


CMR INSTITUTE OF TECHNOLOGY		USN								
Internal Assessment Test – III										
Sub:	Object Oriented Modeling And Design						Code:	18MCA43		
Date:	15-06-2021	Duration:	90 mins	Max Marks:	50	Sem:	IV	Branch:	MCA	
Answer <b>ONE FULL QUESTION</b> from each part										
								Marks	OBE	
								CO	RBT	
<b>Part – I</b>										
1	<p>What do you mean by Domain State Model? Explain the steps performed in constructing the domain state model.</p> <p><b>12.3 Domain State Model</b></p> <p>Some <b>domain</b> 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 <b>state</b> diagram of such a <b>domain</b> class. The <b>state</b> 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 <b>state</b> to another.</p> <p>Most <b>domain</b> classes do not require <b>state</b> diagrams and can be adequately described by a list of operations. For the minority of classes that do exhibit distinct states, however, a <b>state model</b> can help in understanding their behavior.</p> <p>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.</p> <p>The following steps are performed in constructing a domain state model.</p> <ul style="list-style-type: none"> <li>■ Identify domain classes with states. [12.3.1]</li> <li>■ Find states. [12.3.2]</li> <li>■ Find events. [12.3.3]</li> <li>■ Build state diagrams. [12.3.4]</li> <li>■ Evaluate state diagrams. [12.3.5]</li> </ul> <p><b>12.3.1 Identifying Classes with States</b></p> <p>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 <i>Being written</i> to <i>Under consideration</i> to <i>Accepted</i> or <i>Rejected</i>. 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 <i>Maintenance</i>, <i>Loading</i>, <i>Flying</i>, and <i>Unloading</i>. 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.</p> <p><b>ATM example.</b> <i>Account</i> is an important business concept, and the appropriate behavior for an ATM depends on the state of an <i>Account</i>. The life cycle for <i>Account</i> is a mix of progressive and cycling to and from problem states. No other ATM classes have a significant domain state model.</p> <p><b>12.3.2 Finding States</b></p> <p>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.</p> <p>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.</p> <p>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.</p> <p><b>ATM example.</b> Here are some states for an <i>Account</i>: <i>Normal</i> (ready for normal access), <i>Closed</i> (closed by the customer but still on file in the bank records), <i>Overdrawn</i> (customer withdrawals exceed the balance in the account), and <i>Suspended</i> (access to the account is blocked for some reason).</p>						10	CO3	L2	

### 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.

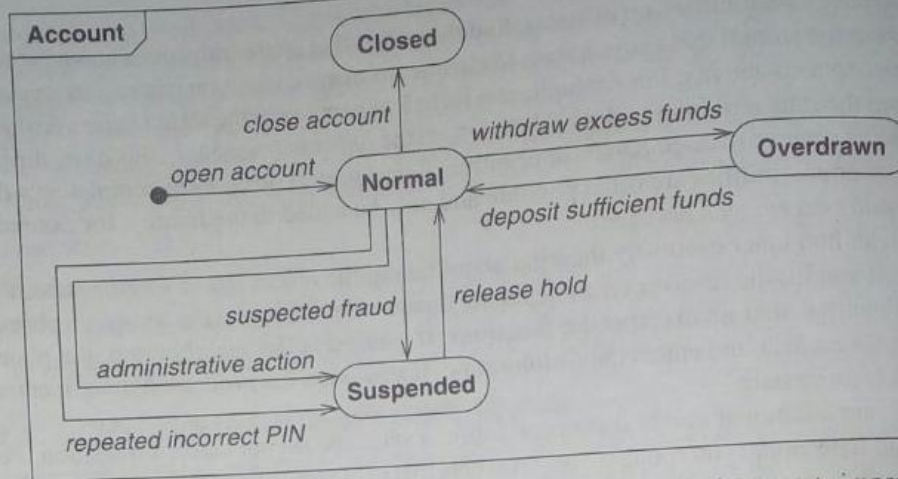
Once you have specified the transitions, consider the meaning of an event in states for which there is no transition on the event. Is it ignored? Then everything is fine. Does it represent an error? Then add a transition to an error state. Does it have some effect that you forgot? Then add another transition.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

(OR)

2

What do you mean by system conception? What are the ways to find new system conception? Explain in detail what a good system concept must answer?

System conception deals with the genesis of an application. Initially some person, who understands both business needs and technology, thinks of an idea for an application. Developers must then explore the idea to understand the needs and devise possible solutions. The purpose of system conception is to defer details and understand the big picture – what need does the proposed system meet, can it be developed at a reasonable cost, and will the demand for the result justify the cost of building it? Ways to find new system conception:

**New Functionality:** Add functionality to an existing system

**Streamlining:** Remove restrictions or generalize the way a system works

**Simplification:** Let ordinary persons perform tasks previously assigned to specialists

**Automation:** Automate manual processes

**Integration:** Combine functionality from different systems

**Analogies:** Look for analogies in other problem domains and see if they have useful ideas

**Globalization:** Travel to other countries and observe their cultural and business practices

A good system conception should answer the following questions.

**Who is the application for?** (person/organization/stakeholder/financial sponsor/end user)

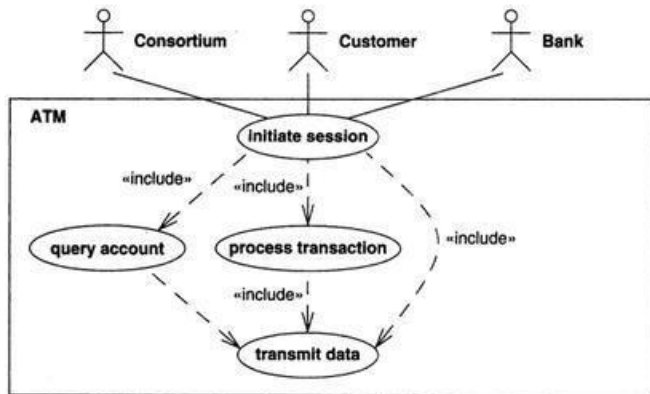
10

CO3

L2

	<b>What problem will it solve?</b> <b>Where will it be used?</b> <b>When is it needed?</b> <b>Why is it needed?</b> <b>How will it work?</b>			
<b>Part – II</b>				
3	<p>What do you mean by Domain Class Model? Explain the steps performed in constructing the domain Class model.</p> <p><b>12.2.7 Keeping the Right Attributes</b>  Eliminate unnecessary and incorrect attributes with the following criteria.</p> <ul style="list-style-type: none"> <li>■ <b>Objects.</b> If the independent existence of an element is important, rather than just its value, then it is an object. For example, <i>boss</i> refers to a class and <i>salary</i> is an attribute. The distinction often depends on the application. For example, in a mailing list <i>city</i> might be considered as an attribute, while in a census <i>City</i> 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.</li> <li>■ <b>Qualifiers.</b> If the value of an attribute depends on a particular context, then consider restating the attribute as a qualifier. For example, <i>employeeNumber</i> is not a unique property of a person with two jobs; it qualifies the association <i>Company employs person</i>.</li> <li>■ <b>Names.</b> 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, <i>department-Name</i> 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.</li> <li>■ <b>Identifiers.</b> 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, <i>accountCode</i> is a genuine attribute; <i>Banks</i> assign <i>accountCodes</i> and customers see them. In contrast, you should not list an internal <i>transactionID</i> as an attribute, although it may be convenient to generate one during implementation.</li> <li>■ <b>Attributes on associations.</b> 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</li> </ul>	10	CO3	L1

	<p>multiplicity. For example, in an association between <i>Person</i> and <i>Club</i> the attribute <i>membershipDate</i> 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.</p> <ul style="list-style-type: none"> <li>■ <b>Internal values.</b> If an attribute describes the internal state of an object that is invisible outside the object, then eliminate it from the analysis.</li> <li>■ <b>Fine detail.</b> Omit minor attributes that are unlikely to affect most operations.</li> <li>■ <b>Discordant attributes.</b> 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.</li> <li>■ <b>Boolean attributes.</b> Reconsider all boolean attributes. Often you can broaden a boolean attribute and restate it as an enumeration [Coad-95].</li> </ul> <p><b>ATM example.</b> 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.</p> <ul style="list-style-type: none"> <li>■ <i>BankCode</i> and <i>cardCode</i> are present on the card. Their format is an implementation detail, but we must add a new association <i>Bank issues CashCard</i>. <i>CardCode</i> is a qualifier on this association; <i>bankCode</i> is the qualifier of <i>Bank</i> with respect to <i>Consortium</i>.</li> <li>■ 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.</li> <li>■ Avoid the temptation to omit <i>Consortium</i>, even though it is currently unique. It provides the context for the <i>bankCode</i> qualifier and may be useful for future expansion.</li> </ul> <p>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.</p>			
(OR)				
4	<p>What is Application class model? Explain the steps to construct the application class model.</p> <p><b>13.2 Application Class Model</b></p> <p>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.</p>	10	CO3	L2



**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

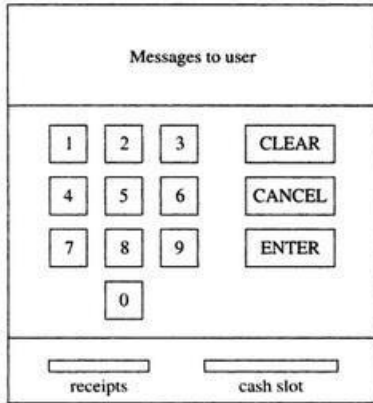
Most interactions can be separated into two parts: application logic and the user interface. A *user interface* is an object or group of objects that provides the user of a system with a coherent way to access its domain objects, commands, and application options. During analysis the emphasis is on the 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.).

## PART - III

5	Specify the steps to construct the application interaction model. Explain any three steps with examples.	10	CO3	L1
---	--	----	-----	----

## 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*.



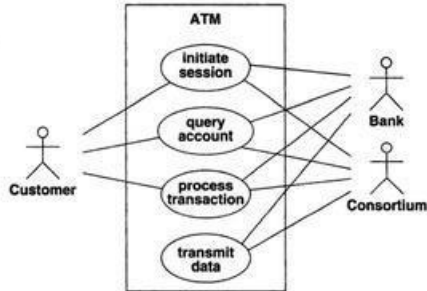
### 13.1.3 Finding Use Cases

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

<b>Initiate session</b>	<p>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</p>
<b>Query account</b>	<p>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.</p>
<b>Process transaction</b>	<p>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.</p>
<b>Transmit data</b>	<p>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.</p>

**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

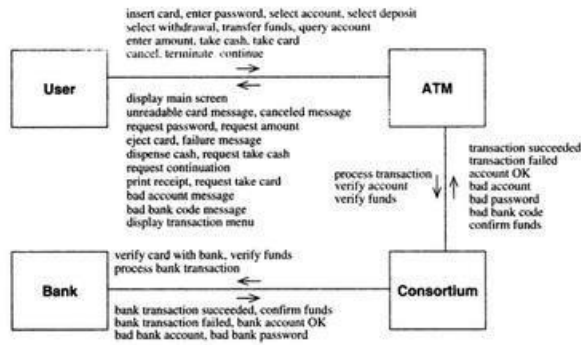


Figure 13.4 Events for the ATM case study. Tally the events in the scenarios and note the classes that send and receive each event.

### 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.

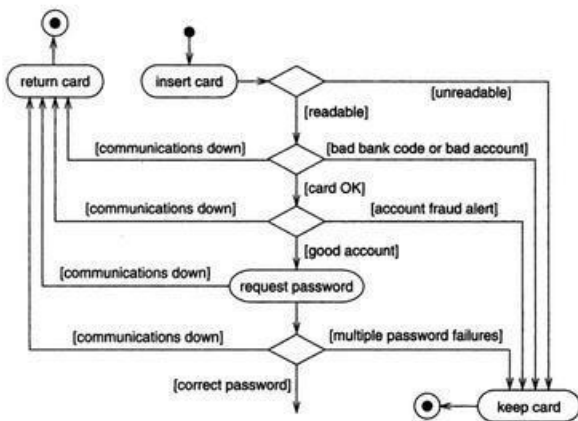


Figure 13.5 Activity diagram for card verification. You can use activity diagrams to document business logic, but do not use them as an excuse to begin premature implementation.

### 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.

(OR)

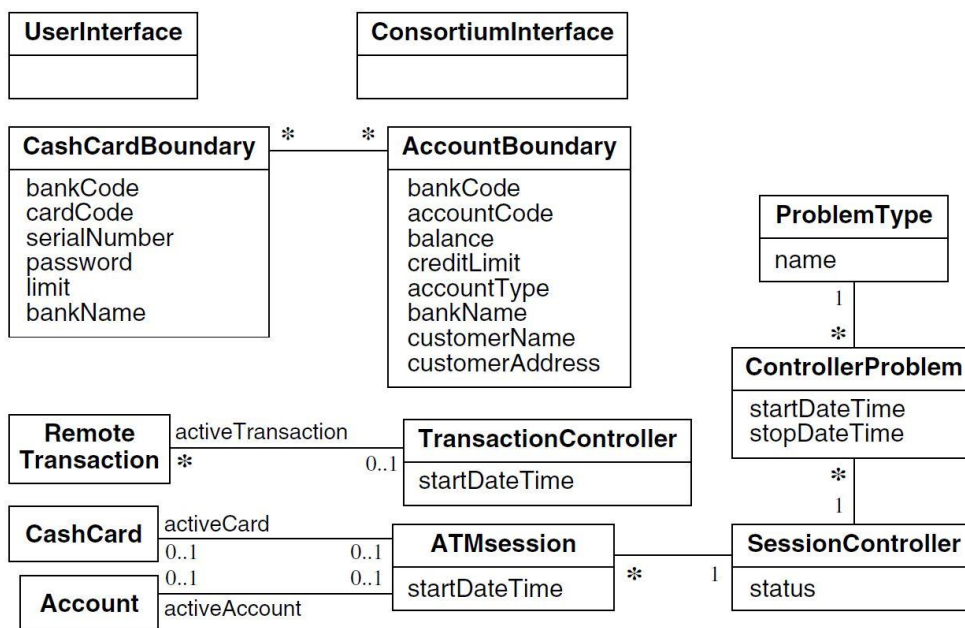
6 Discuss Application state model? Explain the steps to construct the application state model..  
The application state model focuses on application classes and augments the domain state model. Application classes are more likely to have important temporal behavior than domain classes.

10

CO3

L3

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**

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.

<p>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 <i>number of failures</i>, to remember information. At least one transition must depend on the parameter.</p> <p>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 <i>No error</i> and <i>One error</i>.</p> <p>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 <i>time-out</i> 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.</p> <p>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.</p> <p>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.</p> <p><b>ATM example.</b> Figure 13.9 shows the state diagram for the <i>SessionController</i>. 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.</p>		
---	--	--

<p>Figure 13.10 and Figure 13.11 show the state diagram for the <i>TransactionController</i> that is spawned by the <i>SessionController</i>. (See the exercises for the other subdiagrams of Figure 13.10.) We have separated the <i>TransactionController</i> and the <i>SessionController</i> because their purposes are much different—the <i>SessionController</i> focuses on verifying users, while the <i>TransactionController</i> services account inquiries and balance changes.</p> <p><i>13.3.4 Checking Against Other State Diagrams</i></p> <p>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.</p>		
---	--	--

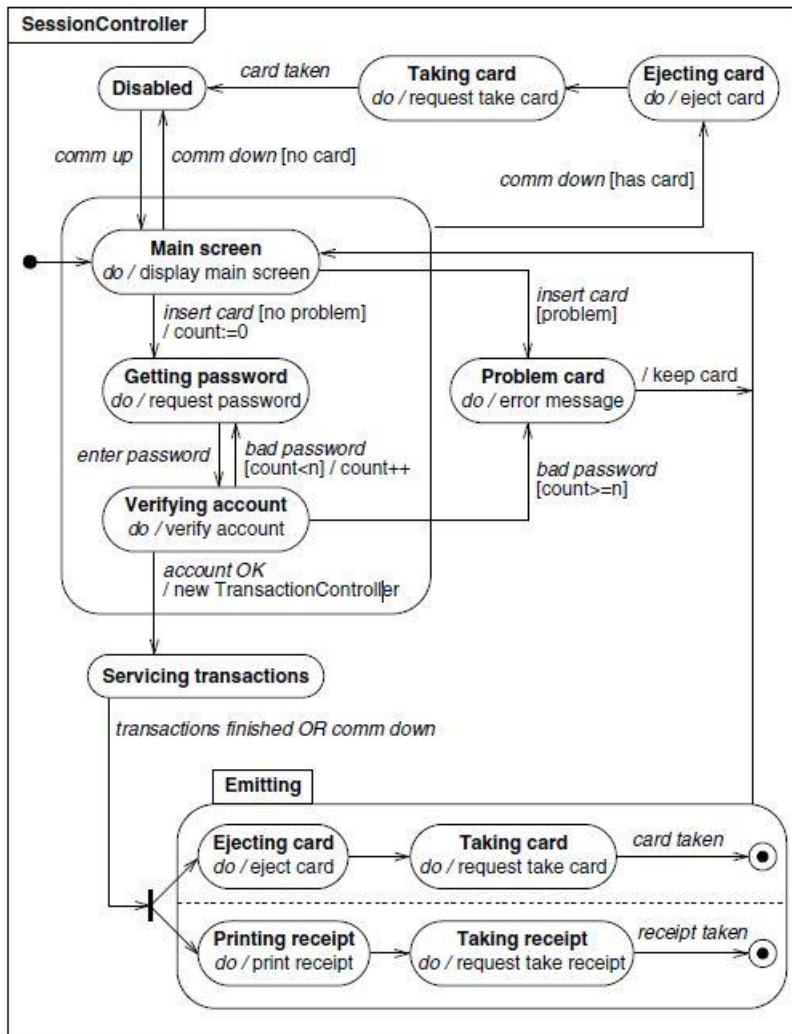


Figure 13.9 State diagram for *SessionController*. Build a state diagram for each application class with temporal behavior.



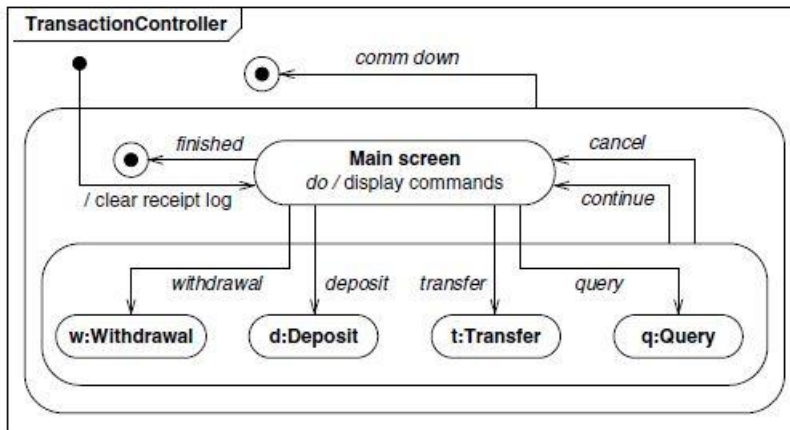


Figure 13.10 State diagram for *TransactionController*. Obtain information from the scenarios of the interaction model.

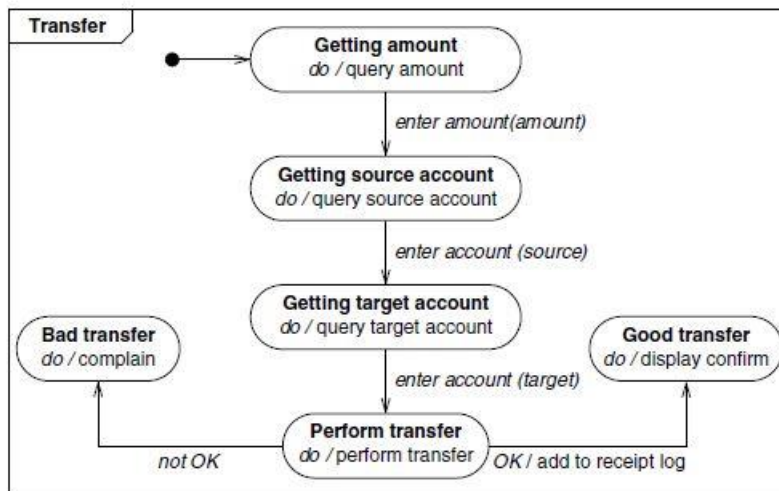


Figure 13.11 State diagram for *Transfer*. This diagram elaborates the *Transfer* state in Figure 13.10.

**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.

## Part – IV

7

What is reusability? What are the reusable things? Explain.

10

CO4

L2

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.

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.

### *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. 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.
- **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.

	<p><b>2 Frameworks</b>  A <i>framework</i> 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.</p> <p><b>3 Patterns</b>  A <i>pattern</i> 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.</p> <p>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. Patterns are prototypical model fragments that distill some of the knowledge of experts.</p> <p>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.</p>			
(OR)				
8	<p>Explain the steps to design algorithms</p> <p>Now formulate an <i>algorithm</i> for each operation. The analysis specification tells <i>what</i> the operation does for its clients, but the algorithm shows <i>how</i> it is done. Perform the following steps to design algorithms.</p> <ul style="list-style-type: none"> <li>■ Choose algorithms that minimize the cost of implementing operations.</li> <li>■ Select data structures appropriate to the algorithms.</li> <li>■ Define new internal classes and operations as necessary.</li> <li>■ Assign operations to appropriate classes.</li> </ul> <p><b>1 Choosing Algorithms</b>  Many operations are straightforward because they simply traverse the class model to retrieve or change attributes or links. The OCL provides a convenient notation for expressing such traversals.</p> <p>However, a class-model traversal cannot fully express some operations. We often use pseudocode to handle these situations. Pseudocode helps us think about the algorithm while deferring programming details. For example, many applications involve graphs and the use of transitive closure.</p> <p>When efficiency is not an issue, you should use simple algorithms. In practice, only a few operations tend to be application bottlenecks. Typically, 20% of the operations consume 80% of execution time. For the remaining operations, it is better to have a design that is simple, understandable, and easy to program than to wring out minor improvements. You can</p>	10	CO4	L2

	<p>focus your creativity on the algorithms for the operations that are a bottleneck. Here are some considerations for choosing among alternative algorithms.</p> <ul style="list-style-type: none"> <li>■ <b>Computational complexity.</b> How does processor time increase as a function of data structure size? It is essential to think about algorithm complexity—that is, how the execution time (or memory) grows with the number of input values: constant time, linear, quadratic, or exponential.</li> <li>■ <b>Ease of implementation and understandability.</b> It is worth giving up some performance on noncritical operations if you can use a simple algorithm.</li> <li>■ <b>Flexibility.</b> You will find yourself extending most programs, sooner or later. A highly optimized algorithm often sacrifices ease of change. One possibility is to provide two versions of critical operations.</li> </ul> <p><i>2 Choosing Data Structures</i></p> <p>Algorithms require data structures on which to work. During analysis, you focused on the logical structure of system information, but during design you must devise data structures that will permit efficient algorithms. The data structures do not add information to the analysis model, but they organize it in a form convenient for algorithms. Many of these data structures are instances of <i>container classes</i>. Such data structures include arrays, lists, queues, stacks, sets, bags, dictionaries, trees, and many variations, such as priority queues and binary trees. Most OO languages provide an assortment of generic data structures as part of their predefined class libraries.</p> <p><i>3 Defining Internal Classes and Operations</i></p> <p>You may need to invent new, low-level operations during the decomposition of high-level operations. Some of the low-level operations may be in the “shopping list” of operations from analysis. But usually you will need to add new internal operations as you expand high-level operations.</p> <p>The expansion of algorithms may lead you to create new classes of objects to hold intermediate results. Typically, the client’s description of the problem will not mention these low-level classes because they are artifacts.</p> <p><i>4 Assigning Operations to Classes</i></p> <p>When a class is meaningful in the real world, the operations on it are usually clear. During design, however, you introduce internal classes that do not correspond to real-world objects but merely some aspect of them. Since the internal classes are invented, they are somewhat arbitrary, and their boundaries are more a matter of convenience than of logical necessity. How do you decide what class owns an operation? Ask yourself the following questions.</p> <ul style="list-style-type: none"> <li>■ <b>Receiver of action.</b> Is one object acted on while the other object performs the action? In general, it is best to associate the operation with the <i>target</i> of the operation, rather than the <i>initiator</i>.</li> <li>■ <b>Query vs. update.</b> Is one object modified by the operation, while other objects are only queried for their information? The object that is changed is the target of the operation.</li> <li>■ <b>Focal class.</b> Looking at the classes and associations that are involved in the operation, which class is the most centrally located in this subnetwork of the class model? If the classes and associations form a star about a single central class, it is the operation’s target.</li> <li>■ <b>Analogy to real world.</b> If the objects were not software, but were the real-world objects, what real object would you push, move, activate, or otherwise manipulate to initiate the operation?</li> </ul> <p>Sometimes it is difficult to assign an operation to a class within a generalization hierarchy. It is common to move operations up and down in the hierarchy during design, as their scope is adjusted. Furthermore, the definitions of the subclasses within the hierarchy are often fluid</p>			
<b>Part – V</b>				
9	What tasks are involved in the process of design optimization? Explain any one in detail.	10	CO4	L1

First get a clean design working. Then you can optimize it. You might find that your concern was misplaced.

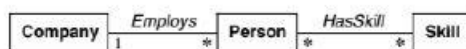
The design model builds on the analysis model. The analysis model captures the logic of a system, while the design model adds development details. You can optimize the inefficient but semantically correct analysis model to improve performance, but an optimized system is more obscure and less likely to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

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

### 1 Adding Redundant Associations for Efficient Access

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

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



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

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

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

### 2 Rearranging Execution Order for Efficiency

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

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

### 3 Saving Derived Values to Avoid Recomputation

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

There are three ways to handle updates.

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

10	<p><b>Explain allocation of subsystems</b></p> <p>You must allocate each concurrent subsystem to a hardware unit, either a general-purpose processor or a specialized functional unit as follows.</p> <ul style="list-style-type: none"> <li>■ Estimate performance needs and the resources needed to satisfy them.</li> <li>■ Choose hardware or software implementation for subsystems.</li> <li>■ Allocate software subsystems to processors to satisfy performance needs and minimize interprocessor communication.</li> <li>■ Determine the connectivity of the physical units that implement the subsystems.</li> </ul> <p><i>1 Estimating Hardware Resource Requirements</i></p> <p>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.</p> <p><i>2 Making Hardware-Software Trade-offs</i></p> <p>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</p>	10	CO4	L2
----	---	----	-----	----

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.

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.

### *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.

### *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 topology of repeated units usually has a regular pattern, such as a linear sequence, a matrix, a tree, or a star.
- **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.

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.