UNIT – III

OBJECT ORIENTED ANALYSIS: USE-CASE DRIVEN

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 first step in finding an appropriate solution to a given problem is to understand the problem and its domain.

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. This is accomplished by constructing several models of the system that concentrate on describing what the system does rather than how it does it. Separating the behavior of a system from the way that behavior is implemented requires viewing the system from the perspective of the user rather than that of the machine. Analysis is the process of transforming a problem definition from a fuzzy set of facts and myths into a coherent statement of a system's requirements.

3.1 WHY ANALYSIS IS A DIFFICULT ACTIVITY

Analysis is a creative activity that involves understanding the problem, its associated constraints, and methods of overcoming those constraints. This is an iterative process that goes on until the problem is well understood. Norman explains the three most common sources of requirement difficulties:

1. Fuzzy descriptions - such as "fast response time" or "very easy and very secure updating mechanisms." A requirement such as fast response time is open to interpretation, which might lead to user dissatisfaction if the user's interpretation of a fast response is different from the systems analyst's interpretation

2. Incomplete requirements - mean that certain requirements necessary for successful system development are not included for a variety of reasons. These reasons could include the users' forgetting to identify them, high cost, politics within the business,or oversight by the system developer. However, because of the iterative nature of object- oriented analysis and the unified approach most of the incomplete requirements can be identified in subsequent tries.

3. Unnecessary features - When addressing features of the system, keep in mind that every additional feature could affect the performance, complexity, stability, maintenance, and support costs of an application. Features implemented by a small extension to the application code do not necessarily have a proportionally small effect on a user interface.

Analysis is a difficult activity. You must understand the problem in some application domain and then define a solution that can be implemented with software. Experience often is the best teacher. If the first try reflects the errors of an incomplete understanding of the problems, refine the application and try another run.

3.2 BUSINESS OB.JECT ANALYSIS: UNDERSTANDING THE BUSINESS LAYER

Business object analysis is a process of understanding the system's requirements and establishing the goals of an application. The main intent of this activity is to understand users' requirements. The outcome of the business object analysis is to identify classes that make up the business layer and the relationships that play a role in achieving system goals.

3.3 USECASE DRIVEN OBJECT-ORIENTED ANALYSIS: THE UNIFIEDAPPROACH

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 interactiondiagrams:

* Determine thesequence.
* 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.

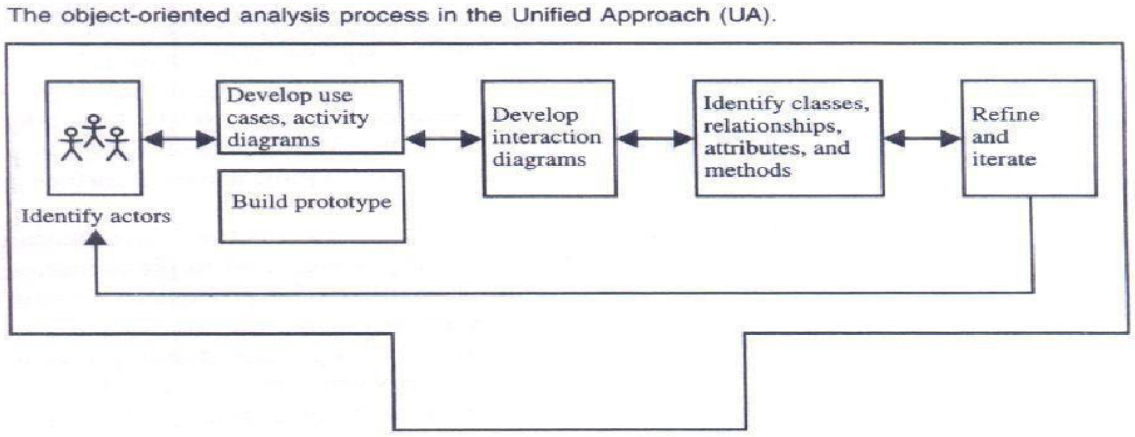


Fig 3.1. Object Oriented analysis process in the Unified Approach (UA)

3.4 BUSINESS PROCESS MODELING

This may include modeling as-is processes and the applications that support them and any number of phased, would-be models of reengineered processes or implementation of the system. These activities would be enhanced and supported by using an activity diagram. Business process modeling can be very time consuming, so the main idea should be to geta 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. (See Fig- 3.2) Developing an activity diagram of the business process can give us a better understanding of what sort of activities are performed in a library by a library member.

FIGURE

This activity diagram (AD) shows some activities that can be performed by a library member.

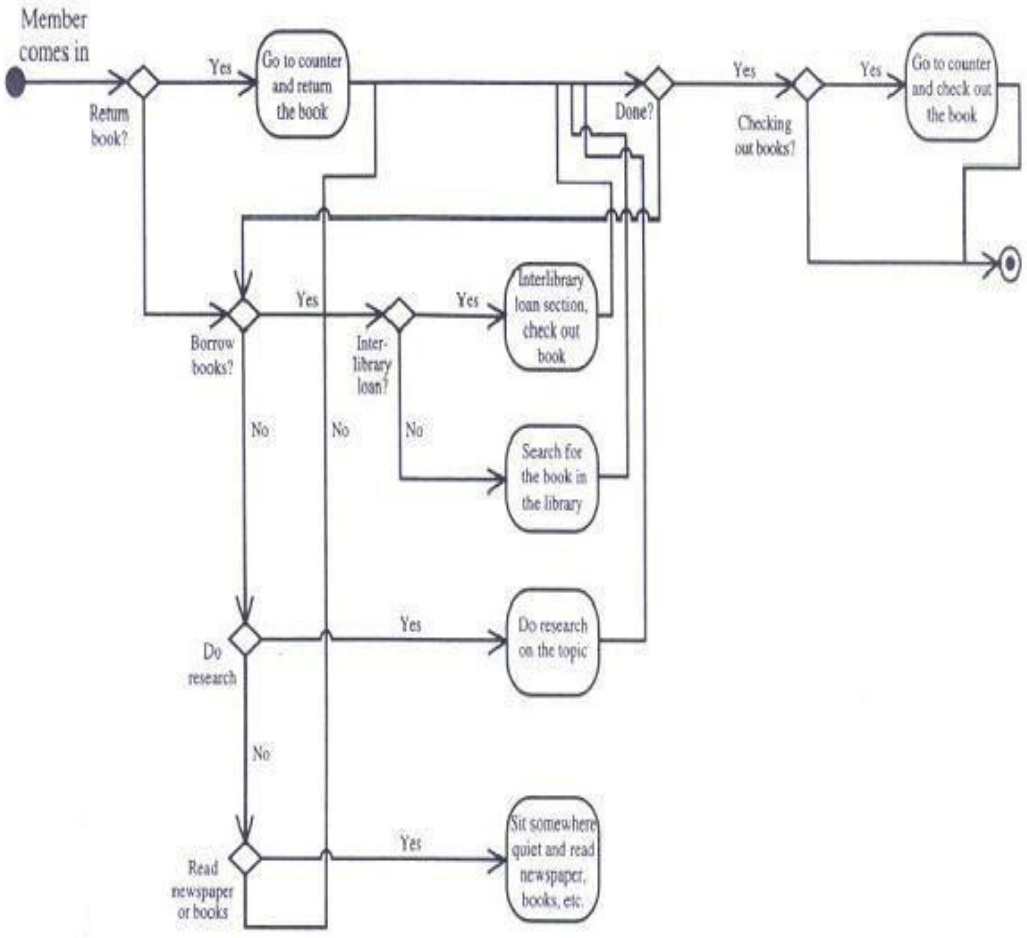


Fig. 3.2. This activity diagram shows some activities that can be performed by a library member.

3.5 USE-CASE MODEL

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). For example, take a car; typical uses of a car include "take you different places" or "haul your stuff" or a user may wantto use it "off the road."
- The use-case model describes the uses of the system and shows the courses of eventsthat can be performed.
- A use-case model also can discover classes and the relationships among subsystems ofthe systems.
- Use-case model can be developed by talking to typical users and discussing thevarious things they might want to do with the application being prepared.
- 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 fewparagraphs.
- 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(see Figure 3.3).
- The use-case model expresses what the business or application will do and not how;that is the responsibility of the UML class diagram

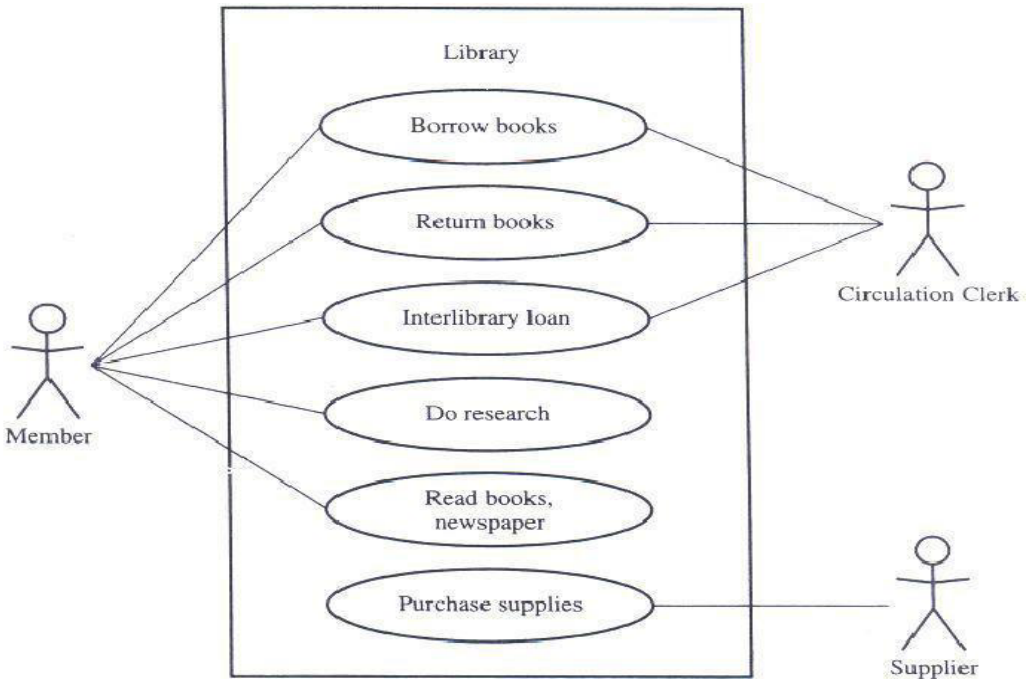


Fig 3.3. Use Case Diagram – Library System

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 object model shows how the business is run. Jacobson, Ericsson, and Jacobson call the use-case model a "what model," in contrast to the object model, whichis a "how model.".

Guidelines for developing Use Case Models:

1. Use Cases under the Microscope
2. Uses and Extends Associations
3. Identifying the Actors
4. Guidelines for Finding Use Cases
5. How Detailed Must a Use Case Be? When to Stop Decomposing and When to Continue
6. Dividing Use Cases into Packages
7. Naming a Use Case

1. Use Cases under the Microscope:

Use cases represent the things that the user is doing with the system, which can be different from the users' goals.

Definition of use case by Jacobson "A Use Case is a sequence of transactions in a system whose task is to yield results of measurable value to an individual actor of the system."

Now let us take a look at the **key words of this definition:**

- **Use case** - Use case is a special flow of events through the system. By definition, many courses of events are possible and many of these are very similar.
- **Actors** - An actor is a user playing a role with respect to the system. When dealing withactors, it is important to think about roles rather than just people and their job titles.
- **Ina system** -. This simply means that the actors communicate with the system's use case.
- **A measurable value**- A use case must help the actor to perform a task that has some identifiable value.
- **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.

The following are some examples of use cases for the library (see Figure 3.4).

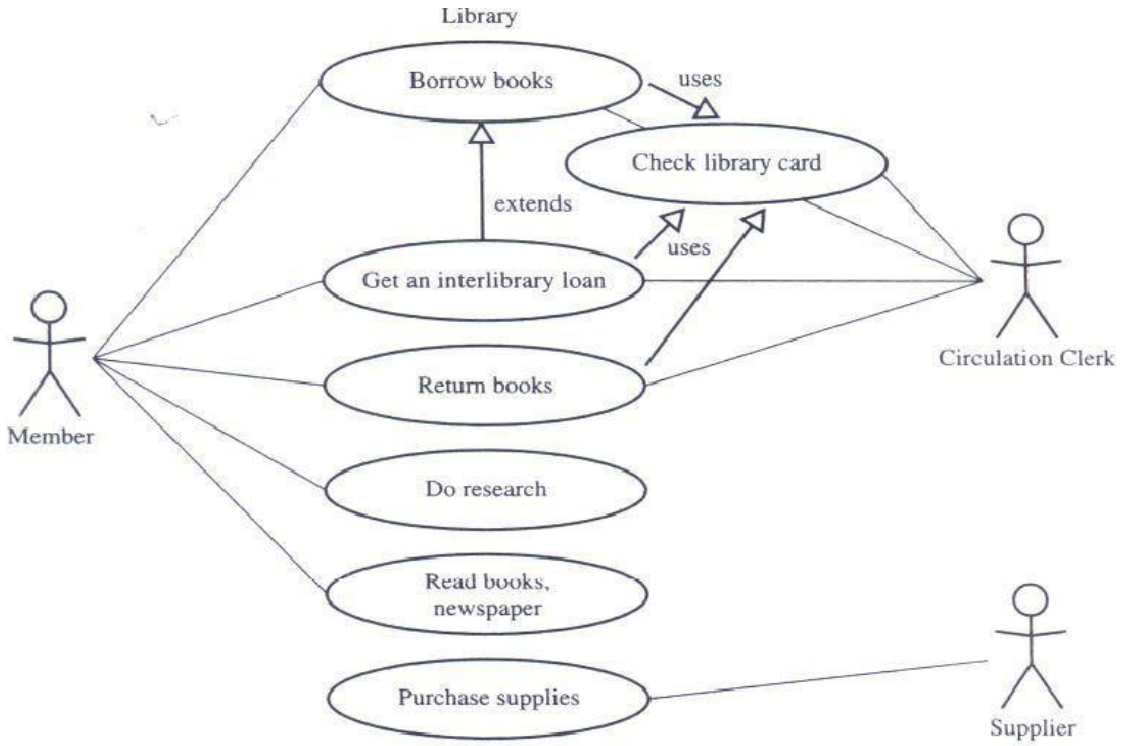


Fig. 3.4. A member, a circulation clerk, and a supplier.

1. **Use-case name: Borrow books**. A member takes books from the library to read at home, registering them at the checkout desk so the library can keep track of its books. Depending on the member's record, different courses of events will follow.
2. **Use-case name: Get an interlibrary loan**. A member requests a book that the library does not have. The book is located at another library and ordered through an interlibrary loan.
3. **Use-case name: Return books**. A member brings borrowed books back to the library.
4. **Use-case name: Check library card.** A member submits his or her library card to the clerk, who checks the borrower's record.
5. **Use-case name: Do research**. A member comes to the library to do research. The member can search in a variety of ways (such as through books, journals, CDROM, WWW) to find information on the subjects of that research.
6. **Use-case name: Read books, newspaper**. A member comes to the library for aquiet place to study or read a newspaper, journal, or book.
7. **Use-case name: Purchase supplies**. The supplier provides the books, journals, and newspapers purchased by the library.

2. Uses and Extends Associations:

A use-case description can be difficult to understand if it contains too many alternatives or exceptional flows of events that are performed only if certain conditions are met as the use-case instance is carried out.

A way to simplify the description is to take advantage of extends and uses associations. The extends association is used when you have one use case that is similar to another use case but does a bit more or is more specialized; in essence, it is like a subclass.

The uses association occurs when you are describing your use cases and notice that some of them have sub flows in common. To avoid describing a sub flow more than once in several use cases, you can extract the common sub flow and make it a use case of its own. This new use case then can be used by other use cases. The relationships among the other use cases and this new extracted use case are called a uses association. The uses association helps us avoid redundancy by allowing a use case to be shared. For example, checking a library card is common among the borrow books, return books, and interlibrary loan use cases (see Figure 3.4).

The similarity between extends and uses associations is that both can be viewed as a kind of inheritance. When you want to share common sequences in several use cases, utilize the uses association by extracting common sequences into a new, shared use case. The extends association is found when you add a bit more specialized, new use case that extends some of the use cases that you have.

Use cases could be viewed as concrete or abstract. An abstract use case is not complete and has no initiation actors but is used by a concrete use case, which does interact with actors. This inheritance could be used at several levels. Abstract use cases also are the use cases that have uses or extends associations.

Fowler and Scott provide us excellent (guidelines for addressing variations in use case modeling : .

1. Capture the simple and normal use case first.
2. For every step in that use case, ask
   *. What could go wrong here?
   *. How might this work out differently?
3. Extract common sequences into a new, shared use case with the uses association. If you are adding more specialized or exceptional uses cases, take advantage of usecases you already have with the extends association.

3. Identifying the Actors:

Identifying the actors is (at least) as important as identifying classes, structures, associations, attributes, and behavior.

The term actor represents the role a user plays with respect to the system. When dealing with actors, it is important to think about roles rather than people or job titles.

A user may play more than one role. For instance, a member of a public library also may play the role of volunteer at the help desk in the library. However, an actor should represent a single user; in the library example, the member can perform tasks some of which can be done by others and others that are unique. However, try to isolate the roles that the users can play. (Fig 3.5)

You have to identify the actors and understand how they will use and interact with the system. In a thought-provoking book on requirement analysis, Gause and Weinberg , explain what is known as the railroad paradox:

*When trying to find all users, we need to beware of the **Railroad Paradox**. When railroads were asked to establish new stops on the schedule, they "studied the requirements," by sending someone to the station at the designated time to see if anyone was waiting for a train. Of course, nobody was there because no stop was scheduled, so the railroad turned down the request because there was no demand.*

Gause and Weinberg concluded that the railroad paradox appears everywhere thereare products and goes like this (which should be avoided):

1. The product is not satisfying the users.
2. Since the product is not satisfactory, potential users will not use it.
3. Potential users ask for a better product.
4. Because the potential users do not use the product, the request is denied.

Therefore, since the product does not meet the needs of some users, they are not identified as potential users of a better product. They are not consulted and the product stays bad. The railroad paradox suggests that a new product actually can create users where none existed before Candidates for actors can be found through the answers to thefollowing questions:

- Who is using the system? Or, who is affected by the system? Or, which groups need help from the system to perform a task?
- Who affects the system? Or, which user groups are needed by the system to perform itsfunctions? These functions can be both main functions and secondary functions, suchas administration.
- Which external hardware or other systems (if any) use the system to perform tasks?
- What problems does this application solve (that is, for whom)? And, finally, howdo users use the system (use case)? What are they doing with the system?

When requirements for new applications are modeled and designed by a group that excludes the targeted users, not only will the application not meet the users' needs, but potential users will feel no involvement in the process and not be committed to giving the application a good try. Always remember Veblen's principle: 'There's no change, no matter how awful, that won't benefit some people; and no change, no matter how good, that won't hurt some.”,

Another issue worth mentioning is that **actors need not be human**, although actors are represented as stick figures within a use case diagram. An actor alsocan be an external system. For example, an accounting system that needs information from a system to update its accounts is an actor in that system.

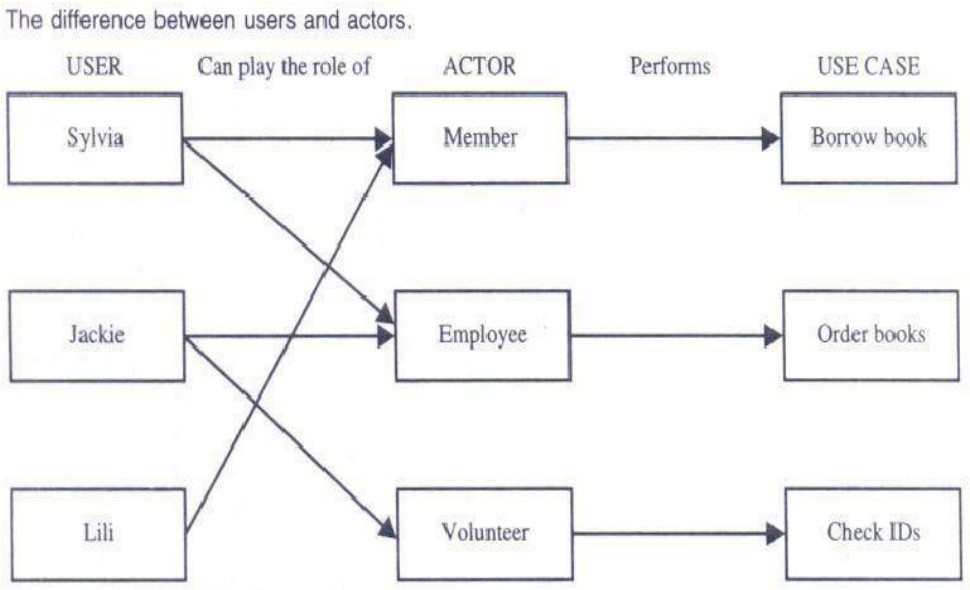


Fig. 3.5. The difference between users and actors

4. Guidelines for Finding Use Cases:

When you have defined a set of actors, it is time to describe the way they interact with the system. This should be carried out sequentially, but an iterated approach may be necessary.
Here are the **steps** for **finding use cases:**

1. For each actor, find the tasks and functions that the actor should be able to perform or that the system needs the actor to perform. The use case should represent a course of events that leads to a clear goal (or, in some cases, several distinct goals that could be alternatives for the actor or for the system).
2. Name the use cases
3. Describe the use cases briefly by applying terms with which the user is familiar. This makes the description less ambiguous.

Once you have identified the use-cases candidates, it may not be apparent that all of these use cases need to be described separately; some may be modeled as variants of others. Consider what the actors want to do.

It is important to separate actors from users. The actors each represent a role that one or several users can play. Therefore, it is not necessary to model different actors that can perform the same use case in the same way. The approach should allow different users to be different actors and play one role when performing a particular actor's use case. Thus, each use case has only one main actor. To achieve this, you have to

- **Isolate users from actors**.
- **Isolate actors from other actors** (separate the responsibilities of each actor).
- **Isolate use cases that have different initiating actors and slightly different behavior** (If the actor had been the same, this would be modeled by a use-case alternativebehavior).

5. How Detailed Must a Use Case Be? When to Stop Decomposing and When to Continue

A use case, as already explained, describes the courses of events that will be carried out by the system. Jacobson et al. believe that, in most cases, too much detail maynot be very useful.

During analysis of a business system, you can develop one use-case diagram asthe system use case and draw packages on this use case to represent the various business domains of the system. For each package, you may create a child usecase diagram. On each child use-case diagram, you can draw all of the use cases of the domain, with actions and interactions. You can further refine the way the use cases are categorized.The extends and uses relationships can be used to eliminate redundant modeling of scenarios.

When should use cases be employed? Use cases are an essential tool in capturing requirements and planning and controlling any software development project.

Capturing use cases is a primary task of the analysis phase. Although most use cases are captured at the beginning of the project, you will uncover more as you proceed.

The UML specification recommends that at least one scenario be prepared for each significantly different kind of use case instance

6. Dividing Use Cases into Packages

Each use case represents a particular scenario in the system.

You may model either how the system currently works or how you want it to work.

A design is broken down into packages.

You must narrow the focus of the scenarios in your system.

For example, in a library system, the various scenarios involve a supplier providing books or a member doing research or borrowing books. In this case, there should be threeseparate packages, one each for Borrow books, Do research, and Purchase books.

Many applications may be associated with the library system and one or more databases used to store the information (see Figure 3.6).

7. Naming a Use Case

Use-case names should provide a general description of the use-case function.

The name should express what happens when an instance of the use case is performed."

Jacobson et al. recommend that the name should be active, often expressed in the form of **a verb** (Borrow) or **verb and noun** (Borrow books).

The naming should be done with care; the description of the use case should be descriptive and consistent.

For example, the use case that describes what happens when a person deposits money into an ATM machine could be named either receive money or deposit money. A library system can be divided into many packages, each of which encompasses multiple use cases.

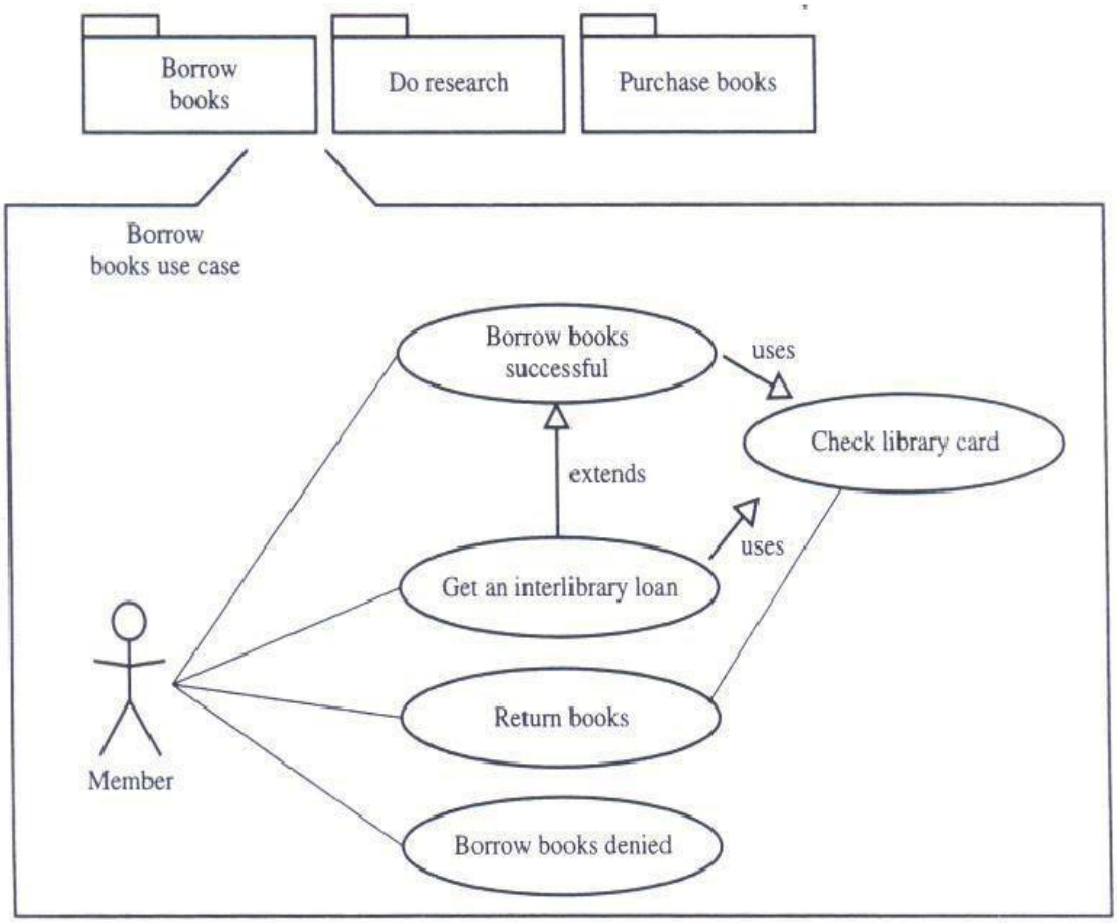


Fig. 3.6. A library system can be divided into many packages, each of whichencompasses multiple use cases.

3.6 DEVELOPING EFFECTIVE DOCUMENTATION

Documenting your project helps reveal issues and gaps in the analysis and design. A document can serve as a communication vehicle among the project's team members, or it can serve as an initial understanding of the requirements. Blum concludes that management has responsibility for resources such as software, hardware, and operational expenses.

In many projects, documentation can be an important factor in making a decision about committing resources. Application software is expected to provide a solution to a problem. It is very difficult, if not impossible, to document a poorly understood problem. The main issue in documentation during the analysis phase is to determine what the system must do. Decisions about how the system works are delayed to the design phase. Blum raises the following questions for determining the importance of documentation: How will a document be used? (If it will not be used, it is not necessary.) What is the objective of the document? What is the management view of the document? Who are the readers of the document?

1. Organization Conventions for Documentation

The documentation depends on the organization's rules and regulations. Most organizations have established standards or conventions for developing documentation. However, in many organizations, the standards border on the nonexistent. In other cases, the standards may be excessive. Too little documentation invites disaster; too much documentation, as Blum put it, transfers energy from the problem-solving tasks to a mechanical and unrewarding activity. Each organization determines what is best for it, and you must respond to that definition and refinement.

Bell and Evans provide us with guidelines and a template for preparing a document that has been adapted for documenting the unified approach's systems development Remember that your modeling effort becomes the analysis, design, and testing documentation. However, this template which is based on the unified approach lifecycle assists you in organizing and composing your models into an effectivedocumentation.

2. Guidelines for Developing Effective Documentation

Bell and Evans provide us the following guidelines for making documents fit theneeds and expectations of your audience:

1. **1.Common cover.** All documents should share a common cover sheet that identifiesthe document, the current version, and the individual responsible for the content. As thedocument proceeds through the life cycle phases, the responsible individual may change. That change must be reflected in the cover sheet.
2. **2.80-20 rule**. As for many applications, the 80-20 rule generally applies for documentation: 80 percent of the work can be done with 20 percent of the documentation. The trick is to make sure that the 20 percent is easily accessible and the rest (80 percent) is available to those (few) who need to know.
3. **Familiar vocabulary.** The formality of a document will depend on how it is used and who will read it. When developing a documentation use a vocabulary that your readers understand and are comfortable with. The main objective here is to communicate with readers and not impress them with buzz words.
4. **Make the document as short as possible**. Assume that you are developing a manual. The key in developing an effective manual is to eliminate all repetition; present summaries, reviews, organization chapters in less than three s; and make chapter headings task oriented so that the table of contents also could serve as an index.
5. **Organize the document**. Use the rules of good organization (such as the organization’s standards, college handbooks, Strunk and White's Elements of Style or the University of Chicago Manual of Style) within each section. Appendix A provides a template for developing documentation for a project. Most CASE tools provide documentation capability by providing customizable reports. The purpose of these guidelines is to assist you in creating an effective documentation.

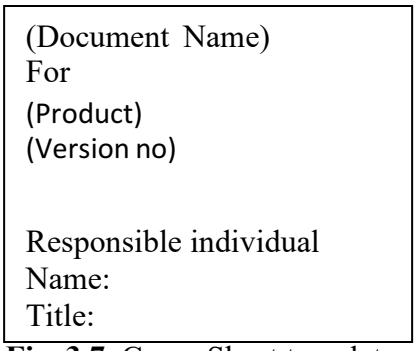


Fig. 3.7. Cover Sheet template.

3.7 Case Study: ANALYSING THE VIANET BANK ATM – THE USE CASE DRIVEN PROCESS

The Following section provides the description of the vianet bank atm system's requirement.

- The Bank client must be able to deposit an amount to and withdraw an amount fromhis or her accounts using the touch screen at the vianet bank atm. Each transaction must be recorded, and the client must be able to review all transactions performed against the given account. Recorded transactions must include the date, time, Transaction type,amount and account balance after the transactions.
- A ViaNet bank client can have two types of accounts: a checking account and savings account. For each checking account, one related savings account can exist.
- Access to the ViaNet bank accounts is Provided by a PIN code consisting of four integerdigits between 0 and 9.
- One PIN code allows access to all accounts held by a bankclient.
- No receipts will be provided for any account transactions.
- The bank application operates for a single banking institution only.
- Neither a checking nor a savings account can have a negative balance. The system should automatically withdraw money from a related savings account if the requested withdrawal amount on the checking account is more than its current balance. If the balance on a savings account is less than the withdrawal amount requested, the transaction will stop and the bank client will be notified.

i) Identifying Actors and Use Cases for the ViaNet Bank ATM System

The bank application will be used by one category of users: bank clients. Notice that identifying the actors of the system is an iterative process and can be modified as you learn more about the system. The actor of the bank system is the bank client. The bank client must be able to deposit an amount to and withdraw an amount from his or her accounts using the bank application. The following scenarios show use-case interactions between the actor (bank client) and the bank. In real life application these use cases are

created by system requirements, examination of existing system documentation, interviews, questionnaire, observation, etc.

- Use-case name: Bank ATM transaction. The bank clients interact with the bank system by going through the approval process. After the approval process, the bank clientcan perform the transaction. Here are the steps in the ATM transaction use case:

1. Insert ATM card.
2. Perform the approval process.
3. Ask type of transaction.
4. Enter type of transaction.
5. Perform transaction.
6. Eject card.
7. Request take card.
8. Take card.

These steps are shown in the Figure activity diagram. .

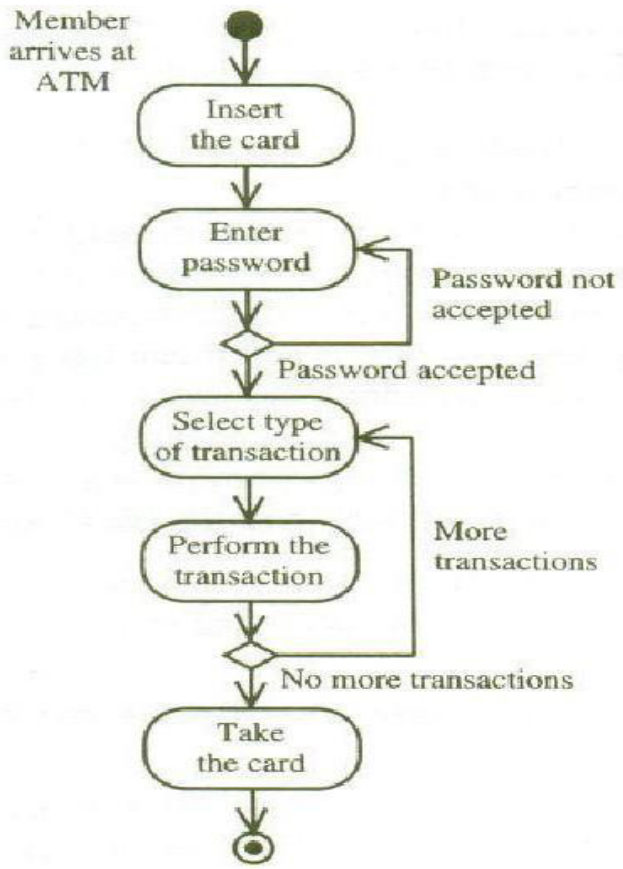


Fig. 3.8. Activities involved in an ATM transaction

- **Use-case name:** Approval process. The client enters a PIN code that consists of 4digits.Activities involved in an ATM transaction.

1. Request password.
2. Enter password.
3. Verify password.

- **Usecase name:** Invalid PIN. If the PIN code is not valid, an appropriate message is displayed to the client. This use case extends the approval process. (See Figure.)
- **Usecase name:** Deposit amount. The bank clients interact with the bank system after the approval process by requesting to deposit money to an account. The client selects the account for which a deposit is going to be made and enters an amount in dollar currency. The system creates a record of the transaction. (See Figure) This usecase extends the bank ATM transaction use case. Here are the steps:

1. Request account type.
2. Request deposit amount.
3. Enter deposit amount.
4. Put the check or cash in the envelope and insert it into ATM.

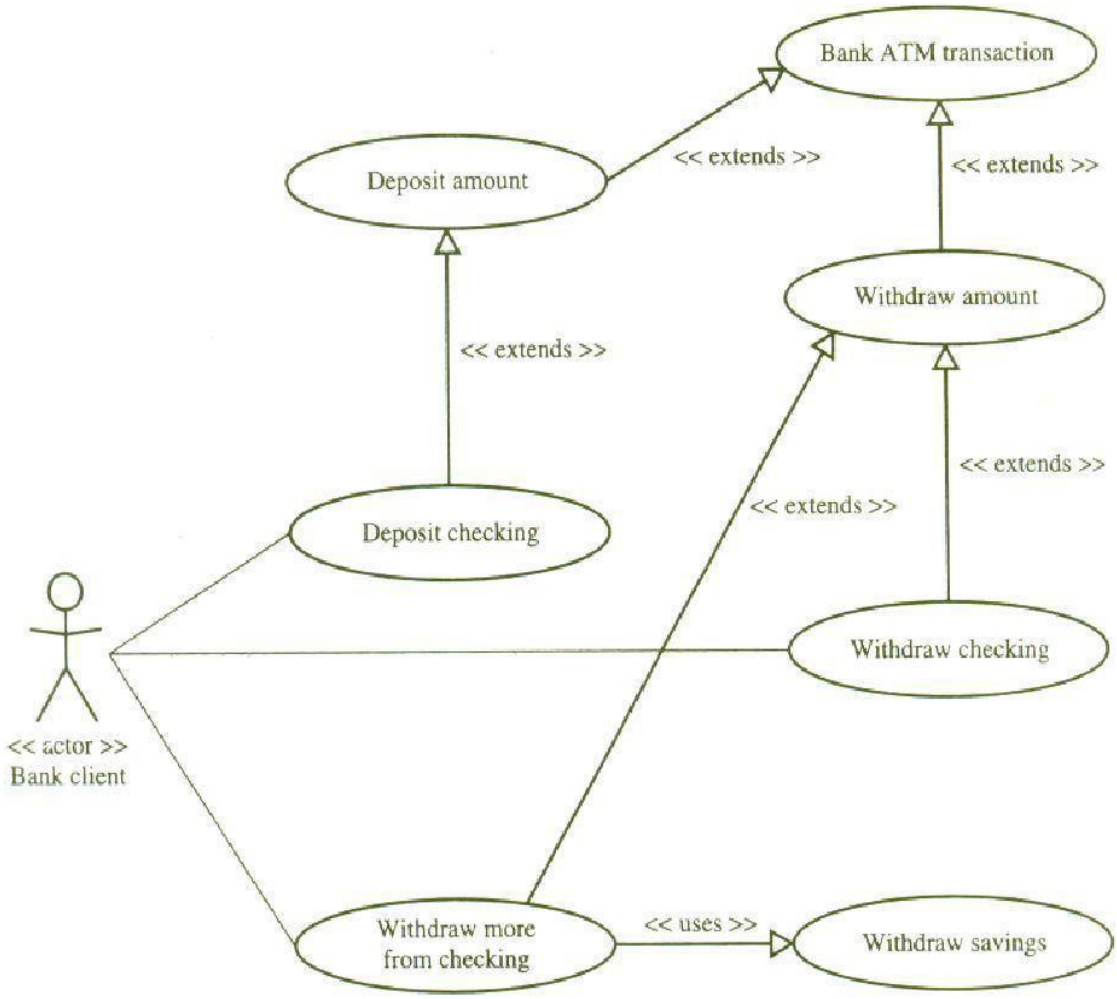


Fig. 3.9. Transaction use cases

- **Usecase name:** Deposit savings. The client selects the savings account for which a deposit is going to be made. All other steps are similar to the deposit amount use case. The system creates a record of the transaction. This use case extends the deposit amount use case. (See Figure 6-11.)

- **Usecase name:** Withdraw checking. The client tries to withdraw an amount from his or her checking account. If the amount is more than the checking account's balance, the insufficient amount is withdrawn from the related savings account. The system creates a record of the transaction and the withdrawal is successful. This use case extends the withdraw checking use case and uses the withdraw savings use case. (See Figure)
- **Usecase name**: Withdraw savings. The client tries to withdraw an amount from a savings account. The amount is less than or equal to the balance and the transaction is performed on the savings account. The system creates a record of the transaction since the withdrawal is successful. This use case extends the withdraw amount use case.

- **Usecase name:** Withdraw savings denied. The client withdraws an amount from a savings account. If the amount is more than the balance, the transaction is halted and a message is displayed. The savings account use-cases package. This use case extends the bank transaction use case. (See Figure 3.10))
- **Usecase name:** Savings transaction history. The bank client requests a history of transactions for a savings account. The system displays the transaction history for the savings account. This use case extends the bank transaction use case. (See Figure)

The use-case list contains at least one scenario of each significantly different kind of use-case instance. Each scenario shows a different sequence of interactions between actors and the system, with all decisions definite. If the scenario consists of an if statement, for each condition create one scenario.

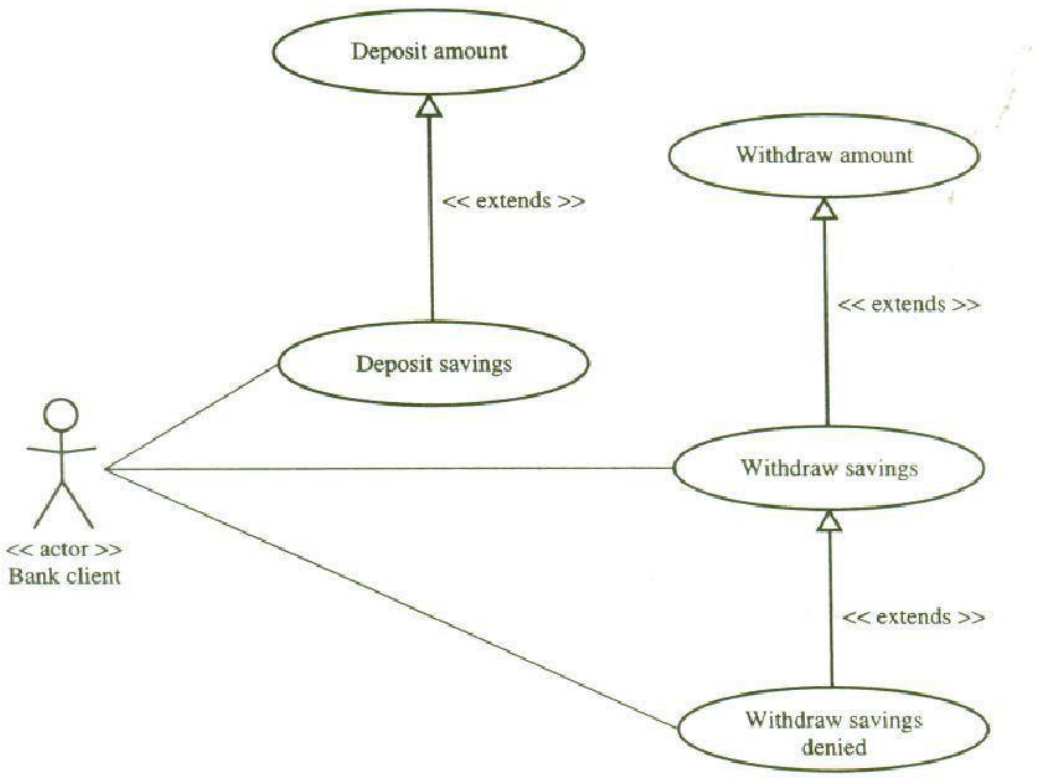


Fig. 3.10. The checking account use-cases

OBJECT ANALYSIS: CLASSIFICATION

3.8 CLASSIFICATIONS THEORY

Classification, the process of checking to see if an object belongs to a category or a class, is regarded as a basic attribute of human nature.

Booch explains that, intelligent classification is part of all good science. Classification guides us in making decisions about modularization. We maychoose to place certain classes and objects together in the same module or in different modules, depending upon the sameness we find among these declarations; coupling and cohesion are simply measures of this sameness. Classification also plays a role in allocating processes to procedures. We place certain processes together in the same processor or different processors, depending upon packaging, performance, or reliability concerns.

Human beings classify information every instant of their waking lives. We recognize the objects around us, and we move and act in relation to them. A human being is sophisticated information system, partly because he or she possesses a superior classification capability. For example, when you see a new model of a car, you have no trouble identifying it as a car. What has occurred here, even though you may never have seen this particular car before, is that you not only can immediately identify it as a car, but you also can guess the manufacturer and model. Clearly, you have some general idea of what cars look like, sound like, do, and are good for-you have a notion of car-kind or, in object-oriented terms, the class car.

Classes are an important mechanism for classifying objects. The chief role of class is to define the attributes, methods, and applicability of its instances. The class car, for example, defines the property color. Each individual car (formally, each instance of the class car) will have a value for this property, such as maroon, yellow, or white. It is early natural to partition the world into objects that have properties (attributes)and methods (behaviors). It is common and useful partitioning or classification, but we also routinely divide the world along a second dimension: We distinguish classes from instances.

A class is a specification of structure, behavior, and the description of an object. Classification is concerned more with identifying the class of an object than the individual objects within a system.

The problem of classification may be regarded as one of discriminating things,not between the individual objects but between classes, via the search for features or invariant attributes or behaviors among members of a class.

Classification can be defined as the categorization of input data (things) into identifiable classes via the extraction of significant features of attributes of the data from a background of irrelevant detail.

Another issue in relationships among classes is studied.

3.9 APPROACHES FOR IDENTIFYING CLASSES

In the following sections, we look at four alternative approaches for identifying classes:

1. The **Noun Phrase** approach;
2. The **Common Class Patterns** approach;
3. The **Usecase Driven, Sequence/Collaboration Modelling** approach;
4. The **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.

3.10. NOUN PHRASE APPROACH

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

All plurals are changed to singular, the nouns are listed, and the list divided into three categories (see Figure 3.11): **relevant classes, fuzzy classes** (the "fuzzyarea," classes we are not sure about), **and irrelevant classes**.

It is safe to scrap the irrelevant classes, which either have no purpose or will be unnecessary. Candidate classes then are selected from the other two categories. Keep in mind that identifying classes and developing a UML class diagram just like other activities is an iterative process. Depending on whether such object modeling is for the analysis or design phase of development, some classes may need to be added or removed from the model and, remember, flexibility is a virtue. You must be able to formulate a statement of purpose for each candidate class; if not, simply eliminate it.

Fig. 3.11. Using the noun phrase strategy, candidate classes can be divided into 3categories.

i) Identifying Tentative Classes

The following are **guidelines** for selecting classes in an application: .

- Look for nouns and noun phrases in the use cases. .
- Some classes are implicit or taken from general knowledge.

- All classes must make sense in the application domain; avoid computer implementation classes-defer them to the design stage.
- Carefully choose and define class names.

ii) 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.

- **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 thatis 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.
- **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. Does the object represented by the noun behave differently when the adjective is applied to it? If the use of the adjective signals that the behavior of the object is different, then make a new class". For example, Adult Members behave differently than Youth Members, so, the two should be classified as different classes.
- **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.
- **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.

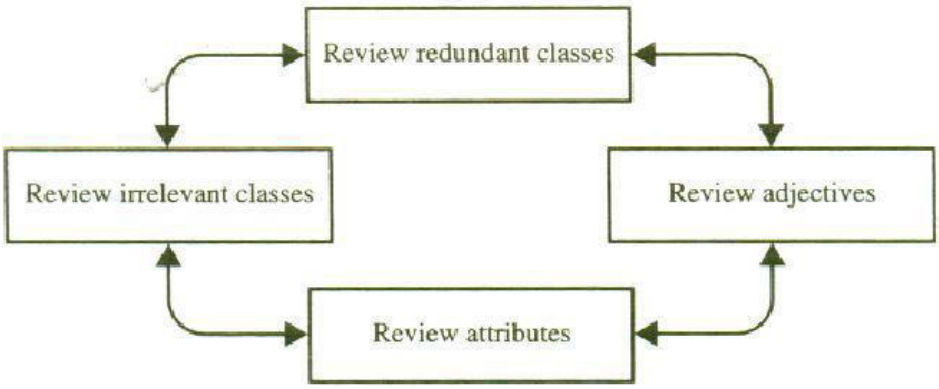


Fig. 3.12. The process of eliminating the redundant classes and refining the remaining classes is not sequential. You can move back and forth among these steps as often as you like.

Example: 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.

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 Balance
AmountApproval
Process ATM
Card
ATM Machine
Bank
Bank Client
Card
Cash
Check
Checking
Checking Account
Client
Client's Account
Currency Dollar
Envelope

Four Digits
Fund

Invalid PIN
Message
Money
Password
PIN

PIN Code
Record
Savings
Savings Account
Step.
System Transaction
History

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 Balance
AmountApproval
Process ATM
Card
ATM Machine
Bank
BankClient
Card
Cash
Check
Checking
Checking Account
Client
Client's Account
Currency Dollar
~~Envelope~~

~~Four Digits~~
Fund
Invalid PIN
Message
Money,
Password
PIN

PIN Code
Record
Savings
Savings Account

~~Step~~
System
Transaction -
Transaction History

Reviewing the Redundant Classes and Building a Common Vocabulary

We need to review the candidate list to see which classes are redundant. Ifdifferent words are being used to describe the same idea, we must select the one that isthe most **meaningful 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, BankClient Account, Client's Account PIN, PIN Code
Checking, Checking Account = BankClient (the term chosen)
Checking Account = Account
Checking Account = PIN

Checking Account = Checking Account
Savings, Savings Account = Savings Account
Fund, Money = Fund
ATM Card, Card = ATM Card
Here is the revised list of candidate classes:
Account
Account Balance
Amount Approval
Process ATM
Card

Bank
BankClient
~~Card~~
Cash
Check
~~Checking~~
Checking Account
~~Client~~
~~Client's account~~
Currency
Dollar
~~Envelope~~
~~Fund digits~~
Fund
Invalid PIN
Message
Message~~M~~
~~oney~~
Password
PIN

~~PIN Code~~
Record
~~Savings~~
Savings Account
~~Step~~
System
Transaction
Transaction History

Reviewing the Classes Containing Adjectives

We 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? If an adjective suggests a different kind of class or the class represented by the noun behaves differently when the adjective is applied to it, then we need to make a new class. However (it is a different use of the same object or the classis irrelevant, we must eliminate it) In this example, we have no classes containingadjectives that we can eliminate.

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.

Amount: a value, not a class.

Account Balance: An attribute of the Account class.
Invalid PIN: It is only a value, not a class.

Password: An attribute, possibly of the BankClient class.
Transaction History: An attribute, possibly of the Transaction class.
PIN: An attribute, possibly of the BankClientclass.

Here is the revised list of candidate classes. Notice that the eliminated classes are strikeouts (they have a line through them).

Account ~~Account~~
~~Balance Amount~~
Approval Process
ATM Card
Bank
BankClient
Cash
~~Card~~
Check
~~Checking~~
Checking Account
Currency
Dollar

~~Envelope~~
~~Fund digits~~
Fund
Message
MaBey
PIN

~~PIN Code~~
Record
Savings Account
System
~~step~~Transa
ction
~~Transaction History~~

Reviewing the Class Purpose

Identifying the classes that play a 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. If you cannot formulate a statement of purpose for a class, simply eliminate it. The classes that add no purpose to the system have been deleted from the list. 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.
- **BankClient 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.
- **Account class:** An Account class is a formal (or abstract) class, it defines the common behaviors that can be inherited by more specific classes such as CheckingAccount and SavingsAccount.
- **CheckingAccount class:** It models a client's checking account and provides more specialized withdrawal service.
- **savingsAccount class:** It models a client's savings account.
- **Transaction class:** Keeps track of transaction, time, date, type, amount, and 'balance.

3.11 COMMON CLASS PATTERN APPROACH

The second method for identifying classes is using *common class patterns*, which is **based on a knowledge base of the common classes that have been proposed by various researchers, such as Shlaer and Mellor [10], Ross[8], and Coad and Yourdon [3].** They have compiled and listed the following patterns for finding the candidate class and object :

- Name.**Concept class**

Context: A concept is a particular idea or understanding that we have of our world. The concept class encompasses principles that are not tangible but used to organize or keep track of business activities or communications. Marin and Odell describe concepts elegantly," Privately held ideas or notions are called **conceptions**. When an understanding is shared by another, it becomes a **concept.** To **communicate** with others, we must share our individually held conceptions and arrive at agreed concepts." Furthermore, Martin and Odell explain that, without concepts, mental life would be total chaos since every item we encountered wouldbe different.
*Example.*Performance is an example of concept class object.

- Name.**Events class**

Context: Events classes are **points in time that must be recorded**. Things happen, usually to something else at a given date and time or as a step in an ordered sequence. Associated with things remembered are attributes (after all, the things to remember are objects) such as who, what, when, where, how, or why.
Example. Landing, interrupt, request, and order are possible events.

- Name. ***Organization class***

Context:. An organization class is a **collection of people, resources, facilities, or groups to which the users belong; their capabilities** havea defined mission, whose existence is largely independent of the individuals.

Example. An accounting department might be considered a potential class.

- Name. ***People class*** (also known as person, roles, and roles played class)

Context. The people class **represents the different roles users play in interacting with the application**. People carry out some function. What roles does a personplay in the system? Coad and Yourdon [3] explain that a class which is represented by a person can be divided into two types: those representing users of the system, such as an operator or clerk who interacts with the system; and those representing people who donot use the system but about whom information is kept by the system.

Example. Employee, client, teacher, and manager are examples of people.

- Name. ***Places class***

Context. Places are **physical locations** that the system must keep information about.
Example. Buildings, stores, sites, and offices are examples of places.

- Name. ***Tangible things and devices class***

Context. This class **includes physical objects or groups of objects** that are tangible and devices with which the application interacts.

Example. Cars are an example of tangible things, and pressure sensors are an example of devices.

3.12 USE-CASE DRIVEN APPROACH: IDENTIFYING CLASSES AND THEIR BEHAVIORS THROUGH SEQUENCE/COLLABORATION MODELING

The use cases are employed to model the scenarios in the system and specify what external actors interact with the scenarios. The scenarios are described in text or through a sequence of steps. Use-case modeling is considered a problem-driven approach to object-oriented analysis, in that the designer first considers the problem at hand and not the relationship between objects, as in a data-driven approach.

Modeling with use cases is a recommended aid in finding the objects of a system and is the technique used by the unified approach. Once the system has been described in terms of its scenarios, the modeler can examine the textual description or steps of each scenario to determine what objects are needed for the scenario to occur. However, this is not a magical process in which you start with use cases, develop a sequence diagram, and voila, classes appear before your eyes.

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.

When building a new system, designers model the scenarios of the way the system of business should work. When redesigning an existing system, many modelers choose to first model the scenarios of the current system, and then model the scenarios of the way the system should work.

i) Implementation Of Scenarios

The UML specification recommends that at least one scenario be prepared for each significantly different use-case instance. Each scenario shows a different sequenceof interaction between actors and the system, with all decisions definite. In essence, 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.

Like use-case diagrams, sequence diagrams are used to model scenarios in the systems. Whereas use cases and the steps or textual descriptions that define them offer a high-level view of a system, the sequence diagram enables you to model a more specific analysis and also assists in the design of the system by modeling the interactions between objects in the system.

As explained in a sequence diagram, the objects involved are drawn on thediagram as a vertical dashed line, with the name of the objects at the top. Horizontal lines corresponding to the events that occur between objects are drawn between the vertical object lines. The event lines are drawn in sequential order, from the top of the diagram to the bottom. They do not necessarily correspond to the steps defined for a usecase scenario.

CASE STUDY: 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. Sequence diagrams are among the most popular UML diagrams and, if used with an object model or class diagram, can capture most of the information about a system. Most object-to-object interactions and operations are considered events, and events include signals, inputs, decisions, interrupts, transitions, and actions to or from users or external devices. An event also is considered to be any action by an object that sends information. 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. Developing sequence or collaboration diagrams requires us to think about objects that generate these events and therefore will help us in identifying classes.

To identify objects of a system, we further analyze the lowest level use cases with a sequence and collaboration diagram pair (actually, most CASE tools such as SA/Object allow you to create only one, either a sequence or a collaboration diagram, and the system generates the other one). Sequence and collaboration diagrams represent the order in which things occur and how the objects in the system send messages to one another.

These diagrams provide a macro-level analysis of the dynamics of a system. Once you start creating these diagrams, you may find that objects may need to be added to satisfy the particular sequence of events for the given use case.

You can draw sequence diagrams to model each scenario that exists when a BankClient withdraws, deposits, or needs information on an account. By walking through the steps, you can determine what objects are necessary for those steps to take place. Therefore, the process of creating sequence or collaboration diagrams can assist you in Identifying Classes or objects of the system. This approach can be combined with noun phrase and class categorization for the best results. We identified the use cases for the bank system. The following are the low level (executable) use cases:

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

Let us create a sequence/collaboration diagram for the following use cases:

- Invalid PIN use case
- Withdraw Checking use case
- Withdraw More from Checking use case

Sequence/collaboration diagrams are associated with a use case. For example, to model the sequence/collaboration diagrams in SA/Object, you must first select a use case, such as the Invalid PIN use case, then associate a sequence or collaboration child process.

To create a sequence, you must think about the classes that probably will be involved in a use-case scenario. Keep in mind that use case refers to a process, not a class. However, a use case can contain many classes, and the same class can occur in many different use cases. 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. Here, we need to think about the sequence of activities that the actor BankClient performs:

- Insert ATM Card.
- Enter PIN number.
- Remove the ATM Card.

Based on these activities, the system should either grant the access right to the account or reject the card. Next, we need to more explicitly define the system. With what are we interacting? We are interacting with an ATMMachine and the BankClient. So, the other objects of this use case are ATMMachine and BankClient.

Now that we have identified the objects involved in the use case, we need to list them in a line along the top of a and drop dotted lines beneath each object (see Figure 3.13). The client in this case is whoever tries to access an account through the ATM, and may or may not have an account. The BankClient on the other hand has an account.

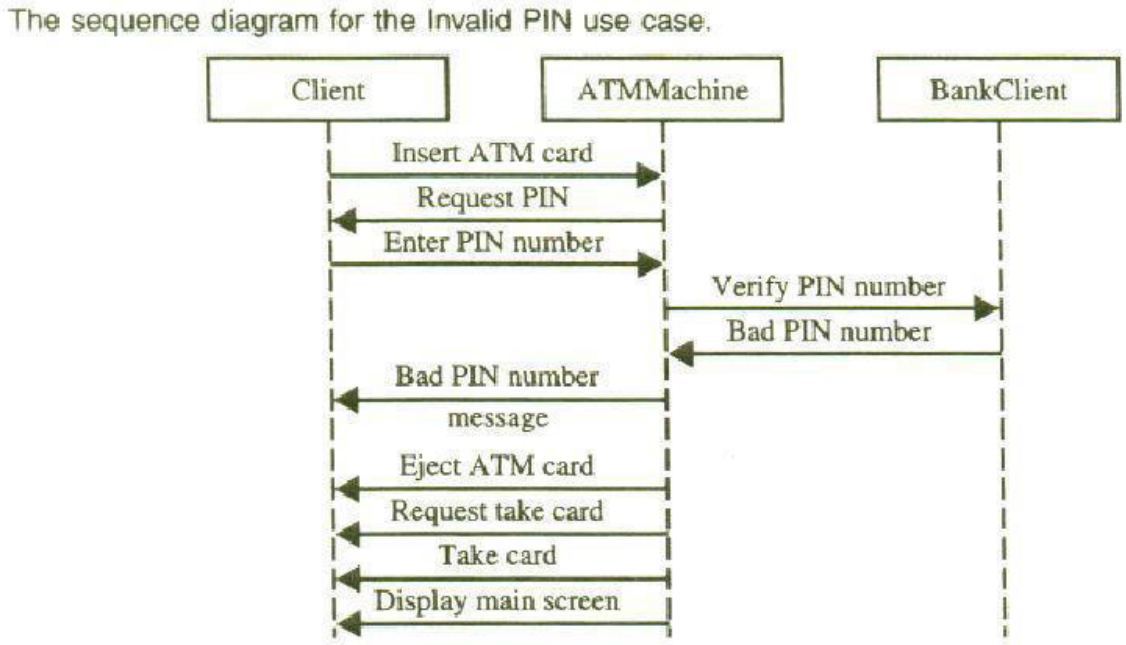


Fig. 3.13. The sequence diagram for the INVALID PIN use case

The dotted lines are the lifelines. The line on the right represents an actor, in this case the BankClient, or an event that is outside the system boundary. An event arrow connects objects. In effect, the event arrow suggests that a message is moving between those two objects. An example of an event message is the request for a PIN. An event line can pass over an object without stopping at that object. Each event must ha"\'e'a descriptive name. In some cases, several objects are active simultaneously, even if they are only waiting for another object to return information to them. In other cases, an object becomes active when it receives a message and then becomes inactive as soon as it responds. Similarly, we can develop sequence diagrams for other use cases (as in Figures 3.14 and 3.16). Collaboration diagrams are just another view of the sequence diagrams and therefore can be created automatically; most UML modeling tools automatically create them (see Figures 3.15)

The following classes have been identified by modeling the UML sequence / collaboration diagrams: Bank, BankClient, ATMMachine, Account, Checking Account, and Savings Account. Similarly other classes can be identified by developing the remaining sequence/ collaboration diagrams.

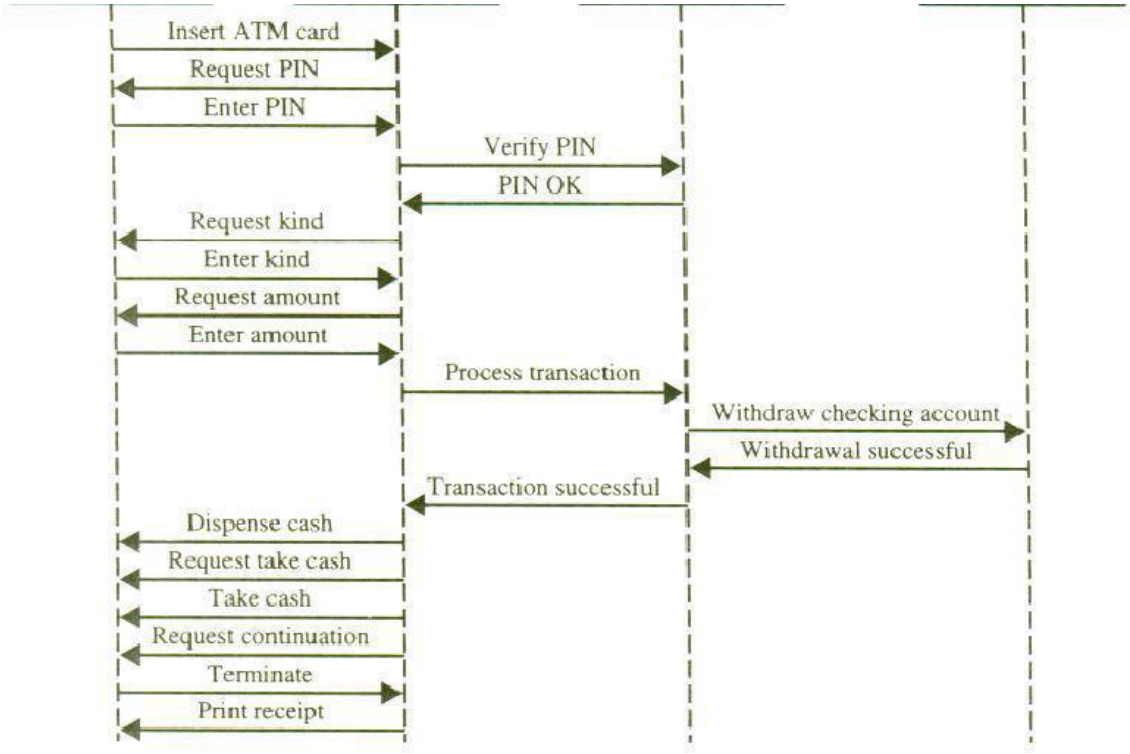


Fig. 3.14. Sequence Diagram for the Withdraw Checking use case

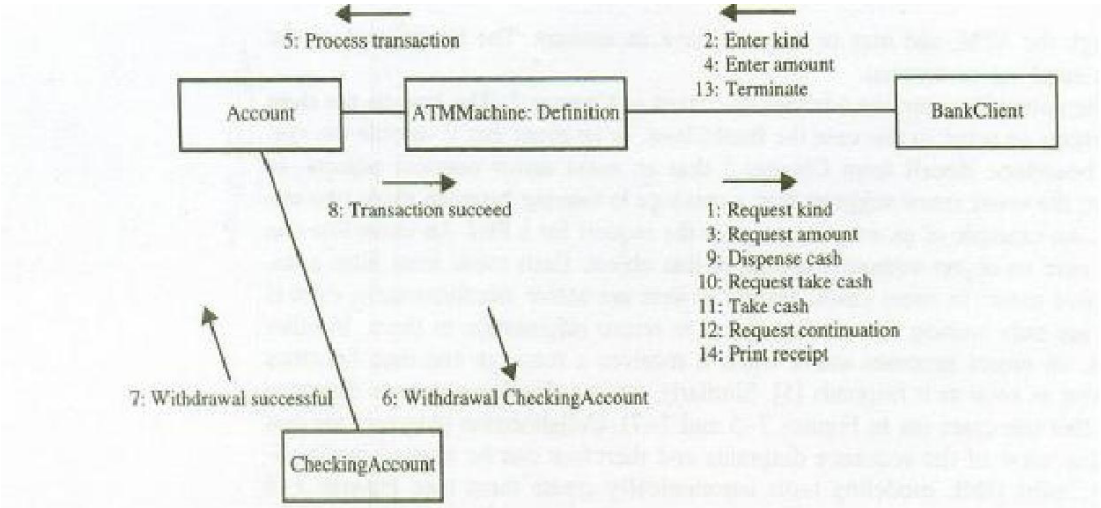
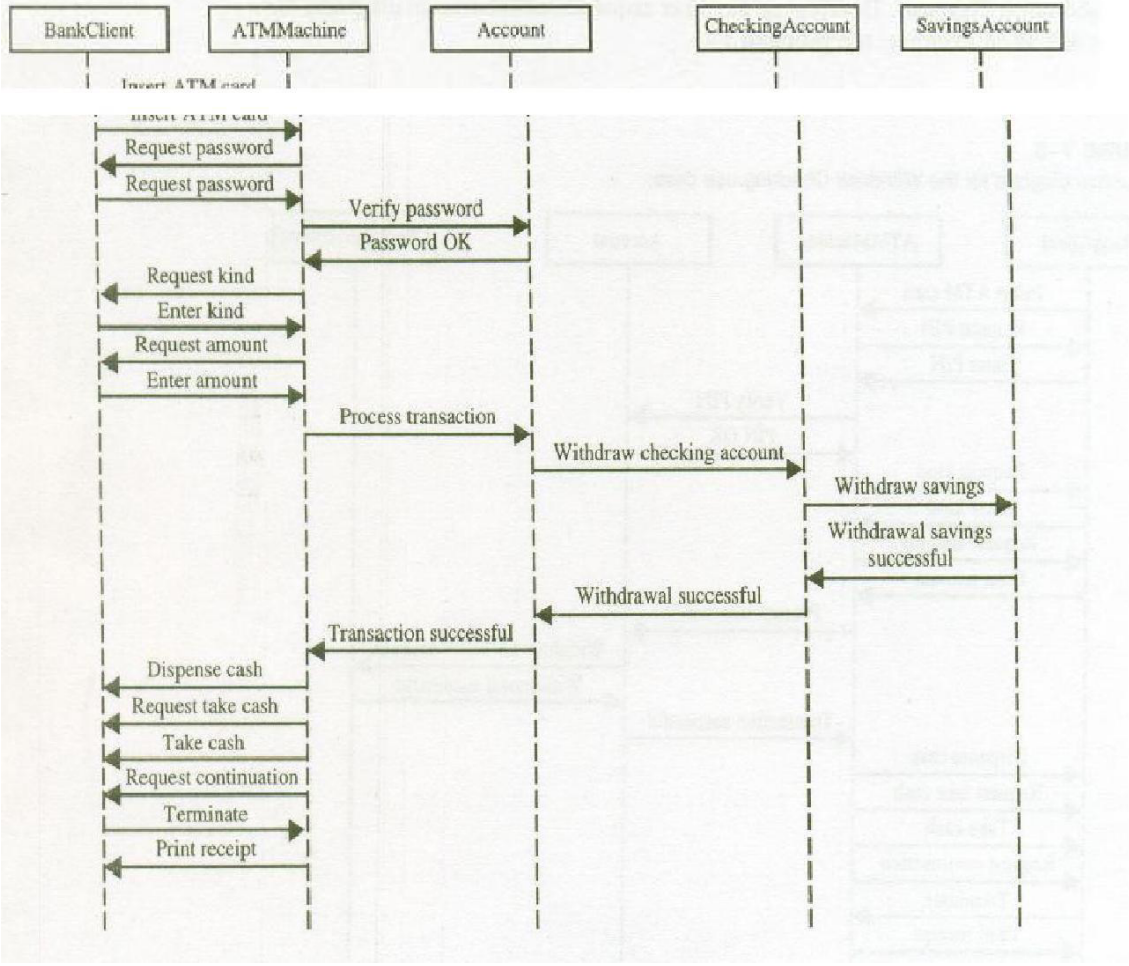


Fig 3.15 : The Collaboration diagram for the Withdraw Checking use case

Fig 3.16
The sequence diagram for the Withdraw More from Checking use case.



3.13 CLASSES, RESPONSIBILITIES, AND COLLABORATORS (CRC) APPROACH

Classes, responsibilities, and collaborators (CRC), developed by Cunningham, Wilkerson, and Beck, was first presented as a way of teaching the basic concepts of object-oriented development.

Classes, Responsibilities, and Collaborators is a technique used for identifying classes' responsibilities and therefore their attributes and methods.

Furthermore, Classes, Responsibilities, and Collaborators can help us identify classes. Classes, Responsibilities, and Collaborators is more a teaching technique than a method for identifying classes.

Classes, Responsibilities, and Collaborators is based on the idea that an object either can accomplish a certain responsibility itself or it may require the assistance of other objects. It requires the assistance of other objects; it must collaborate with those objects to fulfill its responsibility. By identifying **an object's responsibilities** and **collaborators** (cooperative objects with which it works) you can identify its attributes andmethods.

Classes, Responsibilities, and Collaborators cards are 4" X 6" index cards. All the information for an object is written on a card, which is cheap, portable, readily available, and familiar. Figure 3.17 shows an idealized card.

The class name should appear in the upper left-hand corner, a bulleted list of responsibilities should appear under it in the left two thirds of the card, and the list of collaborators should appear in the right third.

Classes, Responsibilities, and Collaborators cards place the designer's focus on the motivation for collaboration by representing (potentially) many messages as phrases of English text.

.

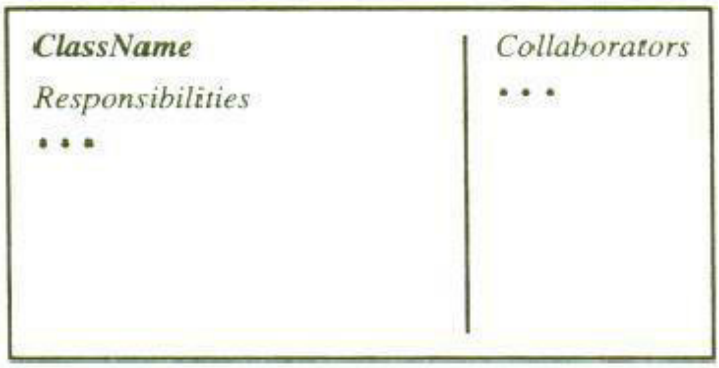


Fig. 3.17. A Classes, Responsibilities and Collaborators (CRC) Index Card

CRC starts with only one or two obvious cards. If the situation calls for a responsibility not already covered by one of the objects: Add, or create a new object to address that responsibility.

– Finding classes is not easy.

– The more practice you have, the better you get at identifying classes.

– There is no such thing as the ―right set of classes.

– Finding classes is an incremental and iterative process.

i) Classes, Responsibilities, And Collaborators Process

The Classes, Responsibilities, and Collaborator's process consists of three steps(Figure 3.17)
1. Identify classes' responsibilities (and identify classes).
2. Assign responsibilities.
3. Identify collaborators.

Classes are identified and grouped by common attributes, which also provides candidates for super classes. The class names then are written onto Classes, Responsibilities, and Collaborators cards. The card also notes sub- and super classes to show the class structure. The application's requirements then are examined for actions and information associated with each class to find the responsibilities of each class.

Next, the responsibilities are distributed; they should be as general as possible and placed as high as possible in the inheritance hierarchy. The idea in locating collaborators is to identify how classes interact. Classes (cards) that have a close collaboration are grouped together physically.

The Classes, Resposibilities, and Collaborators process.

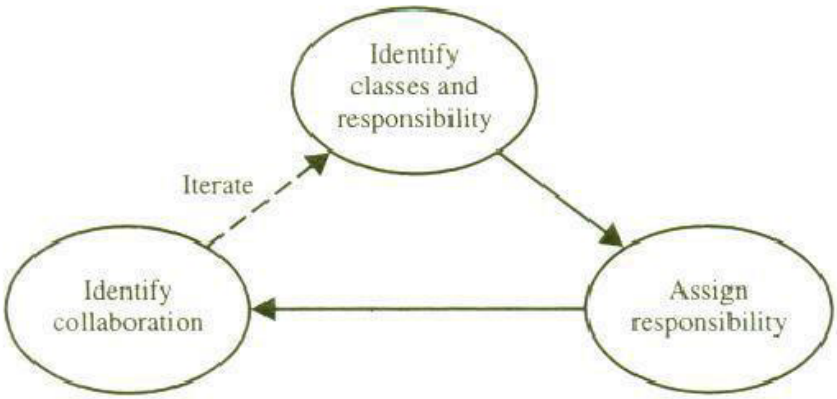


Fig. 3.18. The Classes, Responsibilities and Collaborators process.

CASE STUDY: The ViaNet Bank ATM System: Identifying Classes by Using Classes, Responsibilities, and Collaborators

We already identified the initial classes of the bank system. The objective of this example is to identify objects' responsibilities such as attributes and methods in that system. Account and Transaction provide the banking model. Note that Transaction assumes an active role while money is being dispensed and a passive role thereafter. The class Account is responsible mostly to the BankClient class and it collaborates with several objects to fulfill its responsibilities. Among the responsibilities of the Account class to the BankClient class is to keep track of the BankClient balance, account number, and other data that need to be remembered. These are the attributes of the Account class. Furthermore, the Account class provides certain services or methods, such as means for

BankClient to deposit or withdraw an amount and display the account's Balance (see Figure 3.18).

Classes, Responsibilities, and Collaborators for the Account object.

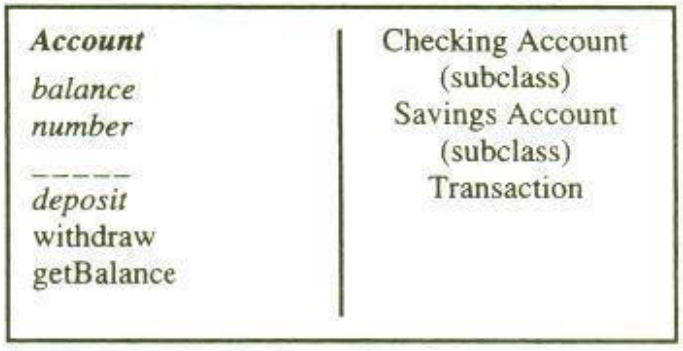


Fig. 3.19. Classes, Responsibilities and Collaborators for the Account Object.

Classes, Responsibilities, and Collaborators encourages team members to pick up the card and assume a role while "executing" a scenario. It is not unusual to see a designer with a card in each hand, waving them about, making a strong identificationwith the objects while describing their collaboration.

As you can see from Figure, 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.

3.14 NAMING CLASSES

Naming a class in an important activity.

Guidelines for Naming Classes

- The class should describe a single object, so it should be the singular form of noun.
- Use names that the users are comfortable with.
- The name of a class should reflect its intrinsic nature.
- By the convention, the class name must begin with an upper-case letter.
- For compound words, capitalize the first letter of each word - for example, Loan Window.

IDENTIFYING OBJECT, RELATIONSHIPS, ATTRIBUTES, & METHODS

In an object-oriented environment, objects take on an active role in a system. All objects stand in relationship to others on whom they rely for services and control. The relationship among objects is based on the assumption each makes about the otherobjects, including what operations can be performed and what behavior results. **Three types of relationships among objects are:**

- **Association**: How are objects associated? This information will guide using designing classes.
- **Super-sub structure** (also known as generalization hierarchy): How are objects organized into super classes and subclasses? This information provides us the direction of inheritance.
- **Aggregation and a-part-of structure:** What is the composition of complex classes? This information guides us in defining mechanisms that properly manage object within-object.

Generally, the relationships among objects are known as **associations.**

The **hierarchical or super-sub relation** allows the sharing of properties or inheritance.

A-part-of structure is a familiar means of organizing components of a bigger object.

.

3.15 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.

we learnt that the **binary associations are shown as lines connecting two class symbols. Ternary and higher-order associations are shown as diamonds connecting to a class symbol by lines, and the association name is written above or below the line. The association name can be omitted if the relationship is obvious.**

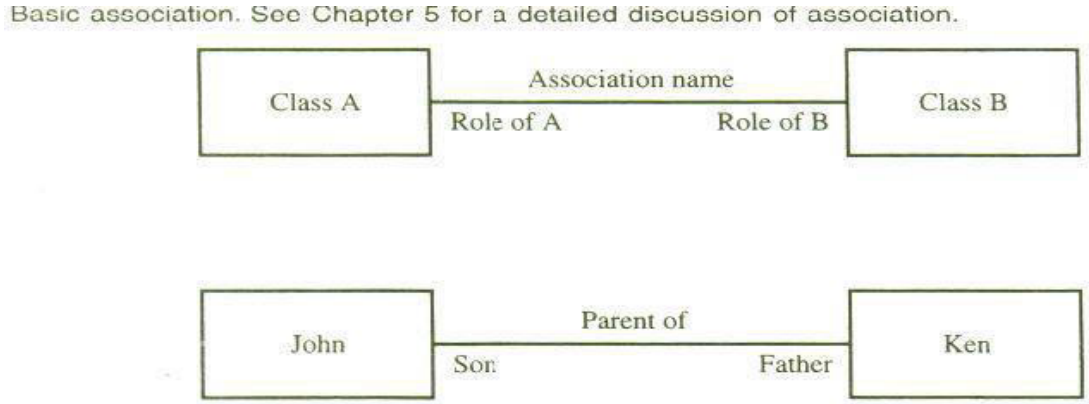


Fig. 3.20. Basic Associations

i) Identifying Associations

Identifying associations begins by analysing the interactions between classes. After all, **any dependency relationship between two or more classes is an association**.

You must examine the responsibilities to determine dependencies.

In other words, **if an object is responsible for a specific task (behavior) and lacks all the necessary knowledge needed to perform the task, then the object must delegate the task to another object that possesses such knowledge.**

Wirfs-Brock, Wilkerson,"'arid Wiener provide the following questions that can help us to identify associations:

- As the class capable of fulfilling the required task by itself?
- If not, what does it need?
- From what other class can it acquire what it needs?

Answering these questions helps us identify association. The approach you should take to identify association is flexibility. **First, extract all candidates' associations from the problem statement and get them down on paper.**

ii) Guidelines For Identifying Association

The Following are general guidelines for identifying the tentative associations:

- 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.

iii) Common Association Patterns

The common association patterns are based on some of the common associations defined by researchers and practioners: Rumbaugh et al. Coad and Yourdon, and others.

These include.**Location association** - -next to, part of, contained in. For example, consider a soup object, cheddar cheese is a-part-of soup. The a-part-of relation is a special type of association.

Communication association –talk to, order to. For example, a customer places an order (communication association) with an operator person (see Figure 3.20).

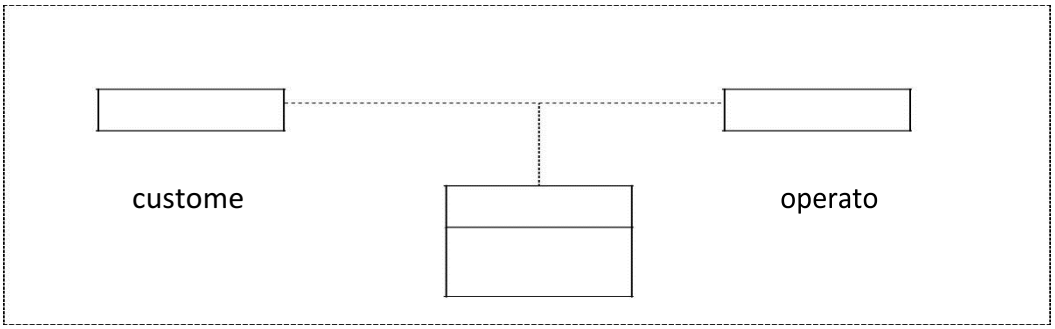


Fig. 3.21. A customer places an order (communication association) with an

operator person.

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

IV) Eliminate Unnecessary Associations

- **Implementation association.** Defer implementation-specific associations to the design phase. Implementation associations are concerned with theimplementation or design of the class within certain programming or developmentenvironments and not relationships among business objects.
- **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 ofthe parent of association (see Figure 3.21).

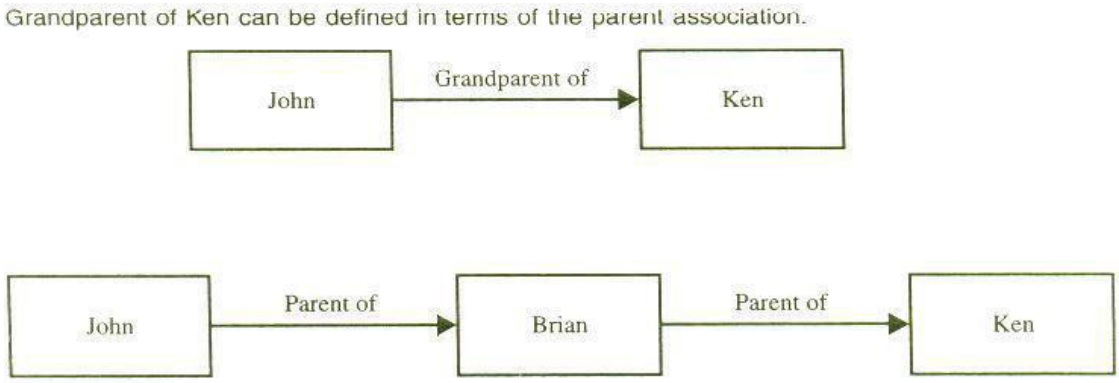


Fig. 3.22. Grandparent of Ken can be defined in terms of the parent association.

3.16 SUPER-SUB CLASS RELATIONSHIPS

The other aspect of classification is identification of super-sub relations among classes. For the most part, **a class is part of a hierarchy of classes, where the top classis the most general one and from it descend all other, more specialized classes**.

The super-sub class relationship represents the **inheritance relationshipsbetween related classes, and the class hierarchy determines the lines of inheritance between classes.**

Class inheritance is useful for a number of reasons. For example, in some cases, **you want to create a number of classes that are similar in all but a few characteristics**. In other cases, someone already has developed a class that you can use, but you need to modify that class.

Subclasses are more specialized versions of their superclasses. Superclass-subclass relationships, also known as generalization hierarchy, allow objects to be built from other objects. Such relationships allow us to explicitly take advantage of the commonality of objects when constructing new classes.

The super-sub class hierarchy is a relationship between classes, where one class is the parent class of another (derived) class. The **parent class also is known as the base or super class or ancestor.**

The super-sub class hierarchy is **based on inheritance**, which is programming by extension as opposed to programming by reinvention. The real **advantage of using this technique is that we can build on what we already have and, more important, reuse what we already have. Inheritance allows classes to share and reuse behaviors and attributes. Where the behavior of a class instance is defined in that class's methods**, **a class also inherits the behaviors and attributes of all of its super classes.**

i) Guidelines For Identifying Super-Sub Relationship, A Generalization

The following are guidelines for identifying super-sub relationships in the application:

- **Top-down.** Look for noun phrases composed of various adjectives in a classname. Often, you can discover additional special cases. Avoid excessive refinement. Specialize only when the subclasses have significant behavior. For example, a phone operator employee can be represented as a cook as well as a clerk or manager because they all have similar behaviors.
- **Bottom-up.** Look for classes with similar attributes or methods. In most cases, you can group them by moving the common attributes and methods to an abstract class. You may have to alter the definitions a bit; this is acceptable as long as generalization truly applies.
- **Reusability.** Move attributes and behaviors (methods) as high as possible in the hierarchy. At the same time, do not create very specialized classes at the top of the hierarchy. This is easier said than done. The balancing act can be achieved through several iterations. This process ensures that you design objects that can be reused in another application.
- **Multiple inheritance**. Avoid excessive use of multiple inheritances. Multiple inheritance brings with it complications such as how to determine which behavior to get from which class, particularly when several ancestors define the same method. It also is more difficult to understand programs written in a multiple inheritance system. One way of achieving the benefits of multiple inheritance is to inherit from the most appropriate class and add an object of another class as an attribute. (See fig 3.22)

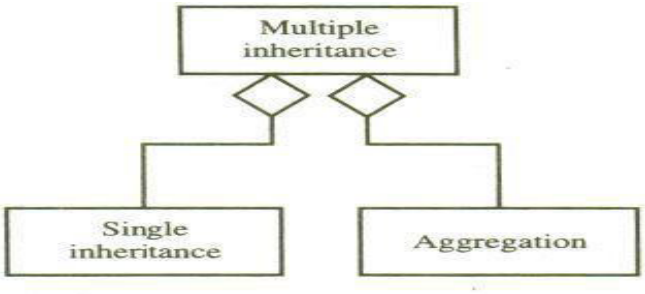


Fig. 3.23. One way of achieving the benefits of multiple inheritance from the most appropriate class.

3.17 A PART OF RELATIONSHIPS-AGGREGATION

A-part-of relationship, also called **aggregation**, represents the situation where class consists of several component classes. A class that is composed of other classes does not behave like its parts; actually, it behaves very differently.

For example, a car consists of many other classes, one of which is a radio, but a car does not behave like a radio (see Figure 3.23).

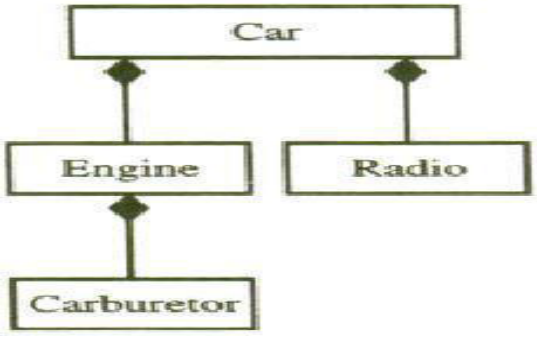


Fig. 3.24. A-part-of composition. A carburetor is a part of an engine and an engine and a radio are parts of a car.

Two major properties of a-part-of relationship are transitivity and antisymmetric,

- **Transitivity**. The property where, if A is part of Band B is part of C, then A is part of C. For example, a carburetor is part of an engine and an engine is part of a car; therefore, a carburetor is part of a car. Figure3.23 shows a-part-of structure.
- **Antisymmetric.** The property of a-part-of relation where, if A is part of B, then . B is not part of A. For example, an engine is part of a car, but acar is not part of an engine.

A clear distinction between the part and the whole can help us determine where responsibilities for certain behavior must reside. This is done mainly by asking the following questions:

- Does the part class belong to a problem domain?
- Is the part class within the system's responsibilities?

□ Does the part class capture more than a single value? (If it captures only a single value, then simply include it as an attribute with the whole class.)

- Does it provide a useful abstraction in dealing with the problem domain?

We saw that the UML uses hollow or filled diamonds to represent aggregations. A filled diamond signifies the strong form of aggregation, which is composition. For example, one might represent aggregation such as container and collection as hollow diamonds (see Figures 3.24, 3.25) and use a solid diamond to represent composition, which is a strong form of aggregation (see Figure 3.23).

A house is a container.

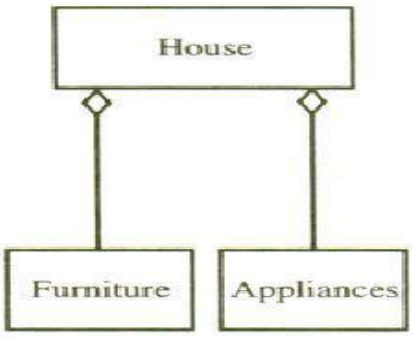


Fig. 3.24. A house is a container

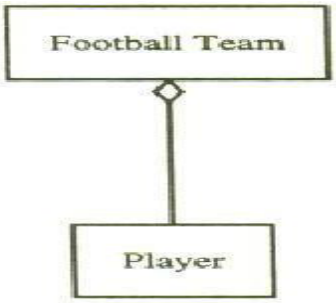


Fig. 3.25. A football team is a collection of players.

i) Part-Of Relationship Patterns

To identify a-part-of structures, Coad and Yourdon provide the following guidelines:

- **Assembly.** An assembly is constructed from its parts and an assembly- part situation physically exists; for example, a French onion soup is an assembly of onion, butter, flour, wine, French bread, cheddar cheese, and so on.

- **Container.** A physical whole encompasses but is not constructed fromphysical parts; for example, a house can be considered as a container for furniture and appliances (see Figure 3.24).
- **Collection-member**. A conceptual whole encompasses parts that may be physical or conceptual; for example, a football team is a collection of players. (fig.3.25).

CASE STUDY: RELATIONSHIP ANALYSIS FOR THE VIANET BANK ATM SYSTEM

1) 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.

2) 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.

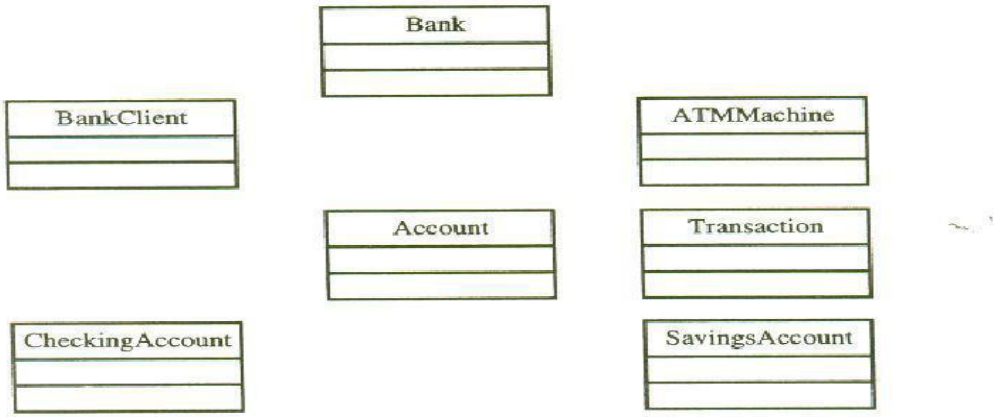


Fig 3.26. UML class diagram for the ViaNet bank ATM system. Some CASE tools such as the SA/Object Architect can automatically define classes and draw them from use cases or collaboration/ sequence diagrams. However, presently, it cannot identifyall the classes. For this example, S/A Object was able to identify only the BankClient class.

- **Multiple class diagrams** that represent various pieces, or views, of the systemclass diagram.
- **Multiple class diagrams,** that show the specific static relationshipsbetween various classes.

we will add relationships later (see Figure 3.26).

3) 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:

- 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.

Some common patterns of associations are these:

- **Location association**. For example, next to, part of, contained in (noticethat apart- of relation is a special type of association).
- **Directed actions association**.
- □ **Communication association**. For example, talk to, order from.

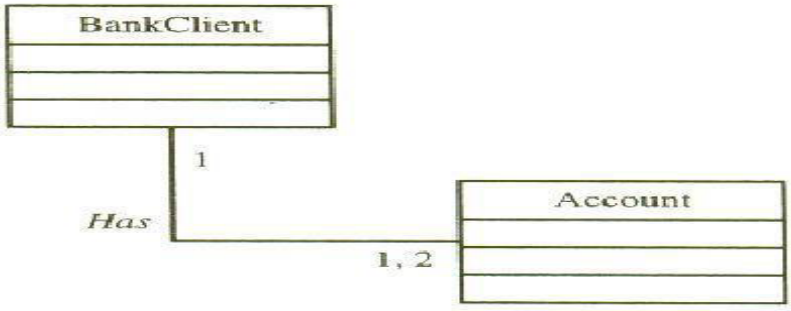


Fig. 3.27. Defining the BankClient-Account association multiplicity. One Client can have one or more Accounts (checking and savings accounts).

The first obvious relation is that each account belongs to a bank client since each BankClient has an account. Therefore, there is an association between the BankClient and Account classes. We need to establish cardinality among these classes.

By default, in most CASE tools such as SNObject Architect, all associations are considered one to one (one client can have only one account and vice versa). However, since each BankClient can have one or two accounts we need to change the cardinality of the association (see Figure 3.27). Other associations and their cardinalities are defined in Table 8-1 and demonstrated in Figure 3.28.

Table 8.1

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

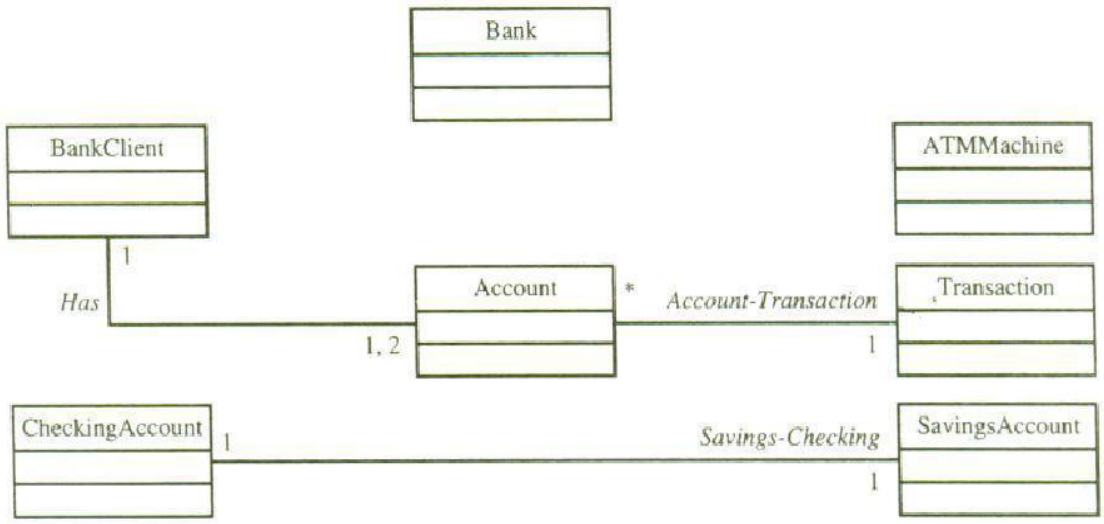


Fig. 3.28. Associations among the ViaNet bank ATM system classes.

4) Defining Super-Sub Relationships

Let us review the guidelines for identifying super-sub relationships:

*. **Top-down**. Look for noun phrases composed of various adjectives in the class name.

*. **Bottom-up**. Look for classes with similar attributes or methods. In most cases, you can group them by moving the common attributes and methods to an abstract class.

*. **Reusability.** Move attributes and behaviors (methods) as high as possible in the hierarchy.

***. Multiple inheritance.** Avoid excessive use of multiple inheritance.

CheckingAccount and SavingsAccount both are types of accounts. They can be defined as specializations of the Account class. When implemented, class will define attributes and services common to all kinds of accounts, with CheckingAccount and SavingsAccount each defining methods that make them more specialized. Fig 3.29.

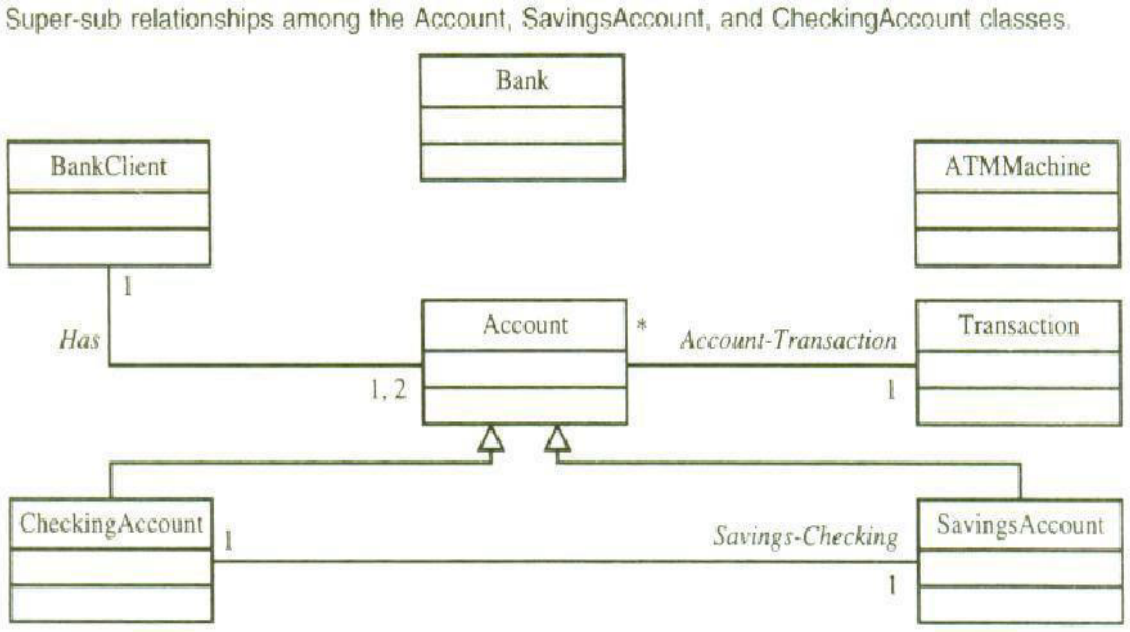


Fig 3.29- Super-sub relationships among the Account, SavingsAccount and CheckingAccount Classes

5) 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.

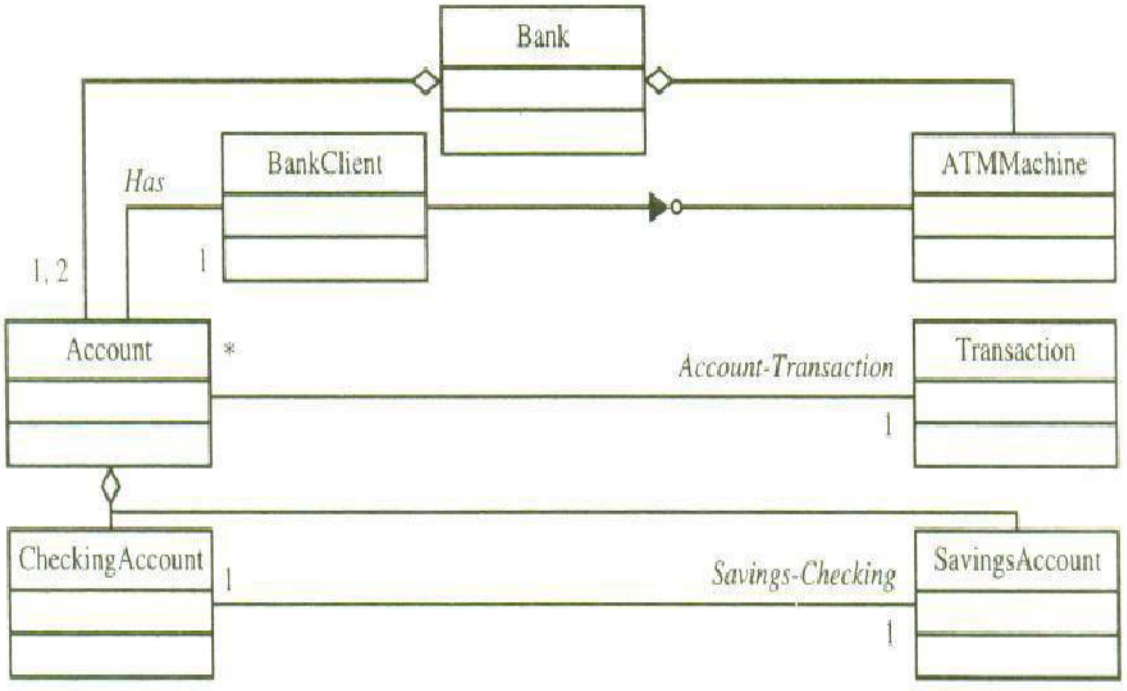


Fig 8.30. Association, generalization, and aggregation among the ViaNet bank classes. Notice that the super-sub arrows for CheckingAccount and SavingsAccount have merged. The relationship between BankClient and ATMMachine is an interface.

A bank consists of ATM machines, accounts, buildings, employees, and so forth. However, since buildings and employees are outside the domain of this application, we define the Bank class as an aggregation of ATM Machine and Account classes. Aggregation is a special type of association. Figure 3.30 depicts the association, generalization, and aggregation among the bank systems classes. If you are wondering what is the relationship between the BankClient and ATM Machine, it is an interface. Identifying a class interface is a design activity of object-oriented system development.

3.18 CLASS RESPONSIBILITY : IDENTIFYING ATTRIBUTES AND METHODS

Identifying attributes and methods is like finding Classes, still a difficult activity and an iterative process.

Responsibilities identify problems to be solved.

Attributes are things an object must remember such as color, cost and manufacturer. Identifying attributes of a system's classes starts with understanding the system' s responsibilities.

The following questions help in identifying the responsibilities of classes and deciding what data elements to keep track of:

What information about an object should we keep track of?
What services must a class provide?

3.19CLASS RESPONSIBILITY: DEFING ATTRIBUTES BY ANALYZING USE CASES AND OTHER UML DIAGRAMS

Attributes can be derived from scenario testing; therefore, by analyzing the use cases and sequence/collaboration, activity and state diagrams, you can begin to understand classes responsibilities and how they must interact to perform their tasks.

The main goal is to understand what the class is responsible for knowledge. Responsibility is the issue.

What kind of questions what kind of question would you like ask;

- How am I going to be used?
- How am I going to collaborate with other classes?
- How am I described in the context of this system's responsibility?

Guidelines for Defining Attributes.

Attributes usually correspond to nouns followed by prepositional phrases such as cost of the soup. Attributes also may correspond to adjectives or adverbs. Keep the class simple; state only enough attributes to define the object state. Attributes are less to be fully described in the problem statement. Omit derived attributes. Do not carry discovery of attributes to excess. You can add more attributes in subsequentiterations.

Important point to remember is that you may think of many attributes that can be associated with a class. You must careful to add only those attributes necessary to the design at hand.

3.20 OBJECT RESPONSIBILITY: METHODS AND MESSAGES.

Objects not only describe abstract data but also must provide some services.
Methods and messages are the workhorses of object-oriented systems.

In an object-oriented environment, every piece of data or object is surrounded by a rich set of routines called **methods.**

These methods do everything from printing the object to initializing its variables. Every class is responsible for storing certain information from the domain knowledge. It also is logical to assign the responsibility for performing any operation necessary on that information.

Operations (methods or Behavior) in the o-o-system usually correspond to queries about attributes.

Methods are responsible for managing the value of attributes such as query, updating, reading and writing;

CASE STUDY : DEFINING ATTRIBUTES FOR VIANET BANK OBJETCTS.

1. **Defining Attributes for the BankClient Class**

   By analyzing the use cases, the sequence/collaboration diagrams and activity diagram of bank atm process, it is apparent that, for the BankClient Class, theproblem domain and system dictate certain attributes. In essence, what does thesystem need to know about the BankClient?

By looking at the activity diagram (See Fig 3.9) we notice that the BankClient must have a PIN number and CardNumber. Therefore, the PIN number and CardNumber are appropriate attributes for the BankClient.

The Attributes of the BankClient are
firstName

lastName

pinNumber

cardNumbe
account:Account

At this stage of the design, we are concerned with the functionality of the BankClient object and not with implementation attributes.

2. **Defining Attributes for the AccountClass**.

Similarly, what information does the system need to know about an account? Based on the ATM Usecase diagram, Sequence/Collaboration diagram and activity diagram, BankClient can interact with its account by entering the account number and they could deposit money, get an account history, or get the balance. Therefore, we have defined the following attributes for the Account Class: number, balance.

CASE STUDY: DEFINING METHODS BY ANALYZING UML DIAGRAMS AND USE CASES.

We know that, in a sequence diagram, the objects involved are drawn on the diagram as vertical dashed lines. Furthermore, the events that occur between objects are drawn between the vertical object lines. An Event is considered tobe an action that transmits information.

For example, to define methods for the Account class, we look at sequencediagrams for the following use cases.

Deposit Checking

Deposit Savings
Withdraw Checking

Withdraw More from
Checking Withdraw Savings

Withdraw Savings Denied
Checking Transaction History
Savings Transaction History

CASE STUDY: DEFINING METHODS BY BANK OBJECTS

Operations (methods or behavior) in the object-oriented system usually correspond to events or actions that transmit information in the sequence diagram or queries about attributes of the objects. In other words, methods are responsible for managing the value for attributes such as query, updating, reading and writing.

1) Defining Account Class operations.

Deposit and withdrawal operations are available to the Client through the bank application, but they are provided as services by the Account Class, since the account objects must be able to manipulate their internal attributes. Account objects also must be able to create transaction records of any deposit or withdrawal they perform.

Here are the methods that we need to define for the Account Class:

deposit

Withdraw create

Transaction.

The services added to the Account class are those that apply to all subclasses of Account; namely, CheckingAccount and SavingsAccount. The subclass will either inherit these generic services without charge or enhance them to suit their own needs.

2) Defining BankClient Class Operation

Analyzing the sequence diagram (fig 3.13), it is apparent that the BankClient requires a method to validate client's passwords.

3) Defining CheckingAccount Class Operations

The requirement specification states that, when a checking account has insufficient funds to cover a withdrawal, it must try to withdraw the insufficient amount from its related saving account. To provide the service, the CheckingAccount class needs awithdrawal service that enables the transfer. Similarly, we must add the withdrawal service to the CheckingAccount class.

TEXT/ REFERENCE BOOKS:

1. Ali Bahrami, “Object oriented systems development using the unified modelling language”, 1st Edition, McGraw-Hill, 1998.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson, “The Unified Modelling Language User Guide”, 3rd Edition Addison Wesley,2007.
3. John Deacon, "Object Oriented Analysis and Design", 1st Edition, Addison Wesley, 2005.
4. Grady Booch, James Rumbaugh, and Ivar Jacobson, “The Unified Modelling Language User Guide”, 3rd Edition Addison Wesley.
5. John Deacon, “Object Oriented Analysis and Design”, 1st Edition, Addison Wesley.
6. Bernd Oestereich, “Developing Software with UML, Object - Oriented Analysis and Design in Practice”, Addison-Wesley

Questions

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Illustrate the tools available for extracting information about a system?	Analyze	BTL 4
2.	Describe the purpose of analysis? Why do we need analysis?	Knowledge	BTL 1
3.	Describe use-case model?	Knowledge	BTL 1
4.	Illustrate how would you identify actors?	Analyze	BTL 4
5.	Illustrate how would you name classes?	Analyze	BTL 4
6.	Explain why is identifying class hierarchy important In object-oriented analysis?	Knowledge	BTL 1
7.	Sketch generalization?	Apply	BTL 3
8.	Illustrate how would you identify a super-subclass structure?	Analyze	BTL 4
9.	Classify association different from an a-part-of relation?	Understand	BTL 2
10.	Sketch repeating attributes	Apply	BTL 3
	Part-B		
Q.No	**Questions**	**Competence**	**BT Level**
1.	Explain the guidelines for finding use cases and developing effective documentation.	Knowledge	BTL 1
2.	Illustrate the CRC approach and naming classes.	Analyze	BTL 4
3.	Classify detailed notes about the Noun phrase approach.	Understand	BTL 2

4.	Explain common class pattern approach with example.	Knowledge	BTL 1
5.	Explain use case driven approach with example.	Knowledge	BTL 1
6.	Conclude developing a sequence collaboration diagram a useful activity in identifying classes?	Analyze	BTL 4
7.	Explain how we can identify attributes and methods for classes?	Knowledge	BTL 1