OBJECT-ORIENTED MODELING AND DESIGN

Domain Analysis

CONTENTS :

1.Overview of Analysis

- The problem statement should not be taken as immutable, but rather as a basis for refining the requirements.
- Analysis is divided into two sub stages
- Domain analysis
- Application analysis

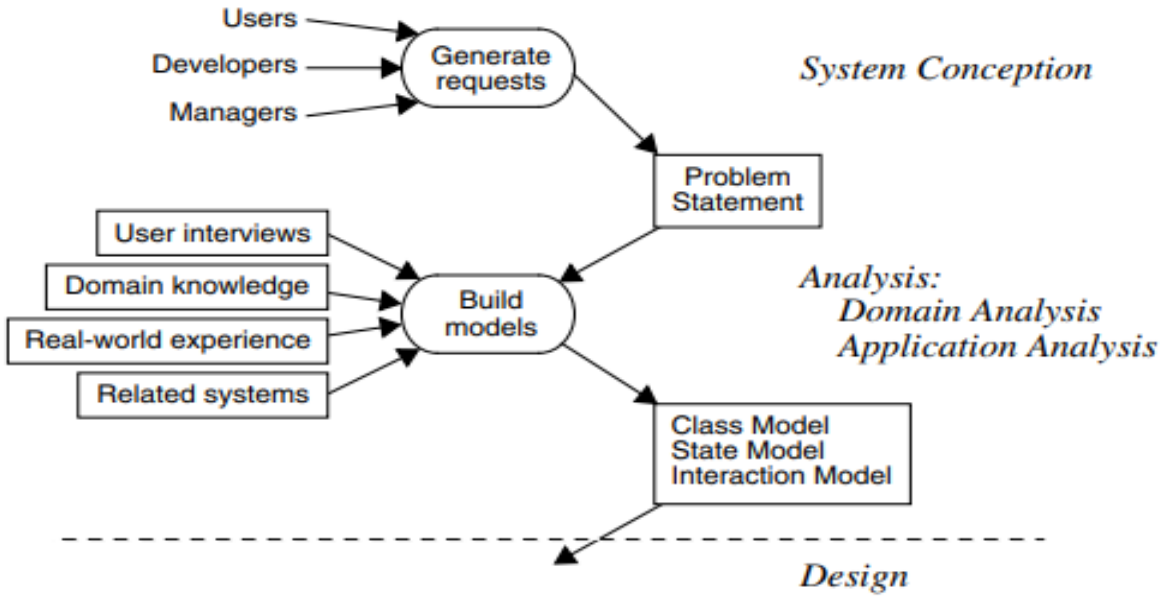

Figure 12.1 Overview of analysis. The problem statement should not be taken as immutable, but rather as a basis for refining the requirements.

Fig 12.1 Overview of Analysis

2.Domain Class Model

- First step in analyzing the requirements is to construct a domain model.
- Static structure of the real world system is captured.
- The domain model describes real-world classes and their relationships to each other.
- Information for the domain model comes from the problem statement, artifacts from related systems, expert knowledge of the application domain and general knowledge of the real world.

The steps to be performed to construct a domain class model:

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

Finding classes

- Classes often correspond to nouns. Ref figure 12.2.
- Eg- ” a reservation system sell tickets to performances at various theater”- Tentative classes would be Reservation, System, Tickets, Performance and Theaters.
- Idea is to capture concepts. not all nouns are concepts, and concepts are also expressed in other parts of speech.

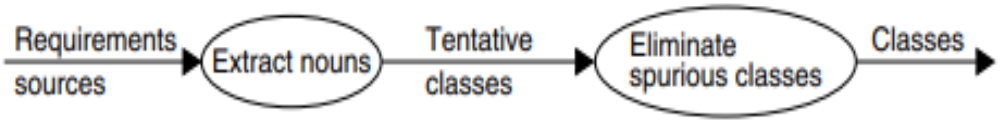

Figure 12.2 Finding classes. You can find many classes by considering nouns.

Figure: 12.2 Finding Classes-by considering nouns

For the Case study of the ATM: The following are the classes extracted from problem statement nouns. Figure 12.3

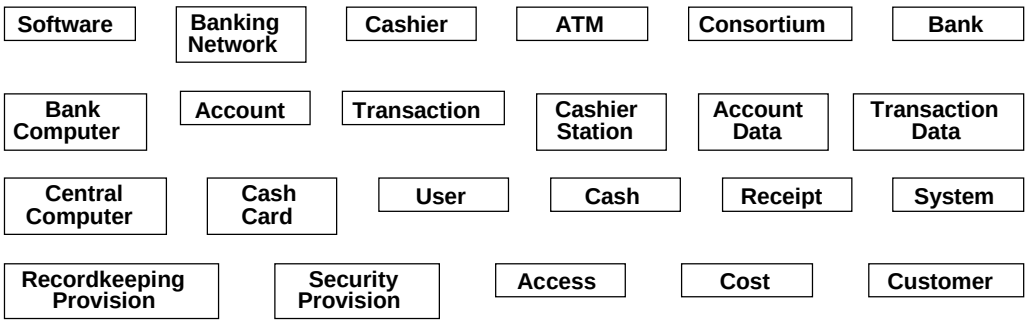

Figure: 12.3 ATM classes extracted from problem statement nouns

- Additional classes that do not appear directly in the statement but can be identified from our knowledge of the problem domain

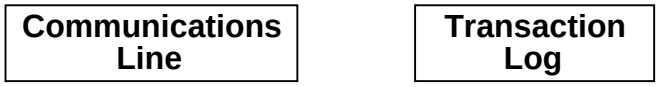

Figure:12.4 ATM classes identified from knowledge of problem domain

Domain Class Model: Keeping the right classes

Discard unnecessary and incorrect classes according to the following criteria

- **Redundant classes**: If two classes express the same concept, you should keep the most descriptive name.

  ATM example. Customer and user are redundant; we retain customer because it is more descriptive.

- **Irrelevant classes:** . If a class has little or nothing to do with the problem, eliminate it. This involves judgment, because in another context the class could be important.

  ATM example. Apportioning Cost is outside the scope of the ATM software.

- **Vague classes**: class should be specific.

  ATM example. RecordkeepingProvision is vague and is handled by Transaction. In other

applications, this might be included in other classes, such as StockSales, TelephoneCalls, or Machine Failure

- **Attributes:** Names that primarily describe individual objects should be restated as attributes.

  ATM example. AccountData is underspecified but in any case probably describes an account. An ATM dispenses cash and receipts, but beyond that cash and receipts are peripheral to the problem, so they should be treated as attributes.

- **Operations:** If a name describes an operation that is applied to objects and not manipulated in its own right, then it is not a class.

  Eg-if we are simply building telephones, then call is part of the state model and not a class

  For example, in a billing system for telephone calls a Call would be an important class with attributes such as date, time, origin, and destination.

- **Roles:** The name of a class should reflect its intrinsic nature and not a role that it plays in an association.

  For Ex-Owner of a car..i n a car manufacturing database, not correct as a class. It can be a person(owner, driver, lessee)

- **Implementation Constructs:** Eliminate constructs from the analysis model that are extraneous to the real world. We may need them during design and not now.

  ATM example. Some tentative classes are really implementation constructs. Transaction Log is simply the set of transactions; its exact representation is a design issue. Communication links can be shown as associations; Communications Line is simply the physical implementation of such a link.

- **Derived classes:**

  As a general rule, omit classes that can be derived from other classes. mark all derived classes with a preceding slash(‘/’)in the class name.

Figure 12.5 shows the classes eliminated from the ATM example

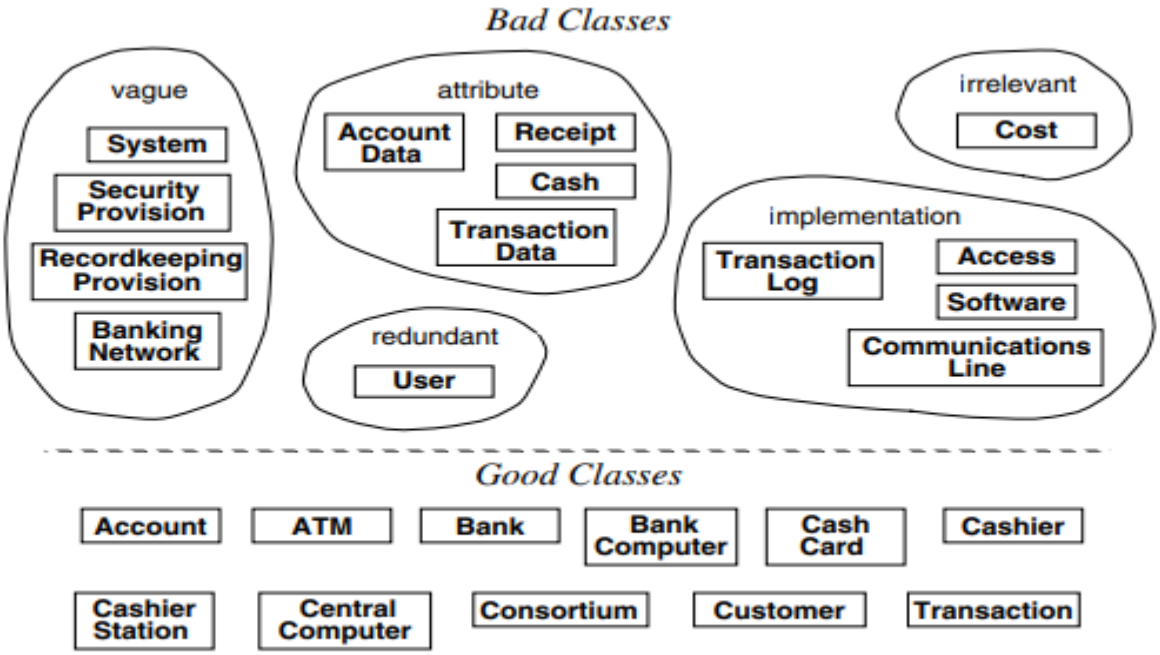

Figure 12.5 Eliminating unnecessary classes from ATM problem

Preparing a Data Dictionary

- Prepare a data dictionary for all modeling elements.
- Describe the scope of the class within the current problem, including all assumptions or restrictions on its use.
- DD also describes associations, attributes, operations and enumeration values

Data Dictionary for the ATM classes

Account	ATM
Bank	BankComputer
CashCard	Cashier
cashierStation	CentralComputer
Consortium	Customer
Transaction	

Account—a single account at a bank against which transactions can be applied. Accounts may be of various types, such as checking or savings. A customer can hold more than one account.

ATM—a station that allows customers to enter their own transactions using cash cards as identification. The ATM interacts with the customer to gather transaction information, sends the transaction information to the central computer for validation and processing, and dispenses cash to the user. We assume that an ATM need not operate independently of the network.

Bank—a financial institution that holds accounts for customers and issues cash cards authorizing access to accounts over the ATM network.

BankComputer—the computer owned by a bank that interfaces with the ATM network and the bank's own cashier stations. A bank may have its own internal computers to process accounts, but we are concerned only with the one that talks to the ATM network.

CashCard—a card assigned to a bank customer that authorizes access of accounts using an ATM machine. Each card contains a bank code and a card number. The bank code uniquely identifies the bank within the consortium. The card number determines the accounts that the card can access. A card does not necessarily access all of a customer's accounts. Each cash card is owned by a single customer, but multiple copies of it may exist, so the possibility of simultaneous use of the same card from different machines must be considered.

Cashier—an employee of a bank who is authorized to enter transactions into cashier stations and accept and dispense cash and checks to customers. Transactions, cash, and checks handled by each cashier must be logged and properly accounted for.

CashierStation—a station on which cashiers enter transactions for customers. Cashiers dispense and accept cash and checks; the station prints receipts. The cashier station communicates with the bank computer to validate and process the transactions.

CentralComputer—a computer operated by the consortium that dispatches transactions between the ATMs and the bank computers. The central computer validates bank codes but does not process transactions directly.

Consortium—an organization of banks that commissions and operates the ATM network. The network handles transactions only for banks in the consortium.

Customer—the holder of one or more accounts in a bank. A customer can consist of one or more persons or corporations; the correspondence is not relevant to this problem. The same person holding an account at a different bank is considered a different customer.

Transaction—a single integral request for operations on the accounts of a single customer. We specified only that ATMs must dispense cash, but we should not preclude the possibility of printing checks or accepting cash or checks. We may also want to provide the flexibility to operate on accounts of different customers, although it is not required yet.

Figure 12.6 Data dictionary for ATM classes. Prepare a data dictionary for all modeling elements.

Finding associations

- A structural relationship between two or more classes is an association.
- A reference from one class to another is an association.
- Associations often correspond to verbs or verb phrases.
- Idea here is to capture relationships

ATM example. Figure 12.7 shows associations.

Verb phrases

Banking network includes cashier stations and ATMs
Consortium shares ATMs
Bank provides bank computer
Bank computer maintains accounts
Bank computer processes transaction against account
Bank owns cashier station
Cashier station communicates with bank computer
Cashier enters transaction for account
ATMs communicate with central computer about transaction
Central computer clears transaction with bank
ATM accepts cash card
ATM interacts with user
ATM dispenses cash
ATM prints receipts
System handles concurrent access
Banks provide software
Cost apportioned to banks

Implicit verb phrases

Consortium consists of banks
Bank holds account
Consortium owns central computer
System provides recordkeeping
System provides security
Customers have cash cards

Knowledge of problem domain

Cash card accesses accounts
Bank employs cashiers

Figure 12.7 Associations from ATM problem statement

Keeping the Right Association

Discard unnecessary associations, using the following criteria:

- Associations between eliminated classes.
- Irrelevant or implementation associations.
- Actions: An association should describe a structural property of the application domain not a transient event.
- Ternary associations.
- Derived associations :they may be redundant

Further specify the semantics of associations as follows:

- Misnamed associations.
- Association end names.
- Qualified associations.
- Multiplicity
- Missing associations
- Aggregation.
- Fig 12.9 shows class diagram with remaining associations

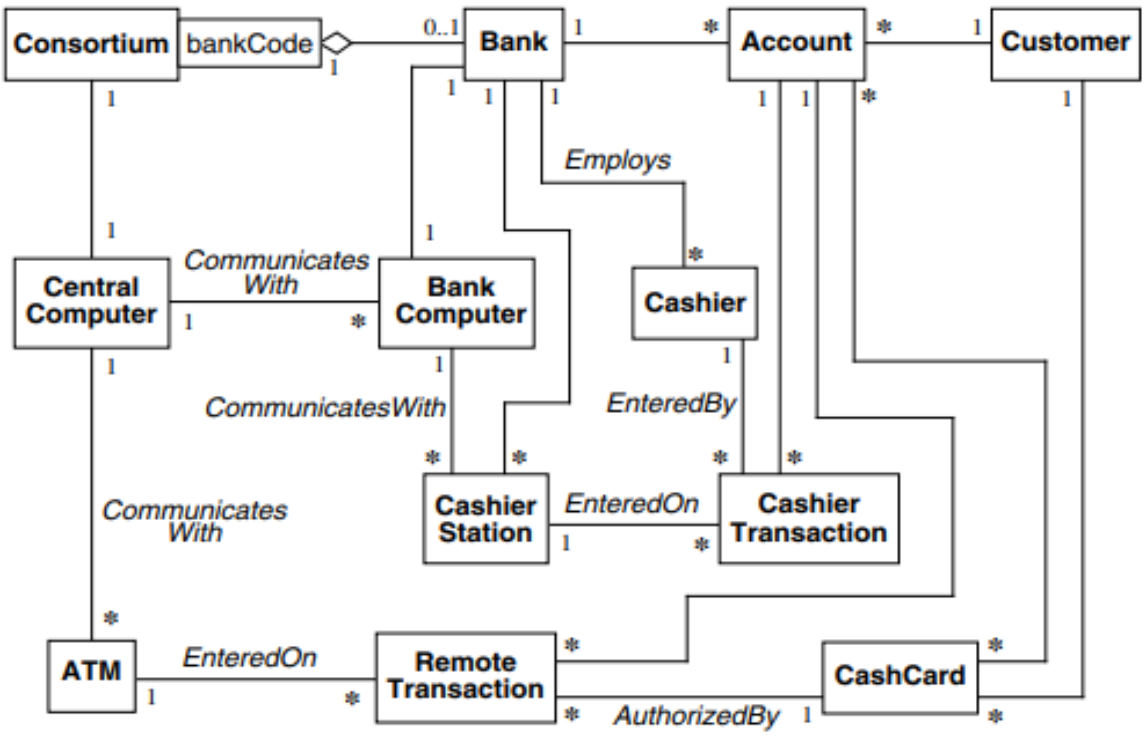

Figure 12.9 Initial class diagram for ATM system

Figure:12.9 Initial class diagram for ATM system

Find attributes of objects and links.

- Attributes are data properties of objects
- Attribute values should not be objects; use an association to show any relationship between objects.
- Attributes usually correspond to nouns followed by possessive phrases, such as "the color of the car

Keeping the right attributes:

Eliminate unnecessary and incorrect attributed with the following criteria:

- **Objects**: If the independent existence of an element is important, rather than just its value, then it is an object. For example, boss refers to a class and salary is an attribute. The distinction often depends on the application.
- **Qualifiers**: If the value of an attribute depends on a particular context, then consider restating the attribute as a qualifier. For example, employee Number is not a unique property of a person with two jobs; it qualifies the association Company employs person.
- **Names:** A name is an attribute when its use does not depend on context, especially when it need not be unique within some set. Names of persons, unlike names of companies, may be duplicated and are therefore attributes.
- Boolean attributes:
- **Identifiers:** . OO languages incorporate the notion of an object identifier for unambiguously referencing an object. Do not include an attribute whose only purpose is to identify an object, as object identifiers are implicit in class models
- **Attributes on associations:** If a value requires the presence of a link, then the property is an attribute of the association and not of a related class. For example, in an association between Person and Club the attribute membership Date belongs to the association, because a person can belong to many clubs and a club can have many members
- **Internal values:** . If an attribute describes the internal state of an object that is invisible outside the object, then eliminate it from the analysis
- **Fine detail:** Omit minor attributes that are unlikely to affect most operations.
- **Discordant attributes**: An attribute that seems completely different from and unrelated to all other attributes may indicate a class that should be split into two distinct classes. A class should be simple and coherent. Mixing together distinct classes is one of the major causes of troublesome models. Unfocused classes frequently result from premature consideration of implementation decisions during analysis.
- **Boolean attributes**: Reconsider all boolean attributes. Often you can broaden a boolean attribute and restate it as an enumeration

ATM example. We apply these criteria to obtain attributes for each class (Figure 12.10).

Some tentative attributes are actually qualifiers on associations. We consider several aspects of the model.

■ BankCode and cardCode are present on the card. Their format is an implementation detail, but we must add a new association Bank issues CashCard. CardCode is a qualifier on this association; bankCode is the

qualifier of Bank with respect to Consortium.

■ The computers do not have state relevant to this problem. Whether the machine is up or down is a transient attribute that is part of implementation.

■ Avoid the temptation to omit Consortium, even though it is currently unique. It provides the context for the bankCode qualifier and may be useful for future expansion.

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

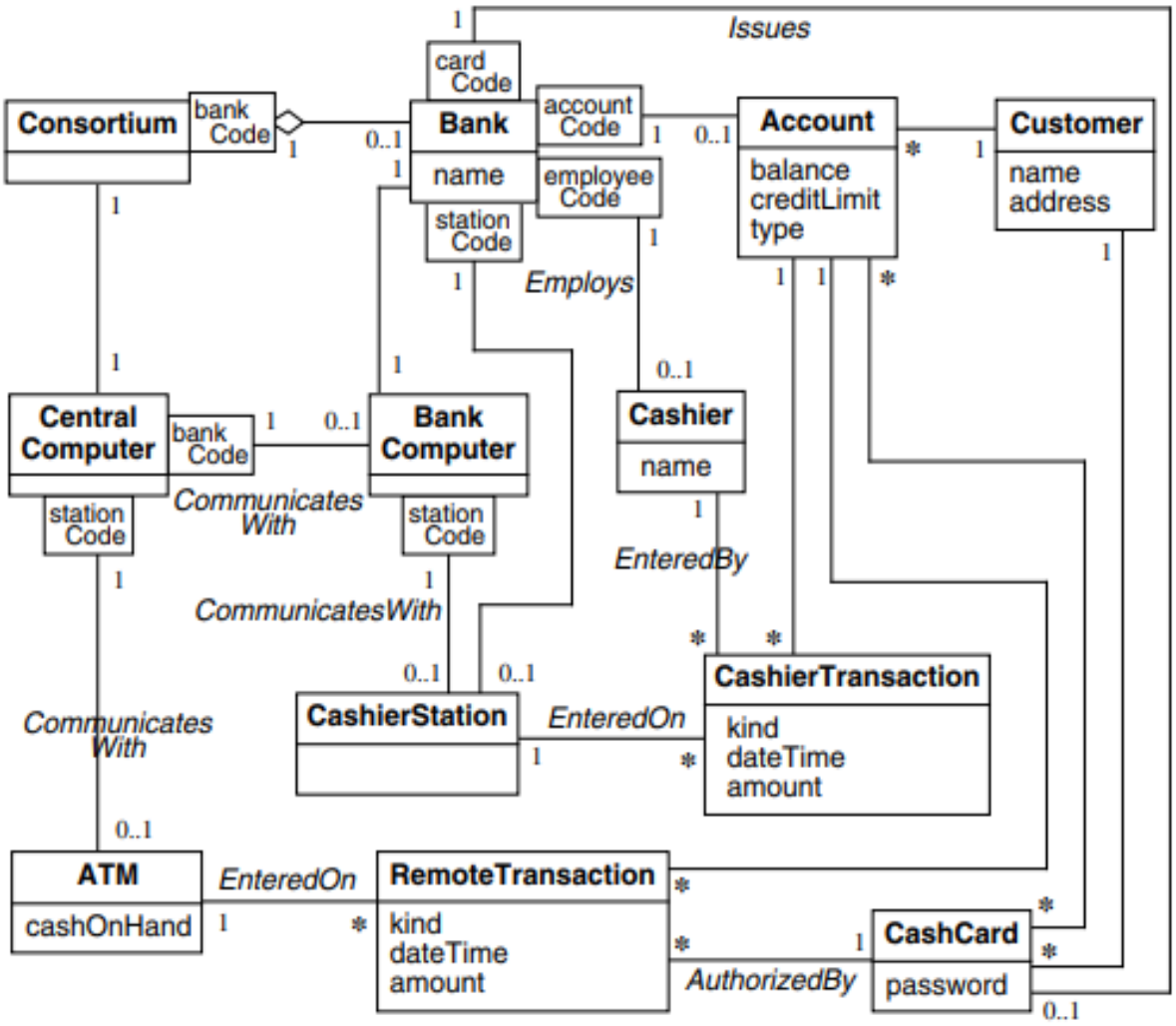

Figure 12.10 ATM class model with attributes

Figure:12.10 ATM class model with attribute

Refining with Inheritance

The next step is to organize classes by using inheritance to share common structure. Inheritance can be added in two directions: by generalizing common aspects of existing classes into a superclass (bottom up) or by specializing existing classes into multiple subclasses (top down).

Organize classes by using inheritance to share common features:

- **Bottom up generalization**.: You can discover inheritance from the bottom up by searching for classes with similar attributes, associations, and operations. For each generalization, define a superclass to share common features.

  ATM example. RemoteTransaction and CashierTransaction are similar, except in their initiation, and can be generalized by Transaction. On the other hand, CentralComputer and BankComputer have little in common for purposes of the ATM example.

- **Top-down generalization:** Top-down specializations are often apparent from the application domain. Look for noun phrases composed of various adjectives on the class name: fluorescent lamp, incandescent lamp; fixed menu, pop-up menu, sliding menu. Avoid excessive refinement. If proposed specializations are incompatible with an existing class, the existing class may be improperly formulated.

- **Generalization vs. enumeration.:** Enumerated subcases in the application domain are the most frequent source of specializations. Often, it is sufficient to note that a set of enumerated subcases exists, without actually listing them. For example, an ATM account could be refined into CheckingAccount and SavingsAccount. While undoubtedly useful in some banking applications, this distinction does not affect behavior within the ATM application; type can be made a simple attribute of Account.

- **Multiple inheritance**: You can use multiple inheritance to increase sharing, but only if necessary, because it increases both conceptual and implementation complexity.

- **Similar associations**: When the same association name appears more than once with substantially the same meaning, try to generalize the associated classes. Sometimes the classes have nothing in common but the association, but more often you will uncover an underlying generality that you have overlooked.

- **Adjusting the inheritance level** :You must assign attributes and associations to specific classes in the class hierarchy. Assign each one to the most general class for which it is appropriate. You may need some adjustment to get everything righ

Fig 12.11 shows the ATM class model after adding inheritance

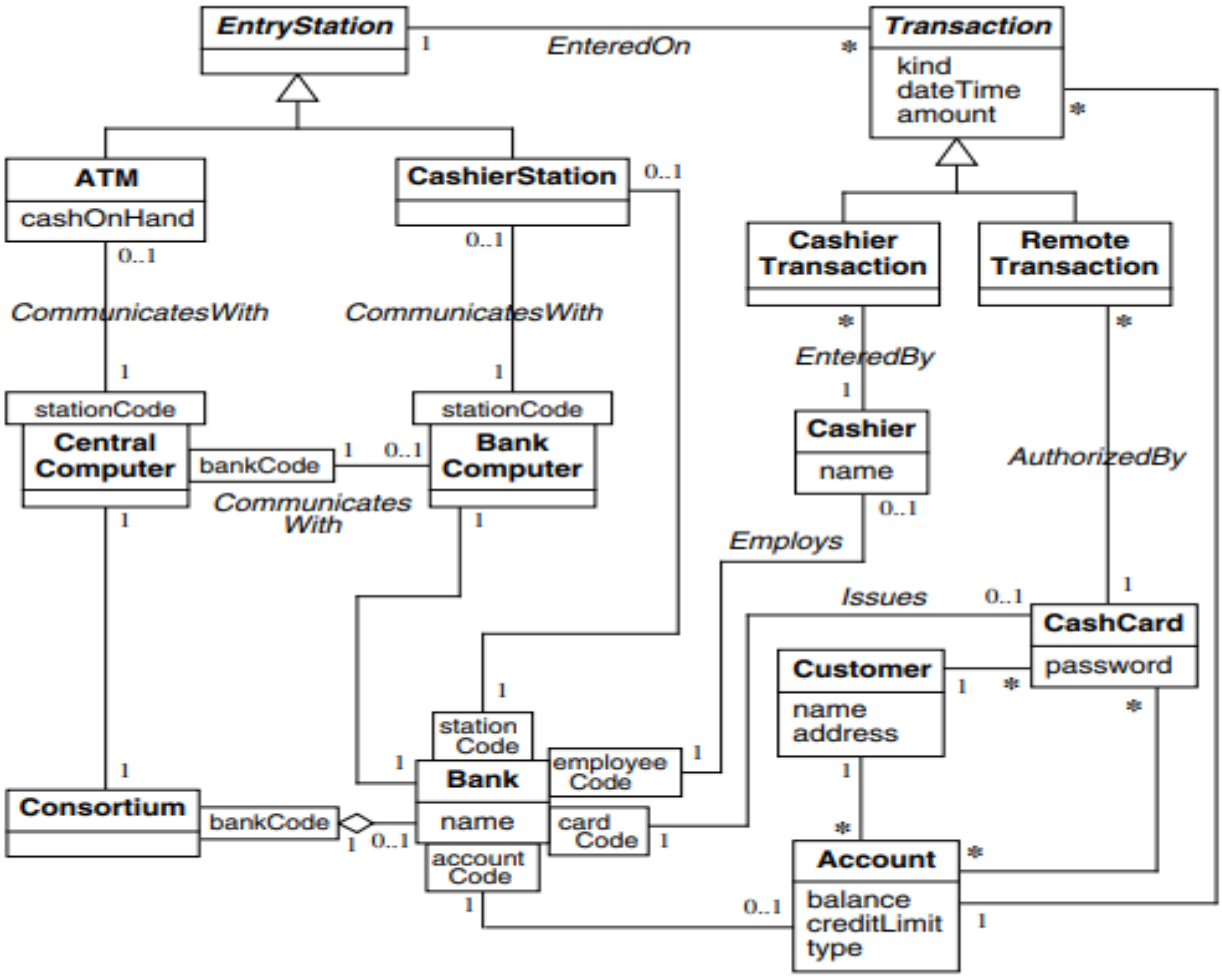

Figure 12.11 ATM class model with attributes and inheritance

Figure:12.11 ATM class model with attributes and inheritance

Testing Access Paths

- Verify that access paths exist for likely queries.
- Trace access paths through the class model to see if they yield sensible results

ATM example. A cash card itself does not uniquely identify an account, so the user must choose an account somehow. If the user supplies an account type (savings or checking), each card can access at most one savings and one checking account. This is probably reasonable, and many cash cards actually work this way, but it limits the system. The alternative is to require customers to remember account numbers. If a cash card accesses a single account, then transfers between accounts are impossible.

Iterating a Class Model

A class model is rarely correct after a single pass, so iterate and refine the model. The entire software development process is one of continual iteration; different parts of a model are often at different stages of completion. If you find a deficiency, go back to an earlier stage if necessary to correct it. Some refinements can come only after completing the state and interaction models.

There are several signs of missing classes.

- **Asymmetries in associations and generalizations**. Add new classes by analogy.
- **Disparate attributes and operations on a class.** Split a class so that each part is coherent.
- **Difficulty in generalizing cleanly**. One class may be playing two roles. Split it up and one part may then fit in cleanly.
- **Duplicate associations with the same name and purpose**. Generalize to create the missing superclass that unites them.
- **A role that substantially shapes the semantics of a class**. Maybe it should be a separate class. This often means converting an association into a class. For example, a person can be employed by several companies with different conditions of employment at each; Employee is then a class denoting a person working for a particular company, in addition to class Person and Company

Also look out for missing associations.

- **Missing access paths for operations.** Add new associations so that you can answer queries. Another concern is superfluous model elements.
- **Lack of attributes, operations, and associations on a class**. Why is the class needed? Avoid inventing subclasses merely to indicate an enumeration. If proposed subclasses are otherwise identical, mark the distinction using an attribute.
- **Redundant information**. Remove associations that do not add new information or mark them as derived.

And finally you may adjust the placement of attributes and associations.

- **Association end names that are too broad or too narrow for their classes.** Move the association up or down in the class hierarchy.
- **Need to access an object by one of its attribute values**. Consider a qualified association.

ATM example. CashCard really has a split personality—it is both an authorization unit within the bank allowing access to the customer's accounts and also a piece of plastic data that the ATM reads to obtain coded IDs. We should split cash card into two classes: CardAuthorization, an access right to one or more customer accounts; and CashCard, a piece of plastic that contains a bank code and a cash card number meaningful to the bank. Each card authorization may have several cash cards, each containing a serial number for security reasons. The card code, present on the physical card, identifies the card authorization within the bank. Each card authorization identifies one or more accounts—for example, one checking account and one savings account.

Transaction is not general enough to permit transfers between accounts because it concerns only a single account. In general, a Transaction consists of one or more updates on individual accounts. An update is a single action (withdrawal, deposit, or query) on a single account. All updates in a single transaction must be processed together as an atomic unit; if any one fails, then they all are canceled.

The distinction between Bank and BankComputer and between Consortium and CentralComputer doesn't seem to affect the analysis. The fact that communications are processed by computers is actually an implementation artifact. Merge BankComputer into Bank and CentralComputer into Consortium. Customer doesn't seem to enter into the analysis so far. However, when we consider operations to open new accounts, it may be an important concept, so leave it alone for now.

Fig 12.12 shows a revised class diagram that is simpler and cleaner.

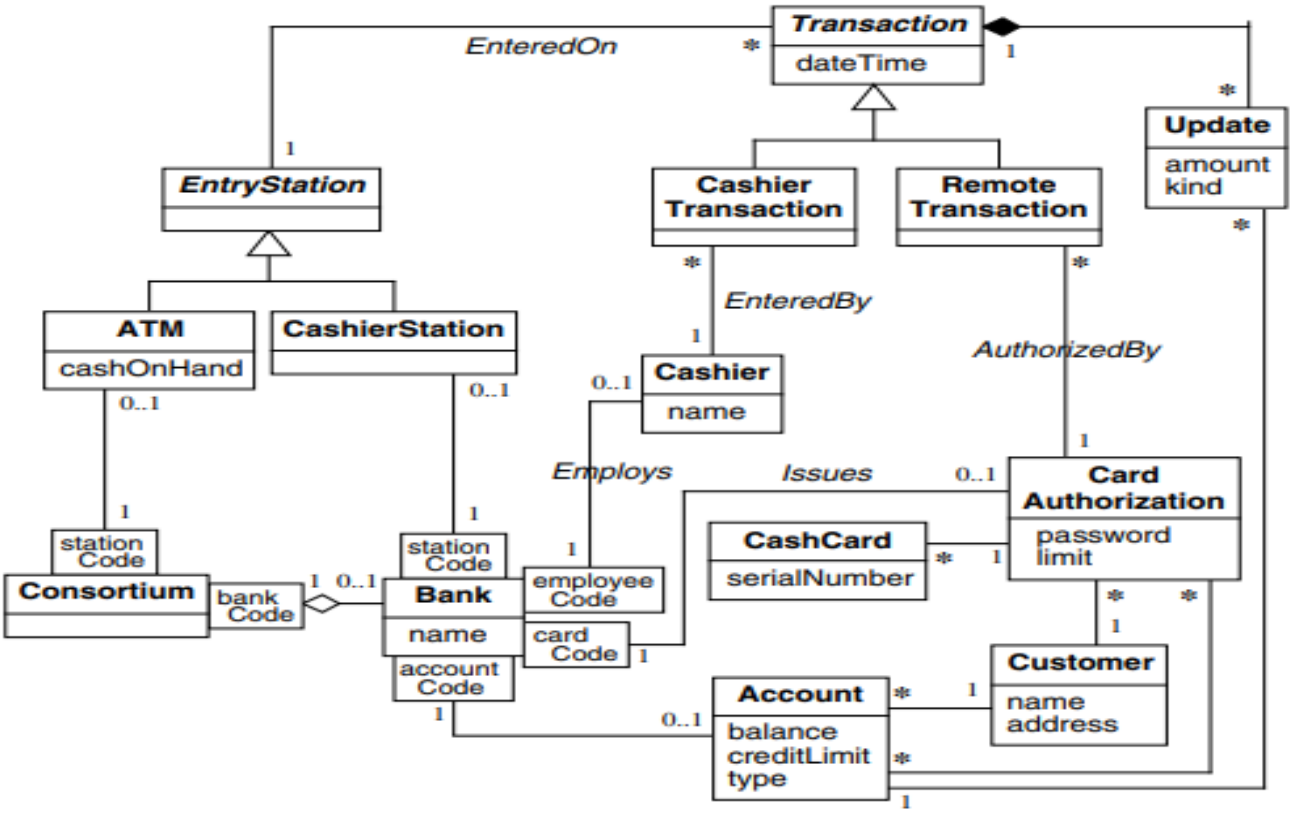

Figure 12.12 ATM class model after further revision

Reconsider the level of abstraction or Shifting the Level of Abstraction

Abstraction makes a model more complex but can increase flexibility and reduce the number of classes.

For example, we encountered one application in which the developers had separate classes for Individual Contributor, Supervisor, and Manager. Individual Contributors report to Supervisors and Supervisors report to Managers. This model certainly is correct, but it suffers from some problems. There is much commonality between the three classes—the only difference is the reporting hierarchy. For example, they

all have phone numbers and addresses. We could handle the commonality with a superclass, but that only makes the model larger. An additional problem arose when we talked to the developers and they said they wanted to add another class for the persons to whom managers reported

Figure 12.13 shows the original model and an improved model that is more abstract. Instead of "hard coding" the management hierarchy in the model, we can "soft code" it with an association between boss and worker. A person who has an employee Type of "individual Contributor" is a worker who reports to another person with an employee Type of "supervisor." Similarly, a person who is a supervisor reports to a person who is a manager. In the improved model a worker has an optional boss, because the reporting hierarchy eventually stops. The improved model is smaller and more flexible. An additional reporting level does not change the model's structure; it merely alters the data that is stored.

Figure 12.13 Shifting the level of abstraction. Abstraction makes a model more complex but can increase flexibility and reduce the number of classes.

Group classes into packages.

- The last step of class modeling is to group classes into packages.
- A package is a group of elements(classes, association, generalizations and lesser packages) with common theme.
- Normally you should restrict each association to a single package, but you can repeat some classes in different packages.
- Reuse a package from a previous design if possible, but avoid forcing a fit. Reuse is easiest when part of the problem domain matches a previous problem

ATM example. The current model is small and would not require breakdown into packages, but it could serve as a core for a more detailed model.

The packages might be:

- tellers—cashier, entry station, cashier station, ATM
- accounts—account, cash card, card authorization, customer, transaction, update, cashier transaction, remote transaction
- banks—consortium, bank

Each package could add details. The account package could contain varieties of transactions, information about customers, interest payments, and fees. The bank package could contain information about branches, addresses, and cost allocations.

3. Domain State Model

The Following steps are performed in constructing a domain state model

- **Identifying classes with states**
- **Finding states**
- **Finding Events**
- **Building state diagrams**
- **Evaluating state diagrams**

Identifying classes with states : Examine the list of domain classes for those that have a distinct life cycle. Look for classes that can be characterized by a progressive history or that exhibit cyclic behavior. Identify the significant states in the life cycle of an object. For example, a scientific paper for a journal goes from Being written to Under consideration to Accepted or Rejected. There can be some cycles, for example, if the reviewers ask for revisions, but basically the life of this object is progressive

Finding states: List the states for each class. Characterize the objects in each class—the attribute values that an object may have, the associations that it may participate in and their multiplicities, attributes and associations that are meaningful only in certain states, and so on. Give each state a meaningful name. Avoid names that indicate how the state came about; try to directly describe the state.
ATM example. Here are some states for an Account: Normal (ready for normal access), Closed (closed by the customer but still on file in the bank records), Overdrawn (customer withdrawals exceed the balance in the account), and Suspended (access to the account is blocked for some reason).

Finding Events: Once you have a preliminary set of states, find the events that cause transitions among states. You can regard an event as completing a do-activity. For example, if a technical paper is in the state Under consideration, then the state terminates when a decision on the paper is reached. In this case, the decision can be positive (Accept paper) or negative (Reject paper). In cases of completing a doactivity, other possibilities are often possible and may be added in the future—for example, Conditionally accept with revisions.
ATM example. Important events include: close account, withdraw excess funds, repeated incorrect PIN, suspected fraud, and administrative action.

Building state diagrams: Note the states to which each event applies. Add transitions to show the change in state caused by the occurrence of an event when an object is in a particular state. If an event terminates a state, it will usually have a single transition from that state to another state. If an event initiates a target state, then consider where it can occur, and add transitions from those states to the target state. Consider the possibility of using a transition on an enclosing state rather than adding a transition from each substate to the target state. If an event has different effects in different states, add a transition for each state.
ATM example. Figure 12.14 shows the domain state model for the Account class.
Evaluating state diagrams: Examine each state model. Are all the states connected? Pay particular

attention to paths through it. If it represents a progressive class, is there a path from the initial state to the final state? Are the expected variations present? If it represents a cyclic class, is the main loop present? Are there any dead states that terminate the cycle? Use your knowledge of the domain to look for missing paths. Sometimes missing paths indicate missing states. When a state model is complete, it should accurately represent the life cycle of the class. ATM example. Our state model for Account is simplistic but we are satisfied with it. We would require substantial banking knowledge to construct a deeper model.

After running through the above steps, the domain state model obtained is a shown in Figure 12.14

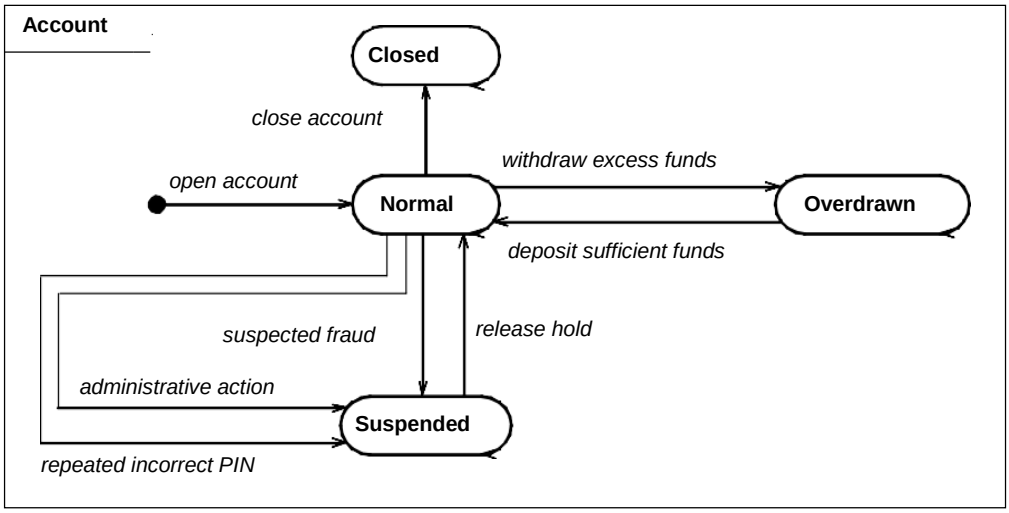

Figure: 12.14 Domain state model.

4. Domain Interaction Model

- The Interaction model is seldom important for domain analysis.
- The Interaction model is an important aspect of application modeling.
- Begin interaction modeling by determining the overall boundary of the system. Then identify use cases and flesh them out with scenarios and sequence diagrams. You should also prepare activity diagrams for use cases that are complex or have subtleties. Once you fully understand the use cases, you can organize them with relationships. And finally check against the domain class model to ensure that there are no inconsistencies.

5.Iterating the Analysis

To understand a problem with all its implications, you must attack the analysis iteratively, preparing a first approximation to the model and then iterating the analysis as your understanding increases.

The iteration to the analysis should be done as follows:

- **Refining the Analysis Model**
- **Restating the Requirements**
- **Analysis and Design.**

Refining the Analysis Model

The overall analysis model may show inconsistencies and imbalances within and across models. Iterate the different portions to produce a cleaner, more coherent model. Try to refine classes to increase sharing and improve structure. Add details that you glossed over during the first pass. Some constructs will feel awkward and won't seem to fit in right. Reexamine them carefully; you may have the wrong concepts. Sometimes major restructuring in the model is needed as your understanding increases.

Restating the Requirements

When the analysis is complete, the model serves as the basis for the requirements and defines the scope of future discourse. Most of the real requirements will be part of the model. In addition you may have some performance constraints; these should be stated clearly, together with optimization criteria. Other requirements specify the method of solution and should be separated and challenged, if possible

The final verified analysis model serves as the basis for system architecture, design, and implementation. You should revise the original problem statement to incorporate corrections and understanding discovered during analysis.

Analysis and Design.

The goal of analysis is to specify the problem fully without introducing a bias to any particular implementation, but it is impossible in practice to avoid all taints of implementation. There is no absolute line between the various development stages, nor is there any such thing as a perfect analysis

Chapter Summary.

- The domain model captures general knowledge about the application-concepts and relationships know to experts in the domain.
- The domain model has class model and state model but seldom has an interaction model.
- A good analysis captures the essential features without introducing the implementation artifacts that prematurely restrict the design decisions.
- The result of analysis replaces the original problem statement and serves as the basis for design

Exercise

1. For each of the following systems, identify the relative importance of the three aspects of modeling
    1) Class modeling 2) state modeling 3) interaction modeling

    a. bridge player
    b. change making machine
    c. car cruise control
    d. electronic typewriter
    e. spelling checker
    f. telephone answering machine.

Answer:

a.bridge player. Interaction modeling, class modeling, and state modeling, in that order, are important for a bridge playing program because good algorithms are needed to yield intelligent play. The game involves a great deal of strategy. Close attention to inheritance and method design can result in significant code reuse. The interface is not complicated, so the state model is simple and could be omitted.

b. change-making machine. Interaction modeling is the most important because the machine must perform correctly. A change making machine must not make mistakes; users will be angry if they are cheated and owners of the machine do not want to lose money. The machine must reject counterfeit and foreign money, but should not reject genuine money. The state model is least important since user interaction is simple.

c. c**ar cruise control**. The order of importance is state modeling, class modeling, and interaction modeling. Because this is a control application you can expect the state model to be important. The interaction model is simple because there are not many classes that interact.

d. electronic typewriter. The class model is the most important, since there are many parts that must be carefully assembled. The interaction between the parts also must be thoroughly understood. The state model is least important.

e.spelling checker. The order of importance is class modeling, interaction modeling, and state modeling. Class modeling is important because of the need to store a great deal of data and to be able to access it quickly. Interaction modeling is important because an efficient algorithm is needed to check spelling quickly. The state model is simple because the user interface is simple: provide a chance to correct each misspelled word that is found.

f. telephone answering machine. The order of importance is state modeling, class modeling, and interaction modeling. The state diagram is non-trivial and important to the behavior of the system. The class model shows relationships between components that complement the state model. There is little computation or state diagrams to interact, so the interaction model is less important.

2. The following is the class model for the scheduler software:

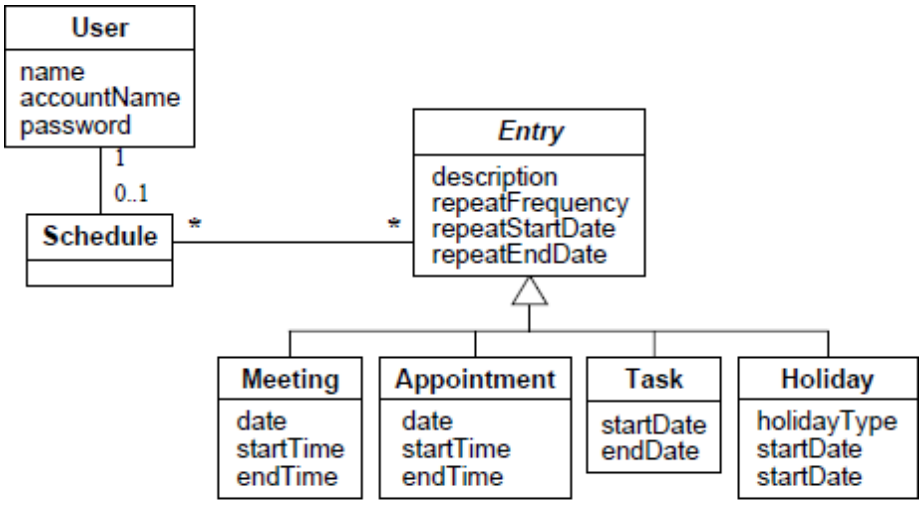

Figure A12.12 Class model for scheduler software

3. Ref to the Question 2. The following is a list of candidate classes. Prepare a list of classes that should be eliminated for any of the reasons listed in the chapter. Give reasons for each elimination. If there is more than one reason, give the main one.

   Scheduling software, meeting user, chairperson, software, meeting entry, schedule, attendee, scheduler, room, time, everyone, attendance, acceptance status, meeting notice, invitation, meeting information, invitee, notice.

Answer:

The following tentative classes should be eliminated.

Redundant classes. *Invitee* is synonymous with *user* and could be a role, depending on the realization of the class model. *Everyone* refers to *user* and is redundant. *Meeting notice* is the same as *notice*; we keep *notice*. *Attendance* is also extraneous and merely refers to an attendee participating in a meeting. *Meeting* and *meeting entry* are synonymous; we keep *meeting*.

Irrelevant or vague classes. We discard *scheduling software*, *scheduler*, *software*, and *meeting information* because they are irrelevant.

Attributes. We model *time* and *acceptance status* as attributes. An *invitation* is a type of notice; we need the enumeration attribute *noticeType* to record whether an invitation is an invitation, reschedule, cancellation, refusal, or confirmation.

Roles. *Chair person* and *attendee* are association ends for user.