may be repeated. Repeat information includes how often the entry should be repeated, when it starts, and when it ends.

12.16 (3) The following is a list of candidate classes. Prepare a list of classes that should be eliminated for any of the reasons given in this chapter. Give a reason for each elimination. If there is more than one reason, give the main one.

scheduling software, function, meeting, appointment, task, holiday, vacation, scheduler, network, user, schedule, entry, meeting entry, day, start time, end time, start date, end date, repeat information.

12.17 (3) Prepare a data dictionary for proper classes from the previous exercise.

12.18 (4) The following is a list of candidate associations and generalizations for the scoring system. Prepare a list of associations and generalizations that should be eliminated or renamed for any of the reasons given in this chapter. Give a reason for each elimination or renaming. If there is more than one reason, give the main one.

- scheduling software that supports the following functions
- the scheduler runs on a network that many users share
- user may have a schedule
- a schedule contains multiple entries
- entries pertain to a single schedule
- a meeting entry may appear in many schedules
- meetings and appointments both occur within a single day and have a start time and end time
- tasks and holidays may extend over several days and just have a start date and end date.

12.19 (5) Construct a class model for the scheduling software.

Exercises 12.20–12.23 are related. Do the exercises in sequence. The following provides requirements for meetings and extends the scheduling software from Exercises 12.16–12.19.

The scheduling software facilitates meetings. When a user (the chairperson) arranges a meeting, the software places a meeting entry in the schedule of each attendee. The chairperson uses the scheduler to reserve a room for the meeting, to identify the attendees, and to find time on their schedules when everyone is available. The chairperson can indicate whether the attendance for each attendee is required or optional. The system tracks the acceptance status for each attendee—whether an attendee has accepted or declined.

The scheduler manages meeting notices. When a meeting is set up, the scheduler sends invitations to all attendees, who are able to view meeting information. Each invitee can accept or refuse as well as possibly cancel later on. The system also manages notices in case the meeting is rescheduled or cancelled.

12.20 (3) The following is a list of candidate classes. Prepare a list of classes that should be eliminated for any of the reasons given in this chapter. Give a reason for each elimination. If there is more than one reason, give the main one.

scheduling software, meeting, user, chairperson, software, meeting entry, schedule, attendee, scheduler, room, time, everyone, attendance, acceptance status, meeting notice, invitation, meeting information, invitee, notice.

12.21 (3) Prepare a data dictionary for proper classes from the previous exercise.

12.22 (4) The following is a list of candidate associations and generalizations. Prepare a list of associations and generalizations that should be eliminated or renamed for any of the reasons given in this chapter. Give a reason for each elimination or renaming. If there is more than one reason, give the main one.

- scheduling software facilitates meetings

- user (the chairperson) arranges a meeting
- software places a meeting entry in the schedule of each attendee
- chairperson uses the scheduler to reserve a room for the meeting, to identify the attendees, and to find time on their schedules when everyone is available
- chairperson can indicate whether the attendance for each attendee is required or optional
- system tracks the acceptance status for each attendee—whether an attendee has accepted or declined
- scheduler manages meeting notices
- scheduler sends invitations to all attendees, who are able to view meeting information
- system also manages notices in case the meeting is rescheduled or cancelled.

12.23 (7) Construct a class model for the extension to the scheduling software. Your answer should resolve a problem from Exercise 12.19. In the class model for our answer to Exercise 12.19, we cannot tell which user owns an entry. (Hint: You should reconcile the chairperson and attendee associations from the extended requirements with the association between *Schedule* and *Entry* from the Exercise 12.16–12.19 requirements.)

13 Application Analysis

This chapter completes our treatment of analysis by adding major application artifacts to the domain model from the prior chapter. We include these application artifacts in analysis, because they are important, visible to users, and must be approved by them. In general, you cannot find the application classes in the domain itself, but must find them in use cases.

13.1 Application Interaction Model

Most domain models are static and operations are unimportant, because a domain as a whole usually doesn't *do* anything. The focus of domain modeling is on building a model of intrinsic concepts. After completing the domain model we then shift our attention to the details of an application and consider interaction.

Begin interaction modeling by determining the overall boundary of the system. Then identify use cases and flesh them out with scenarios and sequence diagrams. You should also prepare activity diagrams for use cases that are complex or have subtleties. Once you fully understand the use cases, you can organize them with relationships. And finally check against the domain class model to ensure that there are no inconsistencies.

You can construct an application interaction model with the following steps.

- Determine the system boundary. [13.1.1]
- Find actors. [13.1.2]
- Find use cases. [13.1.3]
- Find initial and final events. [13.1.4]
- Prepare normal scenarios. [13.1.5]
- Add variation and exception scenarios. [13.1.6]
- Find external events. [13.1.7]
- Prepare activity diagrams for complex use cases. [13.1.8]

- Organize actors and use cases. [13.1.9]
- Check against the domain class model. [13.1.10]

13.1.1 Determining the System Boundary

You must know the precise scope of an application—the boundary of the system—in order to specify functionality. This means that you must decide what the system includes and, more importantly, what it omits. If the system boundary is drawn correctly, you can treat the system as a black box in its interactions with the outside world—you can regard the system as a single object, whose internal details are hidden and changeable. During analysis, you determine the purpose of the system and the view that it presents to its actors. During design, you can change the internal implementation of the system as long as you maintain the external behavior.

Usually, you should not consider humans as part of a system, unless you are modeling a human organization, such as a business or a government department. Humans are actors that must interact with the system, but their actions are not under the control of the system. However, you must allow for human error in your system.

ATM example. The original problem statement from Chapter 11 says to "design the software to support a computerized banking network including both human cashiers and automatic teller machines..." Now it is important that cashier transactions and ATM transactions be seamless—from the customer's perspective either method of conducting business should yield the same effect on a bank account. However, in commercial practice an ATM application would be separate from a cashier application—an ATM application spans banks while a cashier application is internal to a bank. Both applications would share the same underlying domain model, but each would have its own distinct application model. For this chapter we focus on ATM behavior and ignore cashier details.

13.1.2 Finding Actors

Once you determine the system boundary, you must identify the external objects that interact directly with the system. These are its *actors*. Actors include humans, external devices, and other software systems. The important thing about actors is that they are not under control of the application, and you must consider them to be somewhat unpredictable. That is, even though there may be an expected sequence of behavior by the actors, an application's design should be robust so that it does not crash if an actor fails to behave as expected.

In finding actors, we are not searching for individuals but for archetypical behavior. Each actor represents an idealized user that exercises some subset of the system functionality. Examine each external object to see if it has several distinct faces. An actor is a coherent face presented to the system, and an external object may have more than one actor. It is also possible for different kinds of external objects to play the part of the same actor.

ATM example. A particular person may be both a bank teller and a customer of the same bank. This is an interesting but usually unimportant coincidence—a person approaches the bank in one or the other role at a time. For the ATM application, the actors are *Customer*, *Bank*, and *Consortium*.

13.1.3 Finding Use Cases

For each actor, list the fundamentally different ways in which the actor uses the system. Each of these ways is a *use case*. The use cases partition the functionality of a system into a small number of discrete units, and all system behavior must fall under some use case. You may have trouble deciding where to place some piece of marginal behavior. Keep in mind that there are always borderline cases when making partitions; just make a decision even if it is somewhat arbitrary.

Each use case should represent a kind of service that the system provides—something that provides value to the actor. Try to keep all of the use cases at a similar level of detail. For example, if one use case in a bank is "apply for loan," then another use case should not be "withdraw cash from savings account using ATM." The latter description is much more detailed than the former; a better match would be "make withdrawal." Try to focus on the main goal of the use case and defer implementation choices.

At this point you can draw a preliminary use case diagram. Show the actors and the use cases, and connect actors to use cases. Usually you can associate a use case with the actor that initiates it, but other actors may be involved as well. Don't worry if you overlook some participating actors. They will become apparent when you elaborate the use cases. You should also write a one or two sentence summary for each use case.

ATM example. Figure 13.1 shows the use cases, and the bullets summarize them.

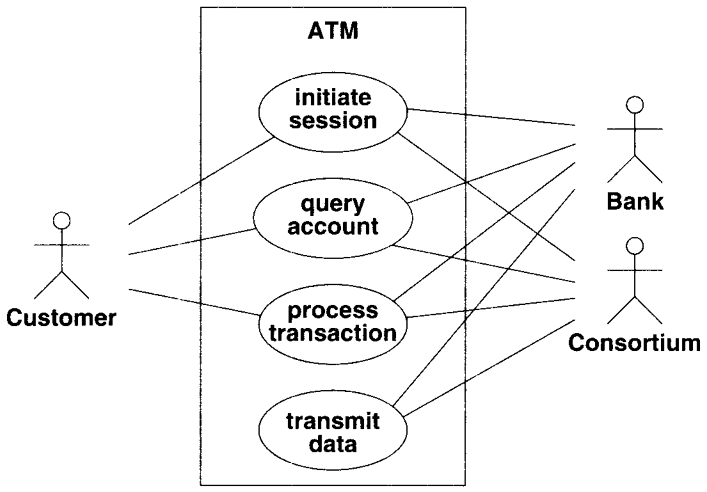


Figure 13.1 Use case diagram for the ATM. Use cases partition the functionality of a system into a small number of discrete units that cover its behavior.

- **Initiate session.** The ATM establishes the identity of the user and makes available a list of accounts and actions.
- **Query account.** The system provides general data for an account, such as the current balance, date of last transaction, and date of mailing for last statement.

- **Process transaction**. The ATM system performs an action that affects an account's balance, such as deposit, withdraw, and transfer. The ATM ensures that all completed transactions are ultimately written to the bank's database.
- **Transmit data**. The ATM uses the consortium's facilities to communicate with the appropriate bank computers.

13.1.4 Finding Initial and Final Events

Use cases partition system functionality into discrete pieces and show the actors that are involved with each piece, but they do not show the behavior clearly. To understand behavior, you must understand the execution sequences that cover each use case. You can start by finding the events that initiate each use case. Determine which actor initiates the use case and define the event that it sends to the system. In many cases, the initial event is a request for the service that the use case provides. In other cases, the initial event is an occurrence that triggers a chain of activity. Give this event a meaningful name, but don't try to determine its exact parameter list at this point.

You should also determine the final event or events and how much to include in each use case. For example, the use case of applying for a loan could continue until the application is submitted, until the loan is granted or rejected, until the money from the loan is delivered, or until the loan is finally paid off and closed. All of these could be reasonable choices. The modeler must define the scope of the use case by defining when it terminates.

ATM example. Here are initial and final events for each use case.

- **Initiate session**. The initial event is the customer's insertion of a cash card. There are two final events: the system keeps the cash card or the system returns the cash card.
- **Query account**. The initial event is a customer's request for account data. The final event is the system's delivery of account data to the customer.
- **Process transaction**. The initial event is the customer's initiation of a transaction. There are two final events: committing or aborting the transaction.
- **Transmit data**. The initial event could be triggered by a customer's request for account data. Another possible initial event could be recovery from a network, power, or another kind of failure. The final event is successful transmission of data.

13.1.5 Preparing Normal Scenarios

For each use case, prepare one or more typical dialogs to get a feel for expected system behavior. These scenarios illustrate the major interactions, external display formats, and information exchanges. A *scenario* is a sequence of events among a set of interacting objects. Think in terms of sample interactions, rather than trying to write down the general case directly. This will help you ensure that important steps are not overlooked and that the overall flow of interaction is smooth and correct.

For most problems, logical correctness depends on the sequences of interactions and not their exact times. (Real-time systems, however, do have specific timing requirements on interactions, but we do not address real-time systems in this book.)

Sometimes the problem statement describes the full interaction sequence, but most of the time you will have to invent (or at least flesh out) the interaction sequence. For example, the ATM problem statement indicates the need to obtain transaction data from the user but is vague about exactly what parameters are needed and in what order to ask for them. During analysis, try to avoid such details. For many applications, the order of gathering input is not crucial and can be deferred to design.

Prepare scenarios for "normal" cases—interactions without any unusual inputs or error conditions. An event occurs whenever information is exchanged between an object in the system and an outside agent, such as a user, a sensor, or another task. The information values exchanged are event parameters. For example, the event *password entered* has the password value as a parameter. Events with no parameters are meaningful and even common. The information in such an event is the fact that it has occurred. For each event, identify the actor (system, user, or other external agent) that caused the event and the parameters of the event.

ATM example. Figure 13.2 shows a normal scenario for each use case.

13.1.6 Adding Variation and Exception Scenarios

After you have prepared typical scenarios, consider "special" cases, such as omitted input, maximum and minimum values, and repeated values. Then consider error cases, including invalid values and failures to respond. For many interactive applications, error handling is the most difficult part of development. If possible, allow the user to abort an operation or roll back to a well-defined starting point at each step. Finally consider various other kinds of interactions that can be overlaid on basic interactions, such as help requests and status queries.

ATM example. Some variations and exceptions follow. We could prepare scenarios for each of these but will not go through the details here. (See the exercises.)

- The ATM can't read the card.
- The card has expired.
- The ATM times out waiting for a response.
- The amount is invalid.
- The machine is out of cash or paper.
- The communication lines are down.
- The transaction is rejected because of suspicious patterns of card usage.

There are additional scenarios for administrative parts of the ATM system, such as authorizing new cards, adding banks to the consortium, and obtaining transaction logs. We will not explore these aspects.

13.1.7 Finding External Events

Examine the scenarios to find all external events—include all inputs, decisions, interrupts, and interactions to or from users or external devices. An event can trigger effects for a target object. Internal computation steps are not events, except for computations that interact with

Initiate session	The ATM asks the user to insert a card. The user inserts a cash card. The ATM accepts the card and reads its serial number. The ATM requests the password. The user enters "1234." The ATM verifies the password by contacting the consortium and bank. The ATM displays a menu of accounts and commands. . . . The user chooses the command to terminate the session. The ATM prints a receipt, ejects the card, and asks the user to take them. The user takes the receipt and the card. The ATM asks the user to insert a card
Query account	The ATM displays a menu of accounts and commands. The user chooses to query an account. The ATM contacts the consortium and bank which return the data. The ATM displays account data for the user. The ATM displays a menu of accounts and commands.
Process transaction	The ATM displays a menu of accounts and commands. The user selects an account withdrawal. The ATM asks for the amount of cash. The user enters $100. The ATM verifies that the withdrawal satisfies its policy limits. The ATM contacts the consortium and bank and verifies that the account has sufficient funds. The ATM dispenses the cash and asks the user to take it. The user takes the cash. The ATM displays a menu of accounts and commands.
Transmit data	The ATM requests account data from the consortium. The consortium accepts the request and forwards it to the appropriate bank. The bank receives the request and retrieves the desired data. The bank sends the data to the consortium. The consortium routes the data to the ATM.

Figure 13.2 Normal ATM scenarios. Prepare one or more scenarios for each use case.

the external world. Use scenarios to find normal events, but don't forget unusual events and error conditions.

A transmittal of information to an object is an event. For example, *enter password* is a message sent from external agent *User* to application object *ATM*. Some information flows are implicit. Many events have parameters.

Group together under a single name events that have the same effect on flow of control, even if their parameter values differ. For example, *enter password* should be an event, whose parameter is the password value. The choice of password value does not affect the flow of control; therefore events with different password values are all instances of the same kind of event. Similarly, *dispense cash* is also an event, since the amount of cash dispensed does not affect the flow of control. Event instances whose values affect the flow of control should be distinguished as different kinds of events. *Account OK, bad account,* and *bad password* are all different events; don't group them under *card status*.

You must decide when differences in quantitative values are important enough to distinguish as distinct events. For example, the different digits from a keyboard would usually be considered the same event, since the high-level control does not depend on numerical values. Pushing the "enter" key, however, might be considered a distinct event, since an application could treat it differently. The distinction among events depends on the application.

Prepare a sequence diagram for each scenario. A *sequence diagram* shows the participants in an interaction and the sequence of messages among them; each participant is assigned a column in a table. The sequence diagram clearly shows the sender and receiver of each event. If more than one object of the same class participates in the scenario, assign a separate column to each object. By scanning a particular column in the diagram, you can see the events that directly affect a particular object. From the sequence diagrams you can then summarize the events that each class sends and receives.

ATM example. Figure 13.3 shows a sequence diagram for the *process transaction* scenario. Figure 13.4 summarizes events with the arrows indicating the sender and receiver. For brevity, we do not show event parameters in Figure 13.4.

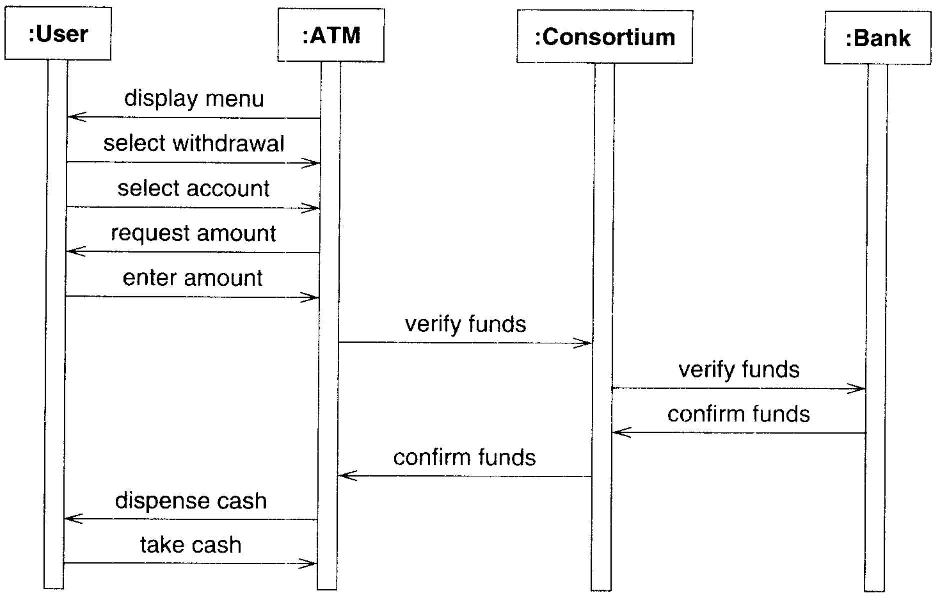


Figure 13.3 Sequence diagram for the *process transaction* scenario. A sequence diagram clearly shows the sender and receiver of each event.

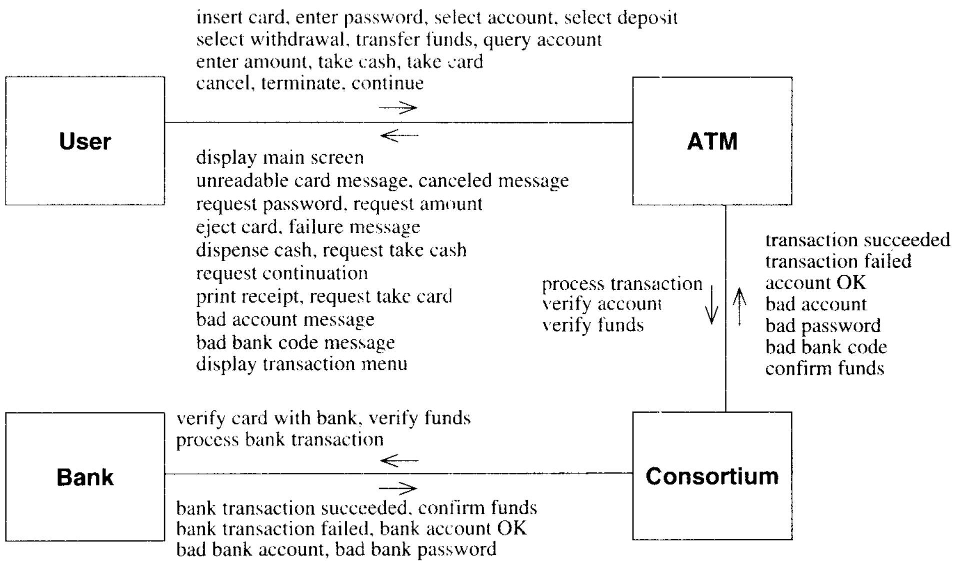

User
ATM
Bank
Consortium
insert card, enter password, select account, select deposit
select withdrawal, transfer funds, query account
enter amount, take cash, take card
cancel, terminate, continue
display main screen
unreadable card message, canceled message
request password, request amount
eject card, failure message
dispense cash, request take cash
request continuation
print receipt, request take card
bad account message
bad bank code message
display transaction menu
process transaction
verify account
verify funds
transaction succeeded
transaction failed
account OK
bad account
bad password
bad bank code
confirm funds
verify card with bank, verify funds
process bank transaction
bank transaction succeeded, confirm funds
bank transaction failed, bank account OK
bad bank account, bad bank password

Figure 13.4 Events for the ATM case study. Tally the events in the scenarios and note the classes that send and receive each event.

13.1.8 Preparing Activity Diagrams for Complex Use Cases

Sequence diagrams capture the dialog and interplay between actors, but they do not clearly show alternatives and decisions. For example, you need one sequence diagram for the main flow of interaction and additional sequence diagrams for each error and decision point. Activity diagrams let you consolidate all this behavior by documenting forks and merges in the control flow. It is certainly appropriate to use activity diagrams to document business logic during analysis, but do not use them as an excuse to begin implementation.

ATM example. As Figure 13.5 shows, when the user inserts a card, there are many possible responses. Some responses indicate a possible problem with the card or account; hence the ATM retains the card. Only the successful completion of the tests allows ATM processing to proceed.

13.1.9 Organizing Actors and Use Cases

The next step is to organize use cases with relationships (include, extend, and generalization—see Chapter 8). This is especially helpful for large and complex systems. As with the class and state models, we defer organization until the base use cases are in place. Otherwise, there is too much of a risk of distorting the structure to match preconceived notions.

Similarly, you can also organize actors with generalization. For example, an *Administrator* might be an *Operator* with additional privileges.

ATM example. Figure 13.6 organizes the use cases with the include relationship.

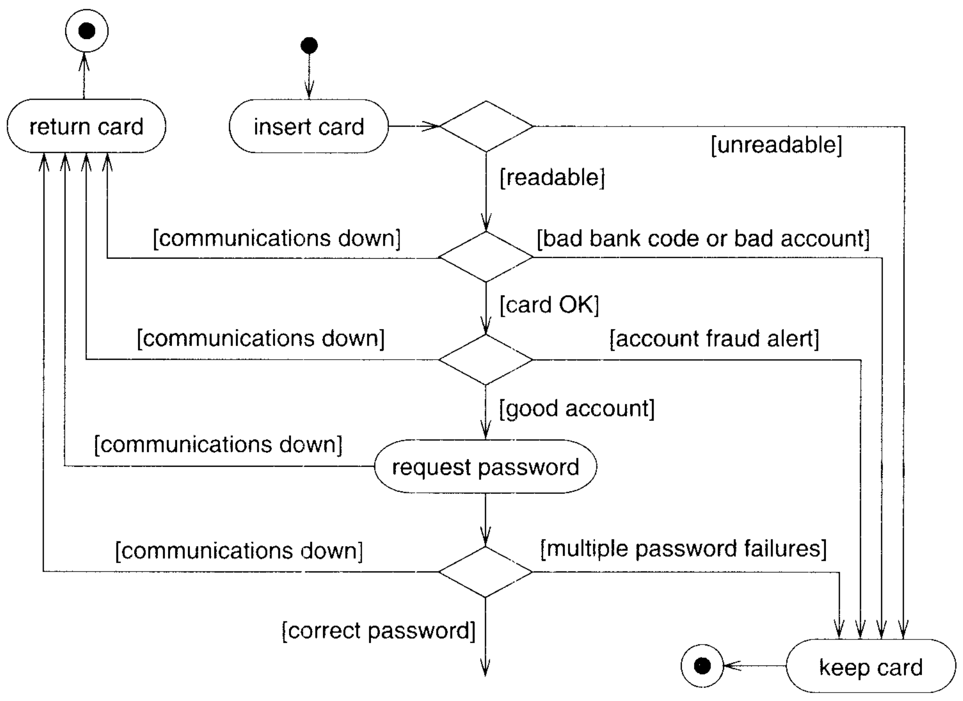


Figure 13.5 Activity diagram for card verification. You can use activity diagrams to document business logic, but do not use them as an excuse to begin premature implementation.

13.1.10 Checking Against the Domain Class Model

At this point, the application and domain models should be mostly consistent. The actors, use cases, and scenarios are all based on classes and concepts from the domain model. Recall that one of the steps in constructing the domain class model is to test access paths. In reality, such testing is a first attempt at use cases.

Cross check the application and domain models to ensure that there are no inconsistencies. Examine the scenarios and make sure that the domain model has all the necessary data. Also make sure that the domain model covers all event parameters.

13.2 Application Class Model

Application classes define the application itself, rather than the real-world objects that the application acts on. Most application classes are computer-oriented and define the way that users perceive the application. You can construct an application class model with the following steps.

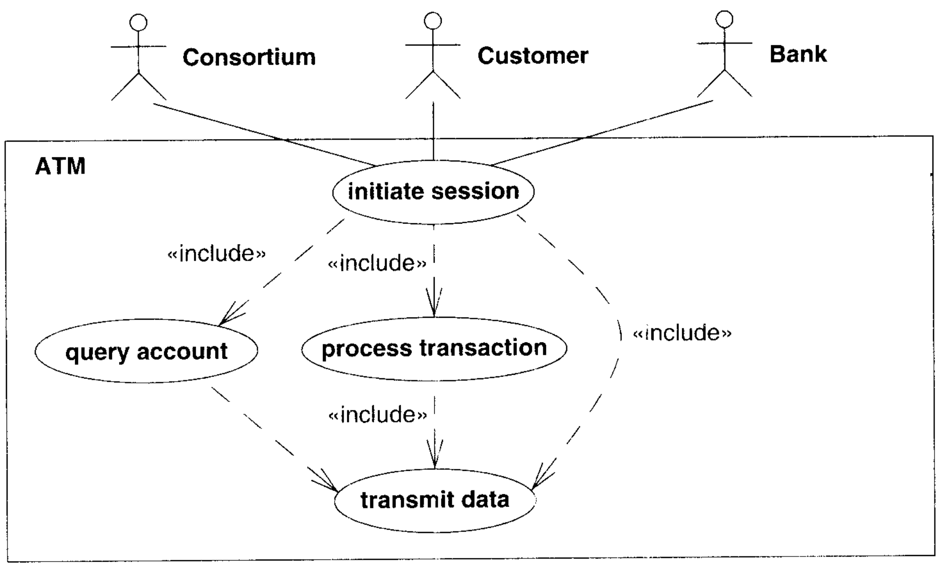


Figure 13.6 Organizing use cases. Once the basic use cases are identified, you can organize them with relationships.

- Specify user interfaces. [13.2.1]
- Define boundary classes. [13.2.2]
- Determine controllers. [13.2.3]
- Check against the interaction model. [13.2.4]

13.2.1 Specifying User Interfaces

Most interactions can be separated into two parts: application logic and the user interface. A ***user interface*** is an object or group of objects that provides the user of a system with a coherent way to access its domain objects, commands, and application options. During analysis the emphasis is on the information flow and control, rather than the presentation format. The same program logic can accept input from command lines, files, mouse buttons, touch panels, physical push buttons, or remote links, if the surface details are carefully isolated.

During analysis treat the user interface at a coarse level of detail. Don't worry about how to input individual pieces of data. Instead, try to determine the commands that the user can perform—a ***command*** is a large-scale request for a service. For example, "make a flight reservation" and "find matches for a phrase in a database" would be commands. The format of inputting the information for the commands and invoking them is relatively easy to change, so work on defining the commands first.

Nevertheless, it is acceptable to sketch out a sample interface to help you visualize the operation of an application and see if anything important has been forgotten. You may also want to mock up the interface so that users can try it. Dummy procedures can simulate application logic. Decoupling application logic from the user interface lets you evaluate the "look and feel" of the user interface while the application is under development.

ATM example. Figure 13.7 shows a possible ATM layout. Its exact details are not important at this point. The important thing is the information exchanged.

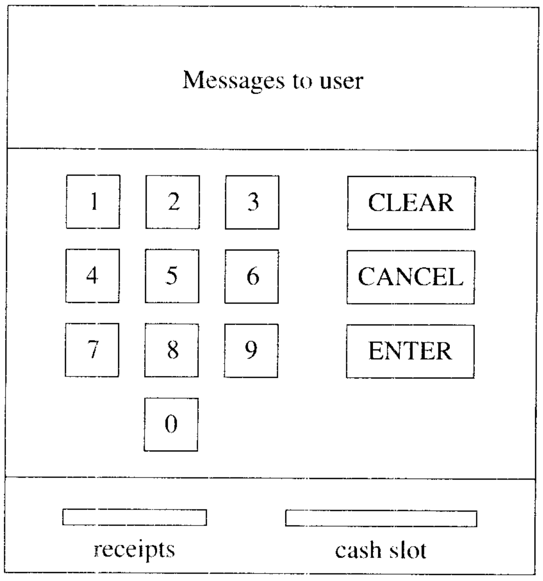


Figure 13.7 Format of ATM interface. Sometimes a sample interface can help you visualize the operation of an application.

13.2.2 Defining Boundary Classes

A system must be able to operate with and accept information from external sources, but it should not have its internal structure dictated by them. It is often helpful to define boundary classes to isolate the inside of a system from the external world. A ***boundary class*** is a class that provides a staging area for communications between a system and an external source. A boundary class understands the format of one or more external sources and converts information for transmission to and from the internal system.

ATM example. It would be helpful to define boundary classes (*CashCardBoundary*, *AccountBoundary*) to encapsulate the communication between the ATM and the consortium. This interface will increase flexibility and make it easier to support additional consortiums.

13.2.3 Determining Controllers

A ***controller*** is an active object that manages control within an application. It receives signals from the outside world or from objects within the system, reacts to them, invokes operations on the objects in the system, and sends signals to the outside world. A controller is a piece of reified behavior captured in the form of an object—behavior that can be manipulated and transformed more easily than plain code. At the heart of most applications are one or more controllers that sequence the behavior of the application.

Most of the work in designing a controller is in modeling its state diagram. In the application class model, however, you should capture the existence of the controllers in a system, the control information that each one maintains, and the associations from the controllers to other objects in the system.

ATM example. It is apparent from the scenarios in Figure 13.2 that the ATM has two major control loops. The outer loop verifies customers and accounts. The inner loop services transactions. Each of these loops could most naturally be handled with a controller.

13.2.4 Checking Against the Interaction Model

As you build the application class model, go over the use cases and think about how they would work. For example, if a user sends a command to the application, the parameters of the command must come from some user-interface object. The requesting of the command itself must come from some controller object. When the domain and application class models are in place, you should be able to simulate a use case with the classes. Think in terms of navigation of the models, as we discussed in Chapter 3. This manual simulation helps to establish that all the pieces are in place.

ATM example. Figure 13.8 shows a preliminary application class model and the domain classes with which it interacts. There are two interfaces—one for users and the other for communicating with the consortium. The application model just has stubs for these classes, because it is not clear how to elaborate them at this time.

Note that the boundary classes "flatten" the data structure and combine information from multiple domain classes. For simplicity, it is desirable to minimize the number of boundary classes and their relationships.

The *TransactionController* handles both queries on accounts and the processing of transactions. The *SessionController* manages *ATMsessions*, each of which services a customer. Each *ATMsession* may or may not have a valid *CashCard* and *Account*. The *SessionController* has a status of *ready*, *impaired* (such as out of paper or cash but still able to operate for some functions), or *down* (such as a communications failure). There is a log of *ControllerProblems* and the specific problem type (bad card reader, out of paper, out of cash, communication lines down, etc.).

13.3 Application State Model

The application state model focuses on application classes and augments the domain state model. Application classes are more likely to have important temporal behavior than domain classes.

First identify application classes with multiple states and use the interaction model to find events for these classes. Then organize permissible event sequences for each class with

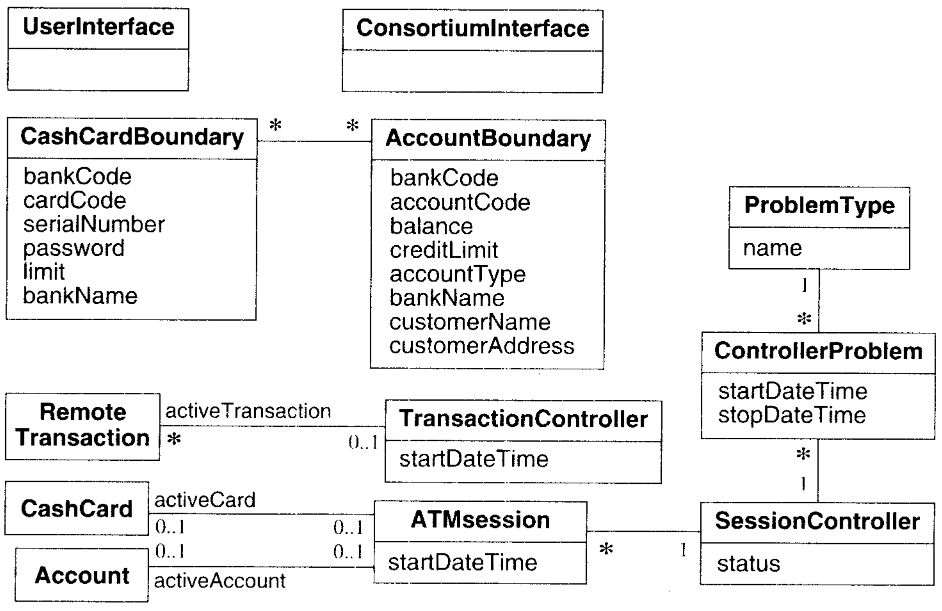


Figure 13.8 **ATM application class model.** Application classes augment the domain classes and are necessary for development.

a state diagram. Next, check the various state diagrams to make sure that common events match. And finally check the state diagrams against the class and interaction models to ensure consistency.

You can construct an application state model with the following steps.

- Determine application classes with states. [13.3.1]
- Find events. [13.3.2]
- Build state diagrams. [13.3.3]
- Check against other state diagrams. [13.3.4]
- Check against the class model. [13.3.5]
- Check against the interaction model. [13.3.6]

13.3.1 Determining Application Classes with States

The application class model adds computer-oriented classes that are prominent to users and important to the operation of an application. Consider each application class and determine which ones have multiple states. User interface classes and controller classes are good candidates for state models. In contrast, boundary classes tend to be static and used for staging data import and export—consequently they are less likely to involve a state model.

ATM example. The user interface classes do not seem to have any substance. This is probably because our understanding of the user interface is incomplete at this point in development. The boundary classes also lack state behavior. However, the controllers do have important states that we will elaborate.

13.3.2 Finding Events

For the application interaction model, you prepared a number of scenarios. Now study those scenarios and extract events. Even though the scenarios may not cover every contingency, they ensure that you do not overlook common interactions and they highlight the major events.

Note the contrast between the domain and application processes for state models. With the domain model, first we find states and then we find events. That is because the domain model focuses on data—significant groupings of data form states that are subject to events. With the application model, in contrast, first we find events and then we determine states. The application model's early attention to events is a consequence of the emphasis on behavior— use cases are elaborated with scenarios that reveal events.

ATM example. We revisit the scenarios from the application interaction model. Some events are: *insert card*, *enter password*, *end session*, and *take card*.

13.3.3 Building State Diagrams

The next step is to build a state diagram for each application class with temporal behavior. Choose one of these classes and consider a sequence diagram. Arrange the events involving the class into a path whose arcs are labeled by the events. The interval between any two events is a state. Give each state a name, if a name is meaningful, but don't bother if it is not. Now merge other sequence diagrams into the state diagram. The initial state diagram will be a sequence of events and states. Every scenario or sequence diagram corresponds to a path through the state diagram.

Now find loops within the diagram. If a sequence of events can be repeated indefinitely, then they form a loop. In a loop, the first state and the last state are identical. If the object "remembers" that it has traversed a loop, then the two states are not really identical, and a simple loop is incorrect. At least one state in a loop must have multiple transactions leaving it or the loop will never terminate.

Once you have found the loops, merge other sequence diagrams into the state diagram. Find the point in each sequence diagram where it diverges from previous ones. This point corresponds to an existing state in the diagram. Attach the new event sequence to the existing state as an alternative path. While examining sequence diagrams, you may think of other possible events that can occur at each state; add them to the state diagram as well.

The hardest thing is deciding at which state an alternate path rejoins the existing diagram. Two paths join at a state if the object "forgets" which one was taken. In many cases, it is obvious from knowledge of the application that two states are identical. For example, inserting two nickels into a vending machine is equivalent to inserting one dime.

Beware of two paths that appear identical but can be distinguished under some circumstances. For example, some systems repeat the input sequence if the user makes an error entering information but give up after a certain number of failures. The repeat sequence is almost the same except that it remembers the past failures. The difference can be glossed over by adding a parameter, such as *number of failures*, to remember information. At least one transition must depend on the parameter.

The judicious use of parameters and conditional transitions can simplify state diagrams considerably but at the cost of mixing together state information and data. State diagrams with too much data dependency can be confusing and counterintuitive. Another alternative is to partition a state diagram into two concurrent subdiagrams, using one subdiagram for the main line and the other for the distinguishing information. For example, a subdiagram to allow for one user failure might have states *No error* and *One error*.

After normal events have been considered, add variation and exception cases. Consider events that occur at awkward times—for example, a request to cancel a transaction after it has been submitted for processing. In cases when the user (or other external agent) may fail to respond promptly and some resource must be reclaimed, a *time-out* event can be generated after a given interval. Handling user errors cleanly often requires more thought and code than the normal case. Error handling often complicates an otherwise clean and compact program structure, but it must be done.

You are finished with the state diagram of a class when the diagram covers all scenarios and the diagram handles all events that can affect a state. You can use the state diagram to suggest new scenarios by considering how some event not already handled should affect a state. Posing "what if" questions is a good way to test for completeness and error-handling capabilities.

If there are complex interactions with independent inputs, you can use a nested state diagram, as Chapter 6 describes. Otherwise a flat state diagram suffices. Repeat the above process of building state diagrams for each class that has time-dependent behavior.

ATM example. Figure 13.9 shows the state diagram for the *SessionController*. The middle of the diagram has the main behavior of processing the card and password. A communications failure can interrupt processing at any time. The ATM returns the card upon a communications failure, but keeps it if there are any suspicious circumstances. After finishing transactions, receipt printing occurs in parallel to card ejection, and the user can take the receipt and card in any order.

Figure 13.10 and Figure 13.11 show the state diagram for the *TransactionController* that is spawned by the *SessionController*. (See the exercises for the other subdiagrams of Figure 13.10.) We have separated the *TransactionController* and the *SessionController* because their purposes are much different—the *SessionController* focuses on verifying users, while the *TransactionController* services account inquiries and balance changes.

13.3.4 Checking Against Other State Diagrams

Check the state diagrams of each class for completeness and consistency. Every event should have a sender and a receiver, occasionally the same object. States without predecessors or successors are suspicious; make sure they represent starting or termination points of the interaction sequence. Follow the effects of an input event from object to object through the system to make sure that they match the scenarios. Objects are inherently concurrent; beware of synchronization errors where an input occurs at an awkward time. Make sure that corresponding events on different state diagrams are consistent.

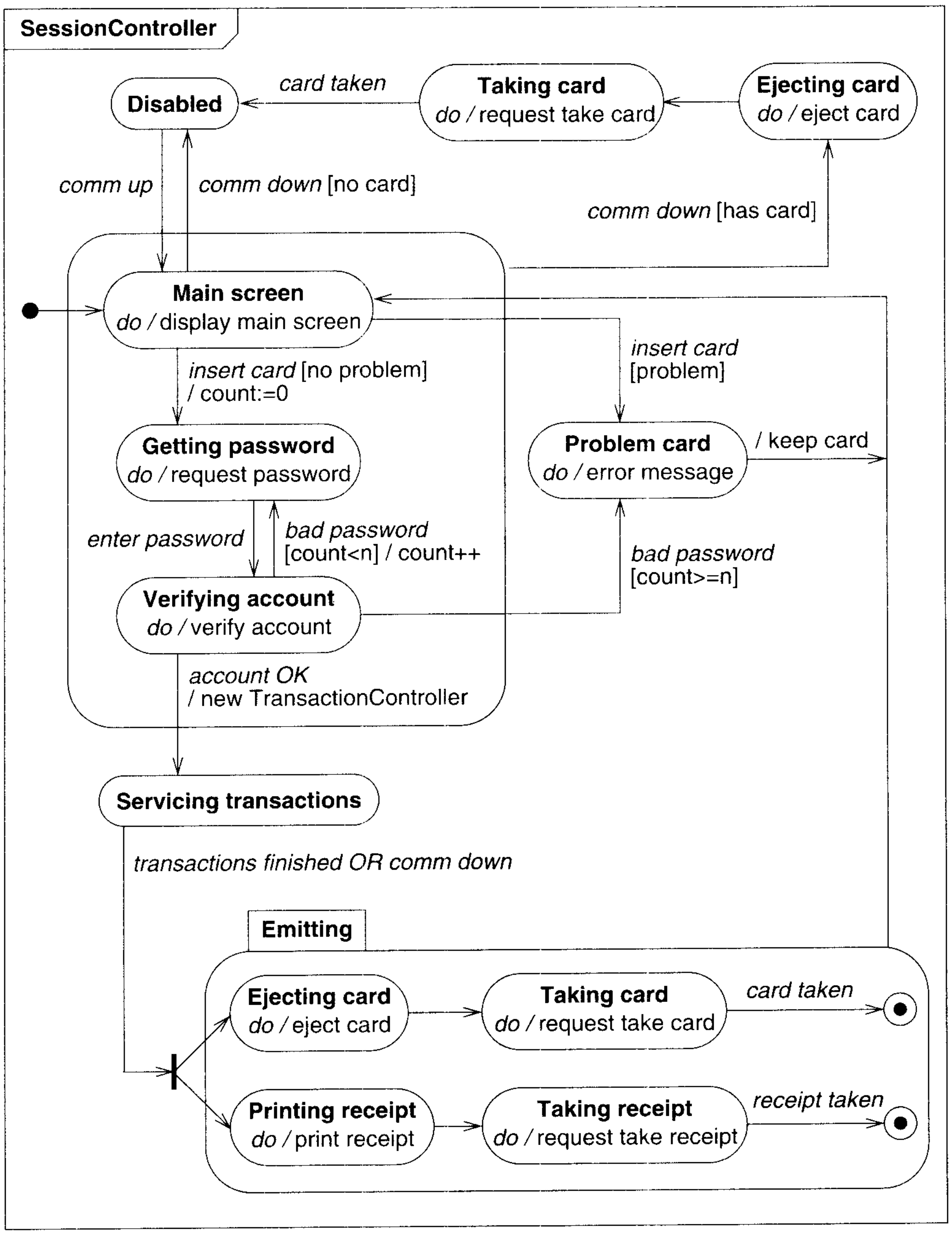


Figure 13.9 State diagram for *SessionController*. Build a state diagram for each application class with temporal behavior.

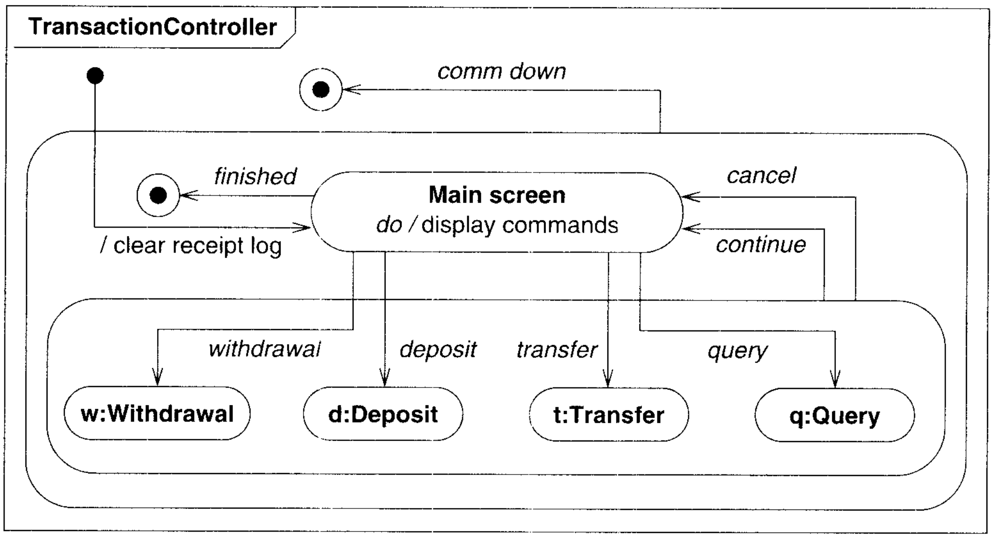


Figure 13.10 State diagram for *TransactionController*. Obtain information from the scenarios of the interaction model.

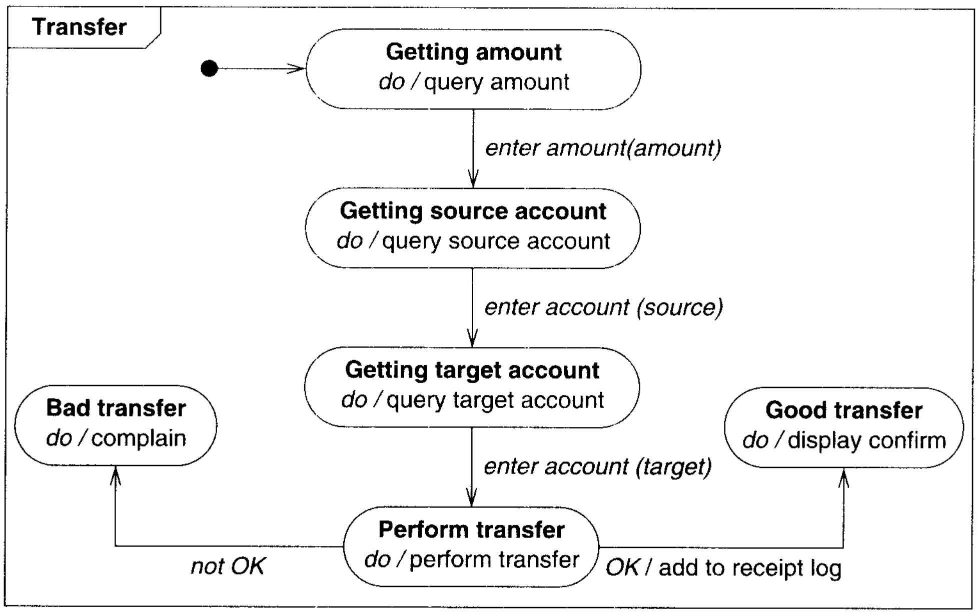


Figure 13.11 State diagram for *Transfer*. This diagram elaborates the *Transfer* state in Figure 13.10.

ATM example. The *SessionController* initiates the *TransactionController*, and the termination of the *TransactionController* causes the *SessionController* to resume.

13.3.5 Checking Against the Class Model

Similarly, make sure that the state diagrams are consistent with the domain and application class models.

ATM example. Multiple ATMs can potentially concurrently access an account. Account access needs to be controlled to ensure that only one update at a time is applied. We will not resolve the details here.

13.3.6 Checking Against the Interaction Model

When the state model is ready, go back and check it against the scenarios of the interaction model. Simulate each behavior sequence by hand and verify that the state diagram gives the correct behavior. If an error is discovered, change either the state diagram or the scenarios. Sometimes a state diagram will uncover irregularities in the scenarios, so don't assume that the scenarios are always correct.

Then take the state model and trace out legitimate paths. These represent additional scenarios. Ask yourself whether they make sense. If not, then modify the state diagram. Often, however, you will discover useful behavior that you had not considered before. The mark of a good design is the discovery of unexpected information that follows from the design, properties that appear meaningful (and often seem obvious) once they are observed.

ATM example. As best as we can tell right now, the state diagrams are sound and consistent with the scenarios.

13.4 Adding Operations

Our style of object-oriented analysis places much less emphasis on defining operations than the traditional programming-based methodologies. We de-emphasize operations because the list of potentially useful operations is open-ended and it is difficult to know when to stop adding them. Operations arise from the following sources, and you should add major operations now. Chapter 15 discusses detailed operations.

13.4.1 Operations from the Class Model

The reading and writing of attribute values and association links are implied by the class model, and you need not show them explicitly. During analysis all attributes and associations are assumed to be accessible.

13.4.2 Operations from Use Cases

Most of the complex functionality of a system comes from its use cases. During the construction of the interaction model, use cases lead to activities. Many of these correspond to operations on the class model.

ATM example. *Consortium* has the activity *verifyBankCode*, and *Bank* has the activity *verifyPassword*. You could implement Figure 13.5 with the operation *verifyCashCard* on class *ATM*.

13.4.3 Shopping-List Operations

Sometimes the real-world behavior of classes suggests operations. Meyer [Meyer-97] calls this a "shopping list," because the operations are not dependent on a particular application but are meaningful in their own right. Shopping-list operations provide an opportunity to broaden a class definition beyond the narrow needs of the immediate problem.

ATM example. Shopping-list operations include:

- account.close()
- bank.createSavingsAccount(customer): account
- bank.createCheckingAccount(customer): account
- bank.createCashCardAuth(customer): cashCardAuthorization
- cashCardAuthorization.addAccount (account)
- cashCardAuthorization.removeAccount (account)
- cashCardAuthorization.close()

13.4.4 Simplifying Operations

Examine the class model for similar operations and variations in form on a single operation. Try to broaden the definition of an operation to encompass such variations and special cases. Use inheritance where possible to reduce the number of distinct operations. Introduce new superclasses as needed to simplify the operations, provided that the new superclasses are not forced and unnatural. Locate each operation at the correct level within the class hierarchy. A result of this refinement is often fewer, more powerful operations that are nevertheless simpler to specify than the original operations, because they are more uniform and general.

ATM example. The ATM example does not require simplification. Figure 13.12 adds some operations to the ATM domain class model from Chapter 12.

13.5 Chapter Summary

The purpose of analysis is to understand the problem 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.

There are two phases of analysis—domain and application. Domain analysis captures general knowledge about an application. Domain analysis involves class models and sometimes state models but seldom has an interaction model. In contrast, application analysis focuses on major application artifacts that are important, visible to users, and must be approved by them. The interaction model dominates application analysis, but the class and state models are also important.

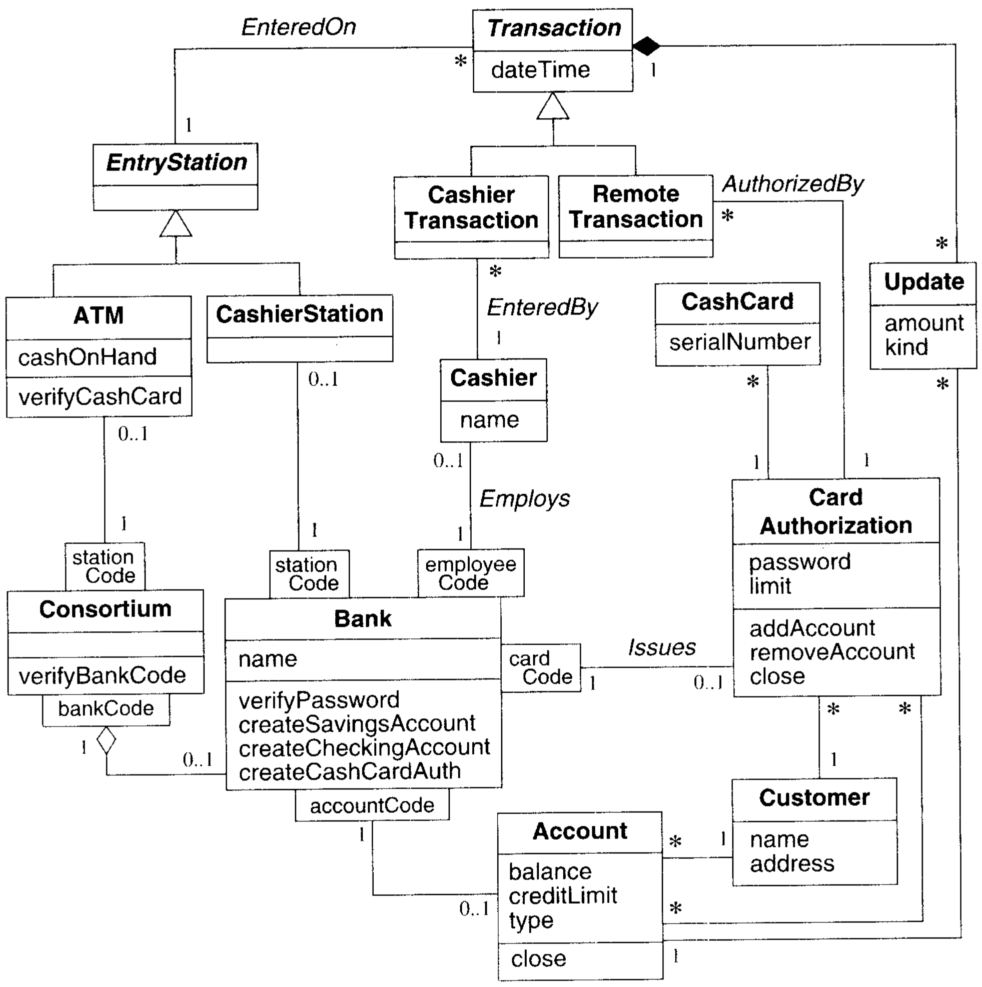


Figure 13.12 ATM domain class model with some operations

The application interaction model shows the interactions between the system and the outside world. First determine the precise scope—the system boundary. Then, define actors for external objects that communicate directly with the system. Also, define use cases for externally visible functionality. For each use case, make up scenarios for normal cases, variations, extreme cases, and exceptions. You can clarify complex use cases with activity diagrams and organize the use cases and actors with relationships. Finally, check the use cases against the domain class model to ensure that there are no inconsistencies.

Next augment the domain classes with application classes. Application classes arise from user interfaces, boundary classes, and controllers. Carefully check the use cases and scenarios to find them.

The last phase of application analysis is to build an application state model. This state model tends to be richer and reveals more behavior than does the domain state model. First identify application classes with multiple states and study the interaction scenarios to find events for these classes. The most difficult aspect is to reconcile the various scenarios and detect overlap and closure of loops. As you complete the state model, check the state diagrams for consistency with each other, as well as the class and interaction models.

We emphasized the need for abstraction during domain analysis, and it is also important for application analysis. Try to think expansively as you construct your models. Do not commit an application to arbitrary business practices that may change over time. Instead, try to build in flexibility that will anticipate and accommodate future changes.

activity diagram	controller
actor	scenario
application analysis	sequence diagram
boundary class	shopping-list operation
building the application class model	system boundary
building the application interaction model	use case
building the application state model	user interface

Figure 13.13 Key concepts for Chapter 13

Bibliographic Notes

Meyer [Meyer-97] provides many useful insights into the principles underlying a good design. He advocates the use of data-directed bottom-up design, discovery of "shopping-list operations," and the lack of any "main program" in a system. He makes effective use of assertions, preconditions, and postconditions for specifying operations.

References

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

Exercises

13.1 (4) Prepare scenarios for the variations and exception bullets in Section 13.1.6.

13.2 (6) Complete the *Deposit*, *Withdrawal*, and *Query* subdiagrams from Figure 13.10.

13.3 (4) Figure E13.1 is a class diagram for Exercise 11.6a. *Sender* and *Receiver* are the only classes with important temporal behavior. Construct a sequence diagram for the following scenario: Sender tries to establish a connection to the receiver by sending a start-of-transaction packet.

The receiver successfully reads the packet and replies with an acknowledgment. The sender then transmits a start-of-file packet, which is acknowledged. Then, the file data is transmitted in three acknowledged packets, followed by end of file and end of transaction, which are also acknowledged.

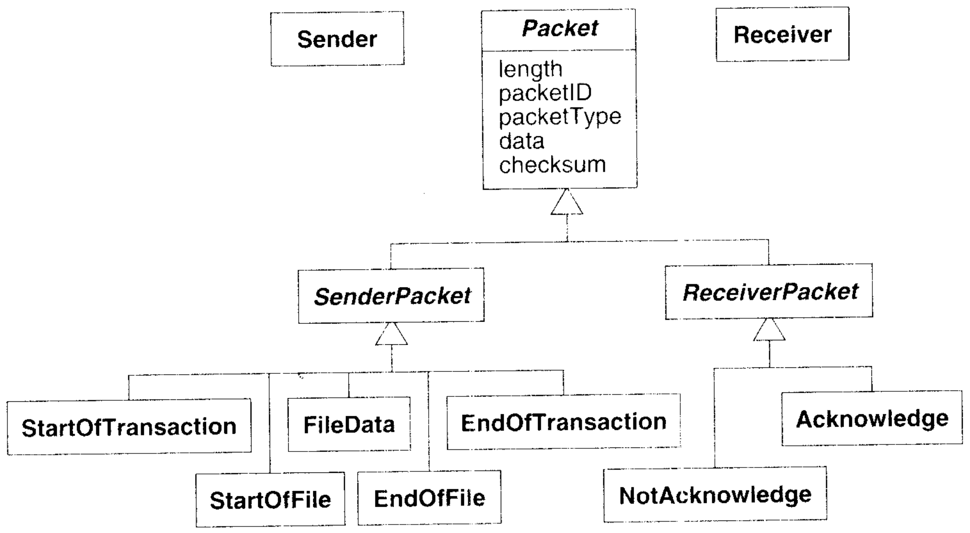


Figure E13.1 A class diagram for a file transfer system

13.4 (3) Prepare additional sequence diagrams for the previous example to include errors caused by noise corruption of each type of sender packet. Revise your previous answer.

13.5 (5) Prepare a state diagram for a file transfer system from the sequence diagrams prepared in Exercises 13.3 and 13.4.

13.6 (8) Prepare a state diagram for a bike odometer from the given scenarios.

- The user turns on the odometer on a bike that is moving.
  The odometer displays the current time. The user presses the mode button.
  The odometer displays the distance biked today. The user presses the mode button.
  The odometer displays the high speed since reset. The user presses the mode button.
  The odometer displays the riding time since reset. The user presses the mode button.
  The odometer displays the distance since reset. The user presses the mode button.
  The odometer displays the average speed since reset. The user presses the mode button.
  The odometer displays the current time...
- The user turns on the odometer on a bike that is stationary.
  The odometer displays the current time. The user presses the mode button.
  The odometer displays the total distance biked. The user presses the mode button.
  The odometer displays the total time biked. The user presses the mode button.
  The odometer displays the distance biked today. The user presses the mode button.
  The odometer displays the high speed since reset. The user presses the mode button.
  The odometer displays the riding time since reset. The user presses the mode button.
  The odometer displays the distance since reset. The user presses the mode button.

The odometer displays the average speed since reset. The user presses the mode button.
The odometer displays the current time...

- The odometer displays the distance biked today.
  The clock rolls over past midnight and begins a new day.
  The odometer sets the distance biked today to zero.
- The user stops biking.
  Four minutes elapse.
  The odometer display dims.
  The user presses the mode button.
  The odometer lights up.
- The user holds the mode button.
  The odometer sets all variables computed since reset to zero.

Consider the simple diagram editor from Exercises 12.3–12.8.

13.7 (2) Describe the system boundary for this application in a few sentences.

13.8 (2) Identify two actors for the application.

13.9 (4) List at least four use cases and define them with a one- or two-sentence bullet. Construct a use case diagram.

13.10 (6) Organize commonality in the use cases with use case relationships. You can create new use cases for common behavior. (Instructor's note: You should give the students the answer to the previous exercise.)

13.11 (4) Prepare a normal scenario for making the drawing in Figure E12.1. Include at least ten editor operations from the problem description in Chapter 12. Do not worry about error conditions.

13.12 (3) Prepare three error scenarios, starting from the previous exercise.

13.13 (4) Prepare sequence diagrams for the scenarios you prepared in the previous exercise.

Consider the computerized scoring system from Exercises 12.9–12.13.

13.14 (2) Describe the system boundary for this application in a few sentences.

13.15 (2) Identify four actors for the application.

13.16 (5) Here are some use cases: register child, schedule meet, schedule season, score figure, judge figure, and compute statistics. Define each one with a one- or two-sentence bullet. Construct a use case diagram.

13.17 (3) Prepare a scenario for setting up the scoring system at the beginning of a season. Enter data on teams, competitors, and judges. Prepare a schedule of meets for the season and select events for each meet. Enter difficulty factors for figures. Include at least 2 teams, 6 competitors, 3 judges, 3 meets, and 12 events. Do not worry about error conditions.

13.18 (3) Prepare three error scenarios, starting from Exercise 13.17.

13.19 (3) Prepare a scenario for printing and processing preregistration forms for the scoring system. In the scenario two children should change their address and another two children should indicate that they are unable to attend. Assign a number to each contestant.

13.20 (6) Prepare an activity diagram for the following computation. Show swim lanes for competitor, computer operator, judge, and scorekeeper.

The computer operator calls the competitor's number as it appears on the display. The competitor verifies her number and then performs the figure. The three judges hold up their scores. A scorekeeper reads the scores. As they are read, the computer operator enters them into the computer.

13.21 (3) Prepare a shopping list of operations for the scoring system and place them in a class diagram.

13.22 (5) For each method listed in the previous exercise, summarize what the method should do.

14 System Design

After you have analyzed a problem, you must decide how to approach the design. During system design you devise the high-level strategy—the ***system architecture***—for solving the problem and building a solution. You make decisions about the organization of the system into subsystems, the allocation of subsystems to hardware and software, and major policy decisions that form the basis for class design.

In this chapter you will learn about the many aspects that you should consider when formulating a system design. We also list several common architectural styles that you can use as a starting point. This list is not meant to be complete; new architectures can always be invented. The treatment in this chapter is intended for small to medium software development efforts; large complex systems, involving more than about ten developers, are limited by human communication issues and require a much greater emphasis on logistics. Most of the suggestions in this chapter are suitable for non-OO as well as OO systems.

14.1 Overview of System Design

During analysis, the focus is on *what* needs to be done, independent of *how* it is done. During design, developers make decisions about how the problem will be solved, first at a high level and then with more detail.

System design is the first design stage for devising the basic approach to solving the problem. During system design, developers decide the overall structure and style. The system architecture determines the organization of the system into subsystems. In addition, the architecture provides the context for the detailed decisions that are made in later stages. You must make the following decisions.

- Estimate system performance. [14.2]
- Make a reuse plan. [14.3]

- Organize the system into subsystems. [14.4]
- Identify concurrency inherent in the problem. [14.5]
- Allocate subsystems to hardware. [14.6]
- Manage data stores. [14.7]
- Handle global resources. [14.8]
- Choose a software control strategy. [14.9]
- Handle boundary conditions. [14.10]
- Set trade-off priorities. [14.11]
- Select an architectural style. [14.12]

You can often choose the architecture of a system by analogy to previous systems. Certain kinds of architecture pertain to broad classes of problems. Section 14.12 surveys several common architectures and describes their corresponding problems. Not all problems can be solved by one of these architectures, but many can. You can construct additional architectures by combining these forms.

14.2 Estimating Performance

Early in the planning for a new system you should prepare a rough performance estimate. Engineers call this a "back of the envelope" calculation. The purpose is not to achieve high accuracy, but merely to determine if the system is feasible. Getting within a factor of two is usually sufficient, although what you can achieve depends on the problem. The calculation should be fast and involve common sense. You will have to make simplifying assumptions. Don't worry about details—just approximate, estimate, and guess, if necessary.

ATM example. Suppose we are planning an ATM network for a bank. We might proceed as follows. The bank has 40 branches. Suppose there are an equal number of terminals in supermarkets and other stores. Suppose on a busy day half the terminals are busy at once. (We could assume all of the terminals are busy without changing the results much. The point is to establish reasonable performance limits.) Suppose that each customer takes one minute to perform a session, and that most transactions involve a single deposit or withdrawal. So we estimate a peak requirement of about 40 transactions a minute, or about one per second. This may not be precise, but it shows that we do not require unusually fast computer hardware. The situation would be much different if we were estimating for an online bookseller or stockbroker, in which case the computer hardware would become a big issue.

You can perform similar estimates for data storage. Count the number of customers, estimate the amount of data for each one, and multiply. In the case of a bank, the requirements for data storage are more severe than for ATM computing power, but they are hardly enormous. Again, the situation would be different for a satellite-based ground imaging system, in which both data storage and access bandwidth would be key architectural issues.

14.3 Making a Reuse Plan

Reuse is often cited as an advantage of OO technology, but reuse does not happen automatically. There are two very different aspects of reuse—using existing things and creating reusable new things. It is much easier to reuse existing things than to design new things for uncertain uses to come. Of course, someone must have designed things in the past in order for us to reuse them now. The point is that most developers reuse existing things, and only a small fraction of developers create new things. Don't feel that you should start with OO technology by building reusable things—that takes a great deal of experience.

Reusable things include models, libraries, frameworks, and patterns. Reuse of models is often the most practical form of reuse. The logic in a model can apply to multiple problems.

14.3.1 Libraries

A ***library*** is a collection of classes that are useful in many contexts. The collection of classes must be carefully organized, so that users can find them. Good organization takes a lot of work, and it can be difficult to decide where to place everything. Online searching can help, but is no substitute for careful organization. In addition, the classes must have accurate and thorough descriptions to help users determine their relevance. [Korson-92] notes several qualities of "good" class libraries.

- **Coherence**. A class library should be organized about a few, well-focused themes.
- **Completeness**. A class library should provide complete behavior for the chosen themes.
- **Consistency**. Polymorphic operations should have consistent names and signatures across classes.
- **Efficiency**. A library should provide alternative implementations of algorithms (such as various sort algorithms) that trade time and space.
- **Extensibility**. The user should be able to define subclasses for library classes.
- **Genericity**. A library should use parameterized class definitions where appropriate.

Unfortunately, problems can arise when integrating class libraries from multiple sources, as shown below [Berlin-90]. Developers often disperse pragmatic decisions across classes and inheritance hierarchies. Class libraries may adopt policies that are individually sensible, but fundamentally incompatible with those of other class libraries. You cannot fix such pragmatic inconsistencies by specializing a class or adding code. Instead, you must break encapsulation and rework the source code. These problems are so severe that they will effectively limit your ability to reuse code from class libraries.

- **Argument validation**. An application may validate arguments as a collection or individually as entered. Collective validation is appropriate for command interfaces; the user enters all arguments, and only then are they checked. In contrast, responsive user interfaces validate each argument or interdependent group of arguments as it is entered. A combination of class libraries, some that validate by collection and others that validate by individual, would yield an awkward user interface.

- **Error handling.** Class libraries use different error-handling techniques. Methods in one library may return error codes to the calling routine, for example, while methods in another library may directly deal with errors.
- **Control paradigms.** Applications may adopt event-driven or procedure-driven control. With event-driven control the user interface invokes application methods. With procedure-driven control the application calls user interface methods. It is difficult to combine both kinds of user interface within an application.
- **Group operations.** Group operations are often inefficient and incomplete. For example, an object-delete primitive may acquire database locks, make the deletion, and then commit the transaction. If you want to delete a group of objects as a transaction, the class library must have a group-delete function.
- **Garbage collection.** Class libraries use different strategies to manage memory allocation and avoid memory leaks. A library may manage memory for strings by returning a pointer to the actual string, returning a copy of the string, or returning a pointer with read-only access. Garbage collection strategies may also differ: mark and sweep, reference counting, or letting the application handle garbage collection (in C++, for example).
- **Name collisions.** Class names, public attributes, and public methods lie within a global name space, so you must hope they do not collide for different class libraries. Most class libraries add a distinguishing prefix to names to reduce the likelihood of collisions.

14.3.2 Frameworks

A ***framework*** [Johnson-88] is a skeletal structure of a program that must be elaborated to build a complete application. This elaboration often consists of specializing abstract classes with behavior specific to an individual application. A class library may accompany a framework, so that the user can perform much of the specialization by choosing the appropriate subclasses rather than programming subclass behavior from scratch. Frameworks consist of more than just the classes involved and include a paradigm for flow of control and shared invariants. Frameworks tend to be specific to a category of applications; framework class libraries are typically application specific and not suitable for general use.

14.3.3 Patterns

A ***pattern*** is a proven solution to a general problem. Various patterns target different phases of the software development lifecycle. There are patterns for analysis, architecture, design, and implementation. You can achieve reuse by using existing patterns, rather than reinventing solutions from scratch. A pattern comes with guidelines on when to use it, as well as trade-offs on its use.

There are many benefits of patterns. One advantage is that a pattern has been carefully considered by others and has already been applied to past problems. Consequently, a pattern is more likely to be correct and robust than an untested, custom solution. Also when you use patterns, you tap into a language that is familiar to many developers. A body of literature is available that documents patterns, explaining their subtleties and nuances. You can regard patterns as extending a modeling language—you need not think only in terms of primitives; you can also think in terms of recurring combinations. Patterns are prototypical model fragments that distill some of the knowledge of experts.

A pattern is different from a framework. A pattern is typically a small number of classes and relationships. In contrast, a framework is much broader in scope (typically at least an order of magnitude larger) and covers an entire subsystem or application.

ATM example. The notion of a transaction offers some possibility of reuse—transactions are a frequent occurrence in computer systems, and there is commercial software to support them. There may also be an opportunity for reuse with the communications infrastructure that connects the consortium to ATMs and bank computers.

14.4 Breaking a System into Subsystems

For all but the smallest applications, the first step in system design is to divide the system into pieces. Each major piece of a system is called a subsystem. Each subsystem is based on some common theme, such as similar functionality, the same physical location, or execution on the same kind of hardware. For example, a spaceship computer might include subsystems for life support, navigation, engine control, and running scientific experiments.

A ***subsystem*** is not an object nor a function but a group of classes, associations, operations, events, and constraints that are interrelated and have a well-defined and (hopefully) small interface with other subsystems. A subsystem is usually identified by the services it provides. A ***service*** is a gro[illegible] related functions that share some common purpose, such as [illegible]essing I/O, drawin[illegible] or performing arithmetic. A subsystem defines a coherent [illegible]tem. For example, the file system within an operating system is a subsystem. It comprises a set of related abstractions that are largely independent of abstractions in other subsystems, such as memory management and process control.

Each subsystem has a well-defined interface to the rest of the system. The interface specifies the form of all interactions and the information flow across subsystem boundaries but does not specify how the subsystem is implemented internally. Each subsystem can then be designed independently without affecting the others.

You should define subsystems so that most interactions are internal, rather than across subsystem boundaries. This reduces the dependencies among subsystems. A system should be divided into a small number of subsystems; 20 is probably too many. Each subsystem may in turn be decomposed into smaller subsystems of its own.

The relationship between two subsystems can be client-server or peer-to-peer. In a ***client-server relationship***, the client calls on the server, which performs some service and replies with a result. The client must know the server's interface, but the server need not know its clients' interfaces because clients initiate all interactions.

In a ***peer-to-peer relationship***, each subsystem may call on the others. A communication from one subsystem to another is not necessarily followed by an immediate response. Peer-to-peer interactions are more complicated, because the subsystems must know each other's interfaces. Communications cycles can occur that are hard to understand and liable to subtle design errors. Look for client-server decompositions whenever possible, because a one-way interaction is much easier to build, understand, and change than a two-way interaction.

The decomposition of systems into subsystems may be organized as a sequence of horizontal layers or vertical partitions.

14.4.1 Layers

A ***layered system*** is an ordered set of virtual worlds (a set of *tiers*), each built in terms of the ones below it and providing the implementation basis for the ones above it. The objects in each layer can be independent, although there is often some correspondence between objects in different layers. Knowledge is one-way only—a subsystem knows about the layers below it, but has no knowledge of the layers above it. A client-server relationship exists between upper layers (users of services) and lower layers (providers of services).

In an interactive graphics system, for example, windows are made from screen operations, which are implemented using pixel operations, which execute as device I/O operations. Each layer may have its own set of classes and operations. Each layer is implemented in terms of the classes and operations of lower layers.

Layered architectures come in two forms: closed and open. In a ***closed architecture***, each layer is built only in terms of the immediate lower layer. This reduces the dependencies between layers and allows changes to be made most easily, because a layer's interface affects only the next layer. In an ***open architecture***, a layer can use features of any lower layer to any depth. This reduces the need to redefine operations at each level, which can result in a more efficient and compact code. However, an open architecture does not observe the principle of information hiding. Changes to a subsystem can affect any higher subsystem, so an open architecture is less robust than a closed architecture. Both kinds of architectures are useful; the designer must weigh the relative value of efficiency and modularity.

Usually the problem statement specifies only the top and bottom layers: The top is the desired system and the bottom is the available resources (hardware, operating system, existing libraries). If the disparity between the two is too great (as it often is), then you must introduce intermediate layers to reduce the conceptual gap between adjoining layers.

You can port a system constructed in layers to other hardware/software platforms by rewriting one layer. It is a good practice to introduce at least one layer of abstraction between the application and any services provided by the operating system or hardware. Define a layer of interface classes providing logical services and map them onto the concrete services that are system dependent.

14.4.2 Partitions

Partitions vertically divide a system into several independent or weakly coupled subsystems, each providing one kind of service. For example, a computer operating system includes a file system, process control, virtual memory management, and device control. The subsystems may have some knowledge of each other, but this knowledge is not deep and avoids major design dependencies.

One difference between layers and partitions is that layers vary in their level of abstraction. In contrast, partitions merely divide a system into pieces, all of which have a similar level of abstraction. Another difference is that layers ultimately depend on each other, usually in a client-server relationship through an open or closed architecture. In contrast, partitions are peers that are independent or mutually dependent (peer-to-peer relationship).

14.4.3 Combining Layers and Partitions

You can decompose a system into subsystems by combining layers and partitions. Layers can be partitioned, and partitions can be layered. Figure 14.1 shows a block diagram of a typical application, which involves simulation and interactive graphics. Most large systems require a mixture of layers and partitions.

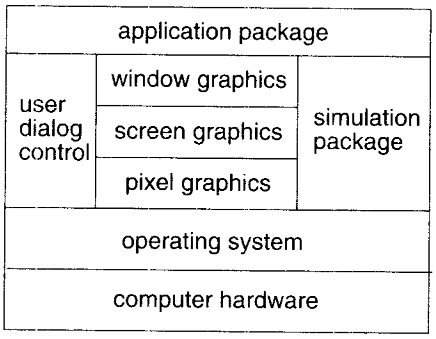


Figure 14.1 Block diagram of a typical application. Most large systems mix layers and partitions.

Once you have identified the top-level subsystems, you should show their information flow. Sometimes, all subsystems interact with all other subsystems, but often the flow is simpler. For example, many computations have the form of a pipeline; a compiler is an example. Other systems are arranged as a star, in which a master subsystem controls all interactions with other subsystems. Use simple topologies when possible to reduce the number of interactions among subsystems.

ATM example. Figure 14.2 shows the architecture of the ATM system. There are three major subsystems: the ATM stations, the consortium computer, and the bank computers. The topology is a simple star; the consortium computer communicates with all the ATM stations and with all the bank computers (comm links). The architecture uses the station code and the bank code to distinguish the phone lines to the consortium computer.

14.5 Identifying Concurrency

In the analysis model, as in the real world and in hardware, all objects are concurrent. In an implementation, however, not all software objects are concurrent, because one processor

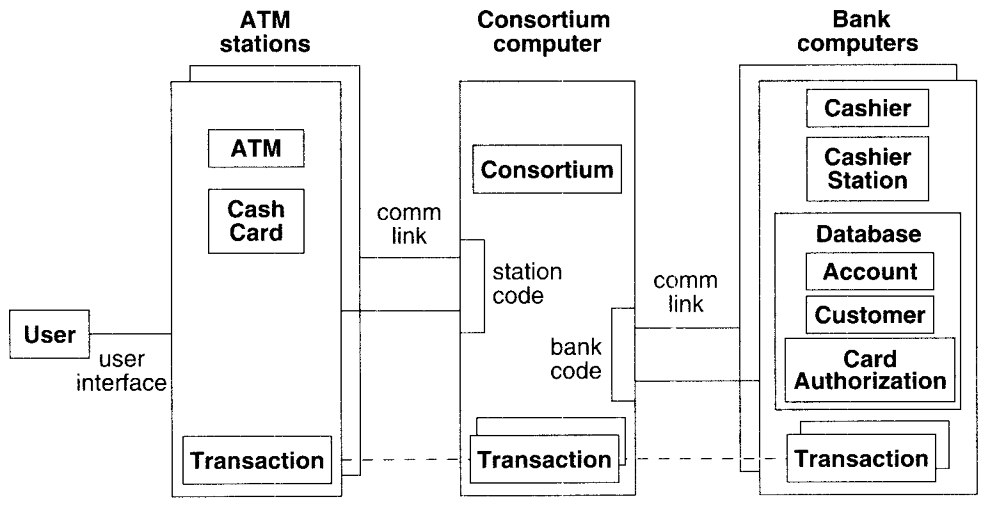


Figure 14.2 Architecture of ATM system. It is often helpful to make an informal diagram showing the organization of a system into subsystems.

may support many objects. In practice, you can implement many objects on a single processor if the objects cannot be active together. One important goal of system design is to identify the objects that must be active concurrently and the objects that have mutually exclusive activity. You can fold the latter objects onto a single thread of control, or task.

14.5.1 Identifying Inherent Concurrency

The state model is the guide to identifying concurrency. Two objects are inherently concurrent if they can receive events at the same time without interacting. If the events are unsynchronized, you cannot fold the objects onto a single thread of control. For example, the engine and the wing controls on an airplane must operate concurrently (if not completely independently). Independent subsystems are desirable, because you can assign them to different hardware units without any communication cost.

You need not implement two subsystems that are inherently concurrent as separate hardware units. The purpose of hardware interrupts, operating systems, and tasking mechanisms is to simulate logical concurrency in a uniprocessor. Separate sensors must, of course, process physically concurrent input, but if there are no timing constraints on response, then a multitasking operating system can handle the computation. Often the problem statement specifies that distinct hardware units must implement the objects.

ATM example. If the ATM statement from Chapter 11 contained the requirement that each machine should continue to operate locally in the event of a central system failure (perhaps with reduced transaction limits), then we would have no choice but to include a CPU in each ATM machine with a full control program.

14.5.2 Defining Concurrent Tasks

Although all objects are conceptually concurrent, in practice many objects in a system are interdependent. By examining the state diagrams of individual objects and the exchange of events among them, you can often fold many objects onto a single thread of control. A ***thread of control*** is a path through a set of state diagrams on which only a single object at a time is active. A thread remains within a state diagram until an object sends an event to another object and waits for another event. The thread passes to the receiver of the event until it eventually returns to the original object. The thread splits if the object sends an event and continues executing.

On each thread of control, only a single object at a time is active. You can implement threads of control as tasks in computer systems.

ATM example. While the bank is verifying an account or processing a bank transaction, the ATM machine is idle. If a central computer directly controls the ATM, we can combine the ATM object with the bank transaction object as a single task.

14.6 Allocation of Subsystems

You must allocate each concurrent subsystem to a hardware unit, either a general-purpose processor or a specialized functional unit as follows.

- Estimate performance needs and the resources needed to satisfy them.
- Choose hardware or software implementation for subsystems.
- Allocate software subsystems to processors to satisfy performance needs and minimize interprocessor communication.
- Determine the connectivity of the physical units that implement the subsystems.

14.6.1 Estimating Hardware Resource Requirements

The decision to use multiple processors or hardware functional units is based on a need for higher performance than a single CPU can provide. The number of processors required depends on the volume of computations and the speed of the machine. For example, a military radar system generates too much data in too short a time to handle in a single CPU, even a very large one. Many parallel machines must digest the data before analyzing a threat.

The system designer must estimate the required CPU processing power by computing the steady-state load as the product of the number of transactions per second and the time required to process a transaction. The estimate will usually be imprecise. Often some experimentation is useful. You should increase the estimate to allow for transient effects, due to random variations in load as well as to synchronized bursts of activity. The amount of excess capacity needed depends on the acceptable rate of failure due to insufficient resources. Both the steady-state load and the peak load are important.

ATM example. The ATM machine itself is relatively simple—all it must do is to provide a user interface and, possibly some local processing. At most a single CPU would suffice for each ATM. The consortium computer is essentially just a routing machine—it receives ATM requests and dispatches them to the appropriate bank computer. A large network might need to be partitioned in some way and involve multiple CPUs, so that the consortium computer does not become a bottleneck. The bank computers perform data processing and involve relatively straightforward database applications. The database vendors have single-processor and multiprocessor versions of their products, and the appropriate choice depends on the needed throughput and reliability.

14.6.2 Making Hardware-Software Trade-offs

Object orientation provides a good way for thinking about hardware. Each device is an object that operates concurrently with other objects (other devices or software). You must decide which subsystems will be implemented in hardware and which in software. There are two main reasons for implementing subsystems in hardware.

- **Cost**. Existing hardware provides exactly the functionality required. Today it is easier to buy a floating-point chip than to implement floating point in software. Sensors and actuators must be hardware, of course.
- **Performance**. The system requires a higher performance than a general-purpose CPU can provide, and more efficient hardware is available. For example, chips that perform the fast Fourier transform (FFT) are widely used in signal-processing applications.

Much of the difficulty of designing a system comes from meeting externally imposed hardware and software constraints. OO design provides no magic solution, but the external packages can be modeled nicely as objects. You must consider compatibility, cost, and performance issues. You should also think about flexibility for future changes, both design changes and future product enhancements. Providing flexibility costs something; the architect must decide how much it is worth.

ATM example. There are no pressing performance issues for the ATM application. Hence general-purpose computers should suffice for the ATMs, consortium, and banks.

14.6.3 Allocating Tasks to Processors

The system design must allocate tasks for the various software subsystems to processors. There are several reasons for assigning tasks to processors.

- **Logistics**. Certain tasks are required at specific physical locations, to control hardware, or to permit independent operation. For example, an engineering workstation needs its own operating system to permit operation when the interprocessor network is down.
- **Communication limits**. The response time or data flow rate exceeds the available communication bandwidth between a task and a piece of hardware. For example, high performance graphics devices require tightly coupled controllers because of their high internal data generation rates.
- **Computation limits**. Computation rates are too great for a single processor, so several processors must support the tasks. You can minimize communication costs by assigning highly interactive subsystems to the same processor. You should assign independent subsystems to separate processors.

ATM example. The ATM does not have any issues with communication and computation limits. The communication traffic and computation that an ATM user initiates are relatively minor. However, there may be an issue with logistics. If the ATM must have autonomy and operate when the communications network is down, then it must have its own CPU and programming. Otherwise, if the ATM is just a dumb terminal that accesses the network and performs all computation via the network, we can simplify ATM logic.

14.6.4 Determining Physical Connectivity

After determining the kinds and relative numbers of physical units, you must determine the arrangement and form of the connections among the physical units.

- **Connection topology**. Choose the topology for connecting the physical units. Associations in the class model often correspond to physical connections. Client-server relationships also correspond to physical connections. Some connections may be indirect; you should try to minimize the connection cost of important relationships.
- **Repeated units**. Choose the topology of repeated units. If you have boosted performance by including several copies of a particular kind of unit or group of units, you must specify their topology. The class model is not a useful guide, because the use of multiple units is primarily a design optimization not required by analysis. The topology of repeated units usually has a regular pattern, such as a linear sequence, a matrix, a tree, or a star. You must consider the expected arrival patterns of data and the proposed parallel algorithm for processing it.
- **Communications**. Choose the form of the connection channels and the communication protocols. The system design phase may be too soon to specify the exact interfaces among units, but often it is appropriate to choose the general interaction mechanisms and protocols. For example, interactions may be asynchronous, synchronous, or blocking. You must estimate the bandwidth and latency of the communication channels and choose the correct kind of connection channels.

Even when the connections are logical and not physical, you must consider them. For example, the units may be tasks within a single operating system connected by interprocess communication (IPC) calls. On most operating systems, such IPC calls are much slower than subroutine calls within the same program and may be impractical for certain time-critical connections. In that case, you must combine the tightly linked tasks into a single task and make the connections by simple subroutine calls.

ATM example. Figure 14.2 summarizes physical connectivity. Multiple ATMs connect to the consortium computer and then are routed to the appropriate bank computer. The topology is a star where the consortium computer mediates communication.

14.7 Management of Data Storage

There are several alternatives for data storage that you can use separately or in combination: data structures, files, and databases. Different kinds of data stores provide trade-offs among cost, access time, capacity, and reliability. For example, a personal computer application may use memory data structures and files. An accounting system may use a database to connect subsystems.

Files are cheap, simple, and permanent. However, file operations are low level, and applications must include additional code to provide a suitable level of abstraction. File implementations vary for different computer systems, so portable applications must carefully isolate file-system dependencies. Implementations for sequential files are mostly standard, but commands and storage formats for random-access files and indexed files vary. Figure 14.3 characterizes the kind of data that belongs in files.

- Data with high volume and low information density (such as archival files or historical records).
- Modest quantities of data with simple structure.
- Data that are accessed sequentially.
- Data that can be fully read into memory.

Figure 14.3 Data suitable for files. Files provide a low-tech solution to data management and should not be overlooked.

Databases, managed by database management systems (DBMSs), are another kind of data store. Various types of DBMSs are available from vendors, including relational and OO. DBMSs cache frequently accessed data in memory in order to achieve the best combination of cost and performance from memory and disk storage. Databases make applications easier to port to different hardware and operating system platforms, since the vendor ports the DBMS code. One disadvantage of DBMSs is their complex interface—many database languages integrate awkwardly with programming languages. Figure 14.4 characterizes the kinds of data that belong in a database.

- Data that require updates at fine levels of detail by multiple users.
- Data that must be accessed by multiple application programs.
- Data that require coordinated updates via transactions.
- Large quantities of data that must be handled efficiently.
- Data that are long-lived and highly valuable to an organization.
- Data that must be secured against unauthorized and malicious access.

Figure 14.4 Data suitable for databases. Databases provide heavyweight data management and are used for most important business applications.

OO-DBMSs have not become popular in the mass market. Consequently you should consider them only for specialty applications that have a wide variety of data types or that must access low-level data management primitives. These applications include engineering applications, multimedia applications, knowledge bases, and electronic devices with embedded software. For most applications that need a database, you should use a relational DBMS (RDBMS). RDBMSs dominate the marketplace, and their features are sufficient for most applications. RDBMSs can also provide a very good implementation of an OO model, *if* they are used properly—Chapter 19 presents the details.

ATM example. The typical bank computer would use a relational DBMS—they are fast, readily available, and cost-effective for these kinds of financial applications.

The ATM might also use a database, but the paradigm for that is less obvious. Relational and OO-DBMSs would both be possibilities. Many OO-DBMSs permit access to low-level primitives, and a stripped-down database might enable mass production of ATM software at a low cost. A stripped-down database might also simplify ATM operation. Alternatively, RDBMSs are mature products with many features that might reduce development effort.

14.8 Handling Global Resources

The system designer must identify global resources and determine mechanisms for controlling access to them. There are several kinds of global resources.

- **Physical units**. Examples include processors, tape drives, and communication satellites.
- **Space**. Examples include disk space, a workstation screen, and the buttons on a mouse.
- **Logical names**. Examples include object IDs, filenames, and class names.
- **Access to shared data**. Databases are an example.

If the resource is a physical object, then it can control itself by establishing a protocol for obtaining access. If the resource is a logical entity, such as an object ID or a database, then there is danger of conflicting access in a shared environment. Independent tasks could simultaneously use the same object ID, for example.

You can avoid conflict by having a "guardian object" own each global resource and control access to it. One guardian object can control several resources. All access to the resource must pass through the guardian object. Allocating each shared global resource to a single object is a recognition that the resource has identity.

You can also partition a resource logically, assigning subsets to different guardian objects for independent control. For example, one strategy for object ID generation in a parallel distributed environment is to preallocate a range of possible IDs to each processor in a network; each processor allocates the IDs within its preallocated range without the need for global synchronization.

In a time-critical application, the cost of passing all access to a resource through a guardian object is sometimes too high, and clients must access the resource directly. In this case, locks can be placed on subsets of the resource. A ***lock*** is a logical object associated with some defined subset of a resource that gives the lock holder the right to access the resource directly. A guardian object must still exist to allocate the locks, but after one interaction with the guardian to obtain a lock the user of the resource can access the resource directly. This approach is more dangerous, because each resource user must be trusted to behave itself in its access to the resource. Do not use direct access to shared resources unless it is absolutely necessary.

ATM example. Bank codes and account numbers are global resources. Bank codes must be unique within the context of a consortium. Account codes must be unique within the context of a bank.

14.9 Choosing a Software Control Strategy

The analysis model shows interactions as events between objects. Hardware control closely matches the analysis model, but there are several ways for implementing control in software. Although all subsystems need not use the same implementation, it is best to choose a single control style for the whole system. There are two kinds of control flows in a software system: external control and internal control.

External control concerns the flow of externally visible events among the objects in the system. There are three kinds of control for external events: procedure-driven sequential, event-driven sequential, and concurrent. The appropriate control style depends on the available resources (language, operating system) and on the kind of interactions in the application.

Internal control refers to the flow of control within a process. It exists only in the implementation and therefore is neither inherently concurrent nor sequential. The designer may choose to decompose a process into several tasks for logical clarity or for performance (if multiple processors are available). Unlike external events, internal transfers of control, such as procedure calls or intertask calls, are under the direction of the program and can be structured for convenience. Three kinds of control flow are common: procedure calls, quasi-concurrent intertask calls, and concurrent intertask calls. Quasi-concurrent intertask calls, such as coroutines or lightweight processes, are programming conveniences in which multiple address spaces or call stacks exist but only a single thread of control can be active at once.

14.9.1 Procedure-driven Control

In a procedure-driven sequential system, control resides within the program code. Procedures request external input and then wait for it; when input arrives, control resumes within the procedure that made the call. The location of the program counter and the stack of procedure calls and local variables define the system state.

The major advantage of procedure-driven control is that it is easy to implement with conventional languages; the disadvantage is that it requires the concurrency inherent in objects to be mapped into a sequential flow of control. The designer must convert events into operations between objects. A typical operation corresponds to a pair of events: an output event that performs output and requests input and an input event that delivers the new values. This paradigm cannot easily accommodate asynchronous input, because the program must explicitly request input. The procedure-driven paradigm is suitable only if the state model shows a regular alternation of input and output events. Flexible user interfaces and control systems are hard to build with this style.

Note that all major OO languages, such as C++ and Java, are procedural languages. Do not be fooled by the OO phrase *message passing*. A message *is* a procedure call with a built-in case statement that depends on the class of the target object. A major drawback of conventional OO languages is that they fail to support the concurrency inherent in objects. Some concurrent OO languages have been designed, but they are not yet widely used.

14.9.2 Event-driven Control

In an event-driven sequential system, control resides within a dispatcher or monitor that the language, subsystem, or operating system provides. Developers attach application procedures to events, and the dispatcher calls the procedures when the corresponding events occur ("callback"). Procedure calls to the dispatcher send output or enable input but do not wait for it in-line. All procedures return control to the dispatcher, rather than retaining control until input arrives. Consequently, the program counter and stack cannot preserve state. Procedures must use global variables to maintain state, or the dispatcher must maintain local state for them. Event-driven control is more difficult to implement with standard languages than procedure-driven control but is often worth the extra effort.

Event-driven systems permit more flexible control than procedure-driven systems. Event-driven systems simulate cooperating processes within a single multithreaded task; an errant procedure can block the entire application, so you must be careful. Event-driven user interface subsystems are particularly useful.

Use an event-driven system for external control in preference to a procedure-driven system whenever possible, because the mapping from events to program constructs is simpler and more powerful. Event-driven systems are also more modular and can handle error conditions better than procedure-driven systems.

14.9.3 Concurrent Control

In a concurrent system, control resides concurrently in several independent objects, each a separate task. Such a system implements events directly as one-way messages (*not* OO language "messages") between objects. A task can wait for input, but other tasks continue execution. The operating system resolves scheduling conflicts among tasks and usually supplies a queuing mechanism, so that events are not lost if a task is executing when they arrive. If there are multiple CPUs, then different tasks can actually execute concurrently.

14.9.4 Internal Control

During design, the developer expands operations on objects into lower-level operations on the same or other objects. Internal object interactions are similar to external object interactions, because you can use the same implementation mechanisms. However, there is an important difference—external interactions inherently involve waiting for events, because objects are independent and cannot force other objects to respond; objects generate internal operations as part of the implementation algorithm, so their form of response is predictable. Consequently, you can think of most internal operations as procedure calls, in which the caller issues a request and waits for the response. There are algorithms for parallel processing, but many computations are well represented sequentially and can easily be folded onto a single thread of control.

14.9.5 Other Paradigms

We assume that the reader is primarily interested in procedural programming, but other paradigms are possible, such as rule-based systems, logic programming systems, and other forms of nonprocedural programs. These constitute another control style in which explicit control is replaced by declarative specification with implicit evaluation rules, possibly nondeterministic or highly convoluted. Developers currently use such languages in limited areas, such as artificial intelligence and knowledge-based programming, but we expect their use to grow in the future. Because these languages are totally different from procedural languages (including OO languages), the remainder of this book has little to say about them.

ATM example. Event-driven control is the appropriate paradigm for the ATM station. The ATM services a single user, so there is little need for concurrent control. The ATM must be responsive in its user interactions, and event-driven control is much better at that than procedure-driven control.

14.10 Handling Boundary Conditions

Although most of system design concerns steady-state behavior, you must consider boundary conditions as well and address the following kinds of issues.

- **Initialization.** The system must proceed from a quiescent initial state to a sustainable steady state. The system must initialize constant data, parameters, global variables, tasks, guardian objects, and possibly the class hierarchy itself. During initialization only a subset of the functionality of the system is usually available. Initializing a system containing concurrent tasks is most difficult, because independent objects must not get either too far ahead or too far behind other independent objects during initialization.
- **Termination.** Termination is usually simpler than initialization, because many internal objects can simply be abandoned. The task must release any external resources that it had reserved. In a concurrent system, one task must notify other tasks of its termination.
- **Failure.** Failure is the unplanned termination of a system. Failure can arise from user errors, from the exhaustion of system resources, or from an external breakdown. The good system designer plans for orderly failure. Failure can also arise from bugs in the system and is often detected as an "impossible" inconsistency. In a perfect design, such errors would never happen, but the good designer plans for a graceful exit on fatal bugs by leaving the remaining environment as clean as possible and recording or printing as much information about the failure as possible before terminating.

14.11 Setting Trade-off Priorities

The system designer must set priorities that will be used to guide trade-offs for the rest of design. These priorities reconcile desirable but incompatible goals. For example, a system can often be made faster by using extra memory, but that increases power consumption and costs more. Design trade-offs involve not only the software itself but also the process of developing it. Sometimes it is necessary to sacrifice complete functionality to get a piece of software into use (or into the marketplace) earlier. Sometimes the problem statement specifies priority, but often the burden falls on the designer to reconcile the incompatible desires of the client and decide how to make trade-offs.

The system designer must determine the relative importance of the various criteria as a guide to making design trade-offs. The system designer does not *make* all the trade-offs, but establishes the priorities for making them. For example, the first video games ran on processors with limited memory. Conserving memory was the highest priority, followed by fast execution. Designers had to use every programming trick in the book, at the expense of maintainability, portability, and understandability. As another example, mathematical subroutine packages run on a wide range of machines. Well-conditioned numerical behavior is crucial to such packages, as well as portability and understandability. These cannot be sacrificed for fast development.

Design trade-offs affect the entire character of a system. The success or failure of the final product may depend on how well its goals are chosen. Even worse, if no system-wide priorities are established, then the various parts of the system may optimize opposing goals ("suboptimization"), resulting in a system that wastes resources. Even on small projects, programmers often forget the real goals and become obsessed with "efficiency" when it is really unimportant.

Setting trade-off priorities is at best vague. You cannot expect numerical accuracy ("speed 53%, memory 31%, portability 15%, cost 1%"). Priorities are rarely absolute; for example, trading memory for speed does not mean that any increase in speed, no matter how small, is worth any increase in memory, no matter how large. We cannot even give a full list of design criteria that might be subject to trade-offs. Instead, the priorities are a statement of design philosophy. Subsequent design will still require judgment and interpretation when trade-offs are actually made.

ATM example. The ATM station is a mass-market product. Consequently, the manufacturing cost is a concern, and the resulting product must have a polished user interface. The software must be robust and resilient in the face of failure. Development cost is a lesser concern, since the cost can be amortized across numerous copies.

14.12 Common Architectural Styles

Several prototypical architectural styles are common in existing systems. Each of these is well suited to a certain kind of system. If you have an application with similar characteristics, you can save effort by using the corresponding architecture, or at least using it as a starting point for your design. Some kinds of systems are listed below.

- **Batch transformation**—a data transformation executed once on an entire input set. [14.12.1]
- **Continuous transformation**—a data transformation performed continuously as inputs change. [14.12.2]
- **Interactive interface**—a system dominated by external interactions. [14.12.3]

- **Dynamic simulation**—a system that simulates evolving real-world objects. [14.12.4]
- **Real-time system**—a system dominated by strict timing constraints. [14.12.5]
- **Transaction manager**—a system concerned with storing and updating data, often including concurrent access from different physical locations. [14.12.6]

This is not meant to be a complete list of known systems and architectures but a list of common forms. Some problems require a new kind of architecture, but most can use an existing style or at least a variation on it. Many problems combine aspects of these architectures.

14.12.1 Batch Transformation

A ***batch transformation*** performs sequential computations. The application receives the inputs, and the goal is to compute an answer; there is no ongoing interaction with the outside world. Examples include standard computational problems such as compilers, payroll processing, VLSI automatic layout, stress analysis of a bridge, and many others. The state model is trivial or nonexistent for batch transformation problems. The class model is important—there are class models for the input, output, and the intervening stages. The interaction model documents the computation and couples the class models. The most important aspect of a batch transformation is to define a clean series of steps.

In the past, when we worked at GE R&D, one of our colleagues (Bill Premerlani) built a compiler that received an ASCII file of graphical pictures as input and generated relational database definition code as output. This work preceded the availability of commercial OO modeling tools. Figure 14.5 shows the sequence of steps. The compiler had five class models—one for the input, one for the output, and three for intermediate representations.

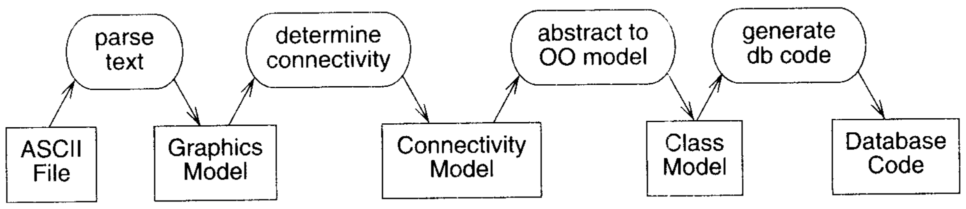


Figure 14.5 Sequence of steps for a compiler. A batch transformation is a sequential input-to-output transformation that does not interact with the outside world.

The steps in designing a batch transformation are as follows.

- Break the overall transformation into stages, with each stage performing one part of the transformation.
- Prepare class models for the input, output, and between each pair of successive stages. Each stage knows only about the models on either side of it.
- Expand each stage in turn until the operations are straightforward to implement.
- Restructure the final pipeline for optimization.

14.12.2 Continuous Transformation

A ***continuous transformation*** is a system in which the outputs actively depend on changing inputs. Unlike a batch transformation that computes the outputs only once, a continuous transformation updates outputs frequently (in theory continuously, although in practice they are computed discretely at a fine time scale). Because of severe time constraints, the system cannot recompute the entire set of outputs each time an input changes (otherwise the application would be a batch transformation). Instead, the system must compute outputs incrementally. Typical applications include signal processing, windowing systems, incremental compilers, and process monitoring systems. The class, state, and interaction models have similar purposes as with the batch transformation.

One way to implement a continuous transformation is with a pipeline of functions. The pipeline propagates the effect of each input change. Developers can define intermediate and redundant objects to improve the performance of the pipeline. Some high-performance systems, such as signal processing, need to synchronize values within the pipeline. Such systems perform operations at well-defined times and carefully balance the flow path of operations so that values arrive at the right place at the right time without bottlenecks.

Figure 14.6 shows the example of a graphics application. The application first maps geometric figures in user-defined coordinates to window coordinates. Then it clips the figures to fit the window bounds. Finally it offsets each figure by its window position to yield its screen position.

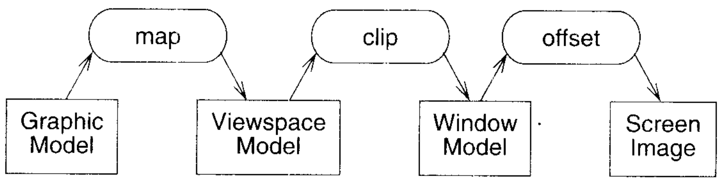


Figure 14.6 Sequence of steps for a graphics application. A continuous transformation repeatedly propagates input changes to the output.

The steps in designing a pipeline for a continuous transformation are as follows.

- Break the overall transformation into stages, with each stage performing one part of the transformation.
- Define input, output, and intermediate models between each pair of successive stages, as for the batch transformation.
- Differentiate each operation to obtain incremental changes to each stage. That is, propagate the incremental effects of each change to an input through the pipeline as a series of incremental updates.
- Add additional intermediate objects for optimization.

14.12.3 Interactive Interface

An ***interactive interface*** is a system that is dominated by interactions between the system and external agents, such as humans or devices. The external agents are independent of the system, so the system cannot control the agents, although it may solicit responses from them. An interactive interface usually includes only part of an entire application, one that can often be handled independently from computations. Examples of interactive systems include a forms-based query interface, a workstation windowing system, and the control panel for a simulation.

The major concerns of an interactive interface are the communications protocol between the system and the external agents, the syntax of possible interactions, the presentation of output (the appearance on the screen, for instance), the flow of control within the system, performance, and error handling. Interactive interfaces are dominated by the state model. The class model represents interaction elements, such as input and output tokens and presentation formats. The interaction model shows how the state diagrams interact.

The steps in designing an interactive interface are as follows.

- Isolate interface classes from the application classes.
- Use predefined classes to interact with external agents, if possible. For example, windowing systems have extensive collections of predefined windows, menus, buttons, forms, and other kinds of classes ready to be adapted to applications.
- Use the state model as the structure of the program. Interactive interfaces are best implemented using concurrent control (multitasking) or event-driven control (interrupts or call-backs). Procedure-driven control (writing output and then waiting for input in-line) is awkward for anything but rigid control sequences.
- Isolate physical events from logical events. Often a logical event corresponds to multiple physical events. For example, a graphical interface can take input from a form, from a pop-up menu, from a function button on the keyboard, from a typed-in command sequence, or from an indirect command file.
- Fully specify the application functions that are invoked by the interface. Make sure that the information to implement them is present.

14.12.4 Dynamic Simulation

A ***dynamic simulation*** models or tracks real-world objects. Examples include molecular motion modeling, spacecraft trajectory computation, economic models, and video games. Simulations are perhaps the simplest system to design using an OO approach. The objects and operations come directly from the application. There are two ways for implementing control: an explicit controller external to the application objects can simulate a state machine, or objects can exchange messages among themselves, similar to the real-world situation.

Unlike an interactive system, the internal objects in a dynamic simulation do correspond to real-world objects, so the class model is usually important and often complex. Like an interactive system, the state and interaction models are also important.

The steps in designing a dynamic simulation are as follows.

- Identify active real-world objects from the class model. These objects have attributes that are periodically updated.
- Identify discrete events. Discrete events correspond to discrete interactions with the object, such as turning power on or applying the brakes. Discrete events can be implemented as operations on the object.
- Identify continuous dependencies. Real-world attributes may be dependent on other real-world attributes or vary continuously with time, altitude, velocity, or steering wheel position, for example. These attributes must be updated at periodic intervals, using numerical approximation techniques to minimize quantization error.
- Generally a simulation is driven by a timing loop at a fine time scale. Discrete events between objects can often be exchanged as part of the timing loop.

Usually, the hardest problem with simulations is providing adequate performance. In an ideal world, an arbitrary number of parallel processors would execute the simulation in an exact analogy to the real-world situation. In practice, the system designer must estimate the computational cost of each update cycle and provide adequate resources. Discrete steps must approximate continuous processes.

14.12.5 Real-time System

A ***real-time system*** is an interactive system with tight time constraints on actions. Hard real-time software involves critical applications that require a guaranteed response within the time constraints. In contrast, soft real-time software must also be highly reliable, but can occasionally violate time constraints. Typical real-time applications include process control, data acquisition, communications devices, device control, and overload relays.

Real-time design is complex and involves issues such as interrupt handling, prioritization of tasks, and coordinating multiple CPUs. Unfortunately, real-time systems are frequently designed to operate close to their resource limits, so that severe, nonlogical restructuring of the design is often needed to achieve the necessary performance. Such contortions come at the cost of portability and maintainability. Real-time design is a specialized topic that we do not cover in this book.

14.12.6 Transaction Manager

A ***transaction manager*** is a system whose main function is to store and retrieve data. Most transaction managers deal with multiple users who read and write data at the same time. They also must secure their data to protect it from unauthorized access as well as accidental loss. Transaction managers are often built on top of a database management system (DBMS)—this is a form of reuse. A DBMS has generic functionality for managing data that you can reuse and need not implement. Examples of transaction managers include airline reservations, inventory control, and order fulfillment.

The class model is dominant. The state model is occasionally important, especially for specifying the evolution of an object as well as constraints and methods that apply at different points in time. The interaction model is seldom significant.

The steps in designing an information system are as follows.

- Map the class model to database structures. See Chapter 19 for advice.
- Determine the units of concurrency—that is, the resources that inherently or by specification cannot be shared. Introduce new classes as needed.
- Determine the unit of transaction—that is, the set of resources that must be accessed together during a transaction. A transaction succeeds or fails in its entirety.
- Design concurrency control for transactions. Most database management systems provide this. The system may need to retry failed transactions several times before giving up.

14.13 Architecture of the ATM System

The ATM system is a hybrid of an interactive interface and a transaction management system. The entry stations are interactive interfaces—their purpose is to interact with a human to gather information needed to formulate a transaction. Specifying the entry stations consists of constructing a class model and a state model. The consortium and banks are primarily a distributed transaction management system. Their purpose is to maintain data and allow it to be updated over a distributed network under controlled conditions. Specifying the transaction management part of the system consists primarily of constructing a class model. Figure 14.2 shows the architecture of the ATM system.

The only permanent data stores are in the bank computers. A database ensures that data is consistent and available for concurrent access. The ATM system processes each transaction as a single batch operation, locking an account until the transaction is complete.

Concurrency arises because there are many ATM stations, each of which can be active at any time. There can be only one transaction per ATM station, but each transaction requires the assistance of the consortium computer and a bank computer. As Figure 14.2 shows, a transaction cuts across physical units; the diagram shows each transaction as three connected pieces. During design, each piece will become a separate implementation class. Although there is only one transaction per ATM station, there may be many concurrent transactions per consortium computer or bank computer. This does not pose any special problem, because the database synchronizes access to any one account.

The consortium computer and bank computers will be event driven. Each of them queues input events but processes them one at a time in the order received. The consortium computer has minimal functionality. It simply forwards a message from an ATM station to a bank computer and from a bank computer to an ATM station. The consortium computer must be large enough to handle the transaction load. It may be acceptable to block an occasional transaction, provided the user receives an appropriate message.

The bank computer is the only unit with any nontrivial procedures, but even those are mostly just database updates. The only complexity might come from failure handling. The bank computers must have capacity to handle the expected worst-case load, and they must have enough disk storage to record all transactions.

The system must contain operations for adding and deleting ATM stations and bank computers. Each physical unit must protect itself against the failure or disconnection from the rest of the network. A database protects against loss of data. However, special attention must be paid to failure during a transaction so that neither the user nor the bank loses money—this may require a complicated acknowledgment protocol before committing the transaction. The ATM station should display an appropriate message if the connection is down. The ATM must handle other kinds of failure as well, such as exhaustion of cash or paper for receipts.

On a financial system such as this, fail-safe transactions are the highest priority. If there is any doubt about the integrity of a transaction, then the ATM must abort the transaction with an appropriate message to the user.

14.14 Chapter Summary

After analyzing an application and before beginning the class design, the system designer must decide on the basic approach to the solution. The form of the high-level strategy for building the system is called the system architecture.

Early in the planning for a new system you should estimate the performance. The intention is to have a rough idea of what to expect. You want to make sure that it is reasonable and that there are no big surprises as development proceeds.

Next, prepare a reuse plan. Reuse is often cited as a benefit of OO technology, but it does not happen automatically. There are two different aspects of reuse. Most developers should focus on reusing existing models, libraries, frameworks, and patterns that are relevant to their applications. In addition, elite developers can create artifacts for reuse by others.

A system can be divided into horizontal layers and vertical partitions. Each layer defines a different abstract world that may differ completely from other layers. Each layer is a client of services of the layer or layers below it and a server of services for the layer or layers above it. Systems can also have partitions, each performing a general kind of service. Simple system topologies, such as pipelines or stars, reduce complexity. Most systems are a mixture of layers and partitions.

Inherently concurrent objects execute in parallel, and a single thread of control cannot combine them; they require separate hardware devices or separate tasks in a processor. You can combine nonconcurrent objects onto a single thread of control and implement them as a single task.

A system must have enough processors and special-purpose hardware units to meet performance goals. You should assign objects to hardware so that hardware use is balanced and meets concurrency constraints. You can do this by estimating computational throughput and allowing for queuing effects in configuring the hardware. You may want to use special-purpose hardware for compute-intensive computations. One goal in partitioning a hardware network is to minimize communications traffic between physically distinct modules.

Data stores can cleanly separate subsystems within an architecture and give application data some degree of permanence. In general, memory data structures, files, and databases can implement data stores. Files are simple, cheap, and permanent but may provide too low a level of abstraction for an application and necessitate much additional programming. Databases provide a higher level of abstraction than files, but they too involve compromises in terms of overhead costs and complexity.

The system designer must identify global resources and determine mechanisms for controlling access to them. Some common mechanisms are: establishing a "guardian" object that serializes all access, partitioning global resources into disjoint subsets which are managed at a lower level, and locking.

Hardware control is inherently concurrent, but software control can be procedure driven, event driven, and concurrent. Control for a procedure-driven system resides within the program code; the location of the program counter and the stack of procedure calls and local variables define the system state. In an event-driven system control resides within a dispatcher or monitor; application procedures are attached to events and are called by the dispatcher when the corresponding events occur. In a concurrent system, control resides concurrently in multiple independent objects. Event-driven and concurrent implementations are much more flexible than procedure-driven control.

Most of system design is concerned with steady-state behavior, but boundary conditions (initialization, termination, and failure) are also important.

An essential aspect of system architecture is making trade-offs between time and space, hardware and software, simplicity and generality, and efficiency and maintainability. These trade-offs depend on the goals of the application. The system designer must state the priorities, so that trade-off decisions during subsequent design will be consistent.

Several kinds of systems are frequently encountered for which standard architectural styles exist. These include two kinds of functional transformations: batch computation and continuous transformation; three kinds of time-dependent systems: interactive interface, dynamic simulation, and real-time; and a database system: transaction manager. Most application systems are usually a hybrid of several forms, possibly one for each major subsystem. Other kinds of architecture are possible.

architecture	hardware requirements	service
client-server	inherent concurrency	subsystem
concurrency	layer	system design
data management	partition	system topology
event-driven system	peer-to-peer	thread of control
framework	reuse plan	trade-off priorities

Figure 14.7 Key concepts for Chapter 14

Bibliographic Notes

Simple software applications do not require much systems engineering, but complex systems must be decomposed and the parts assigned to the appropriate specialists. [Clements-02] presents a process for evaluating software architectures. Essentially a group of stakeholders meet and prioritize criteria that the architecture should satisfy; they quantify the criteria with specific scenarios. Then they analyze the architecture to determine its compliance with the high-priority scenarios.

Patterns are a popular topic in the literature and the subject of a number of books. There are patterns for analysis [Coad-95], architecture [Buschmann-96] [Shaw-96], design [Gamma-95], and implementation [Coplien-92]. There have also been a number of conferences over the years that have focused on patterns, many of which have been sponsored by the Pattern Languages of Programming [PLoP].

References

[Berlin-90] Lucy Berlin. When objects collide: Experiences with reusing multiple class hierarchies. *ECOOP/OOPSLA 1990 Proceedings*, October 21–25, 1990, Ottawa, Ontario, Canada, 181–193.

[Buschmann-96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester, UK: Wiley, 1996.

[Clements-02] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures*. Boston: Addison-Wesley, 2002.

[Coad-95] Peter Coad, David North, and Mark Mayfield. *Object Models: Strategies, Patterns, and Applications*. Upper Saddle River, NJ: Yourdon Press, 1995.

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

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

[Johnson-88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming 1*, 3 (June/July 1988), 22–35.

[Korson-92] Tim Korson and John D. McGregor. Technical criteria for the specification and evaluation of object-oriented libraries. *Software Engineering Journal* (March 1992), 85–94.

[PLoP] jerry.cs.uiuc.edu/~plop

[Shaw-96] Mary Shaw and David Garlan. *Software Architecture*. Upper Saddle River, NJ: Prentice Hall, 1996.

Exercises

14.1 (4) For each of the following systems, list the applicable style(s) of system architecture: batch transformation, continuous transformation, interactive interface, dynamic simulation, real-time system, and transaction manager. Explain your selection(s). For systems that fit more than one style, group features of the system by style.

a. **An electronic chess companion**. The system consists of a chess board with a built-in computer, lights, and membrane switches. The human player registers moves by pressing chess pieces on the board, activating membrane switches mounted under each square. The computer indicates moves through lights also mounted under each square. The human moves the chess pieces for the computer. The computer should make only legal moves, should reject attempted illegal human moves, and should try to win.

b. **An airplane flight simulator for a video game system**. The video game system has already been implemented and consists of a computer with joystick and pushbutton inputs and an output interface for a color television. Your job is to develop the software for the computer to display the view from the cockpit of an airplane. The joystick and pushbutton control the airplane. The display should be based on a terrain description stored in memory. When your program is complete, it will be sold on cartridges that plug into the video game system.

c. **A floppy disk controller chip**. The chip is going to use a microprogram for internal control. You are concerned with the microprogram. The chip bridges the gap between a computer and a floppy disk drive. Your portion of the control will be responsible for positioning the read/write head and reading the data. Information on the diskette is organized into tracks and sectors. Tracks are equally spaced circles of data on the diskette. Data within a track is organized into sectors. Your architecture will need to support the following operations: Find track 0, find a given track, read a track, read a sector, write a track, and write a sector.

d. **A sonar system**. You are concerned with the portion of the system that detects undersea objects and computes how far away they are (range). This is done by transmitting an acoustic pulse and analyzing any resulting echo. A technique called correlation is used to perform the analysis, in which a time-delayed copy of the transmitted pulse is multiplied by the returned echo and integrated for many values of time delay. If the result is large for a particular value of time delay, it is an indication that there is an object with a range that corresponds to that delay.

14.2 (3) Discuss how you would implement control for the applications described in the previous exercise.

14.3 (7) As the system architect for a new signal-processing product, you must decide how to store data in real time. The product uses analog to digital convertors to sample an analog input signal at the rate of 16,000 bytes/second (128,000 bits/second) for 10 seconds. Unfortunately, the needed calculations are too time consuming to do as the samples are received, so you are going to have to store the samples temporarily. The decision has already been made to limit the amount of memory used for buffers to 64,000 bytes. The system has a floppy disk drive that uses diskettes organized into 77 tracks for a total of 243,000 bytes of storage per diskette. It takes 10 milliseconds to move the disk drive read/write head from one track to another and 83 milliseconds, on average, to find the beginning of a track once the head is positioned. The disk drive will be positioned at the correct track prior to the start of data acquisition.

Two solutions to the problem are being considered: (1) Simply write the data samples on the diskette as they become available. Why doesn't this work? (2) Use memory as a buffer. Data samples are placed in memory as they are acquired and written to the floppy disk as fast as possible on sequential tracks. Will this method work? Describe the method in more detail. How much memory is needed for the buffer? How many tracks will be used on the diskette? Prepare a few scenarios. Describe how the control might work.

14.4 (6) Consider a system for automating the drilling of holes in rectangular metal plates. The size and location of holes are described interactively, using a graphical editor on a personal computer. When the user is satisfied with a particular drawing, a peripheral device on the personal computer punches a numerical control (N/C) tape. The tape can then be used on a variety of commercially available N/C drilling machines that have moving drill heads and can change drill sizes.

You are concerned only with the editing of the drawings and the punching of the N/C tapes. The tapes contain sequences of instructions to move the drill head, change drills, and drill. Since it takes some time to move the drill between holes, and even longer to change drills, the system should determine a reasonably efficient drilling sequence. It is not necessary to achieve the absolute minimum time, but the system should not be grossly inefficient either. The drill head is controlled independently in the x and y directions, so the time it takes to move between holes is proportional to the larger of the required displacements in the x and the y direction. Prepare a system architecture. How would you characterize the style of the system?

14.5 (5) Consider a system for interactive symbolic manipulation of polynomials. The basic idea is to allow a mathematician to be more accurate and productive in developing formulas. The user enters mathematical expressions and commands a line at a time. Expressions are ratios of polynomials, which are constructed from constants and variables. Intermediate expressions may be assigned to variables for later recall. Operations include addition, subtraction, multiplication, division, and differentiation with respect to a variable.

Develop an architecture for the system. How would you characterize the style of the system? How would you save work in progress to resume at a future time?

14.6 (4) An architecture for the system described in the previous exercise could involve the following subsystems. Organize them into partitions and layers.

a. line syntax—scan a line of user input for tokens
b. line semantics—determine the meaning of a line of input
c. command processing—execute user input, error checking
d. construct expression—build an internal representation of an input expression
e. apply operation—carry out an operation on one or more expressions
f. save work—save the current context
g. load work—read in previously saved context
h. substitute—substitute one expression for a variable in another expression
i. rationalize—convert an expression to canonical form
j. evaluate—replace a variable in an expression with a constant and simplify the expression

14.7 (6) Consider a system for editing, saving, and printing class diagrams and generating relational database schema. The system supports only a limited subset of the class modeling notation—classes with attributes and binary associations with multiplicity. The system also includes editing functions such as create class, create association, cut, copy, and paste. The editor must understand the semantics of class diagrams. For example, when a class rectangle is moved, the lines representing any attached associations are stretched. If a class is deleted, attached associations are also deleted. When the user is satisfied with the diagram, the system will generate the corresponding relational database schema. Discuss the relative advantages of a single program that performs all functions versus two programs, one that edits class diagrams and the other that generates database schema from class diagrams.

14.8 (6) In the previous exercise, both physical and logical aspects of class diagrams must be considered. Physical aspects include location and sizes of lines, boxes, and text. Logical aspects include connectivity, classes, attributes, and associations. Discuss basing your architecture on the following strategies. Consider the issues involved in editing and saving class diagrams as well as generating database schema.

a. Model only the geometrical aspects of class diagrams. Treat logical aspects as derived.
b. Model both the geometrical and logical aspects of class diagrams.

14.9 (5) Another approach to the system described in Exercise 14.7 is to use a commercially available desktop publishing system for class diagram preparation instead of implementing your own class diagram editor. The desktop editor can dump its output in an ASCII markup language. The vendor supplies the grammar for the markup language.

Compare the two approaches. One approach is to build your own editor that understands the semantics of class diagrams. The other is to use a commercially available desktop publishing system to edit class diagrams. What happens if new versions of the desktop publishing system become available? Can you assume that the user prepares a diagram using a notation that your database generator will understand? Is it worth the effort to implement functions such as cut, copy, and paste that commercial systems already do so well? Who is going to help the users if they run into problems? How is your system going to be supported and maintained? How soon can you get the system completed?

14.10 (6) A common issue in many systems is how to store data so it is preserved in the event of power loss or hardware failure. The ideal solution should be reliable, low-cost, small, fast, maintenance free, and simple to incorporate into a system. Also, it should be immune to heat, dirt, and humidity. Compromises in the available technology often influence the functional requirements. Compare each of the following solutions in terms of the ideal. Note that this is not an exhaustive list of solutions.

a. Do not worry about it at all. Reset all data every time the system is turned on.
b. Never turn the power off if it can be helped. Use a special power supply, including backup generators, if necessary.
c. Keep critical information on a magnetic disk drive. Periodically make full and/or incremental copies on magnetic tape.
d. Use a battery to maintain power to the system memory when the rest of the system is off. It might even be possible to continue to provide limited functionality.
e. Use a special memory component, such as a magnetic bubble memory or an electronically erasable programmable read-only memory.
f. Critical parameters are entered by the user through switches. Several types of switches are commercially available for this use, including several toggle switches in a package that connects the same way as an integrated circuit.

14.11 (7) For each of the following systems, select one or more of the strategies for data storage from the previous exercise. In each case explain your reasoning and give an estimate (order of magnitude) of how much memory, in bytes, is required:

a. **Four-function pocket calculator**. Main source of power is light. Performs basic arithmetic.
b. **Electronic typewriter**. Main source of power is either rechargeable batteries or alternating current. Operates in two modes. In one mode, documents are typed a line at a time. Editing may be performed on a line before it is typed. A liquid crystal display will display up to 16 characters for editing purposes. In the other mode, an entire document can be entered and edited before printing. The typewriter should be able to save the working document for up to a year with the main power off.
c. **System clock for a personal computer**. Main power is direct current supplied by the personal computer when it is on. Provides time and date information to the computer. Must maintain the correct date and time for at least five years with the main power off.
d. **Airline reservation system**. Main power is alternating current. Used to reserve seats on airline flights. The system must be kept running at all times, at all costs. If, for some reason, the system must be shut off, no data should be lost.
e. **Digital control and thermal protection unit for a motor**. The device provides thermal protection for motors over a wide range of horsepower ratings by calculating motor temperature based on measured current and a simulation of motor heat dissipation. If the calculated motor temperature exceeds safe limits, the motor is shut off and not allowed to be restarted until it cools down. The main source of power is alternating current, which may be interrupted. The system must provide protection as soon as it is turned on. Parameters needed for thermal

simulation are initially set at the factory, but provision must be made to change them, if necessary, after the system is installed. Because the motor temperature is not measured directly, it is necessary to continue to simulate the motor temperature for at least an hour after loss of main power, in case power is restored before the motor cools.

14.12 (9) The design of file formats is a common task for system design. A BNF diagram is a convenient way to express file formats. Figure E14.1 is a portion of a BNF diagram of a language for describing classes and binary associations. Nonterminal symbols are shown in rectangles, and terminal symbols are shown in circles or rectangles with rounded corners. With the exception of *character*, the diagram defines all nonterminals. A diagram consists of classes and associations. A class has a unique name and many attributes. An association has an optional name and two ends. An association end contains the name one of the classes being associated and multiplicity information. Textual information is described by quoted strings. A character is any ASCII character except quote.

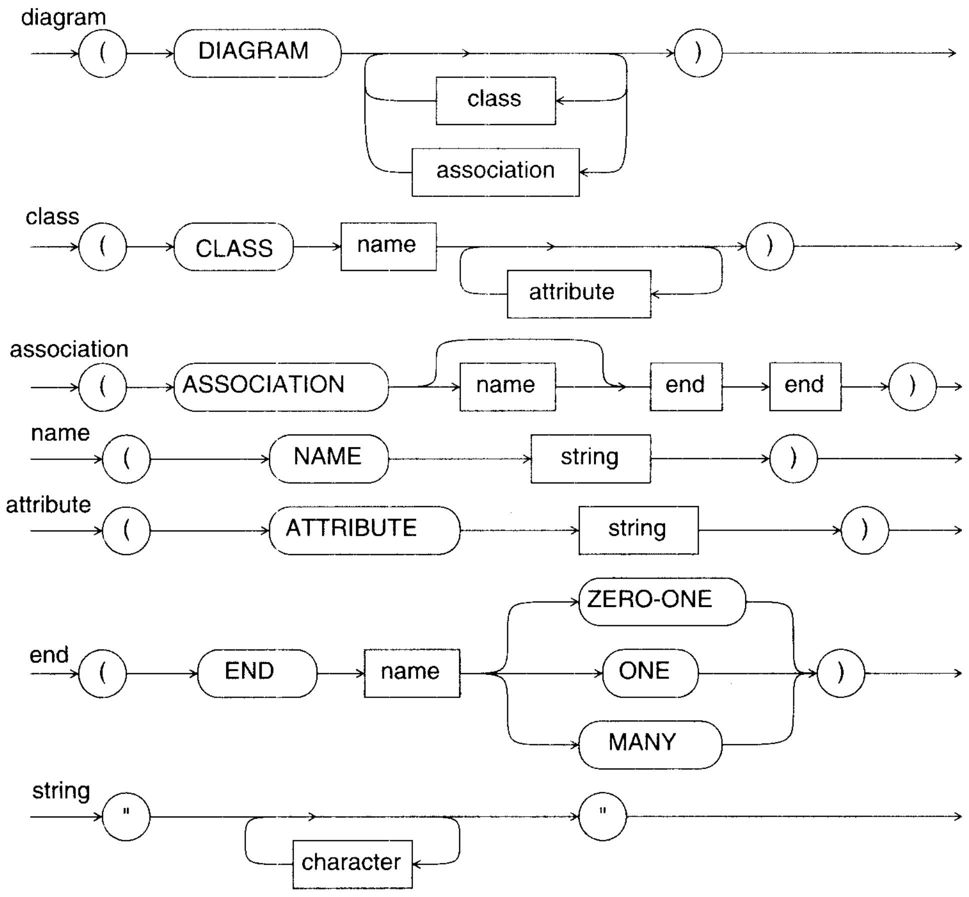


Figure E14.1 BNF diagram for a language that describes classes and associations

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.

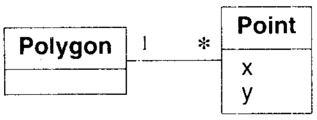


Figure E14.2 Class diagram of polygons

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