Fifth Semester MCA Degree Examination , Dec 2018/Jan-2019

# Object Oriented Modelling and Design Pattern

16MCA51 Max Marks : 80

1a. Outline the various Object Oriented Themes. 6M

**1 Abstraction**

Abstraction focus on essential aspects of an application while ignoring details. This means focusing on what an object is and does, before deciding how to implement it. Use of abstraction preserves the freedom to make decisions as long as possible by avoiding premature commitments to details. Most modem languages provide data abstraction, but inheritance and polymorphism add power. The ability to abstract is probably the most important skill required for 00 development.

**2 Encapsulation**

Encapsulation (also information hiding) separates the external aspects of an object, that are accessible to other objects, from the internal implementation details, that are hidden from other objects. Encapsulation prevents portions of a program from becoming so interdependent that a small change has massive ripple effects. You can change an object's implementation without affecting the applications that use it. You may want to change the implementation of an object to improve performance, fix a bug, consolidate code, or support porting. Encapsulation is not unique to 00 languages, but the ability to combine data structure and behaviour in a single entity makes encapsulation cleaner and more powerful things prior languages, such as Fortran, Cobol, and C.

**3. Combining Data and Behaviour**

The caller of an operation need not consider how many implementations exist. Operator polymorphism shifts the burden of deciding what implementation to use from ilk calling code to the class hierarchy. For example, non-OO code to display the contents of a window must distinguish the type of each figure, such as polygon, circle, or text, and call the appropriate procedure to display it. An 00 program would simply invoke the draw operation on each figure; each object implicitly decides which procedure to use, based on its class. Maintenance is easier, because the calling code need not be modified when a new class is added.

**4. Sharing**

00 techniques promote sharing at different levels. Inheritance of both data structure and behaviour lets subclasses share common code. This sharing via inheritance is one of the main advantages of 00 languages. More important than the savings in code is the conceptual clarity from recognizing that different operations are ail really the same thing. This reduces the number of distinct cases that you must understand and analyze.00 development not only lets you share information within an application, but also offers the prospect of reusing designs and code on future projects. 00 development provides the tools, such as abstraction, encapsulation, and inheritance, to build libraries of reusable components. Unfortunately, reuse has been overemphasized as a justification for 00 technology.

**5 Emphasis the Essence of an Object**

00 technology stresses what an object is, rather than how it is used. The uses of an object depend on the details of the application and often change during development. As

requirements evolve, the features supplied by an object are much more stable than the ways it is used, hence software systems built on object structure are more stable in the long run. 00 development places a greater emphasis on data structure and a lesser emphasis on procedure structure than functional-decomposition methodologies.

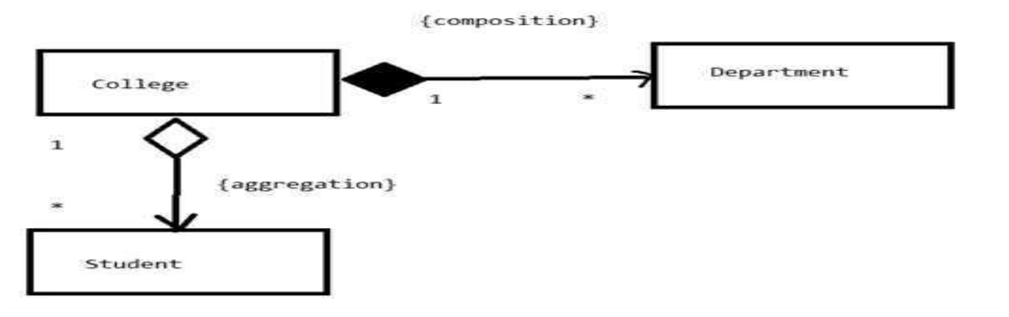
**6 Synergy**

Identity, classification, polymorphism, and inheritance characterize 00 languages. Each of concepts can be used in isolation, but together they complement each other. The benefits of an OQ-approach are greater than they might seem at first. The emphasis on the essential properties of an object forces the developer to think more carefully deeply about what an object IS and does. The resulting system tends to be cleaner, more general, and more robust than it would be if the emphasis were only on the use of data and operations.

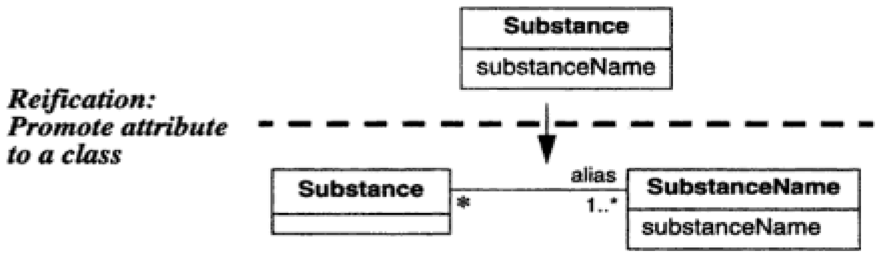
## 1b. Describe the following terms: 10M

i) Enumeration ii) Aggregation iii) Composition iv)Abstract class
v) Reification Give example for each.

**Enumeration:** An enumeration is a complete, ordered listing of all the items in a collection. The term is commonly used in mathematics and computer science to refer to a listing of all of the elements of a set. The precise requirements for an enumeration (for example, whether the set must be finite, or whether the list is allowed to contain repetitions) depend on the discipline of study and the context of a given problem.
Some sets can be enumerated by means of a natural ordering (such as 1, 2, 3, 4, ... for the set of positive integers), but in other cases it may be necessary to impose a (perhaps arbitrary) ordering. In some contexts, such as enumerative combinatory , the term enumeration is used more in the sense of counting – with emphasis on determination of the number of elements that a set contains, rather than the production of an explicit listing of those elements.
**Aggregation** is a stronger form of association. An association is a link connecting two classes. In UML, a link is placed between the "whole" and the "parts" classes with a diamond head attached to the "whole" class to indicate that this association is an aggregation . In aggregation, the part may have an independent lifecycle, it can exist independently. When the whole is destroyed the part may continue to exist.
Example :A car has many parts. A part can be removed from one car and installed into a different car. If we consider a salvage business, before a car is destroyed, they remove all saleable parts. Those parts will continue to exist after the car is destroyed.
**Composition** is really a strong form of aggregation. Composition has only one owner. Composition cannot exist independent of their owner. Composition lives or dies with their owner. It is represented using a filled diamond head. Composition is a stronger form of aggregation. The lifecycle of the part is strongly dependent on the lifecycle of the whole. When the whole is destroyed, the part is destroyed too.
Example : For example, a building has rooms. A room can exist only as part of a building. The room cannot be removed from one building and attached to a different one. When the building ceases to exist so do all rooms that are part of it.



**Abstract** class is a class that has no direct instances but whose descendant classes have direct instances. In UML notation an abstract class name is listed in an italic (or place the keyword {abstract} below or after the name). We can use abstract classes to define the methods that can be inherited by subclasses. Alternatively, an abstract class can define the signature for an operation with out supplying a corresponding method. We call this an abstract operation. Abstract operation defines the signature of an operation for which each concrete subclass must provide its own implementation.
**Reification** is the promotion of something that is not an object into an object and can be helpful for meta applications. It is useful to promote attributes, methods, constraints, and control information into objects so that we can describe and manipulate them as data.



## 2a. State the purpose of building a model. 4M

1. Testing a physical entity before building it: Medieval built scale models of Gothic Cathedrals to test the forces on the structures. Engineers test scale models of airplanes, cars and boats to improve their dynamics. Advancement in computation permit the simulation of many physical structures without the need to build physical models. These models are cheaper than building a complete system and enable early correction of flaws.
2. Communication with customers: Architects and product designers build models to show their customers. (note: mock-ups are demonstration products that imitate some or all of the external behaviour of a system).
3. Visualization: Storyboards of movies, TV shows and advertisements let writers see how their ideas flow.
4. Reduction of complexity: Models reduce complexity to understand directly by separating out a small number of important things to do with at a time.

## 2b. Briefly explain the class model, state model and interaction model. 4M

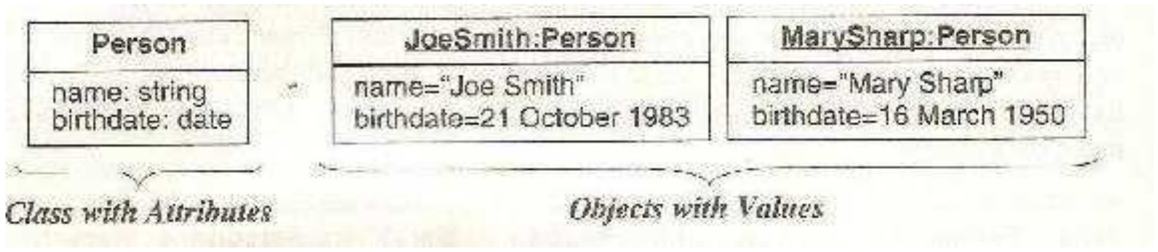
**Class Model**: represents the static, structural, “data” aspects of a system. It describes the structure of objects in a system- their identity, their relationships to other objects, their attributes, and their operations. Goal in constructing class model is to capture those

concepts from the real world that are important to an application. Class diagrams express the class model.
**State Model**: represents the temporal, behavioural, “control” aspects of a system. State model describes those aspects of objects concerned with time and the sequencing of operations – events that mark changes, states that define the context for events, and the organization of events and states. State diagram express the state model. Each state diagram shows the state and event sequences permitted in a system for one class of objects. Actions and events in a state diagram become operations on objects in the class model. References between state diagrams become interactions in the interaction model.
**Interaction model** – represents the collaboration of individual objects, the “interaction” aspects of a system. Interaction model describes interactions between objects – how individual objects collaborate to achieve the behaviour of the system as a whole. The state and interaction models describe different aspects of behaviour, and you need both to describe behaviour fully. Use cases, sequence diagrams and activity diagrams document the interaction model.

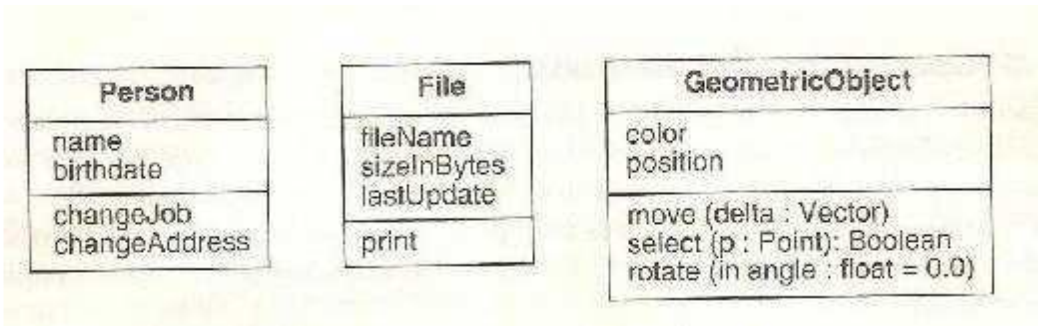
## 2c. Explain with examples : 8M

i) Value and attribute ii) Operation and Method
iii) Link and Association iv) Qualified Association

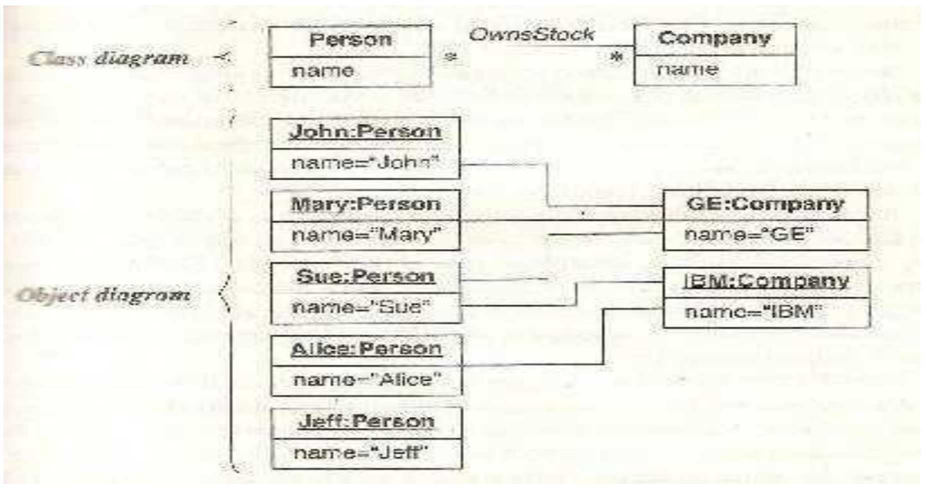
**Value and attribute:** Value is a piece of data. Attribute is a named property of a class that describes a value held by each object of the class. Following analogy holds: Object is to class as value is to attribute.
Example: Attributes of Person object .Name, birthdate, weight.
Values: JoeSmith, 21 October 1983, 64. (Of person object).



In UML List attributes in the 2nd compartment of the class box. Optional details (like default value) may follow each attribute. A colon precedes the type, an equal sign precedes default value. Show attribute name in regular face, left align the name in the box and use small case for the first letter. we may also include attribute values in the 2nd compartment of object boxes with same conventions.
**Operation and Method :** An **operation** is a function or procedure that maybe applied to or by objects in a class. E.g. Hire, fire and pay dividend are operations on Class Company. Open, close, hide and redisplay are operations on class window. All objects in a class share the same operations. A **method** is the implementation of an operation for a class.
E.g. In class file, print is an operation you could implement different methods to print files. Same operation may apply to many different classes. Such an operation is polymorphic.



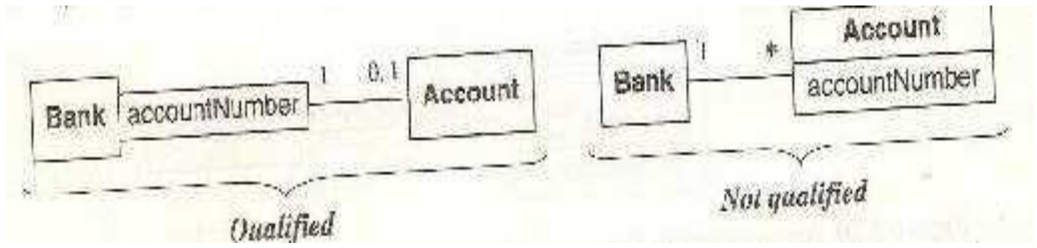
In UML List operations in 3rd compartment of class box. List operation name in regular face, left align and use lower case for first letter. Optional details like argument list and return type may follow each operation name. Parenthesis enclose an argument list, commas separate the arguments. A colon precedes the result type. We do not list operations for objects, because they do not vary among objects of same class.
**Links and associations:** Links and associations are the means for establishing relationships among objects and classes. **Link** is a physical or conceptual connection among objects. E.g. JoeSmith *WorksFor* Simplex Company. We define a link as a tuple – that is, a list of objects. Link is an instance of an association. A**ssociation** is a description of a group of links with common structure and common semantics. E.g. a person *WorksFor* a company. Association describes a set of potential links in the same way that a class describes a set of potential objects.
The below figure shows many-to-many association (model for a financial application).



In UML ,Link is a line between objects; a line may consist of several line segments. If the link has the name, it is underlined. Association connects related classes and is also denoted by a line. Show link and association names in italics. Association name is optional, if the model is unambiguous. Ambiguity arises when a model has multiple associations among same classes. Developers often implement associations in programming languages as references from one object to another. A reference is an attribute in one object that refers to another object.

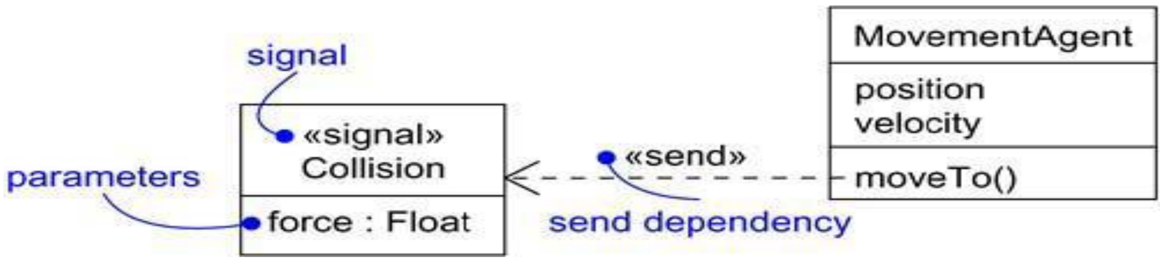
**Qualified Association** is an association in which an attribute called the qualifier disambiguates the objects for a "many" association ends. It is possible to define qualifiers

for one-to-many and many-to-many associations. A qualifier selects among the target objects, reducing the effective multiplicity from "many" to "one".
**Ex 1:** qualifier for associations with one to many multiplicity. A bank services multiple accounts. An account belongs to single bank. Within the context of a bank, the Account Number specifies a unique account. Bank and account are classes, and Account Number is a qualifier. Qualification reduces effective multiplicity of this association from one-to-many to one-to-one. Qualification increases the precision of a model.

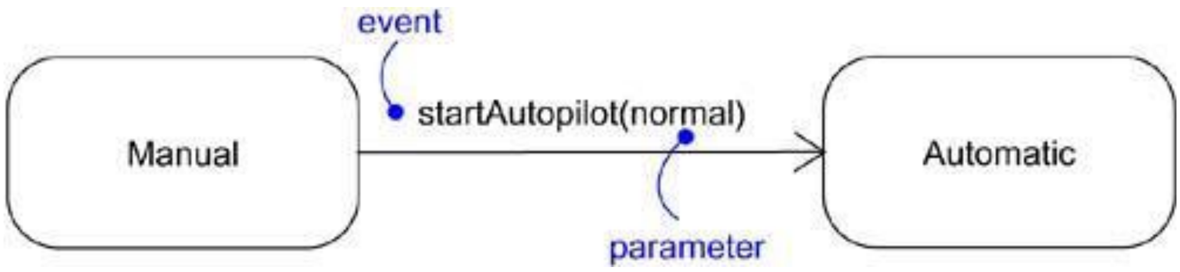


## 3a. What is an event? With example describe the different types of events in state modelling. 8M

An **event** is an occurrence at a point in time, such as user depresses left button of mouse. An event happens instantaneously with regard to the time scale of an application. Events cause state changes which is shown in State Diagrams
**Signal Event :**A signal event represents a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another. Exceptions are an example of internal signal**.** Signal event is an asynchronous event**.** Signal events may have instances, generalization relationships, attributes and operations. Attributes of a signal serve as its parameters**.** A signal event may be sent as the action of a state transition in a state machine or the sending of a message in an interaction
Signals are modeled as stereotyped classes and the relationship between an operation and the events by using a dependency relationship, stereotyped as send.



**Call Event**: A call event represents the dispatch of an operation . Call event is a synchronous event.



**Time and Change Events**: A **time event** is an event that represents the passage of time. It is modeled by using the keyword 'after' followed by some expression that evaluates to a period of time which can be simple or complex.

- when (room temperature < heating set point )
- when (room temperature > cooling set point )
- when (battery power < lower limit )
- when (tire pressure < minimum pressure )

A **change event** is an event that represents a change in state or the satisfaction of some condition. It is modeled by using the keyword 'when' followed by some Boolean expression.

- when (date = jan 1, 2000 )
- after (10 seconds )

## 3b. Describe sequence diagram with active objects, passive objects and transient objects. 8M

With procedural code all objects are not constantly active. Most objects are passive and do not have their own threads of control. A passive object is not activated until it has been called. Once the execution of an operation completes and control returns to the caller, the passive object becomes inactive.

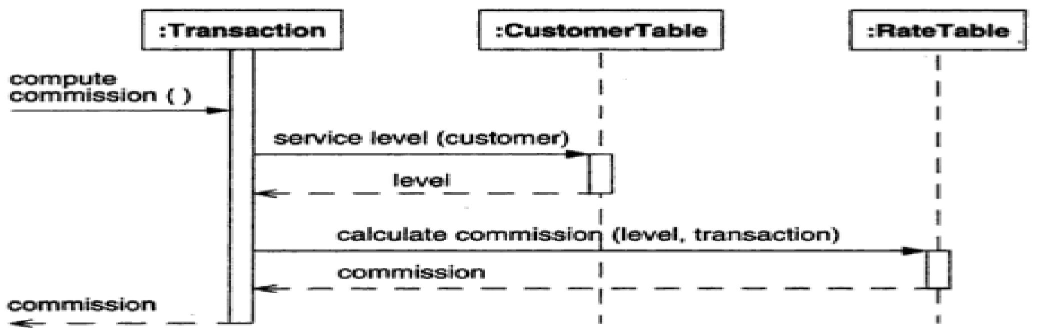


**Figure 8.5** **Sequence diagram with passive objects.** Sequence diagrams can show the implementation of operations.

The UML shows the period of time for an object's execution as a thin rectangle. This is called the ***activation*** or ***focus of control***. An activation shows the time period during which a call of a method is being processed, including the time when the called method has invoked another operation. The period of time when an object exists but is not active is shown as a dashed line. The entire period during which the object exists is called the ***lifeline***, as it shows the lifetime of the object.

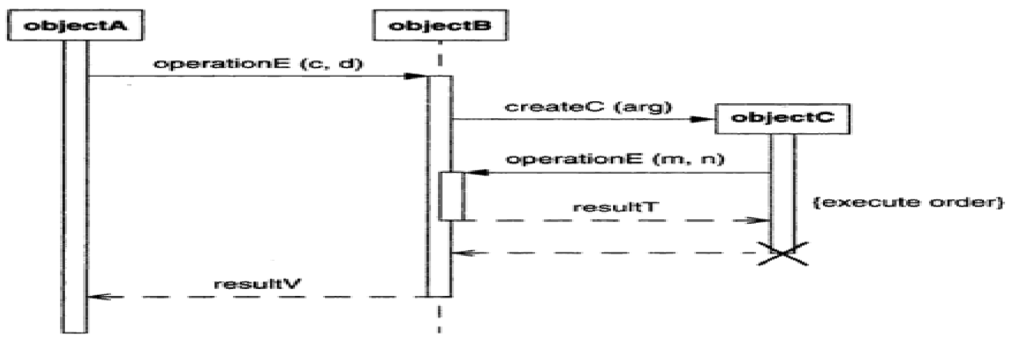


**Figure 8.6** **Sequence diagram with a transient object.** Many applications have a mix of active and passive objects. They create and destroy objects.

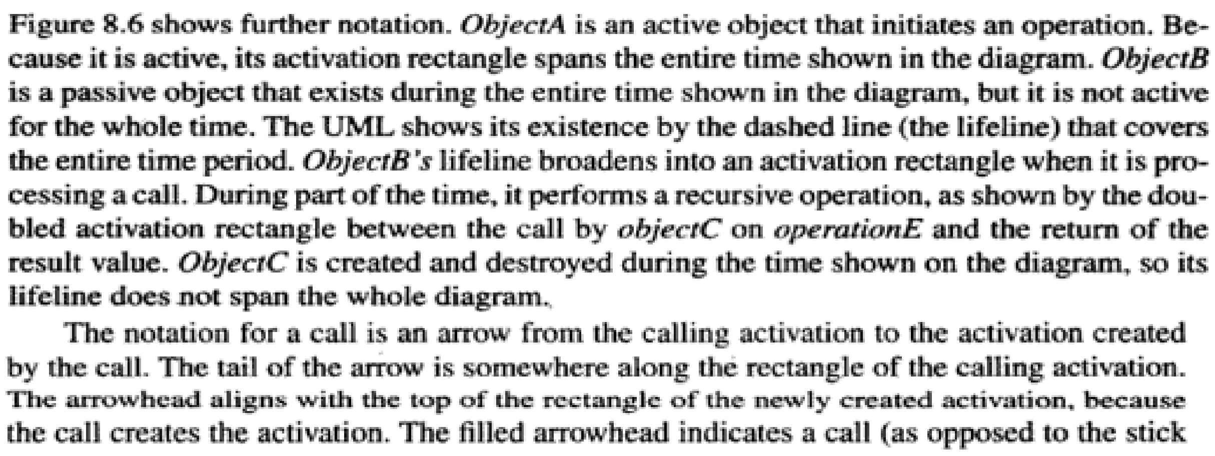
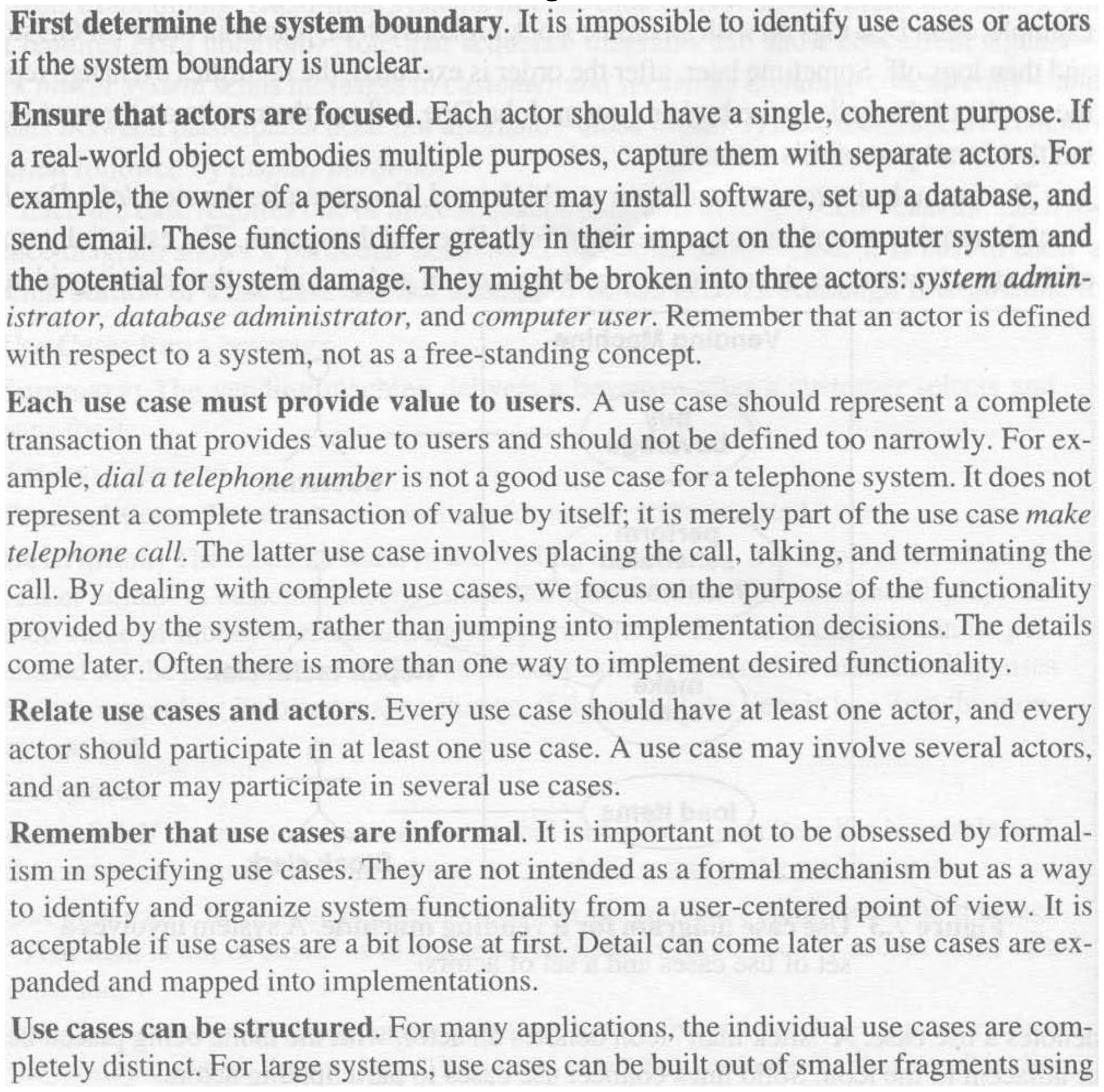


Figure 8.6 shows further notation. *ObjectA* is an active object that initiates an operation. Because it is active, its activation rectangle spans the entire time shown in the diagram. *ObjectB* is a passive object that exists during the entire time shown in the diagram, but it is not active for the whole time. The UML shows its existence by the dashed line (the lifeline) that covers the entire time period. *ObjectB's* lifeline broadens into an activation rectangle when it is processing a call. During part of the time, it performs a recursive operation, as shown by the doubled activation rectangle between the call by *objectC* on *operationE* and the return of the result value. *ObjectC* is created and destroyed during the time shown on the diagram, so its lifeline does not span the whole diagram.

The notation for a call is an arrow from the calling activation to the activation created by the call. The tail of the arrow is somewhere along the rectangle of the calling activation. The arrowhead aligns with the top of the rectangle of the newly created activation, because the call creates the activation. The filled arrowhead indicates a call (as opposed to the stick

4a. What are the guide lines to be followed while drawing use case diagram? Draw the use case model for vending machine. 8M


**First determine the system boundary**. It is impossible to identify use cases or actors if the system boundary is unclear.

**Ensure that actors are focused**. Each actor should have a single, coherent purpose. If a real-world object embodies multiple purposes, capture them with separate actors. For example, the owner of a personal computer may install software, set up a database, and send email. These functions differ greatly in their impact on the computer system and the potential for system damage. They might be broken into three actors: *system administrator, database administrator,* and *computer user*. Remember that an actor is defined with respect to a system, not as a free-standing concept.

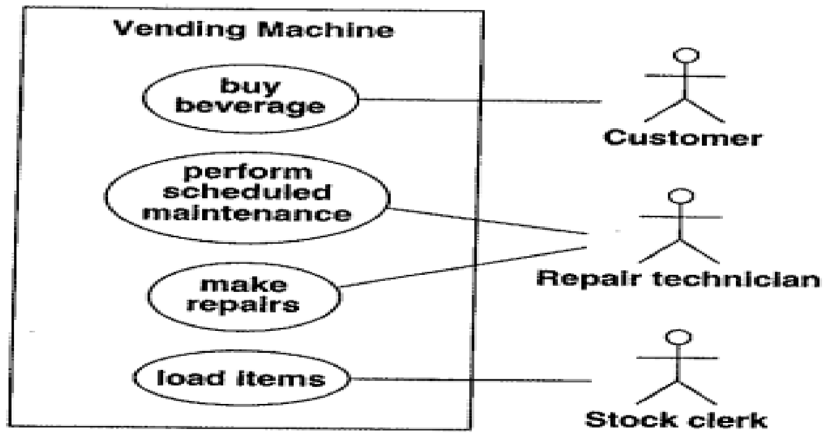
**Each use case must provide value to users**. A use case should represent a complete transaction that provides value to users and should not be defined too narrowly. For example, *dial a telephone number* is not a good use case for a telephone system. It does not represent a complete transaction of value by itself; it is merely part of the use case *make telephone call*. The latter use case involves placing the call, talking, and terminating the call. By dealing with complete use cases, we focus on the purpose of the functionality provided by the system, rather than jumping into implementation decisions. The details come later. Often there is more than one way to implement desired functionality.

**Relate use cases and actors**. Every use case should have at least one actor, and every actor should participate in at least one use case. A use case may involve several actors, and an actor may participate in several use cases.

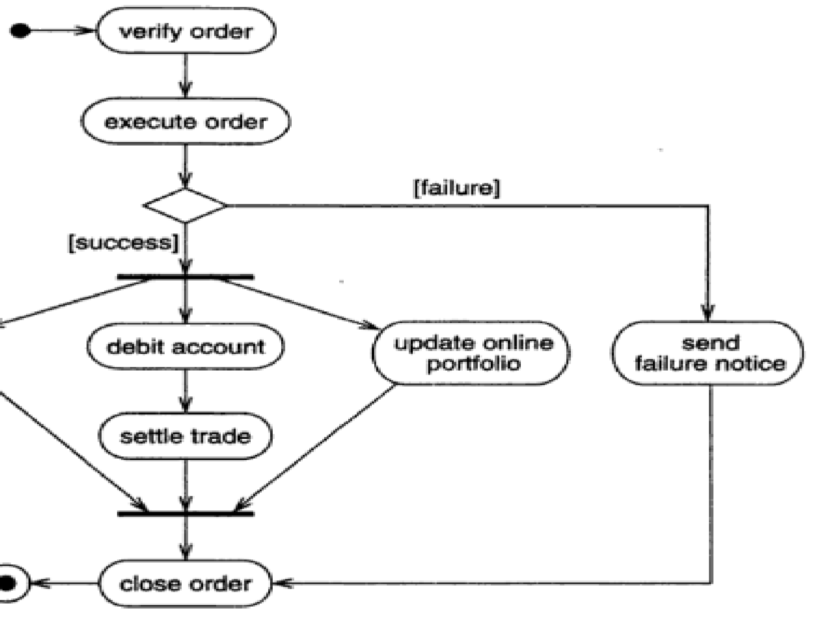
**Remember that use cases are informal**. It is important not to be obsessed by formalism in specifying use cases. They are not intended as a formal mechanism but as a way to identify and organize system functionality from a user-centered point of view. It is acceptable if use cases are a bit loose at first. Detail can come later as use cases are expanded and mapped into implementations.

**Use cases can be structured**. For many applications, the individual use cases are completely distinct. For large systems, use cases can be built out of smaller fragments using

relationships.



4b. Discuss the use of branching and concurrency in activity diagram. 8M



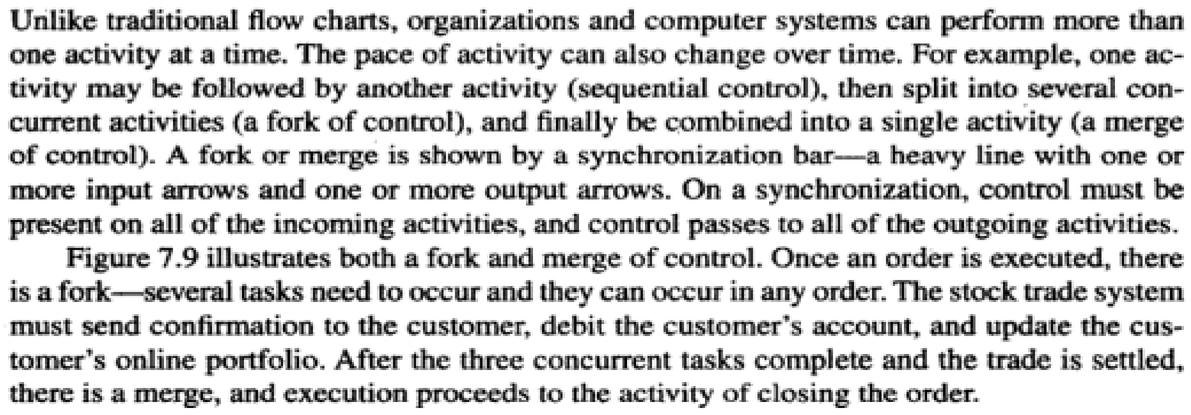
## Branching

If there is more than one successor to an activity, each arrow may be labeled with a condition in square brackets, for example, *[failure]*. All subsequent conditions are tested when an activity completes. If one condition is satisfied, its arrow indicates the next activity to perform. If no condition is satisfied, the diagram is badly formed and the system will hang unless it is interrupted at some higher level. To avoid this danger, you can use the ***else*** condition; it is satisfied in case no other condition is satisfied. If multiple conditions are satisfied, only one successor activity executes, but there is no guarantee which one it will be. Sometimes this kind of nondeterminism is desirable, but often it indicates an error, so the modeler should determine whether any overlap of conditions can occur and whether it is correct.

As a notational convenience, a diamond shows a branch into multiple successors, but it means the same thing as arrows leaving an activity symbol directly. In Figure 7.9 the diamond has one incoming arrow and two outgoing arrows, each with a condition. A particular execution chooses only one path of control.

If several arrows enter an activity, the alternate execution paths merge. Alternatively, several arrows may enter a diamond and one may exit to indicate a merge.

## Concurrent Activities



Unlike traditional flow charts, organizations and computer systems can perform more than one activity at a time. The pace of activity can also change over time. For example, one activity may be followed by another activity (sequential control), then split into several concurrent activities (a fork of control), and finally be combined into a single activity (a merge of control). A fork or merge is shown by a synchronization bar—a heavy line with one or more input arrows and one or more output arrows. On a synchronization, control must be present on all of the incoming activities, and control passes to all of the outgoing activities.

Figure 7.9 illustrates both a fork and merge of control. Once an order is executed, there is a fork—several tasks need to occur and they can occur in any order. The stock trade system must send confirmation to the customer, debit the customer's account, and update the customer's online portfolio. After the three concurrent tasks complete and the trade is settled, there is a merge, and execution proceeds to the activity of closing the order.

## 5a. Explain the procedure to be followed to construct a domain class model. 10M

Find classes
Prepare a data dictionary
Find associations
Find attributes of objects and links
Organize and simplify classes using inheritance
Verify that access paths exists for likely queries
Iterate and refine the model
Reconsider the level of abstraction
Group classes into packages.

Finding Classes:- 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 ,Performances and Theater
Keeping the right classes:
We need to discard the unnecessary and incorrect classes:
Redundant classes: if two classes express same
Irrelevant classes: It has little or nothing to do with the problem
Vague class: too broad in scope
Attributes: describe individual objects
Operations
Roles
Implementation constructs: CPU, subroutine, process and algorithm
Derived classes: omit class that can be derived from other class

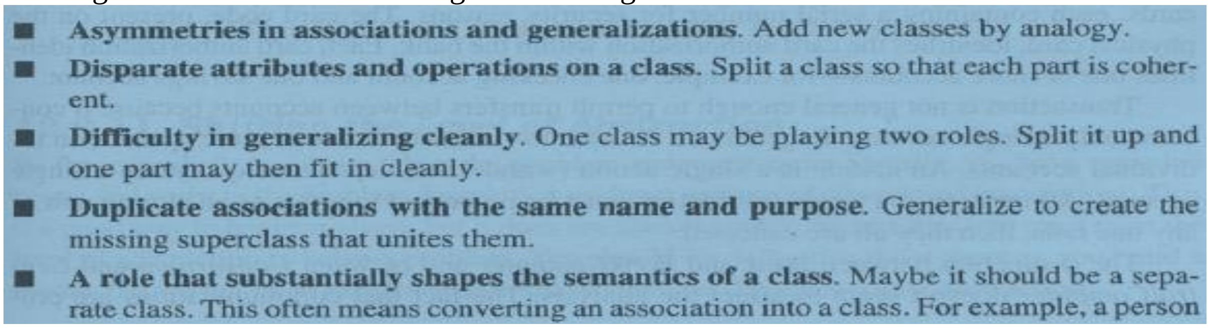
Prepare data dictionary: Information regarding the data is maintained
Finding Associations
Keeping the right associations:
1.Associations between eliminated classes
2.Irrelevant or implementation associations
3.Actions
4.Ternary association
5.Derived Association
6.Misnamed associations
7.Association end names

8.Qualified associations
9.Multiplicity
10.Missing Associations
11.Aggregation

Finding attributes:
Keeping the right attributes:
Refining with inheritance :

1. Bottom-up generalization
2. Top-down generalization
3. Generalization vs enumeration
4. Multiple inheritance
5. Similar associations.
6. Adjusting the inheritance level

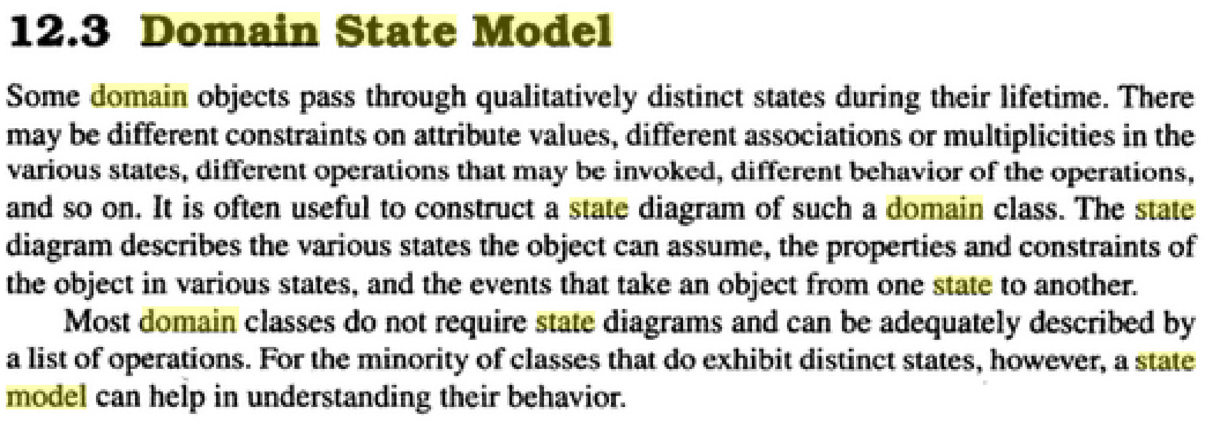
Testing Access paths
Iterating a Class Model : Several signs of missing classes



- **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

Shifting the level of abstraction
Grouping Classes into packages.

5b. Write and explain the steps performed in constructing a domain state model. 6M



## 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.

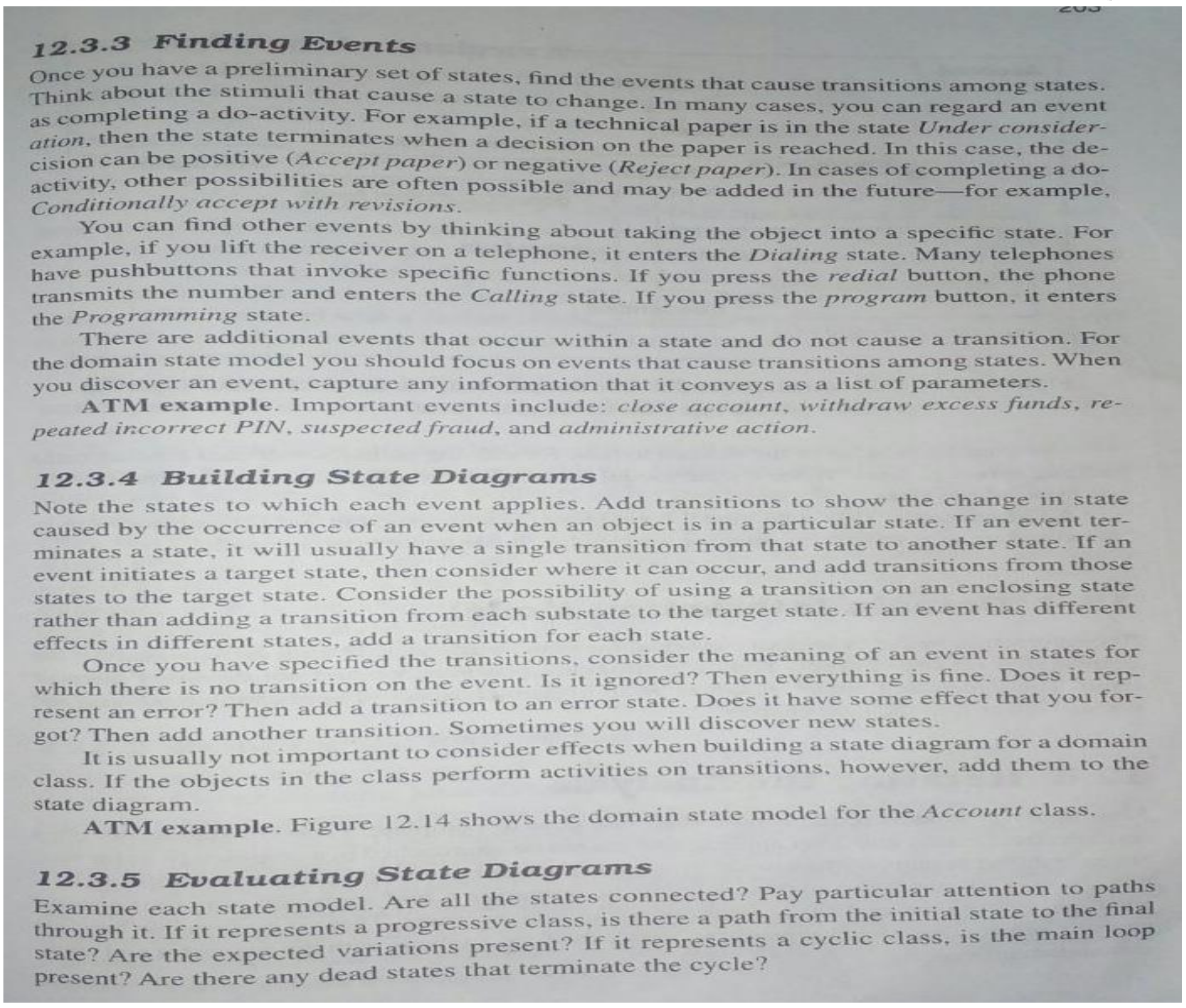
### 12.3.2 Finding States

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?

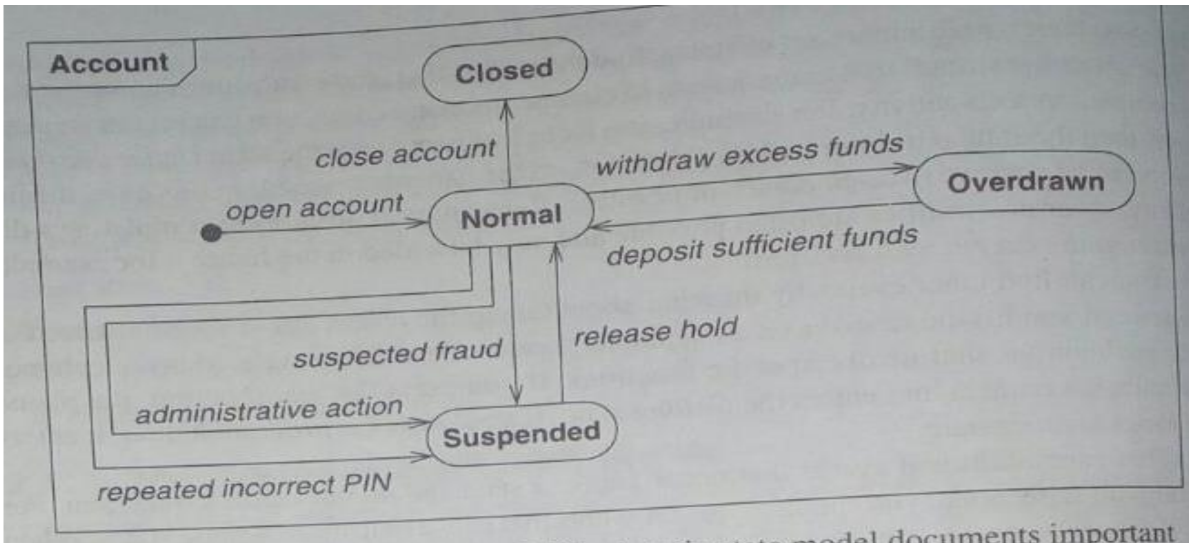


**Figure 12.14 Domain state model**. The domain state model documents important classes that change state in the real world.

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.

simplistic but we are satisfied with it.

6a. Describe the steps for constructing application interaction model.　　10M

## 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.

- **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.

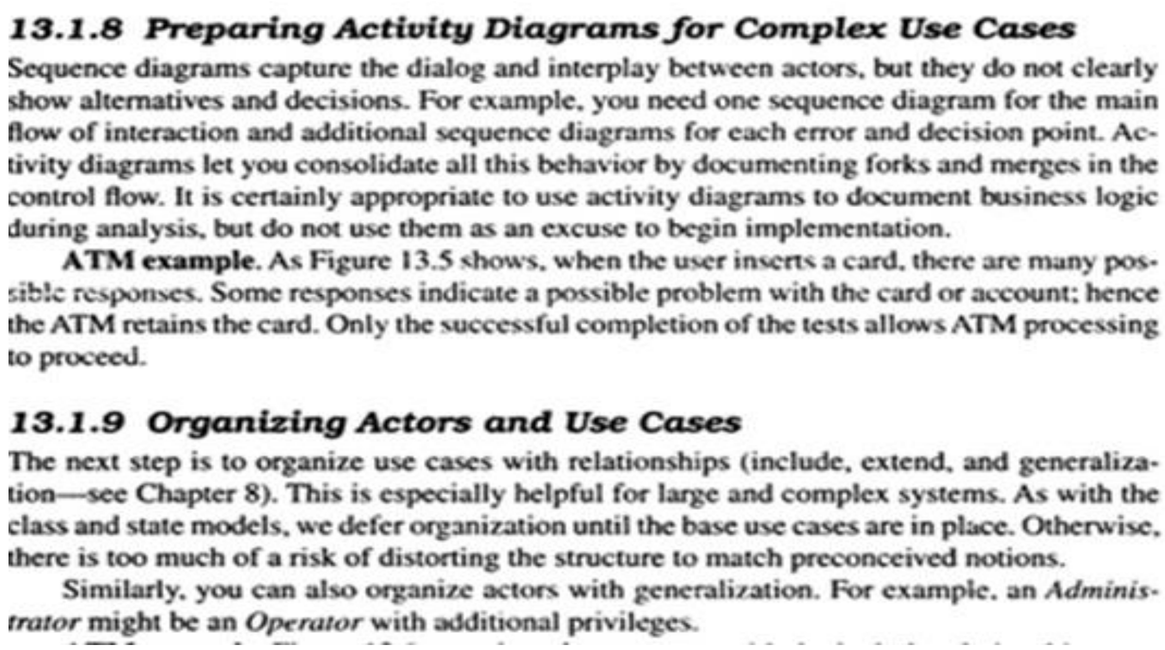
**ATM example.** Some variations and exceptions follow. We could prepare scenarios for each of these but will not go through the details here. (See the exercises.)

- 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



### 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.

10. Checking against the domain class model.

6B. Explain the steps for constructing application state model. 6M

The application state model focuses on application classes

· Augments the domain state model

Application State Model- steps

1. Determine Application Classes with States
2. Find events
3. Build state diagrams
4. Check against other state diagrams
5. Check against the class model
6. Check against the interaction model

**1. Determine Application Classes with States**

· Good candidates for state models

⁻ User interface classes

⁻ Controller classes

· ATM example

⁻ The controllers have states that will elaborate.

**2. Find events**

· Study scenarios and extract events.

· In domain model

⁻ Find states and then find events

· In application model

⁻ Find events first, and then find states

· ATM example

⁻ Revisit the scenarios, some events are:

⁻ Insert card, enter password, end session and take card.

**3. Building State Diagrams**

· To build a state diagram for each application class with temporal behaviour.
Initial state diagram
⁃ Choose one of these classes and consider a sequence 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.
· Find loops
⁃ If a sequence of events can be repeated indefinitely, then they form a loop.
· Merge other sequence diagrams into the state diagram.
· After normal events have been considered, add variation and exception cases.
· The state diagram of a class is finished when the diagram covers all scenarios and the diagram handles all events that can affect a state.
· Identify the classes with multiple states
· Study the interaction scenarios to find events for these classes
· Reconcile the various scenarios
· Detect overlap and closure of loops

**4. check against other state diagrams**

· Every event should have a sender and a receiver.
· 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.
· Make sure that corresponding events on different state diagrams are consistent.
· ATM example
· The SessionController initiates the TransactionController,
· The termination of the TransactionController causes the SessionController to resume.

**5. Check against the class model**

· 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.

**6. Check against the interaction model**

• Check the state model against the scenarios of the interaction model.
• Simulate each behaviour sequence by hand and verify the state diagrams.
• Take the state model and trace out legitimate paths.

7a. With the help of architecture of ATM system describe how to break a system into subsystems in system design. 12M

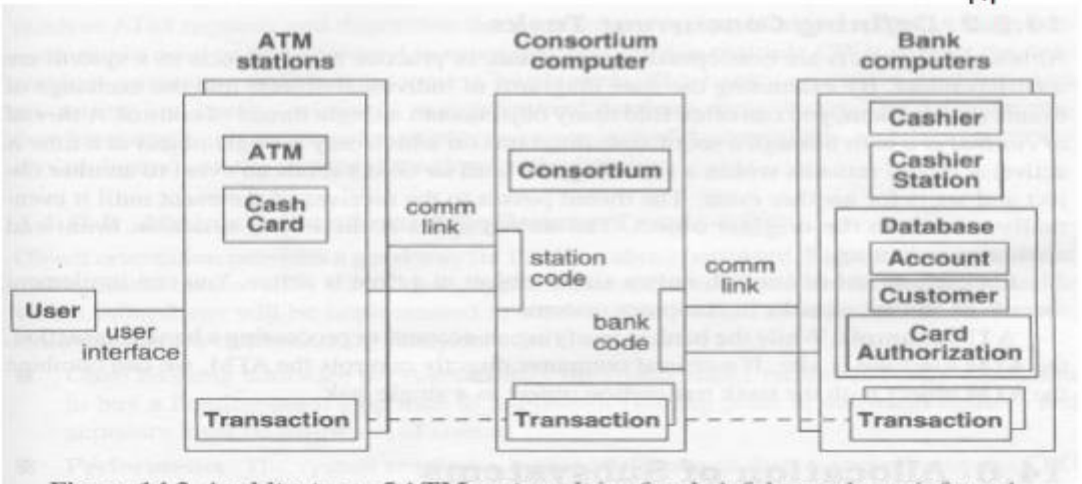


**Figure 14.2 Architecture of ATM system.** It is often helpful to make an informal diagram showing the organization of a system into subsystems.

Divide the system into pieces . Each piece of a system is called subsystem. A subsystem is a group of classes, associations, operations, events, and constrains. A subsystem is usually identified by the services it provides. Each subsystem has a well-defined interface to the rest of the system.

The relation between two subsystems can be – Client-server relationship – Peer-to-peer relationship

The decomposition of systems into subsystems is organized as a sequence of

– Horizontal layers,
– Vertical partitions, or
– Combination of layers and partitions.

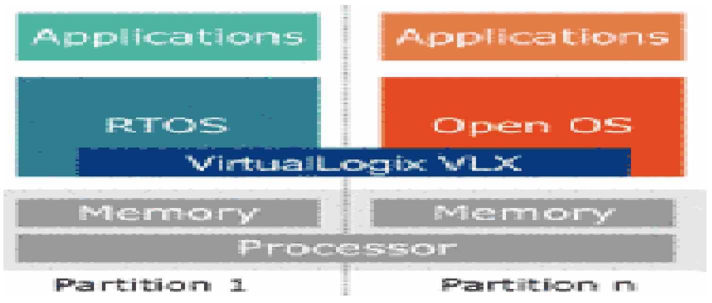
1. **LAYERS :** Each built in terms of the ones below it. The objects in each layer can be independent.

   E.g. A client-server relationship. Problem statement specifies only the top and bottom layers:
   – The top is the desired system.
   – The bottom is the available resources.
   The intermediate layers is than introduced. • Two forms of layered architectures:
   – Closed architecture • Each layer is built only in terms of the immediate lower layer.
   – Open architecture • A layer can use features on any lower layer to any depth
   .• Do not observe the principle of information hiding.

2. **PARTITIONS**

   Vertically divided into several subsystems .Independent or weakly coupled Each providing one kind of service. E.g. A computer operating system includes
   – File system
   – Process control
   – Virtual memory management

3. **COMBINATION OF LAYERS AND PARTITIONS**



7b. Discuss about making a reuse plan in system design 4M

Two aspects of reuse:

– Using existing things

– Creating reusable new things

Reusable things include: Models, Libraries, Frameworks ,Patterns

**1. Libraries** A library is a collection of classes that are useful in many contexts.

Qualities of "Good" class libraries:

– *Coherence* – well focused themes

– *Completeness* – provide complete behaviour

– *Consistency* - polymorphic operations should have consistent names and signatures across classes.

– *Efficiency* – provide alternative implementations of algorithms.

– *Extensibility* – define subclasses for library classes

– *Genericity* – parameterized class definitions

Problems limit the reuse ability:

– Argument validation • Validate arguments by collection or by individual

– Error Handling • Error codes or errors

– Control paradigms • Event-driven or procedure-driven control

– Group operations

– Garbage collection

– Name collisions

**2. Frameworks** A framework is a skeletal structure of a program that must be elaborated to build a complete application. Frameworks class libraries are typically application specific and not suitable for general use.

**3. Patterns** A pattern is a proven solution to a general problem. There are patterns for analysis, architecture, design, and implementation.

Benefits of Pattern

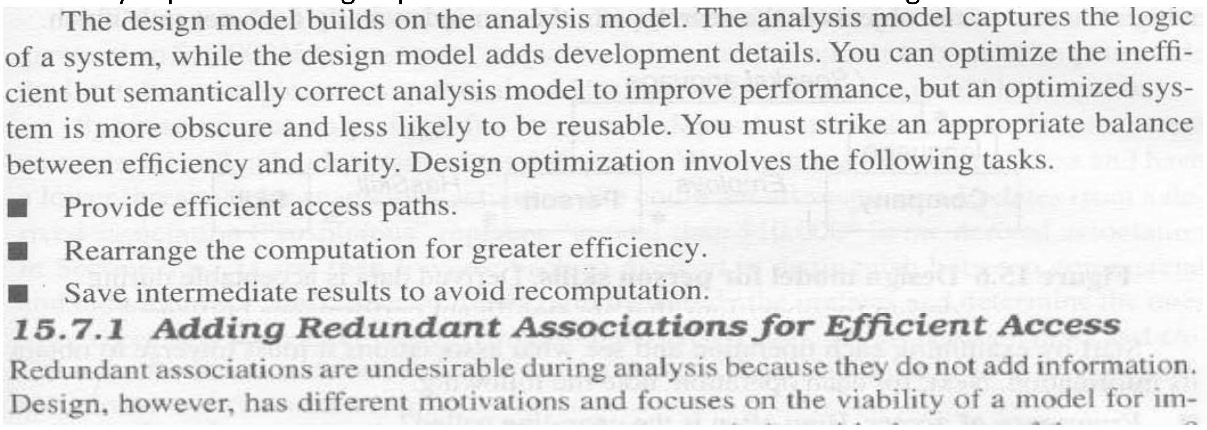
1. A pattern is considered by others and has already been applied to past problems .
2. When you use patterns, you tap into a language that is familiar to many developers
3. Patterns are prototypical model fragments that distil some of the knowledge of experts.

**Pattern vs. Framework**

1.A pattern is typically a small number of classes and relationships.

2 A framework is much broader in scope and covers an entire subsystem or application.

8a. Briefly explain the design optimization with reference to class design. 8M



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

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

### 15.7.1 *Adding Redundant Associations for Efficient Access*

Redundant associations are undesirable during analysis because they do not add information. Design, however, has different motivations and focuses on the viability of a model for im-

implementation.

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.

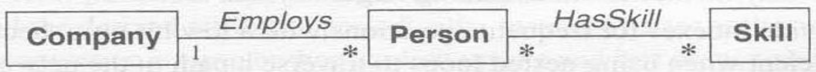


**Figure 15.5 Analysis model for person skills.** Derived data is undesirable during analysis because it does not add information.

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

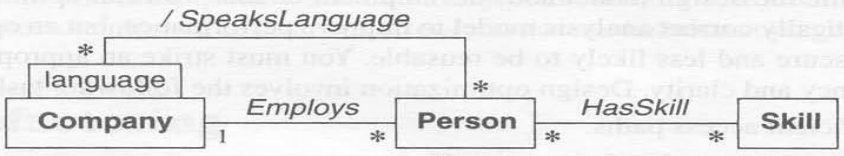


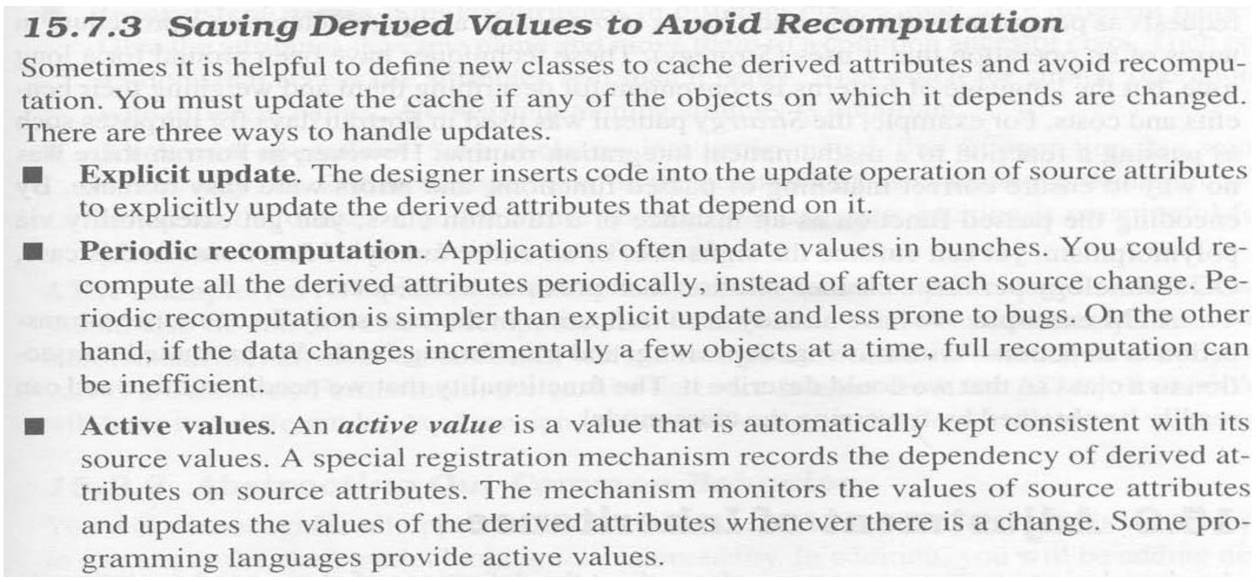
**Figure 15.6 Design model for person skills.** Derived data is acceptable during design for operations that are significant performance bottlenecks.

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

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

### 15.7.2 Rearranging Execution Order for Efficiency

After adjusting the structure of the class model to optimize frequent traversals, the next thing to optimize is the algorithm itself. One key to algorithm optimization is to eliminate dead paths as early as possible. For example, suppose an application must find all employees who speak both Japanese and French. Suppose 5 employees speak Japanese and 100 speak French; it is better to test and find the Japanese speakers first, then test if they speak French. In general, it pays to narrow the search as soon as possible. Sometimes you must invert the execution order of a loop from the original specification.



### 15.7.3 *Saving Derived Values to Avoid Recomputation*

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

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

8b. Explain the steps to be performed in designing algorithm for class design. 8M

Formulate an algorithm for each operation. The analysis specification tells what the operation does for its Clients. The algorithm show how it is done

**1. Choosing algorithms (Choose algorithms that minimize the cost of implementing operations)**

When efficiency is not an issue, you should use simple algorithms. Typically, 20% of the operations consume 80% of execution time. Considerations for choosing alternative algorithms

a. Computational complexity

b. Ease of implementation and understand ability

c. Flexibility

Simple but inefficient. Complex efficient

2. **Choosing Data Structures (select data structures appropriate to the algorithm)**

a. Algorithms require data structures on which to work.

b. They organize information in a form convenient for algorithms.

c. Many of these data structures are instances of container classes.

d. Such as arrays, lists, queues, stacks, set…etc.

**3. Defining New Internal Classes and Operations**

a. To invent new, low-level operations during the decomposition of high-level operations.

b. The expansion of algorithms may lead you to create new classes of objects to hold intermediate results.

ATM Example:

i. Process transaction uses case involves a customer receipt.

ii. A Receipt class is added.

**4. Assigning Operations to Classes (assign operations to appropriate classes)**

a. How do you decide what class owns an operation?

i. Receiver of action: To associate the operation with the target of operation, rather than the initiator.

ii Query vs. update : The object that is changed is the target of the operation

iii Focal class : Class centrally located in a star is the operation's target

iv. Analogy to real world

9a. Define pattern. Explain the pattern description template. 8M

Abstracting from specific problem-solution pairs and distilling out common factors leads to patterns.
Each pattern is a three part rule:

• a relation between a certain context

• a problem

• and a solution

Pattern Categories
Some patterns help in structuring a software system into subsystems. Other patterns support the refinement of subsystems and components, or of the relationships between them. Further patterns help in implementing particular design aspects in a specific programming language.
Patterns also range from domain-independent ones, such as those for decoupling interacting components, to patterns addressing domain-specific.
We group patterns into three categories:
Architectural patterns
Design patterns
Idioms

**Name** The name and a short summary of the pattern.
**Also Known As** Other names for the pattern, if any are known.
**Example** A real-world example demonstrating the existence of the problem and the need for the pattern.
**Context** The situations in which the pattern may apply
**Problem** The problem the pattern addresses, including a discussion of its associated forces.
**Solution** The fundamental solution principle underlying the pattern.
**Structure** A detailed specification of the structural aspects of the pattern.
**Dynamics** Typica1 scenarios describing the run-time behavior of the pattern.
**Implementation :** Guidelines for implementing the pattern. These are only a suggestion, not an immutable rule. You should adapt the implementation to meet your needs, by adding different, extra, or more detailed steps, or by re-ordering the steps.
**Example Resolved :** Discussion of any important aspects for resolving the example that are not yet covered in the Solution, Structure, Dynamics and implementation sections.
**Variants :** A brief description of variants or specializations of a pattern.
**Known Uses :** Examples of the use of the pattern, taken from existing systems.
**Consequences :** The benefits the pattern provides, and any potential liabilities.
**See Also :** References to patterns that solve similar problems, and to patterns
that help us refine the pattern we are describing.

9b. Briefly explain Forwarder-Receiver pattern. 8M

Forwarder-Receiver design pattern Provides transparent inter process communication for software systems with peer-to-peer interaction model. It introduces forwarders and receivers to decouple peers from the underlying communication mechanisms.

**Example:** The company Dwarf Ware offers applications for the management of the computer networks. System consists of agent processes written in Java that run on each available network node.

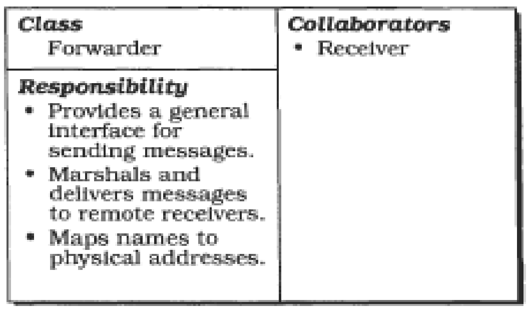
**Context** – peer-to-peer communication

**Problem forces –**

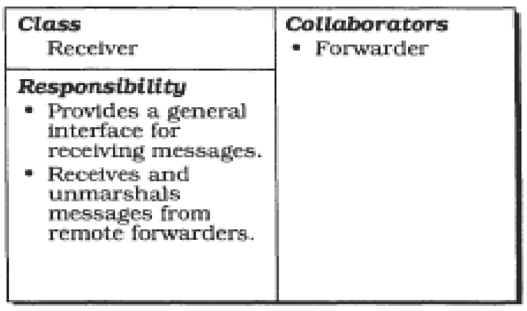
1.The system should allow the exchangeability of the communication mechanisms.
2.The co-operation of components follows a peer-to-peer model, in which a sender only needs to know names of its receivers.
3.The communication between peers should not have a major impact on performance.

**Solution** – peers may act as clients or servers. Therefore the details of the underlying IPC mechanisms for sending or receiving messages are hidden from peers by encapsulating all system-specific functionality into separate components. system specific functionalities are the mapping of names to physical locations, the establishment of communication channels and marshalling and un marshalling messages.

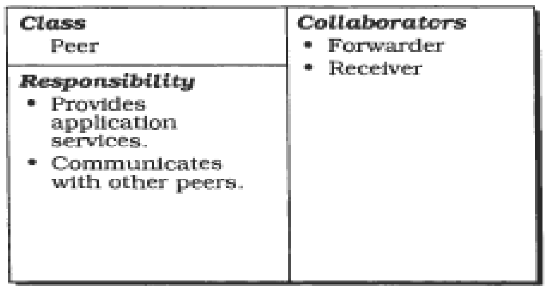
**Structure**



| *Class* Forwarder | *Collaborators* |
|---|---|
| *Responsibility*<br>• Provides a general interface for sending messages.<br>• Marshals and delivers messages to remote receivers.<br>• Maps names to physical addresses. | • Receiver |

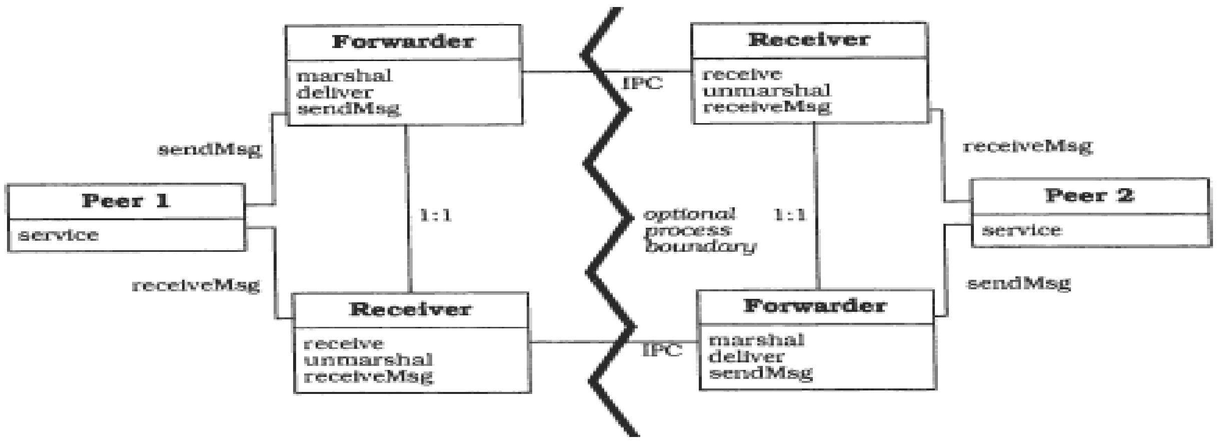


| *Class* Receiver | *Collaborators* |
|---|---|
| *Responsibility*<br>• Provides a general interface for receiving messages.<br>• Receives and unmarshals messages from remote forwarders. | • Forwarder |



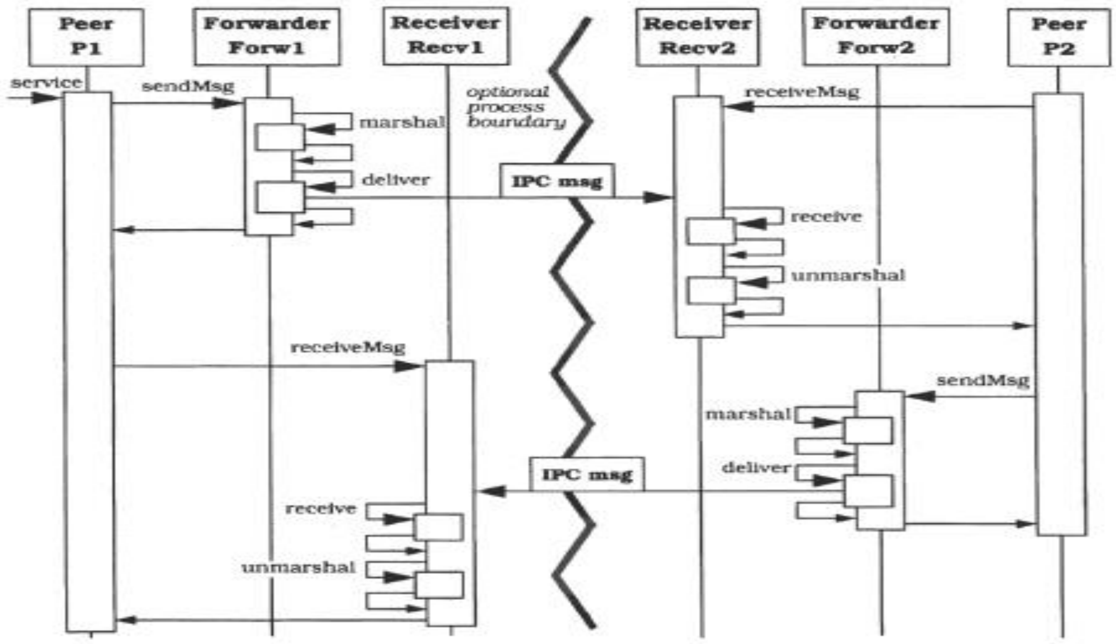
| *Class* Peer | *Collaborators* |
|---|---|
| *Responsibility*<br>• Provides application services.<br>• Communicates with other peers. | • Forwarder<br>• Receiver |

The static relationships **in** the Forwarder-Receiver design pattern are shown in the diagram below.
To send a message to a remote peer, the peer invokes the method **sendMsg** of its forwarder, passing the message as an argument. The method **sendMsg** must convert messages to a format that the underlying IPC mechanism understands. For this purpose, it calls **marshal. sendMsg** uses **deliver** to transmit the IPC message data to a remote receiver. When the peer wants to receive a message from a remote peer, it invokes the **receiveMsg** method of its receiver, and the message is returned. **receiveMsg** invokes **receive,** which uses the functionality
of the underlying IPC mechanism to receive IPC messages. After message reception receiveMsg calls unmarshal to convert **IPC** messages to a format that the peer understands.



**Dynamics**

The following scenario illustrates a typical example of the use of a Forwarder-Receiver structure.
Two peers pl and p2 communicate with each other. For this purpose, PI uses a forwarder forw1 and a
receiver Recv2 handles all message transfers with a forwarder Forw2 and a receiver Recv2 :PI requests a service from a remote peer2.Fo r this purpose, it sends the request to its forwarder forw1 and specifies the name of the recipient.
Forw1 determines the physical location of the remote peer and marshals the message.Forw1 delivers the message to the remote receiver Recv2.At some earlier time P2 **has** requested its receiver Recv2 to wait for an incoming request. Now, Recv2 receives the message arriving from Forw1.
Recv2 unmarshals the message and forwards it to its peer2 .Meanwhile. pl calls Its receiver Recvl to wait for a response.
P2 performs the requested service, and sends the result and the name of the recipient pl to the forwarder forw2.T he forwarder marshals the result and delivers it Recv l . Recvl receives the response from P2, unmarshals it and delivers it to P1.



10a. Explain the structure of Client-Dispatcher server pattern. 8M

**Intent :**

The Client-Dispatcher-Server design pattern introduces an intermediate layer between clients and servers, the dispatcher component. It provides location transparency by means of a name service, and hides the details of the establishment of the communication connection between clients and servers.

**Context** A software system integrating a set of distributed servers, with the servers running locally or distributed over a network.

**Problem** When a software system uses servers distributed over a network it must provide a means for communication between them. In many cases a connection between components may have to be established before the communication can take place, depending on the available communication facilities. However, the core functionality of the components should be separate from the details of communication mechanisms. Clients should not need to know where servers are located.

We have to balance the following forces:

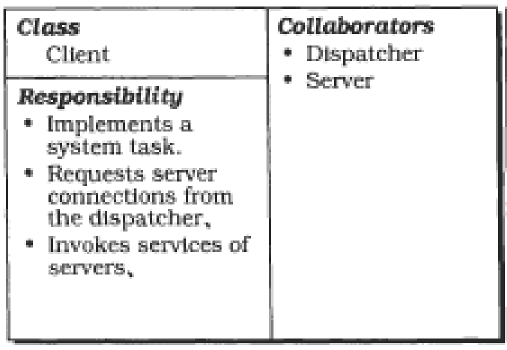
- A component should be able to use a service independent of the location of the service provider.
- The code implementing the functional core of a service consumer should be separate from the code used to establish a connection with service providers.

**Solution :** Provide a dispatcher component to act as an intermediate layer between clients and servers. The dispatcher implements a name service that allows clients to refer to servers by names instead of physical locations, thus providing location transparency. In addition, the dispatcher is responsible for establishing the communication channel between a client and a server.

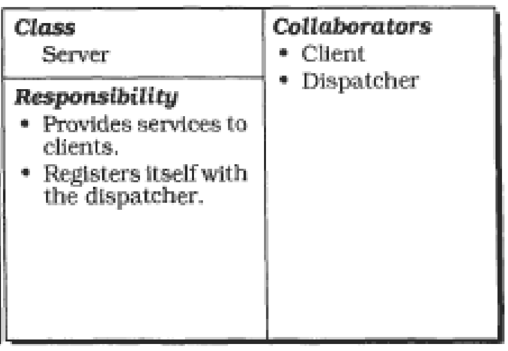
Add servers to the application that provides services to other components. Each server is uniquely identified by its name, and is connected to clients by the dispatcher.

Clients rely on the dispatcher to locate a particular server and to establish a communication link with the server. In contrast to traditional Client-Server computing, the roles of clients and servers can change dynamically.

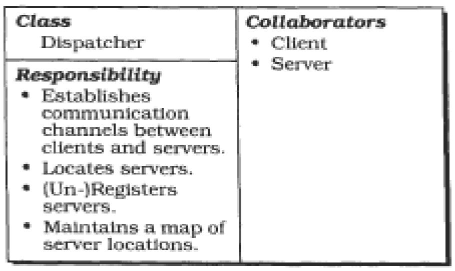
## Structure:



| *Class* Client | *Collaborators* |
|---|---|
| ***Responsibility*** <br>• Implements a system task. <br>• Requests server connections from the dispatcher. <br>• Invokes services of servers. | • Dispatcher <br>• Server |

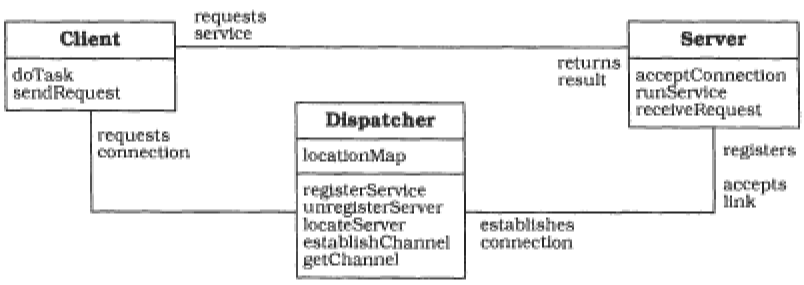


| *Class* Server | *Collaborators* |
|---|---|
| ***Responsibility*** <br>• Provides services to clients. <br>• Registers itself with the dispatcher. | • Client <br>• Dispatcher |



| *Class* Dispatcher | *Collaborators* |
|---|---|
| ***Responsibility*** <br>• Establishes communication channels between clients and servers. <br>• Locates servers. <br>• (Un-)Registers servers. <br>• Maintains a map of server locations. | • Client <br>• Server |

The static relationships between clients, servers and the dispatcher are as follows:



10b. Describe Whole-Part design pattern. 8M

The Whole-Part design pattern helps with the aggregation of components that together form a semantic unit. An aggregate component, the whole, encapsulates its constituent components, the Parts, organizes their collaboration, and provides a common interface to its functionality. Direct access to the Parts Is not possible.

**Context :** Implementing aggregate objects.

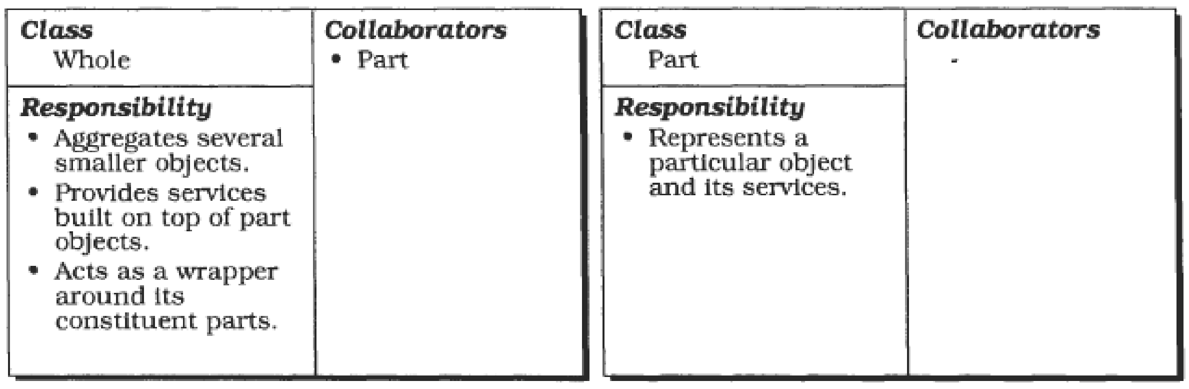
**Problem:**

- A complex object should either be decomposed into smaller objects, or composed of existing objects, to support reusability, changeability and the recombination of the constituent objects in other types of aggregate.
- Clients should see the aggregate object as an atomic object that does not allow any direct access to its constituent parts.

**Solution:**

- Use a component that encapsulates smaller objects, and prevents clients from accessing these constituent parts directly.
- Use an interface as the only means of access to the functionality of the encapsulated objects.
- An assembly-parts relationship differentiates between a product and its parts or subassemblies
- A container-contents relationship.
- A collection-members relationship helps to group similar objects - such as an organization and its members

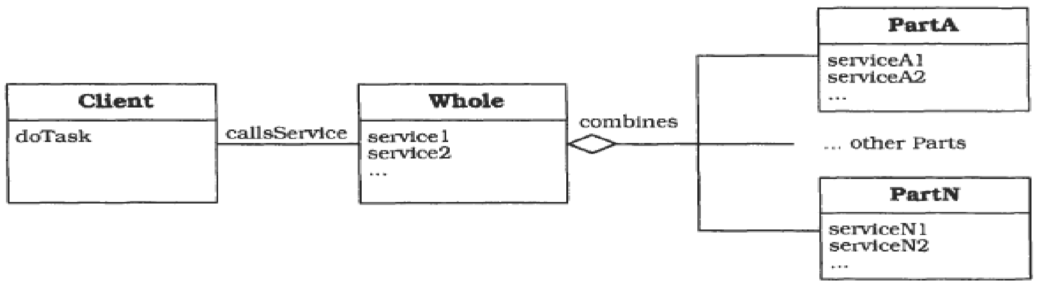
**Structure:**



| *Class* Whole | *Collaborators* |
|---|---|
| *Responsibility*<br>• Aggregates several smaller objects.<br>• Provides services built on top of part objects.<br>• Acts as a wrapper around its constituent parts. | • Part |

| *Class* Part | *Collaborators* |
|---|---|
| *Responsibility*<br>• Represents a particular object and its services. | - |

The static relationships between a Whole and its Parts are illustrated in the OMT diagram below:



**Implementation:**

1.Design the public interface of the Whole.
2.Separate the Whole into Parts, or synthesize it from existing ones. There are two approaches to assembling the Parts you need—either assemble a Whole 'bottom-up' from existing Parts, or decompose it 'top-down' into smaller Parts:

- The bottom-up approach allows you to compose Wholes from loosely-coupled Parts that you can later reuse when implementing other types of Whole.
- The top-down approach makes it is possible to cover all of the Whole 's functionality. Partitioning into Parts is driven by the services the Whole provides to its clients, freeing you from the requirement to implement glue code.

3. If you follow a bottom-up approach, use existing Parts from component libraries or class libraries and specify their collaboration.
4. If you follow a top-down approach, partition the Whole 's services into smaller collaborating services and map these collaborating services to separate Parts.
5.Specify the services of the Whole in terms of services of the Parts. In the structure you found in the previous two steps, the Whole is represented as a set of collaborating Parts with separate responsibilities.
6. Implement the Parts.
7. Implement the Whole.