UNIT – III

Object Oriented Analysis : Use case driven Object analysis – approaches for identifying classes – identifying objects, relationships attributes, methods for ATM banking system –Object oriented design process – design axioms.

OBJECT ORIENTED ANALYSIS

Analysis is the process of extracting the needs of a system and what the system must do to satisfy the users' requirement. The goal of object oriented analysis is to understand the domain of tile problem and the system's responsibilities by understanding how the users use or will use the system. The main objective of the analysis is to capture a complete, unambiguous, and consistent picture of the requirements of the system and what the system must do to satisfy the users' requirements and needs.

Business Object Analysis: Understanding the Business Layer

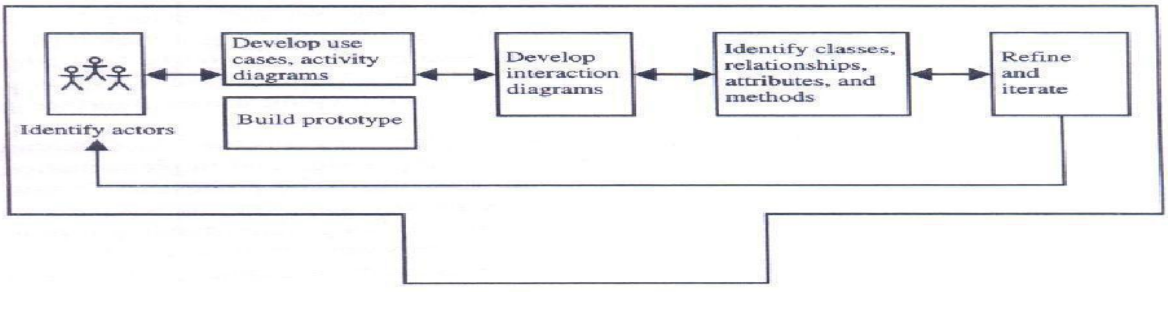
Business object analysis is a process of understanding the system's requirements and establishing the goals of an application. To understand the users' requirements, we need to find out how they "use" the system. This can be accomplishing by developing use cases. Defer unimportant details until later. State *what* must be done, not *how* it should be done. This, of course, is easier said than done. Yet another tool that can be very useful for understanding users' requirements is preparing a prototype of the user interface.

USE CASE DRIVEN OBJECT ORIENTED ANALYSIS

The object-oriented analysis (OOA) phase of the unified approach uses actors and use cases to describe the system from the users' perspective. The *actors* are external factors that interact with the system; *use cases* are scenarios that describe how actors use the system. The use cases identified here will be involved throughout the development process. The OOA process consists of the following steps:

1. *Identify the actors:* Who is using the system? Or, in the case of a new system, who will be using the system?
2. *Develop a simple business process model using UML activity diagram.*
3. *Develop the use case:* What are the users doing with the system? Or, in case of the new system, what will users be doing with the system? .Use cases provide us with comprehensive documentation of the system under study.
4. *Prepare interaction diagrams:* Determine the sequence. .Develop collaboration diagrams.
5. *Classification-develop a static UML class diagram:* Identify classes. . Identify relationships. Identify attributes. Identify methods.
6. *Iterate and refine:* If needed, repeat the preceding steps.

The object-oriented analysis process in the Unified Approach (UA).



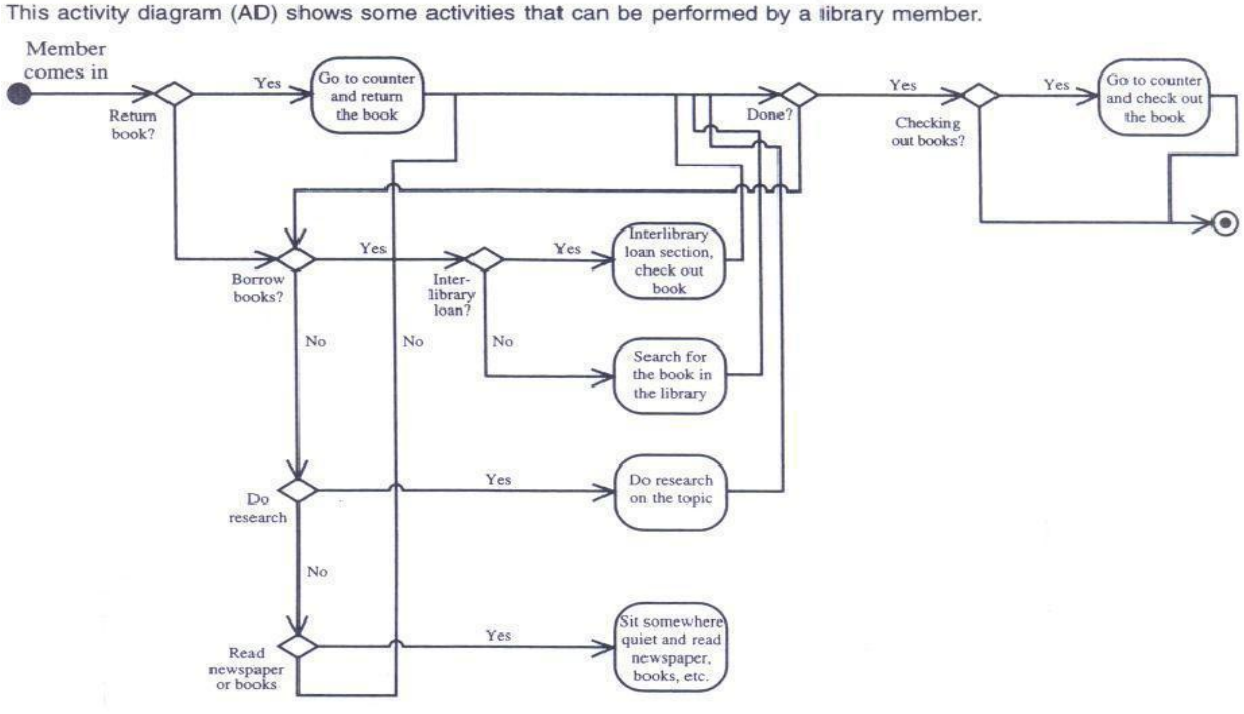
BUSINESS PROCESS MODELING

Business process modeling can be very time consuming, so the main idea should be to get a basic model without spending too much time on the process.

The advantage of developing a business process model is that it makes you more familiar with the system and therefore the user requirements and also aids in developing use cases.

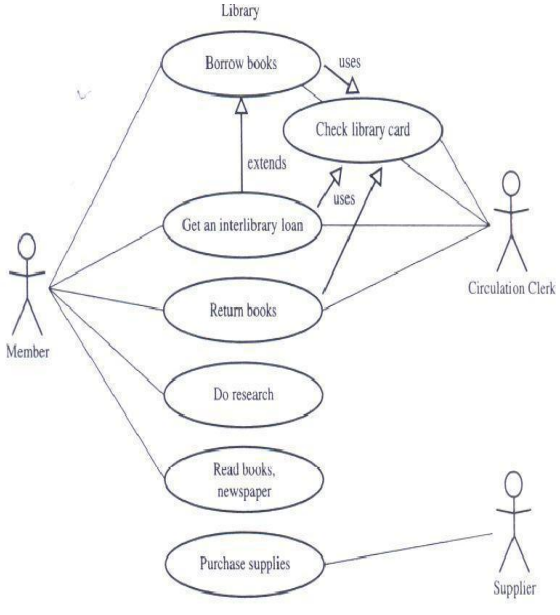
For example, let us define the steps or activities involved in using your school library. These activities can be represented with an activity diagram.

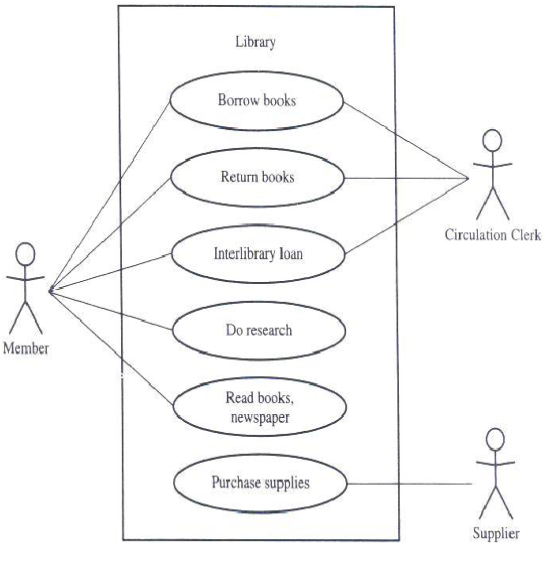
Developing an activity diagram of the business process can give us better understandings of what sort of activities are performed in a library by a library member.



USE-CASEMODEL

Use cases are scenarios for understanding system requirements. A use-case model can be instrumental in project development, planning, and documentation of systems requirements.





A use case is an interaction between users and a system; it captures the goal of the users and the responsibility of the system to its users

The use-case model describes the uses of the

system and shows the courses of events that can be performed. A use-case model also can discover classes and the relationships among subsystems of the systems. Each use or scenario represents what the user wants to do.

Each use case must have a name and short textual description, no more than a few paragraphs.

Since the use-case model provides an external view of a system or application, it is directed primarily toward the users or the "actors" of the systems, not its implementers As you can see, these are uses of external views of the library system by an actor such as a member, circulation clerk, or supplier instead of a developer of the library system. The simpler the use-case model, the more effective it will be. It is not wise to capture all the details right at the start.

The UML class diagram, also called an object model, represents the static relationships between objects, inheritance, association, and the like. The object model represents an internal view of the system, as opposed to the use-case model, which represents the external view of the system.

The use-case diagram depicts the **extends** and uses relationships where the interlibrary loan is a special case of checking out books. Entering into the system is common to get an interlibrary loan, borrow books, and return books use cases, so it is being "used" by all these use cases.

Transaction: A transaction is an atomic set of activities that are performed either fully or not at all. A transaction is triggered by a stimulus from an actor to the system or by a point in time being reached in the system.

IDENTIFYING CLASSES The four alternative approaches for identifying classes are :

1. noun phrase approach
2. common class patterns approach
3. use case driven, sequence/collaboration modeling approach
4. Classes, Responsibilities, and Collaborators (CRC) approach

The first two approaches have been included to increase your understanding of the subject; the unified approach uses the use-case driven approach for identifying classes and understanding the behavior of objects. However, you always can combine these approaches to identify classes for a given problem. Another approach that can be used for identifying classes is Classes, Responsibilities, and Collaborators (CRC) developed by Cunningham, Wilkerson, and Beck. Classes, Responsibilities, and Collaborators, more technique than method, is used for identifying classes responsibilities and therefore their attributes and methods.

Noun Phrase Approach

The noun phrase approach was proposed by Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. In this method, reading through the requirements or use cases and looking for noun phrases. Nouns in the textual description are considered to be classes and verbs.

Using the noun phrase strategy, candidate classes can be divided into three categories:

1. Relevant Classes,

2. Fuzzy Area or Fuzzy Classes and
3. Irrelevant Classes. Database standards and even fourth-generation languages."

Selecting Classes from the Relevant and Fuzzy Categories

The following guidelines help in selecting candidate classes from the relevant and fuzzy categories of classes in the problem domain.

1. **Redundant classes.** Do not keep two classes that express the same information. If more than one word is being used to describe the same idea, select the one that is the most meaningful in the context of the system. This is part of building a common vocabulary for the system as a whole. Choose your vocabulary carefully; use the word that is being used by the user of the system.
2. **Adjectives classes.** Adjectives can be used in many ways. An adjective can suggest a different kind of object, different use of the same object, or it could be utterly irrelevant. If the use of the adjective signals that the behavior of the object is different, and then makes a new class". For example, Adult Members behave differently than Youth Members; so, the two should be classified as different classes.
3. **Attribute classes.** Tentative objects that are used only as values should be defined or restated as attributes and not as a class. For example, Client Status and Demographic of Client are not classes but attributes of the Client class.
4. **Irrelevant classes.** Each class must have a purpose and every class should be clearly defined and necessary. You must formulate a statement of purpose for each candidate class. If you cannot come up with a statement of purpose, simply eliminate the candidate class.

The process of eliminating the redundant classes and refining the remaining classes is not sequential. The process of identifying relevant classes and eliminating irrelevant classes is an incremental process. Each iteration often uncovers some classes that have been overlooked.

The ViaNet Bank ATM System: Identifying Classes by Using Noun Phrase Approach

To better understand the noun phrase method, we will go through a case and apply the noun phrase strategy for identifying the classes. We must start by reading the use cases and applying the principles discussed in this chapter for identifying classes.

Initial List of Noun Phrases: Candidate Classes The initial study of the use cases of the bank system produces the following noun phrases (candidate classes-maybe).

Account	ATM Card	Card
Account Balance	ATM Machine	Cash
Amount Approval	Bank	Check
Process	Bank Client	Checking

It is safe to eliminate the irrelevant classes. The candidate classes must be selected from relevant and fuzzy classes. The following irrelevant classes can be eliminated because they do not belong to the problem statement: Envelope, Four Digits, and Step. Strikeouts indicate eliminated classes.

Account	Account	Approval	Process	ATM Machine
Balance	Amount	ATM Card		Bank . Bank Client

Reviewing the Redundant Classes and Building a Common Vocabulary

We need to review the candidate list to see which classes are redundant. If different words are being used to describe the same idea, we must select the one that is the most meaningfull the context of the system and eliminate the others. The following are the different class names that are being used to refer to the same concept:
Client, Bank Client Account, Client's Account PIN, PIN Code
Checking, Checking Account = Bank Client (the term chosen)
Checking Account = Account
Checking Account = PIN
Checking Account = Checking Account
Savings, Savings Account = Savings

Reviewing the Classes Containing Adjectives

By again review the remaining list, now with an eye on classes with adjectives. The main question is this: Does the object represented by the noun behave differently when the adjective is applied to it?

However (it is a different use of the same object or the class is irrelevant, we must eliminate it)

Reviewing the Possible Attributes

The next review focuses on identifying the noun phrases that are attributes, not classes. The noun phrases used only as values should be restated as attributes. This process also will help us identify the attributes of the classes in the system.
Balance: An attribute of the Account class. Invalid PIN: It is only a value, not a class. *Password:* An attribute, possibly of the Bank Client class.
Transaction History: An attribute, possibly of the Transaction class. PIN: An attribute, possibly of the BankClientclass

Reviewing the Class Purpose

Identifying the classes that play role in achieving system goals and requirements is a major activity of object-oriented analysis) each class must have a purpose. Every class should be clearly defined and necessary in the context of achieving the system's goals The candidate classes are these:
ATM Machine class: Provides an interface to the ViaNet bank.
ATMCard class: Provides a client with a key to an account.
Bank Client class: A client is an individual that has a checking account and, possibly, a savings account.
Bank class: Bank clients belong to the Bank. It is a repository of accounts and processes the accounts' transactions.

The major problem with the noun phrase approach is that it depends on the completeness and correctness of the available document, which is rare in real life. On the other hand, large volumes of text on system documentation might lead to too many candidate classes.

The process of creating sequence or collaboration diagrams is a systematic way to think about how a use case (scenario) can take place; and by doing so, it forces you to think about objects involved in your application

Implementation of Scenarios

This process helps us to understand the behavior of the system's objects. When you have arrived at the lowest use-case level, you may create a child sequence diagram or accompanying collaboration diagram for the use case. With the sequence and collaboration diagrams, you can model the implementation of the scenario.

The Vianet Bank ATM System: Decomposing

Scenario with a Sequence Diagram: Object Behavior Analysis A sequence diagram represents the sequence and interactions of a given use case or scenario. The event line represents a message sent from one object to another, in which the "from" object is requesting an operation be performed by the "to" object. The "to" object performs the operation using a method that its class contains.

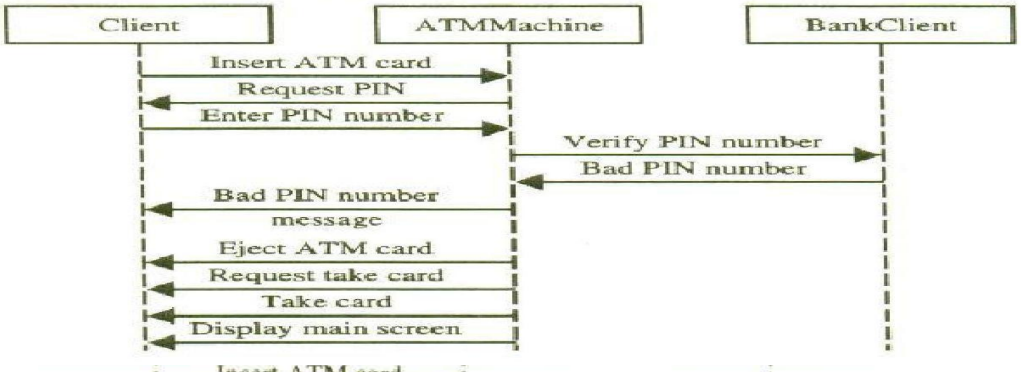
We identified the use cases for the bank system. The following are the low level (executable) use cases:

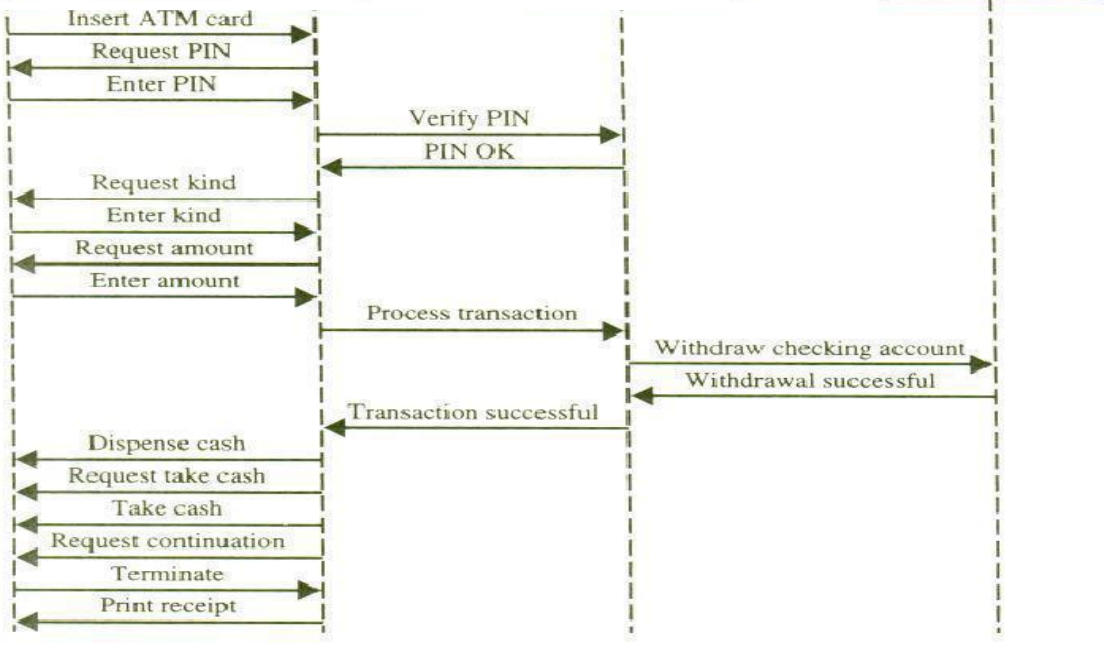
Deposit Checking	Withdraw More from	Denied Checking
Deposit Savings	Checking Withdraw	Transaction History
Invalid PIN	Savings	Savings Transaction
Withdraw Checking	Withdraw Savings	History

Point of caution: you should defer the interfaces classes to the design phase and concentrate on the identifying business classes here. Consider how we would prepare a sequence diagram for the Invalid PIN use case.

The client in this case is whoever tries to access an account through the ATM, and major may not have an account. The Bank Client on the other hand has an account.

The sequence diagram for the Invalid PIN use case.





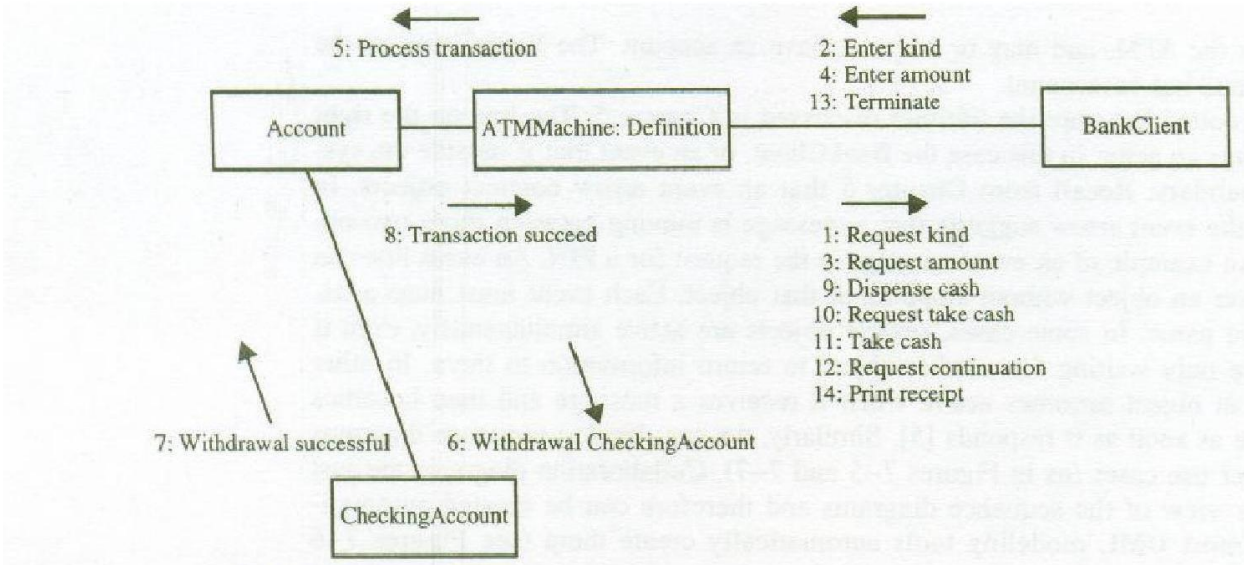
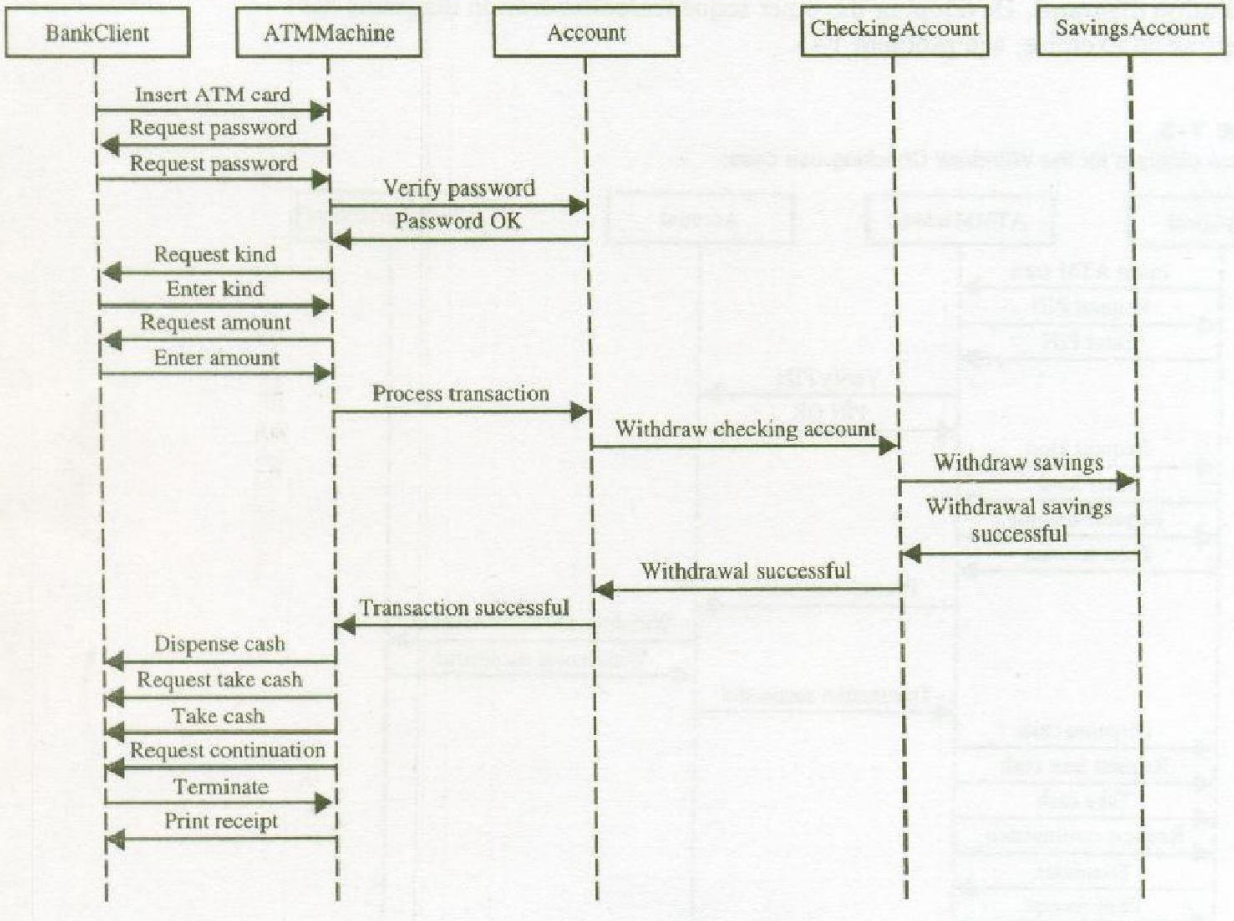
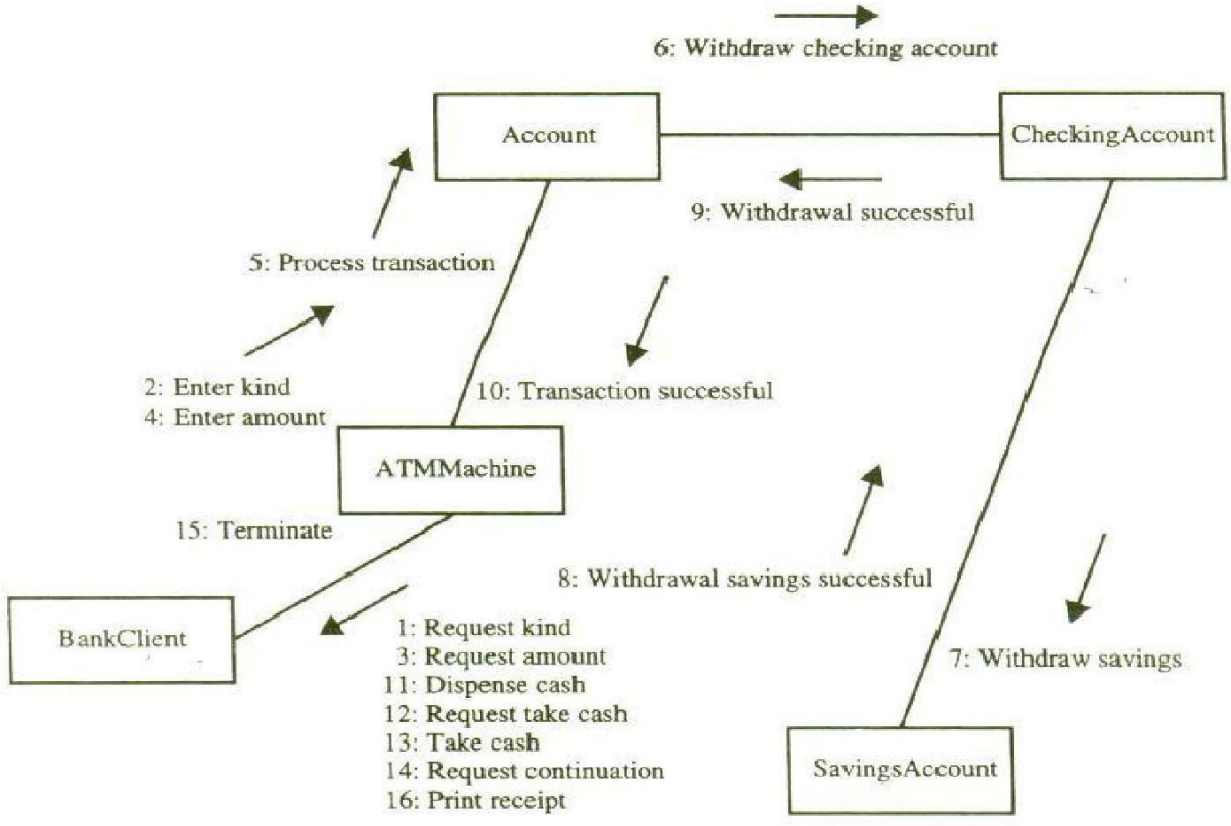


FIGURE 7–6
The collaboration diagram for the Withdraw Checking use case.

FIGURE 7–7
The sequence diagram for the Withdraw More from Checking use case.





CLASSES, RESPONSIBILITIES, AND COLLABORATORS (CRC)

Classes, Responsibilities, and Collaborators is a technique used for identifying classes' responsibilities and therefore their attributes and methods. By identifying an object's responsibilities and collaborators (cooperative objects with which it works) you can identify its attributes and methods. CRC cards are 4" X 6" index cards.

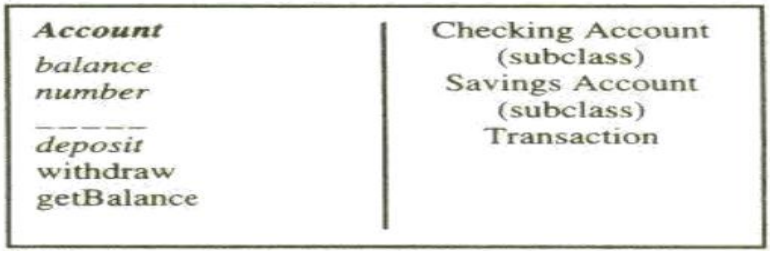
Fig: A Classes, Responsibilities, and Collaborators (CRC) index card. **CRC PROCESS** The Classes, Responsibilities, and Collaborators process consists of three steps

1. Identify classes' responsibilities (and identify classes).
2. Assign responsibilities.
3. Identify collaborators.

The Classes, Resposibilities, and Collaborators process.

As cards are written for familiar objects, all participants pick up the same context and ready themselves for decision making. Then, by waving cards and pointing fingers and yelling statements like, "no, this guy should do that," decisions are made. Finally, the group starts to relax as consensus has been reached and the issue becomes simply finding the right words to record a decision as a responsibility on a card.

Classes, Responsibilities, and Collaborators for the Account object.



This process is iterative. Start with few cards (classes) then proceed to play "what if." If the situation calls for a responsibility not already covered by one of the objects, either add the responsibility to an object or create a new object to address that responsibility. If one of the objects becomes too cluttered during this process, copy the information on

Analyzing relationships among classes.

Identifying association.

Association patterns.

Identifying super- and subclass hierarchies.

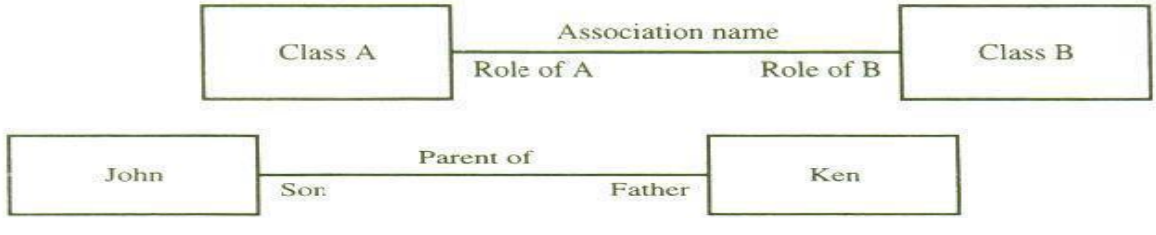
Identifying aggregation or a-part-of compositions.

Class responsibilities.

Identifying attributes and methods by analyzing use cases and other UML diagrams.

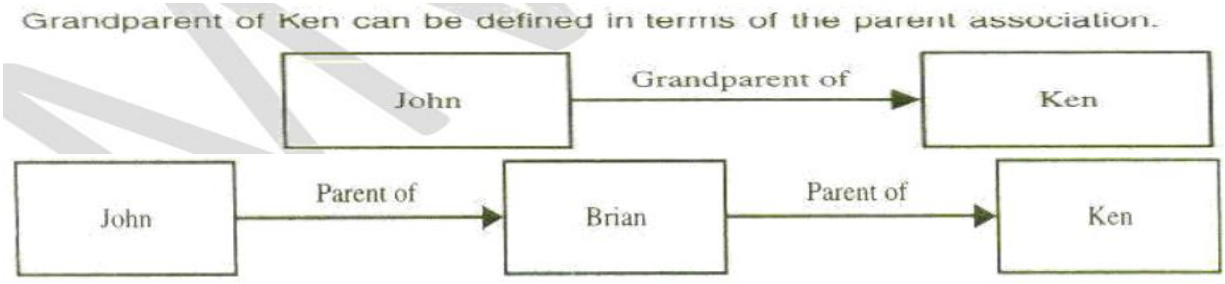
ASSOCIATIONS Association represents a physical or conceptual connection between two or more objects) For example, if an object has the responsibility for telling another object that a credit card number is valid or invalid, the two classes have an association.

Basic association. See Chapter 5 for a detailed discussion of association.



Guideline for Identifying Association A dependency between two or more classes may be an association. Association often corresponds to a verb or prepositional phrase, such as part of, next to, works for, or contained in. A reference from one class to another is an association. Some associations are implicit or taken from general knowledge.

Common Association Patterns Communication association talk to, order to. For example, a customer places an order (communication association)with an operator person



These association patterns and similar ones can be stored in the repository and added to as more patterns are discovered

Ternary associations. Ternary or n-ary association is an association among more than two classes . Ternary associations complicate the representation. When possible, restate ternary associations as binary associations **Directed actions (or derived) association.** Directed actions (derived) associations can be defined in terms of other associations. Since they are redundant, avoid these types of association. For example, Grandparent of can be defined in terms of the parent of association (see Figure). Choose association names carefully

DESIGN AXIOMS

By definition, an *axiom* is a fundamental truth that always is observed to be valid and for which there is no counterexample or exception. A *theorem* is a proposition that may not be self-evident but can be proven from accepted axioms.

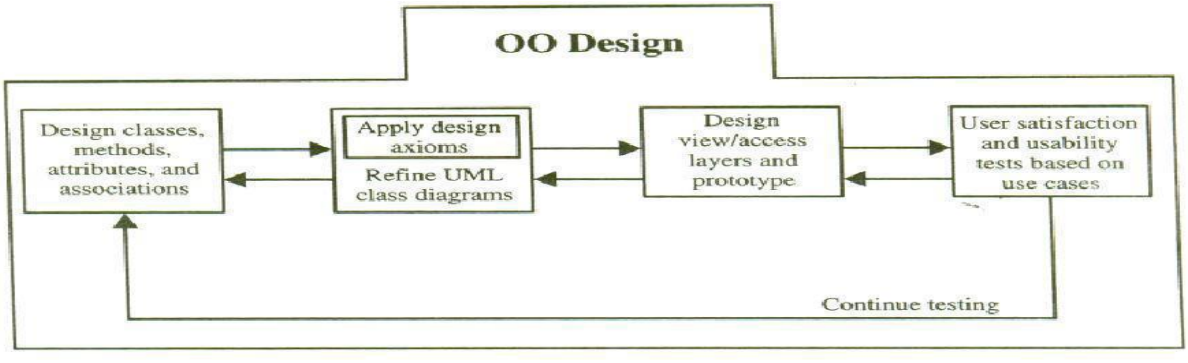
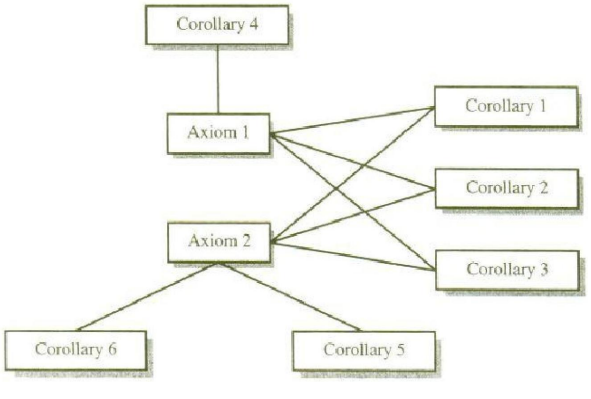


Fig:The object-oriented design process in the unified approach. Axiom 1 deals with relationships between system components (such as classes, requirements, and software components), and Axiom 2 deals with the complexity of design. Axiom 1. *The independence axiom.* Maintain the independence of components. Axiom 2. *The information axiom.* Minimize the information content of the design. Axiom 1 states that, during the design process, as we go from requirement and use case to a system component, each component must satisfy that requirement without affecting other requirements.

Axiom 2 is concerned with simplicity. Occam's razor says that, "The best theory explains the known facts with a minimum amount of complexity and maximum simplicity and straightforwardness."

COROLLARIES AND ITS RELATIONSHIP WITH THE TWO AXIOMS From the two design axioms, many corollaries may be derived as a direct consequence of the axioms. These corollaries may be more useful in making specific design decisions, since they can be applied to actual situations more easily than the original axioms. They even may be called *design rules,* and all are derived from the two basic axioms



The origin of corollaries. Corollaries 1, 2, and 3 are from both axioms, whereas corollary4 is from axiom 1 and corollaries 5 and 6 are from axiom 2. **Corollary 1.** ***Uncoupled design with less information content.*** Highly cohesive objects can improve coupling because only a minimal amount of essential information need be passed between objects.

Corollary 2. ***Single purpose***. Each class must have a single, clearly defined purpose. When you document, you should be able to easily describe the purpose of a class in a few sentences.

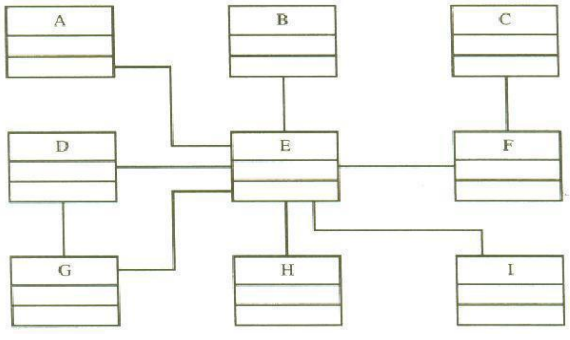
Corollary 3. ***Large number of simple classes***. Keeping the classes simple allows reusability.

Corollary 4. ***Strong mapping.*** There must be a strong association between the physical system (analysis's object) and logical design (design's object).

Corollary 5. ***Standardization.*** Promote standardization by designing interchangeable components and reusing existing classes or components.

Corollary 6. ***Design with inheritance.*** Common behavior (methods) must be moved to super classes. The super class-subclass structure must make logical sense.

Corollary 1. Uncoupled Design with Less Information Content The main goal here is to maximize objects cohesiveness among objects and software components in order to improve coupling because only a minimal amount of essential information need be passed between components. *Coupling* is a measure of the strength of association established by a connection from one object or software component to another. Coupling is a binary relationship: A is coupled with B. The degree of coupling is a function of



1. How complicated the connection is.
2. Whether the connection refers to the object itself or something inside it.
3. What is being sent or received.

Coupling increases (becomes stronger) with increasing complexity or obscurity of the interface. Coupling decreases (becomes lower) when the connection is to the component interface rather than to an internal component

Table contains different types of interaction couplings. Inheritance is a form of coupling between super- and subclasses. A subclass is coupled to its superclass in terms of attributes and methods. Unlike interaction coupling, high inheritance coupling is desirable

TYPES OF COUPLING AMONG OBJECTS OR COMPONENTS (shown from highest to lowest)

Degree of coupling	Name	Description
Very high	Content coupling	The connection involves direct reference to attributes or methods of another object.
High	Common coupling	The connection involves two objects accessing a "global data space," for both to read and write.
Medium	Control coupling	The connection involves explicit control of the processing logic of one object by another.
Low	Stamp coupling	The connection involves passing an aggregate data structure to another object, which uses only a portion of the components of the data structure.
Very low	Data coupling	The connection involves either simple data items or aggregate structures all of whose elements are used by the receiving object. This should be the goal of an architectural design.

Cohesion Coupling deals with interactions between objects or software components. We also need to consider interactions within a single object or software .component, called *cohesion.* Cohesion reflects the "single-purposeness" of an object. Highly cohesive components can lower coupling because only a minimum of essential information need be passed between components. Cohesion also helps in designing classes that have very specific goals and clearly defined purposes. Method cohesion, like function cohesion, means that a method should carry only one function. **Corollary 2. Single Purpose** Every class should be clearly defined and necessary in the context of achieving the system's goals. When you document a class, you should be able to easily explain its purpose in a sentence or two. If you cannot, then rethink the class and try to subdivide it into more independent pieces. In summary, keep it simple; to be more precise, each method must provide only one service. Each method should be of moderate size, no more than a page; half a page is better.

Corollary 3. Large Number of Simpler Classes, Reusability The less specialized the classes are, the more likely future problems can be solved by a recombination of existing classes, adding a minimal number of subclasses. A class that easily can be understood and reused (or inherited) contributes to the overall system, while a complex, poorly designed class is just so much dead weight and usually cannot be reused Coad and Yourdon describe four reasons why people are not utilizing this concept:

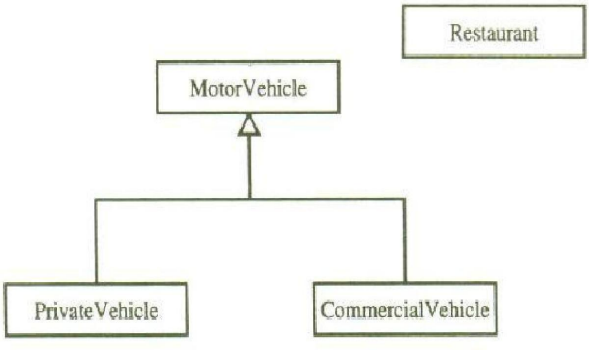
1. Software engineering textbooks teach new practitioners to build systems from "first principles"; reusability is not promoted or even discussed.
2. The "not invented here" syndrome and the intellectual challenge of solving an interesting software problem in one's own unique way mitigates against reusing someone else's software component.
3. Unsuccessful experiences with software reusability in the past have convinced many practitioners and development managers that the concept is not practical.
4. Most organizations provide no reward for reusability; sometimes productivity is measured in tenns of new lines of code written plus a discounted credit (e.g., 50 percent less credit) for reused lines of code.

Griss argues that, although reuse is widely desired and often the...benefit of utilizing object technology, many object - oriented reuse efforts fail because of too narrow a focus on technology and not on the policies set forth by an organization. He recommended an institutionalized approach to software development, in which software assets intentionally are created or acquired to be reusable. **Corollary 4. Strong Mapping** During the design phase, we need to design this class design its methods, its association with other objects, and its view and access classes. A strong mapping links classes identified during analysis and classes designed during the design phase (e.g., view and access classes). With OO techniques, the same paradigm is used for analysis, design, and implementation. The analyst identifies objects' types and inheritance, and thinks about events that change the state of objects. The designer adds detail to this model perhaps designing screens, user interaction, and client-server interaction. The thought process flows so naturally from analyst to design that it may be difficult to tell where analysis ends and design begins.

Corollary 5. Standardization Similarly, object-oriented systems are like organic systems, meaning that they grow as you create new applications. The knowledge of existing classes will help you determine what new classes are needed to accomplish the tasks and where you might inherit useful behavior rather than reinvent the wheel. Furthermore, class libraries must be easily searched, based

on users' criteria. For example, users should be able to search the class repository with commands like "show me all Facet classes. " The concept of design patterns might provide a way to capture the design knowledge, document it, and store it in a repository that can be shared and reused in different applications.

Corollary 6. Designing with Inheritance When you implement a class, you have to determine its ancestor, what attributes it will have, and what messages it will understand. Then, you have to construct its methods and protocols. Ideally, you will choose inheritance to minimize the amount of program instructions This is a simple, easy to understand design, although somewhat limited in the reusability of the classes. For example, if in another project you must build a system that models a vehicle assembly plant, the classes from the licensing application are not appropriate, since these classes have instructions and data that deal with the legal requirements of motor vehicle license acquisition and renewal.

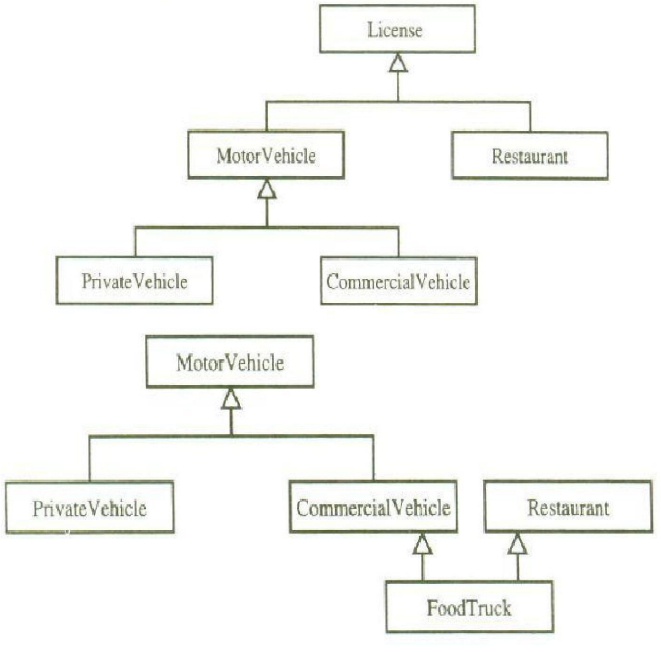


Achieving Multiple Inheritance in a Single Inheritance System *Single inheritance* means that each class has only a single superclass. This technique is used in Smalltalk and several other object-oriented systems. One result of using a single inheritance hierarchy is the absence of ambiguity as to how an object will respond to a given method.

Fig: The initial single inheritance design.

Explain relationship analysis for the ATM banking system. To better gain experience in object relationship analysis, we use the familiar bank system case and apply the concepts for identifying associations, super sub relationships, and a-part-of relationships for the classes identified. Furthermore, object-oriented analysis and design are performed in an iterative process using class diagrams This iterative process is unlike the traditional waterfall technique, in which all analysis is completed before design begins

The single inheritance design modified to allow licensing food trucks.

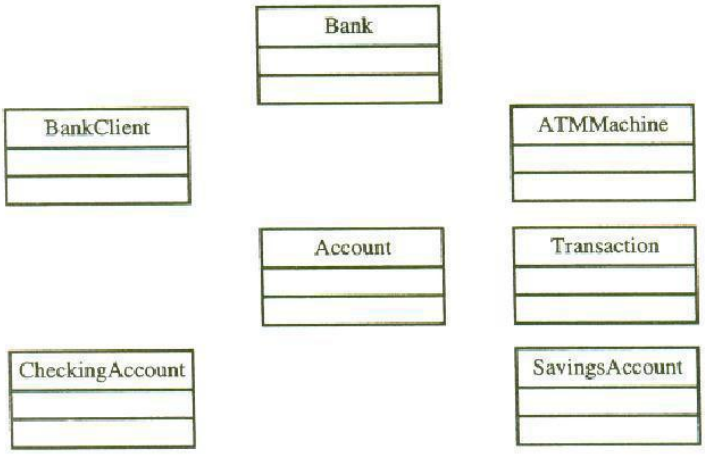


Identifying Classes' Relationships

One of the strengths of object-oriented analysis is the ability to model objects as they exist in the real world. To accurately do this, you must be able to model more than just an object's internal workings. You also must be able to model how objects relate to each other. Several different relationships exist in the ViaNet bank ATM system, so we need to define them

Developing a UML Class Diagram Based on the Use-Case Analysis

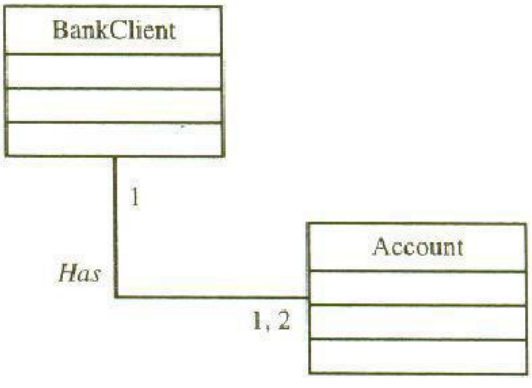
The UML class diagram is the main static analysis and design diagram of a system. The analysis generally consists of the following class diagrams .One class diagram for the system, which shows the identity and definition of classes in the system, their interrelationships, and various packages containing groupings of classes.



UML class diagram for the ViaNet bank ATM system. Some CASE tools such as the SA/Object Architect can automatically define classes and draw them from use cases or collaboration/ sequence diagrams. However, presently, it cannot identify all the classes. For this example, S/A Object was able to identify only the BankClient class. Multiple class diagrams that represent various pieces, or views, of the system class diagram. Multiple class diagrams, that show the specific static relationships between various classes.

Defining Association Relationships

Identifying association begins by analyzing the interactions of each class. Remember that any dependency between two or more classes is an association. The following are general guidelines for identifying the tentative associations, as explained in this chapter: .Association often corresponds to verb or prepositional phrases, such as part of, next to, works for, or contained in. A reference from one class to another is an association. Some associations are implicit or taken from general knowledge.



Other associations and their cardinalities are defined in Table 8-1 and demonstrated in Figure

SOME ASSOCIATIONS AND THEIR CARDINALITIES IN THE BANK SYSTEM

Class	Related class	Association name	Cardinality
Account	BankClient	Has	One
BankClient	Account		One or two
SavingsAccount	CheckingAccount	Savings-Checking	One
CheckingAccount	SavingsAccount		Zero or one
Account	Transaction	Account-Transaction	Zero or more
Transaction	Account		One

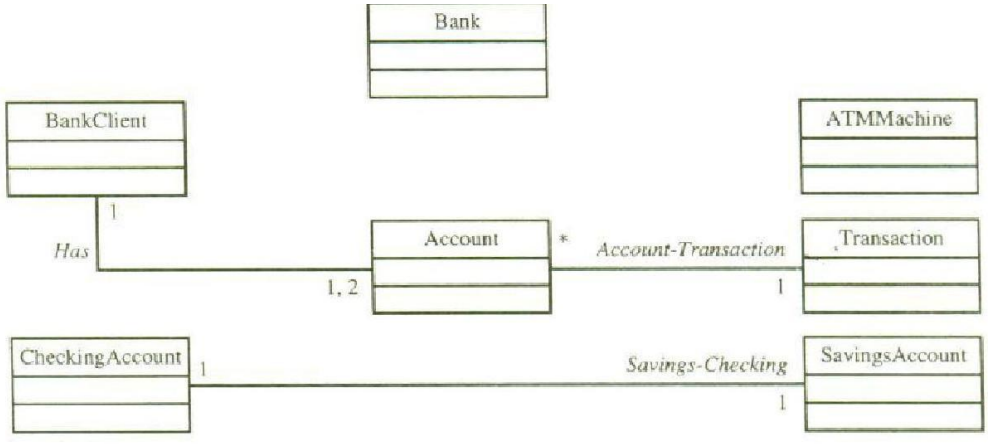
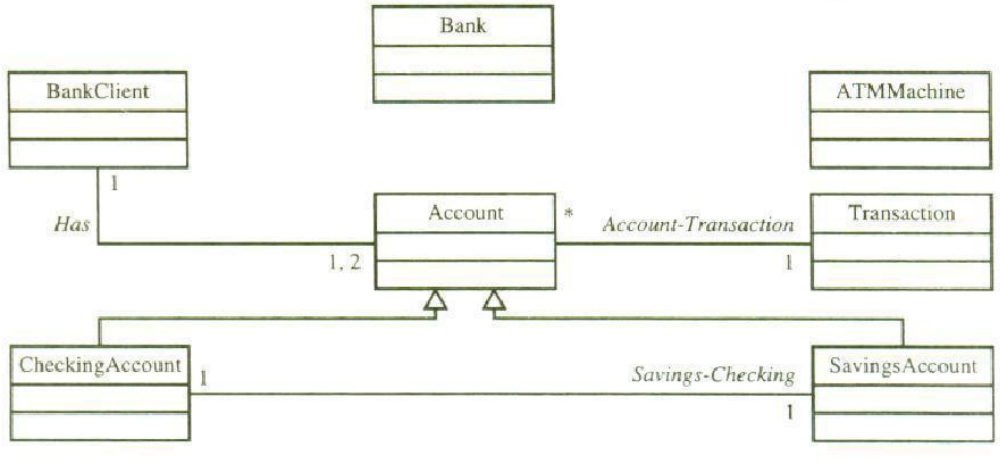


FIG: Associations among the ViaNet bank ATMsystem classes.

Identifying the Aggregation/a-Part-of Relationship To identify a-part-of structures, we look for the following clues: .Assembly. A physical whole is constructed from physical parts. . Container. A physical whole encompasses but is not constructed from physical parts. . Collection-Member. A conceptual whole encompasses parts that may be physical or conceptual.

Super-sub relationships among the Account, SavingsAccount, and CheckingAccount classes.



Association, generalization, and aggregation among the ViaNet bank classes. Notice that the super-sub arrows for CheckingAccount and SavingsAccount have merged. The relationship between BankClient and ATMMachine is an interface.

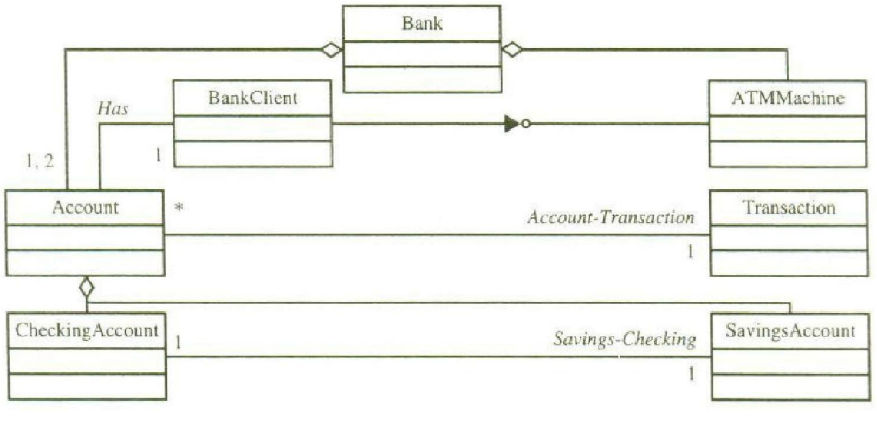


Figure depicts the association, generalization, and aggregation among the bank systems classes. If you are wondering what the relationship between the Bank Client and ATMMachine is, it is an interface. Identifying a class interface is a design activity of object-oriented system development.