CMR
INSTITUTE OF
TECHNOLOGY

USN

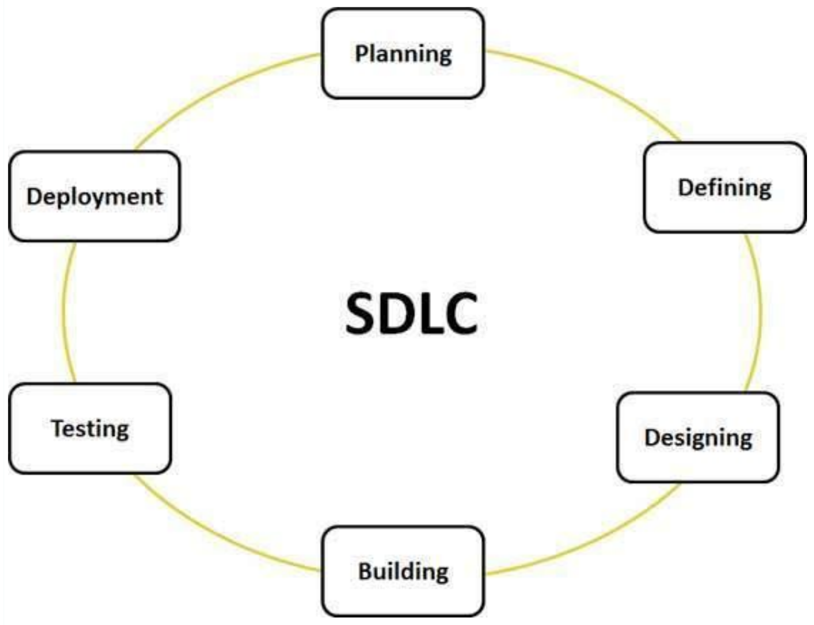
Internal Assesment Test - II

Sub:	OBJECT ORIENTED MODELING & DESIGN							Code:	13MCA51
Date:	03/11/2016	Duration:	90 mins	Max Marks:	50	Sem:	V	Branch:	MCA
Answer Any FIVE FULL Questions									

		Marks	OBE	
			CO	RBT
1	Briefly describe the software development stages.	[10]	CO1, CO4	L1

SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process.

The following figure is a graphical representation of the various stages of a typical SDLC.



A typical Software Development life cycle consists of the following stages:

Stage 1: Planning and Requirement Analysis

Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational, and technical areas.

Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage. The outcome of the technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks.

Stage 2: Defining Requirements

Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts. This is done through .SRS. . Software Requirement Specification document which consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3: Designing the product architecture

SRS is the reference for product architects to come out with the best architecture for the product to be developed. Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification.

This DDS is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity , budget and time constraints , the best design approach is selected for the product.

A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any). The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS.

Stage 4: Building or Developing the Product

In this stage of SDLC the actual development starts and the product is built. The programming code is generated as per DDS during this stage. If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle.

Developers have to follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers etc are used to generate the code. Different high level programming languages such as C, C++, Pascal, Java, and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

Stage 5: Testing the Product

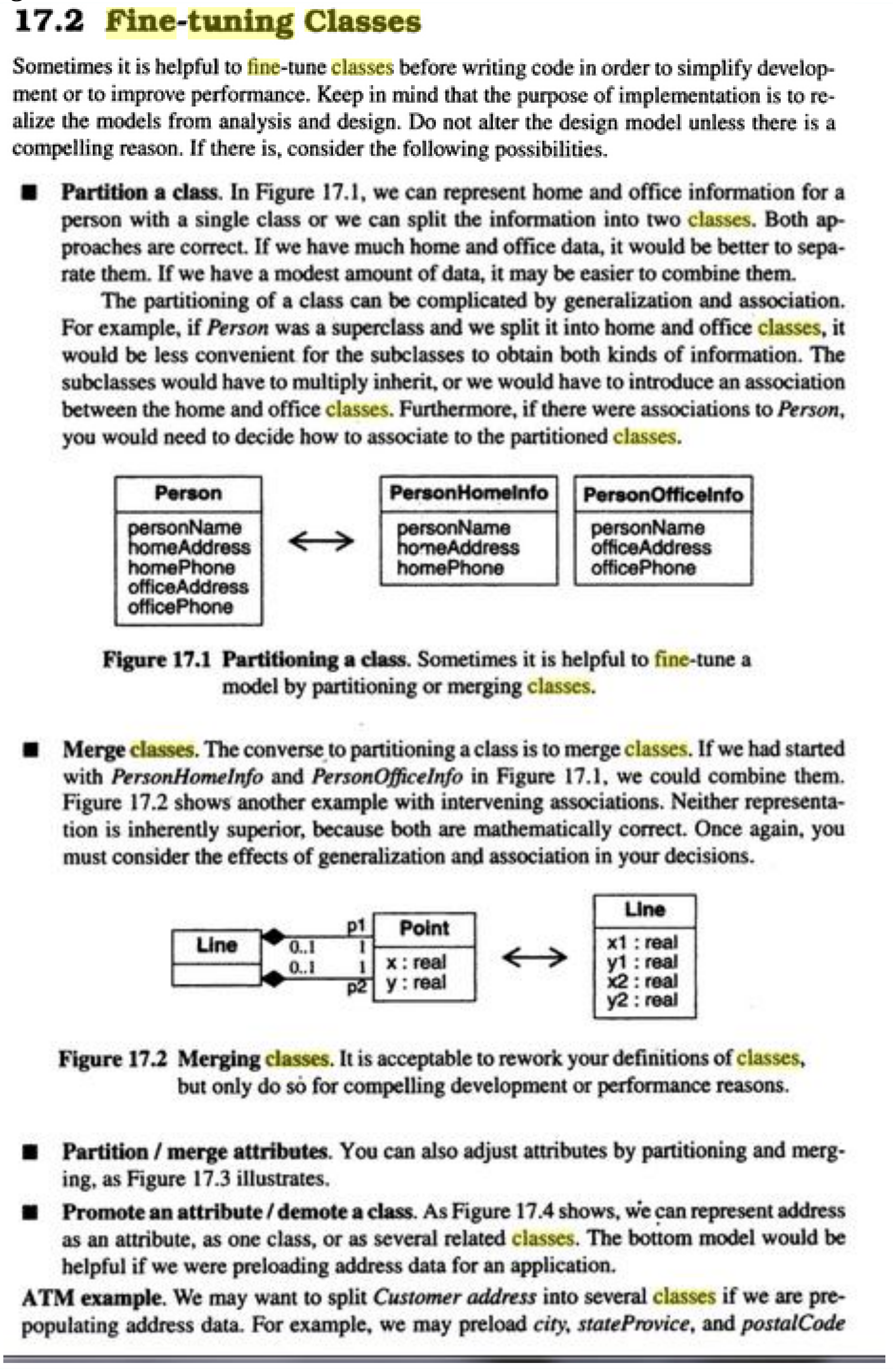
This stage is usually a subset of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC. However this stage refers to the testing only stage of the product where products defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

Stage 6: Deployment in the Market and Maintenance

Once the product is tested and ready to be deployed it is released formally in the appropriate market. Sometime product deployment happens in stages as per the organizations. business strategy. The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing).

Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

2	Explain why tuning of classes is required? What are the different possibilities in doing the same?	[10]	CO2, CO3	L4



17.2 Fine-tuning Classes

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

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

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

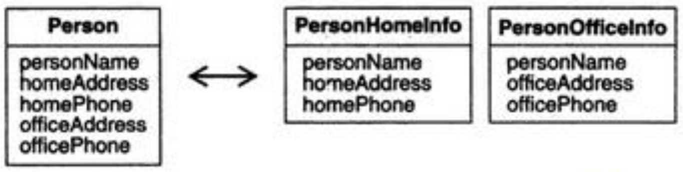


Figure 17.1 Partitioning a class. Sometimes it is helpful to fine-tune a model by partitioning or merging classes.

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

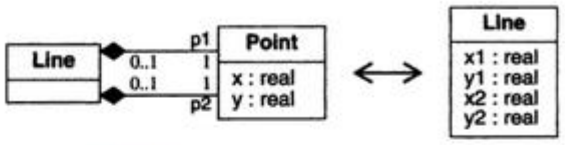
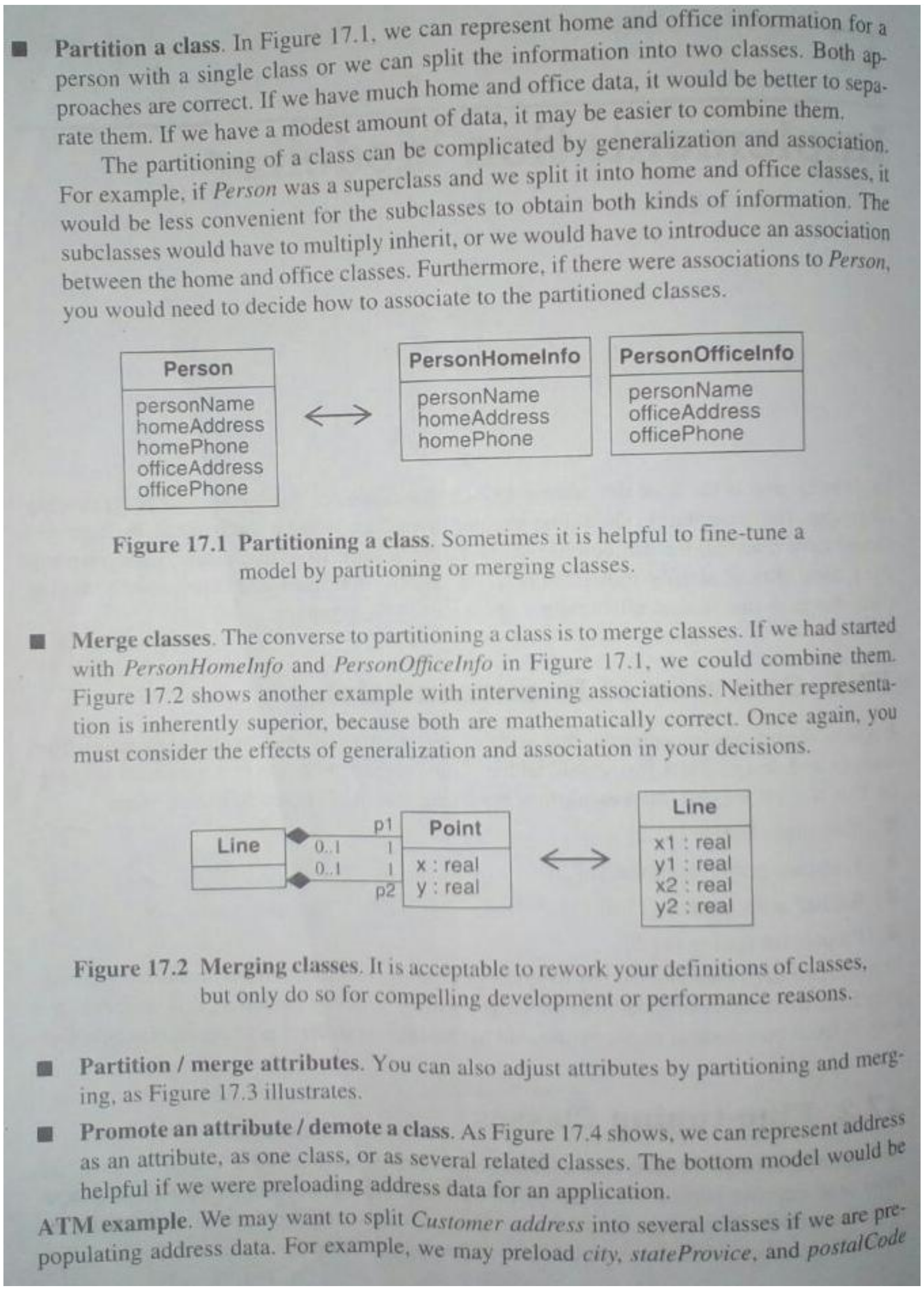


Figure 17.2 Merging classes. It is acceptable to rework your definitions of classes, but only do so for compelling development or performance reasons.

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

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



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

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

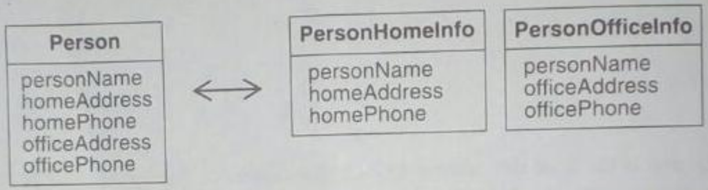


Figure 17.1 Partitioning a class. Sometimes it is helpful to fine-tune a model by partitioning or merging classes.

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

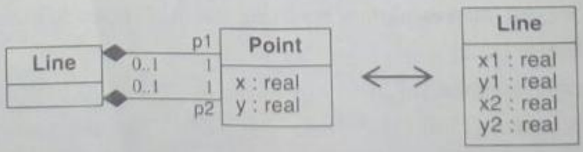
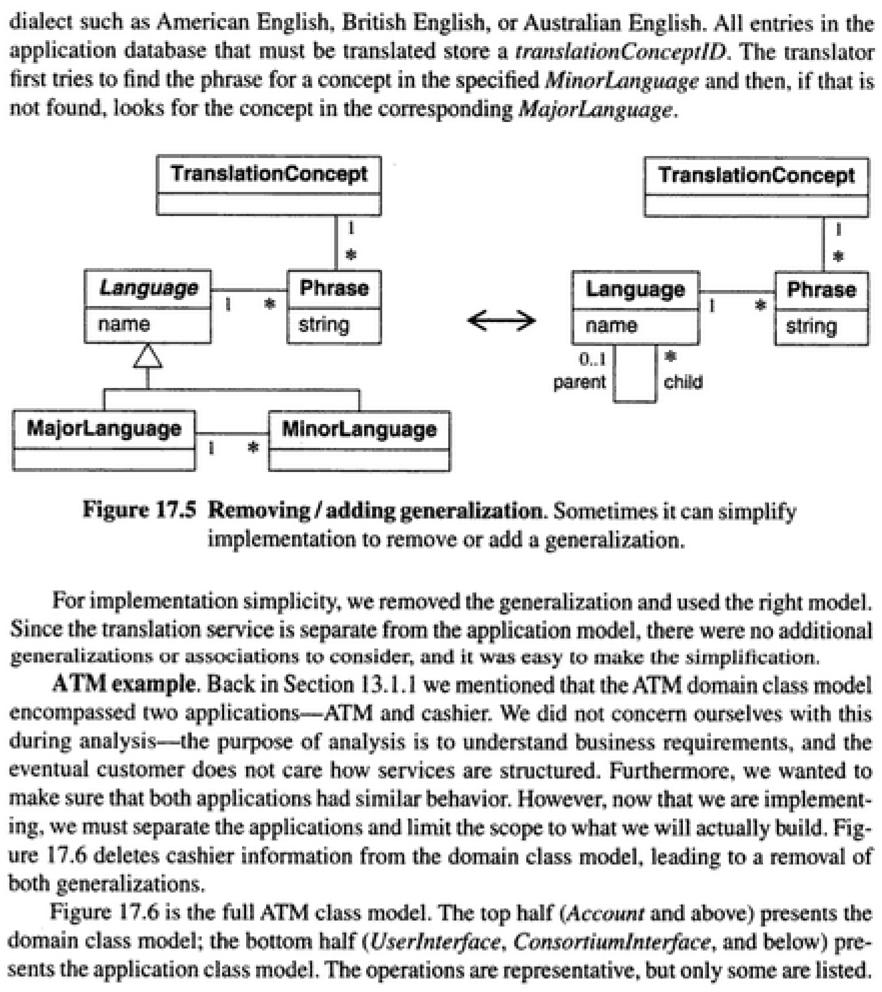


Figure 17.2 Merging classes. It is acceptable to rework your definitions of classes, but only do so for compelling development or performance reasons.

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

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



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

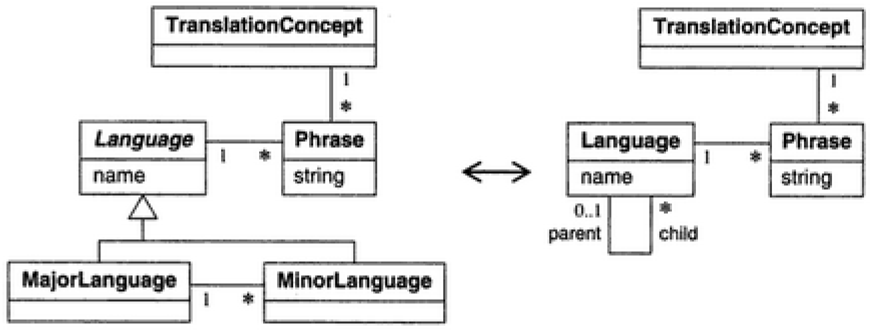


Figure 17.5 Removing / adding generalization. Sometimes it can simplify implementation to remove or add a generalization.

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

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

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

3	List how do you eliminate unnecessary and incorrect attributes while constructing domain class model?	[10]	CO2, CO3	L1

12.2.7 Keeping the Right Attributes

Eliminate unnecessary and incorrect attributes with the following criteria.

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

  A name is an attribute when its use does not depend on context, especially when it need not be unique within some set. Names of persons, unlike names of companies, may be duplicated and are therefore attributes.
- **Identifiers.** OO languages incorporate the notion of an object identifier for unambiguously referencing an object. Do not include an attribute whose only purpose is to identify an object, as object identifiers are implicit in class models. Only list attributes that exist in the application domain. For example, *accountCode* is a genuine attribute; *Banks* assign *accountCodes* and customers see them. In contrast, you should not list an internal *transactionID* as an attribute, although it may be convenient to generate one during implementation.
- **Attributes on associations.** If a value requires the presence of a link, then the property is an attribute of the association and not of a related class. Attributes are usually obvious on many-to-many associations; they cannot be attached to either class because of their multiplicity. For example, in an association between *Person* and *Club* the attribute *membershipDate* belongs to the association, because a person can belong to many clubs and a club can have many members. Attributes are more subtle on one-to-many associations because they could be attached to the "many" class without losing information. Resist the urge to attach them to classes, as they would be invalid if multiplicity changed. Attributes are also subtle on one-to-one associations.
- **Internal values.** If an attribute describes the internal state of an object that is invisible outside the object, then eliminate it from the analysis.
- **Fine detail.** Omit minor attributes that are unlikely to affect most operations.
- **Discordant attributes.** An attribute that seems completely different from and unrelated to all other attributes may indicate a class that should be split into two distinct classes. A class should be simple and coherent. Mixing together distinct classes is one of the major causes of troublesome models. Unfocused classes frequently result from premature consideration of implementation decisions during analysis.
- **Boolean attributes.** Reconsider all boolean attributes. Often you can broaden a boolean attribute and restate it as an enumeration [Coad-95].

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

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

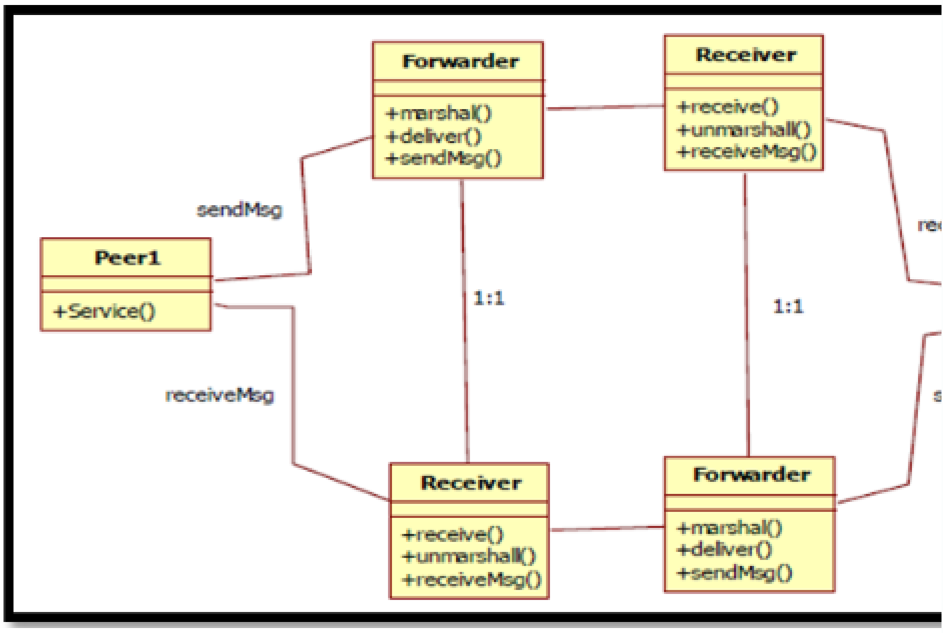
Keep in mind that the ATM problem is just an example. Real applications, when fleshed out, tend to have many more attributes per class than Figure 12.10 shows.

4	Explain forwarder receiver design pattern with its class structure.	[10]	CO5	L5

Forwarder – Receiver Design Pattern

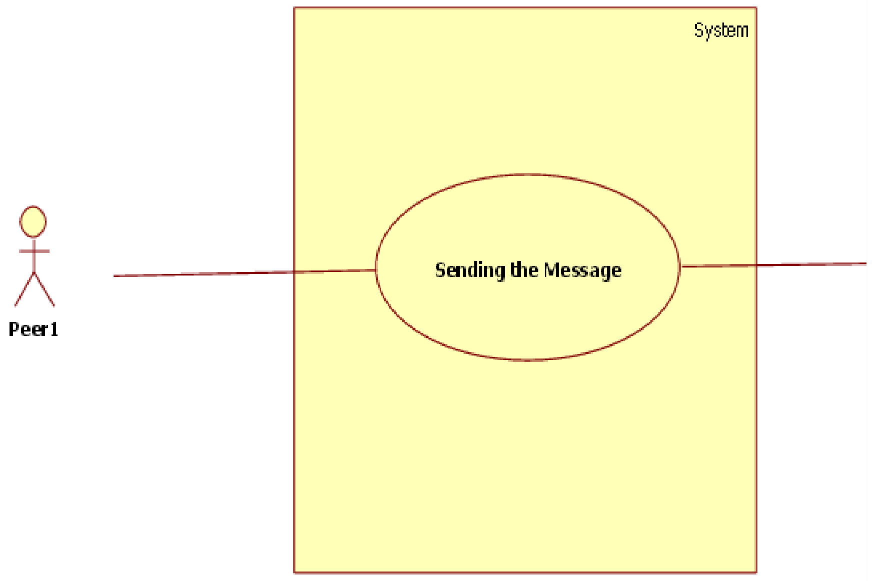
Intent:
The Forwarder-Receiver design pattern provides transparent inter-process communication for software systems with a peer -to-peer interaction model. It introduces forwarders and receivers to decouple peers from the underlying communication mechanisms.

Structure:

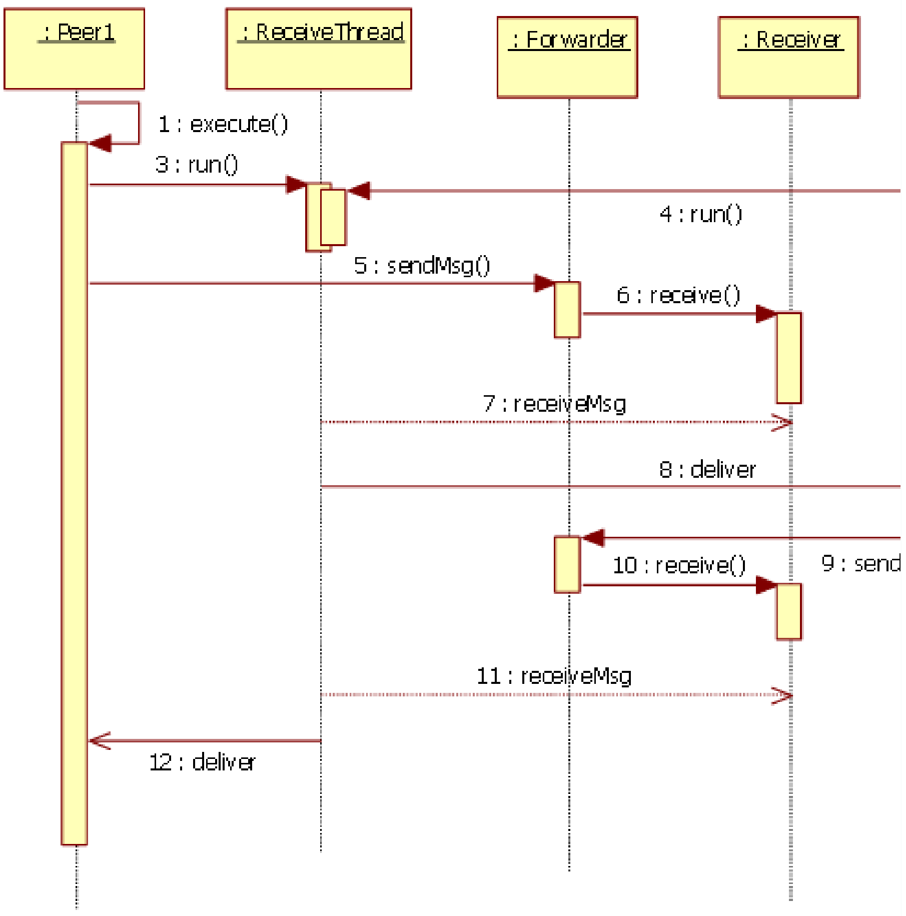


Participant Classes:
Peer components are responsible for application tasks. To carry out their tasks peers need to communicate with other peers.
Forwarder components are responsible for forwarding all these messages to remote network agents without introducing any dependencies on the underlying IPC mechanisms.
Receiver components are responsible for receiving messages. A receiver offers a general interface that is an abstraction of a particular IPC mechanism. It includes functionality for receiving and unmarshaling messages.
Example: A simple peer-to-peer message exchange scenario; Underlying communication protocol is TCP/IP

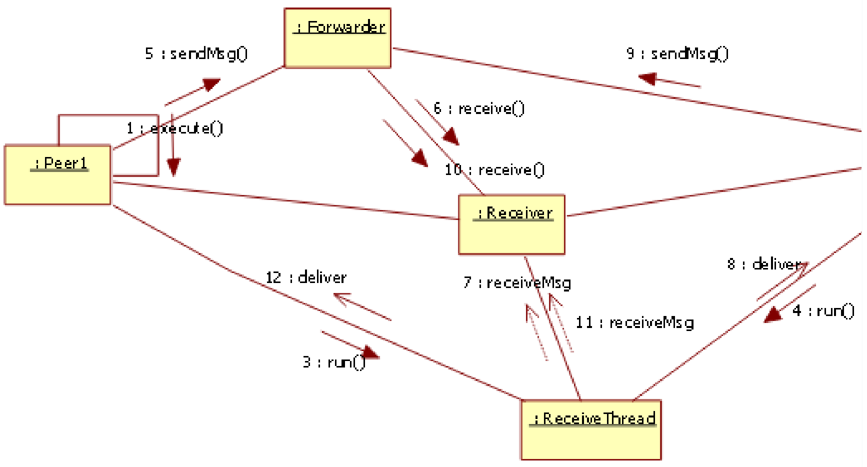
Use Case Diagram:



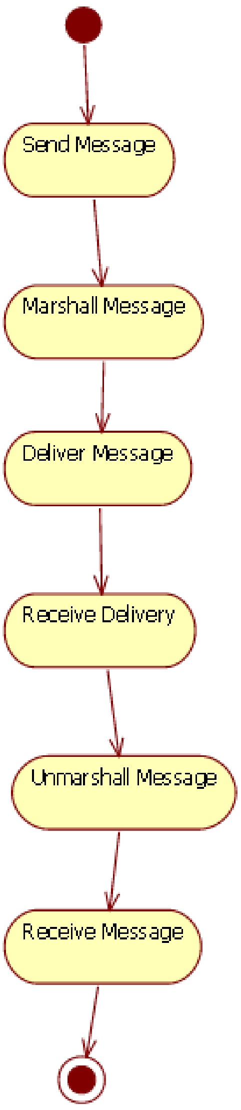
Sequence Diagram:



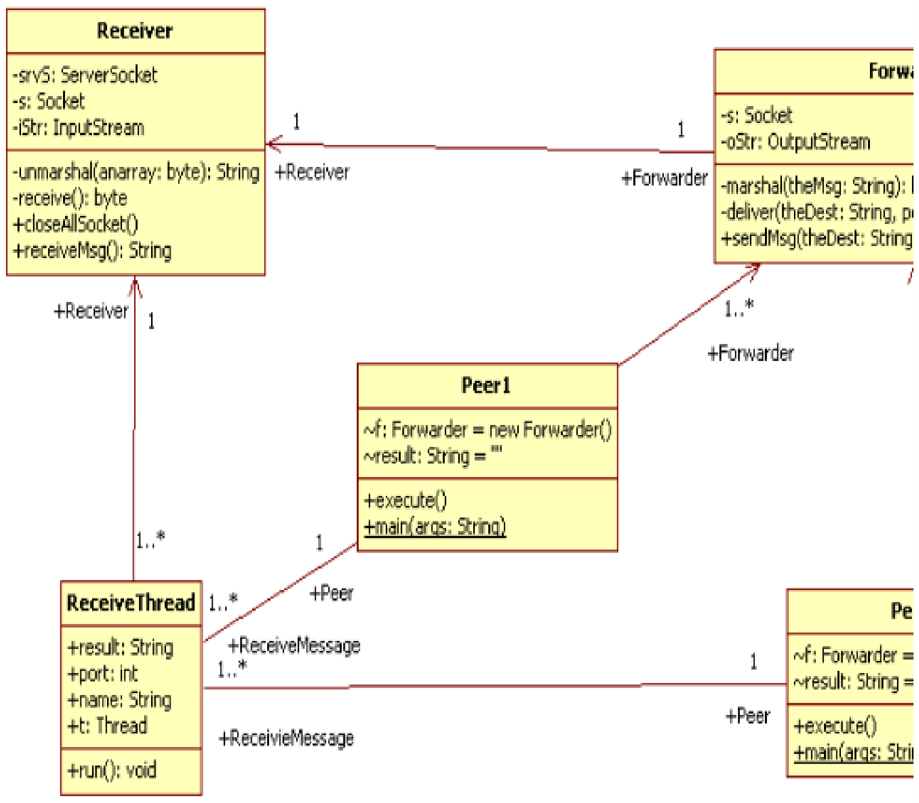
Collaboration Diagram:



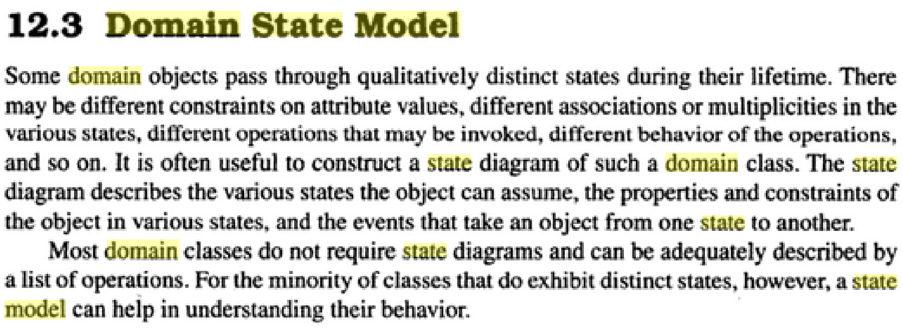
Activity Diagram:



Class Diagram:



5	What do you mean by Domain State Model? Explain the steps performed in constructing the domain state model.	[10]	CO3	L1



12.3 Domain State Model

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

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

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

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

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

12.3.1 Identifying Classes with States

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

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

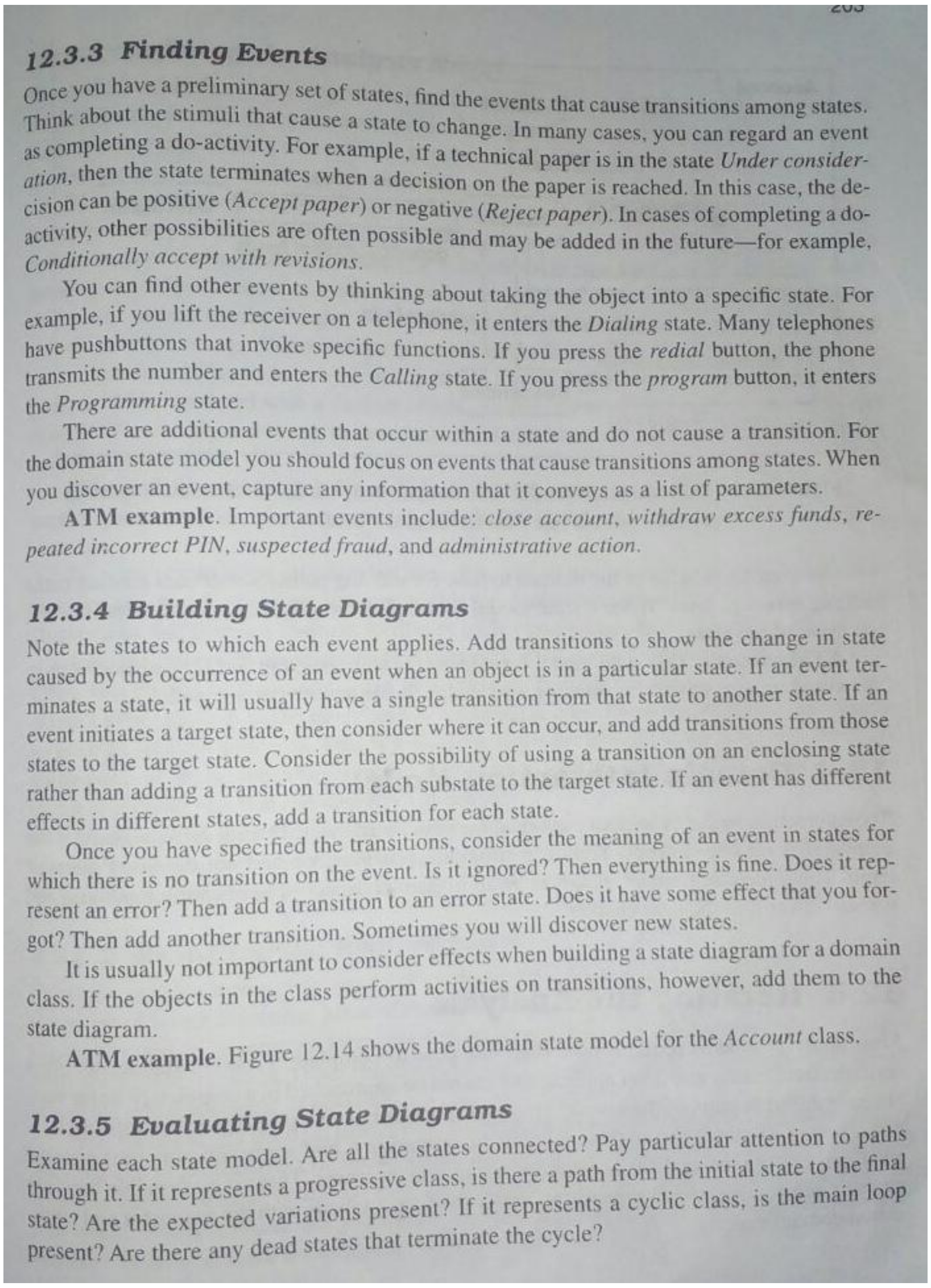
12.3.2 Finding States

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

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

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

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



12.3.3 Finding Events

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

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

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

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

12.3.4 Building State Diagrams

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

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

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

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

12.3.5 Evaluating State Diagrams

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

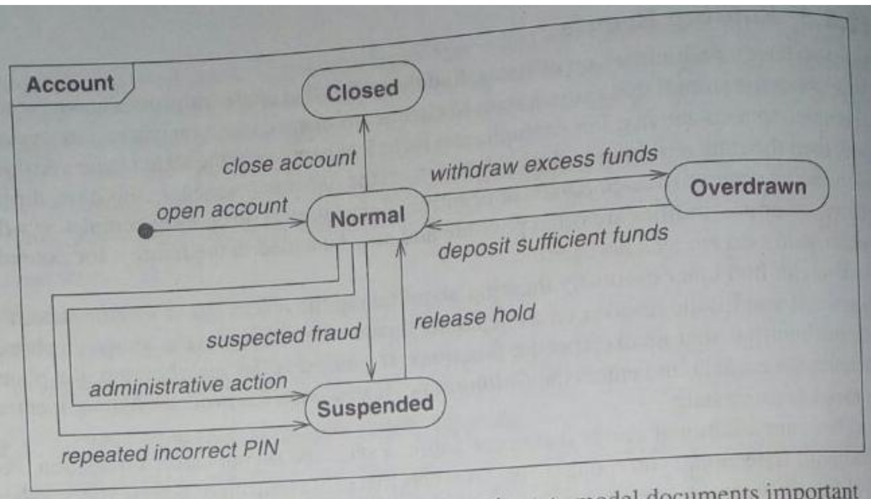


Figure 12.14 Domain state model. The domain state model documents important classes that change state in the real world.

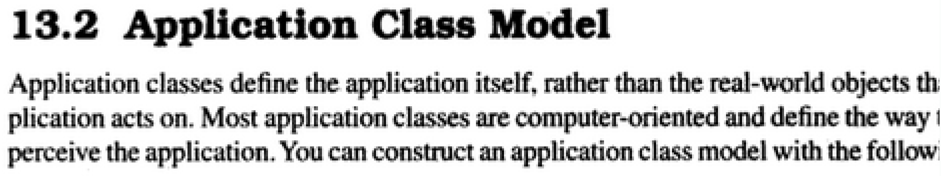
Use your knowledge of the domain to look for missing paths. Sometimes missing paths indicate missing states. When a state model is complete, it should accurately represent the life cycle of the class.

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

12.4 Domain Interaction Model

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

6	Discuss what you understand of Application class model. Explain the steps to construct the application class model.	[10]	CO3	L2



13.2 Application Class Model

Application classes define the application itself, rather than the real-world objects th plication acts on. Most application classes are computer-oriented and define the way t perceive the application. You can construct an application class model with the followi

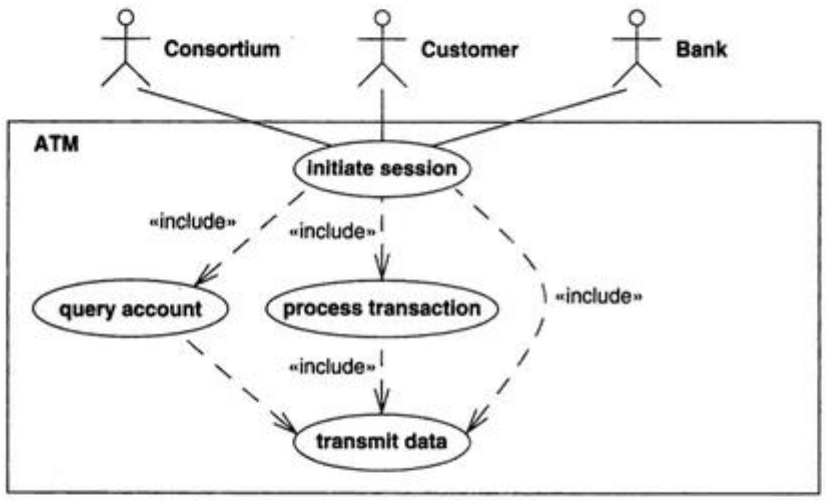


Figure 13.6 Organizing use cases. Once the basic use cases are identified, you can organize them with relationships.

- 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.

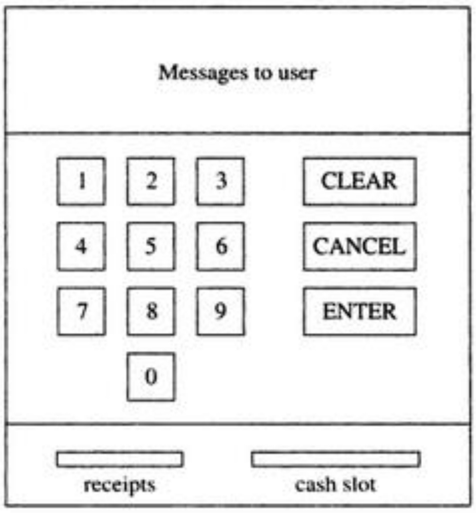


Figure 13.7 Format of ATM interface. Sometimes a sample interface can help you visualize the operation of an application.

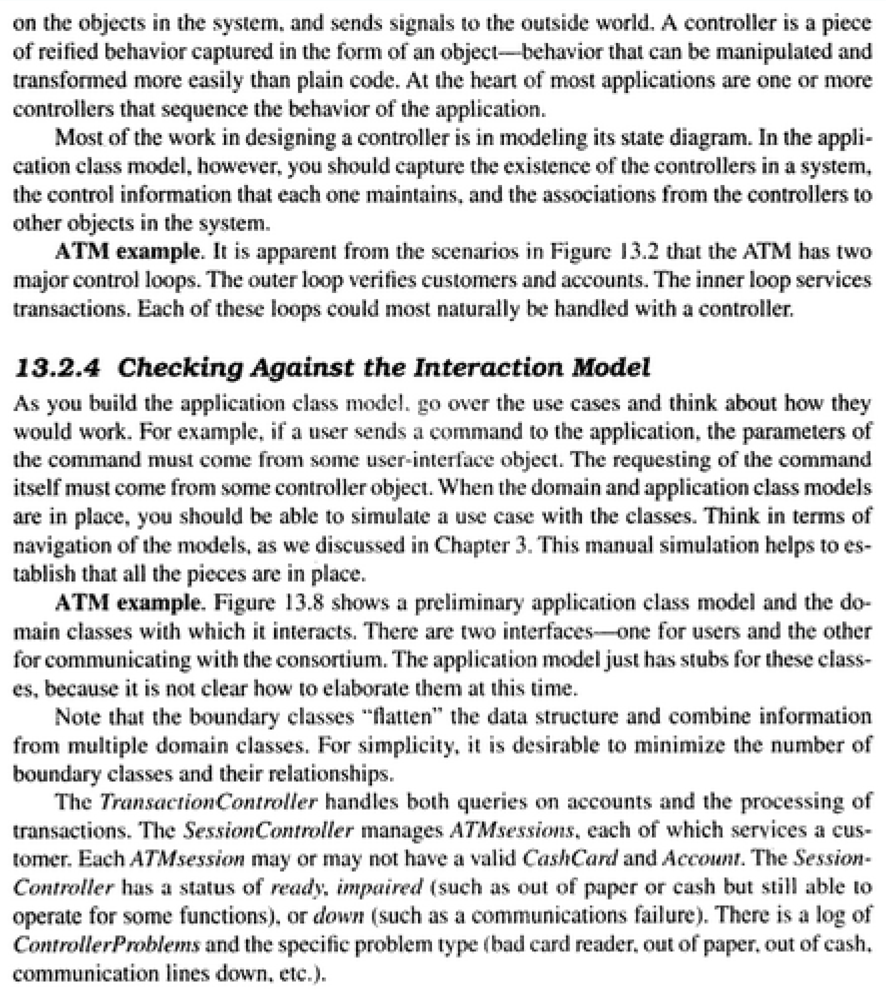
13.2.2 Defining Boundary Classes

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

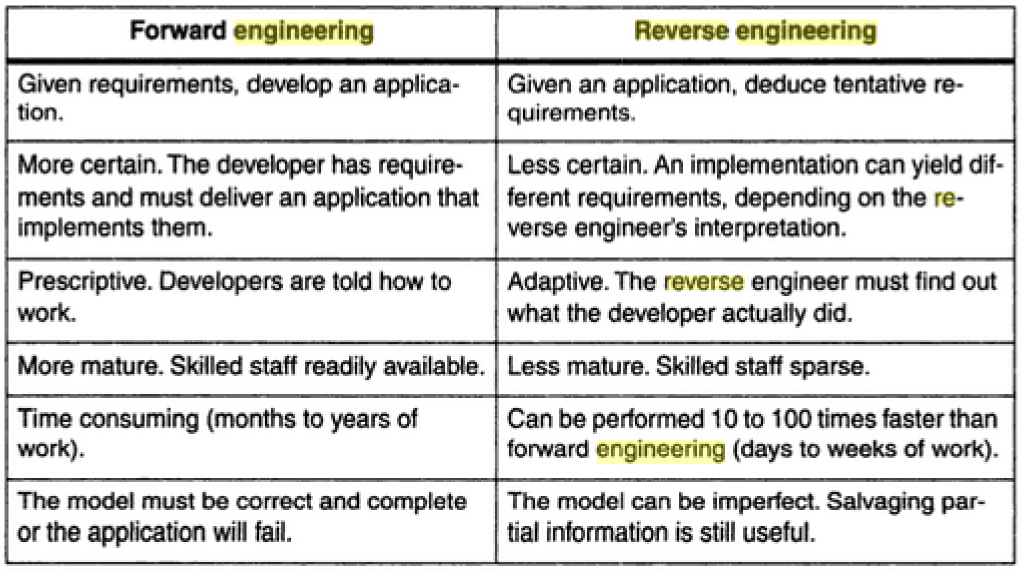
As you build the application class model, go over the use cases and think about how they would work. For example, if a user sends a command to the application, the parameters of the command must come from some user-interface object. The requesting of the command itself must come from some controller 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.).

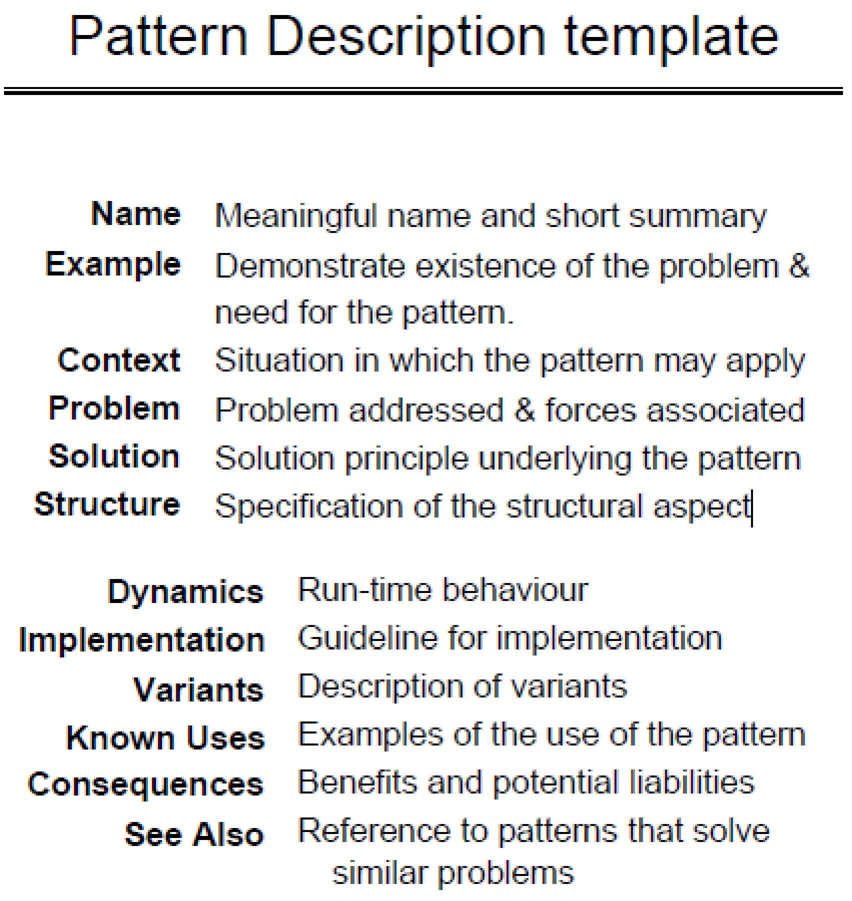
7	What is reverse engineering? Differentiate between reverse engineering and forward engineering.	[10]	CO1, CO4	L5



Forward engineering	Reverse engineering
Given requirements, develop an application.	Given an application, deduce tentative requirements.
More certain. The developer has requirements and must deliver an application that implements them.	Less certain. An implementation can yield different requirements, depending on the reverse engineer's interpretation.
Prescriptive. Developers are told how to work.	Adaptive. The reverse engineer must find out what the developer actually did.
More mature. Skilled staff readily available.	Less mature. Skilled staff sparse.
Time consuming (months to years of work).	Can be performed 10 to 100 times faster than forward engineering (days to weeks of work).
The model must be correct and complete or the application will fail.	The model can be imperfect. Salvaging partial information is still useful.

8	What is pattern? What are the contents of pattern description template?	[10]	CO5	L1

A Pattern in software architecture describes a particular recurring design problem that arises in specific design context, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the way in which they collaborate



Pattern Description template

Name	Meaningful name and short summary
Example	Demonstrate existence of the problem & need for the pattern.
Context	Situation in which the pattern may apply
Problem	Problem addressed & forces associated
Solution	Solution principle underlying the pattern
Structure	Specification of the structural aspect
Dynamics	Run-time behaviour
Implementation	Guideline for implementation
Variants	Description of variants
Known Uses	Examples of the use of the pattern
Consequences	Benefits and potential liabilities
See Also	Reference to patterns that solve similar problems

Course Outcomes		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8
CO1:	Acquire knowledge of o Basic UML Concepts and terminologies o Life Cycle of Object oriented Development o Modeling Concepts	1	1	1	-	-	-	-	2
CO2:	Identify the basic principles of Software modeling and apply them in real world applications	3	2	1	-	-	-	-	-
CO3:	Produce conceptual models for solving operational problems in software and IT environment using UML	2	1	3	-	-	-	-	-
CO4:	Analyze the development of Object Oriented Software models in terms of o Static behaviour o Dynamic behaviour	1	3	-	-	-	-	-	-
CO5:	Evaluate and implement various Design patterns	2	1	3	-	-	-	-	-

Cognitive level	KEYWORDS
L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.

PO1 – Apply *knowledge*; PO2 - *Problem analysis*; PO3 - *Design/development of solutions*; PO4 – team work ; PO5 – *Ethics* ; PO6 -Communication; PO7- *Business Solution*; PO8 – Life-long learning