Chapter 13

Application Analysis

Object-Oriented Analysis and Design
Process overview

- Application Interaction Model
- Application Class Model
- Overview of class Design

- Most domain model focus on building a model of intrinsic concepts.
- While Application model focus on the details of the application and consider interaction.
- You can construct application interaction model with following steps:
  - Determine the system boundary
  - Find actor
  - Find use cases

- Find initial and final events
- Prepare normal scenarios
- Add variation and exception scenarios
- Find external events.
- Prepare activity diagram for complex use cases.
- Organize actors and use cases
- Check against the domain class model.

Determine the system boundary

- Must know scope of an application *to specify functionality*.
- It means, you must decide *what system includes* and *what it omits.*
- If boundary is correct, you can treat system as box where internal details are hidden and changeable.
- At this state, *determine purpose of the system.*

- Don’t consider humans as part of system,
- Ex. From problem statement (Chap 11), mentioned “ design system for human cashiers and Automatic Teller machine (ATM)..”
- So here there will two different system is going to be design.
  - Human Cashier (Will be used at Bank)
  - ATM (At ATM location).

Note: Will focus on ATM behavior only.

Finding Actors

- Identify the external object that interact directly with the system called *Actors*
- Actors includes
  - Humans, external devices and other software systems.
- In finding actors, we are not searching for individual but for *standard behavior.*
- Each actor should be idealized.

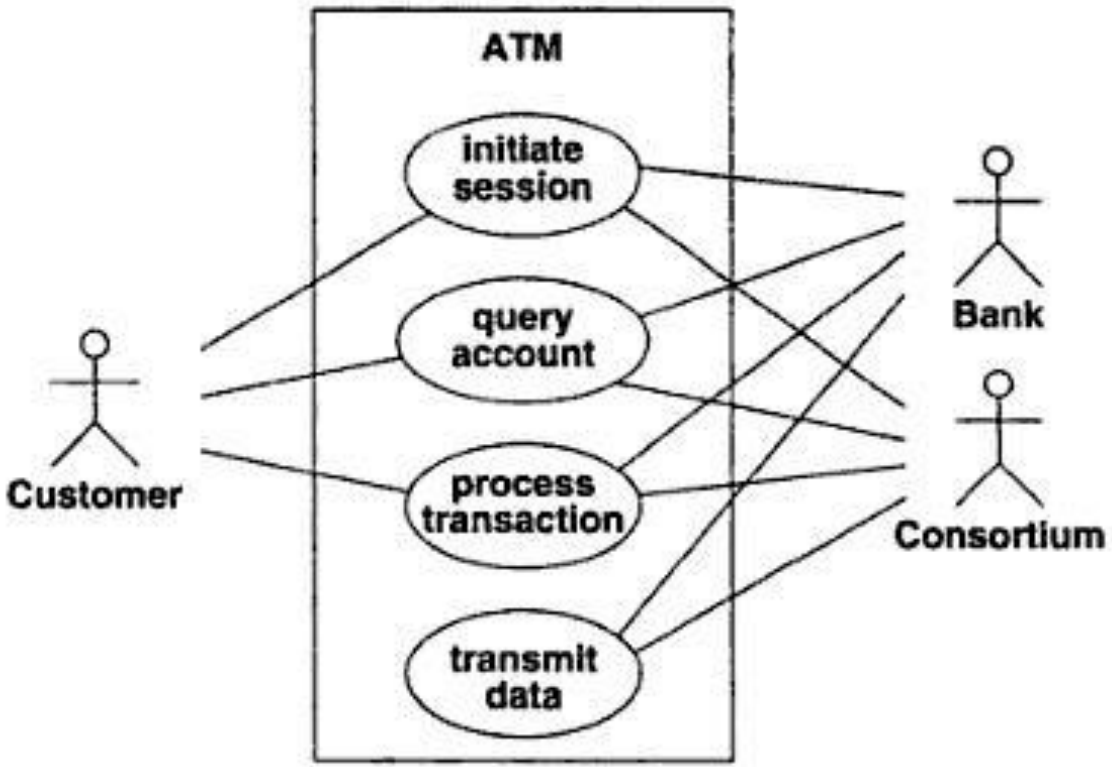
- It is possible for different kinds of external to play the same actor.
  - Ex. ATM application, the actors are Customer, Bank and Consortium.

Finding Use Cases

- For each actor, list the functionality different ways in which the actor uses the system called *Use Cases.*
- Use cases partition the functionality of a system into a small number of discrete units.
- Each use cases should represent a kind of service that system provides – something that provides value to the actor.

- Keep similar level of detail for use cases
  - "Apply for loan"
  - It should not be "Withdraw cash from saving account using ATM" . Restate with "Make Withdrawal"
- Now draw preliminary use case diagram.
- Show actors and use cases, connect actors to use cases.
- Usually, you can associate a use case with the actor that initiates it.

- You should also write a one or more sentence for each use case.



- Initiate session: ATM establish the identity of the user and make a list of accounts and actions.
- Query Account: System provides general data for an account, such as current balance, date of last transaction etc.
- Process Transaction: transaction like deposit, withdraw and transfer.
- Transmit Data: ATM uses the consortium's facilities to communicate with the appropriate bank computers.

Finding Initial and Final Events

- Use case diagram does not show behavior clearly.
- To Understand behavior, you must understand the execution sequences of each use cases.
- Determine which actor initiate the use case.
- In may case, initial event is request for services that use case provides.

- In many cases, initial event is occurrence that triggers a chain of activity.
- Similar, determine final event(s).
  - For ex. For “apply of loan” would continue until
    - Application submit
    - Loan grant or reject
    - Loan is delivered.
    - Paid off and Closed.
- User must define the scope for termination.

- For ATM Example:
  - Initiate session:
    - Initial Event is customer's insertion of a cash card.
    - Two Final Event: system keeps cash card and system returns the cash card.
  - Query Account:
    - Initial Event is customer's request for account data.
    - Final event is system retrieve data for customer.

- Process Transaction:
  - Initial event is customer's initiation of transaction.
  - Two final event : committing or aborting it.
- Transmit Data:
  - Initial event:
    - customer's request for account data.
    - Recovery from network, power or kind of failure.
  - Final event: successful transmission of data.

Preparing Normal Scenarios

- For each use case, prepare one or more story base scenarios.
- Scenario illustrate the major interactions, external display and information exchange.
- Scenario is a sequence of events among a set of interacting objects.
- Think in terms of sample interactions.
- Sometime problem statement describes full interaction sequences, but most of time you will have invent.

- So prepare scenarios for “normal” cases – interaction without any unusual inputs or error conditions.
- Information values exchanged are event parameters.
  - Ex. Entered password has password value as a parameter.
- So for each event, identify the actor that caused the event and the parameters of the events.

Object-Oriented Analysis and Design

Process overview

Initiate session	The ATM asks the user to insert a card. The user inserts a cash card. The ATM accepts the card and reads its serial number. The ATM requests the password. The user enters "1234." The ATM verifies the password by contacting the consortium and bank. The ATM displays a menu of accounts and commands. . . . The user chooses the command to terminate the session. The ATM prints a receipt, ejects the card, and asks the user to take them. The user takes the receipt and the card. The ATM asks the user to insert a card
Query account	The ATM displays a menu of accounts and commands. The user chooses to query an account. The ATM contacts the consortium and bank which return the data. The ATM displays account data for the user. The ATM displays a menu of accounts and commands.
Process transaction	The ATM displays a menu of accounts and commands. The user selects an account withdrawal. The ATM asks for the amount of cash. The user enters $100. The ATM verifies that the withdrawal satisfies its policy limits. The ATM contacts the consortium and bank and verifies that the account has sufficient funds. The ATM dispenses the cash and asks the user to take it. The user takes the cash. The ATM displays a menu of accounts and commands.
Transmit data	The ATM requests account data from the consortium. The consortium accepts the request and forwards it to the appropriate bank. The bank receives the request and retrieves the desired data. The bank sends the data to the consortium. The consortium routes the data to the ATM.

Adding variations and Exception Scenarios

- Once normal scenarios prepared, consider “Special” cases, such as omitted input, maximum and minimum values and repeated values.
- Then consider “error” cases, including invalid values and failures to respond.
- Consider various other kind of interactions also such as help request and status inquiries.

- ATM Example:
  - ATM can't read the card
  - Card has expired
  - ATM times out waiting for a response
  - Amount is invalid
  - Machine is out of cash or paper.
  - Communication lines are down.
  - Transaction rejected because of suspicious patterns of card usage.

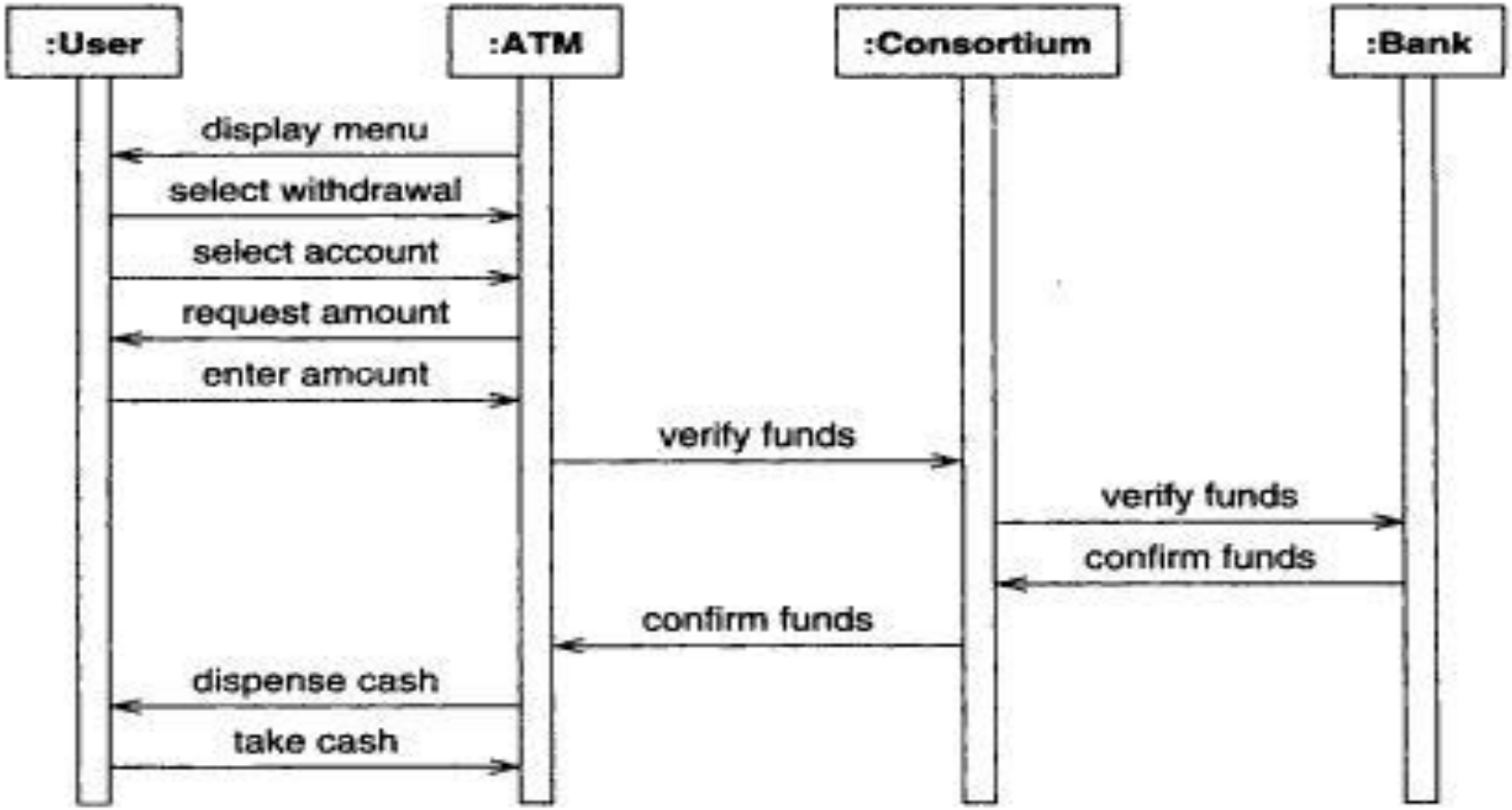
Finding External Events

- To find all external events – includes all inputs, decisions, interrupts, and interaction to or from users or external devices.
- An Event can execute effects for a target object.
- Use scenario to find out normal events, unusual event and error conditions.
- In simple words, transmission of information to an object is Event.

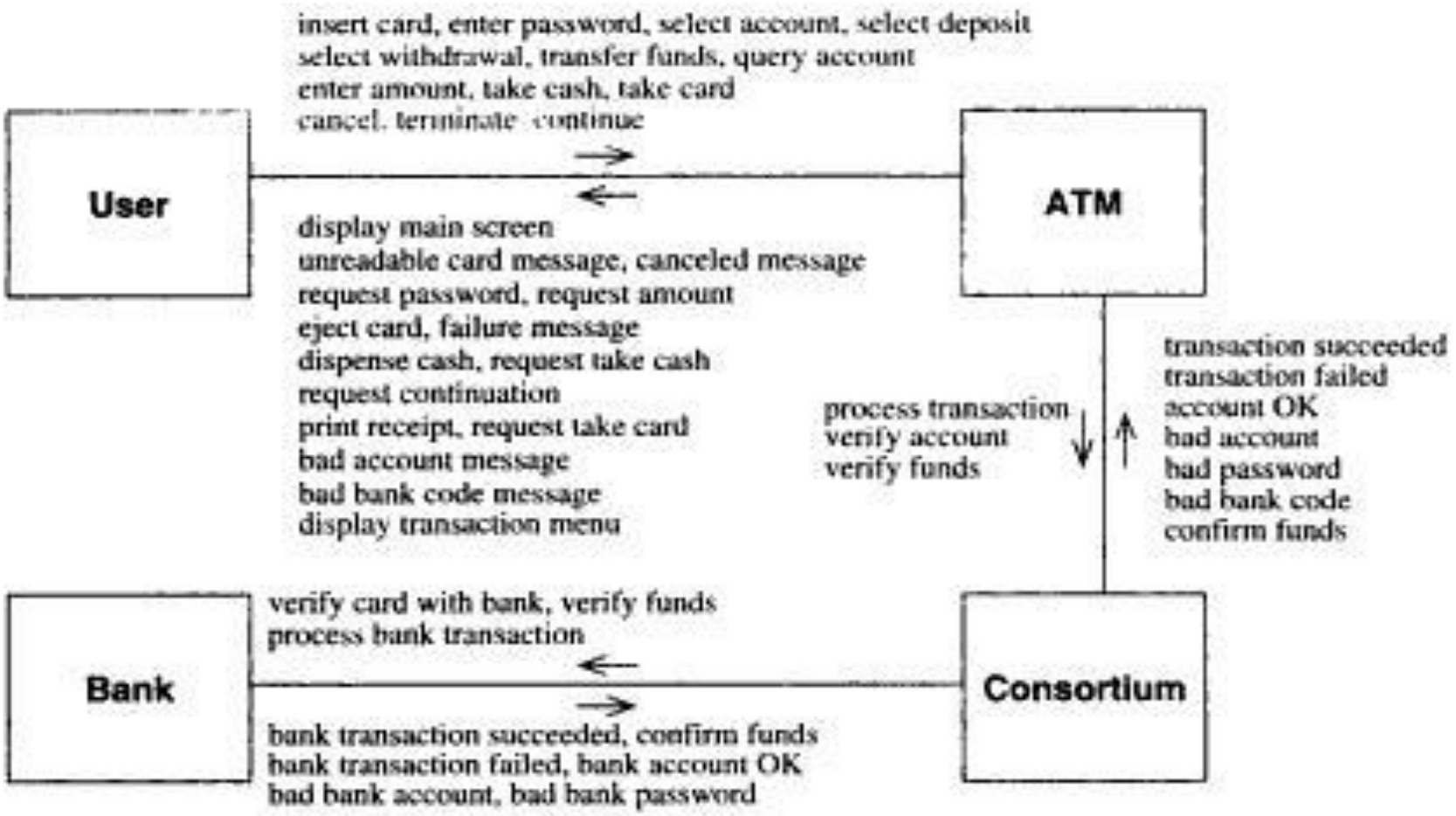
- For ex. *Enter password* is message from external *User* to application object *ATM.*
- Event instances whose values affect the flow of control should be different kinds of events.
  - Account OK, Bad Account and Bad Password are different events.
- Based on event entered, prepared sequence diagram for it.

- Prepare a sequence diagram for each scenario.
- It shows participation in an interaction and sequences of message among them.
- From sequence diagram, you can then summarize the events that each class sends and receives.

Object-Oriented Analysis and Design
Process overview



Object-Oriented Analysis and Design
Process overview



Preparing Activity diagrams for Complex use Cases

- Sequence diagram capture dialog and interplay between actors.
- Do not clearly show alternatives and decisions.
- Activity diagram let you consolidate all the behavior by documenting forks and merges in the control flow.

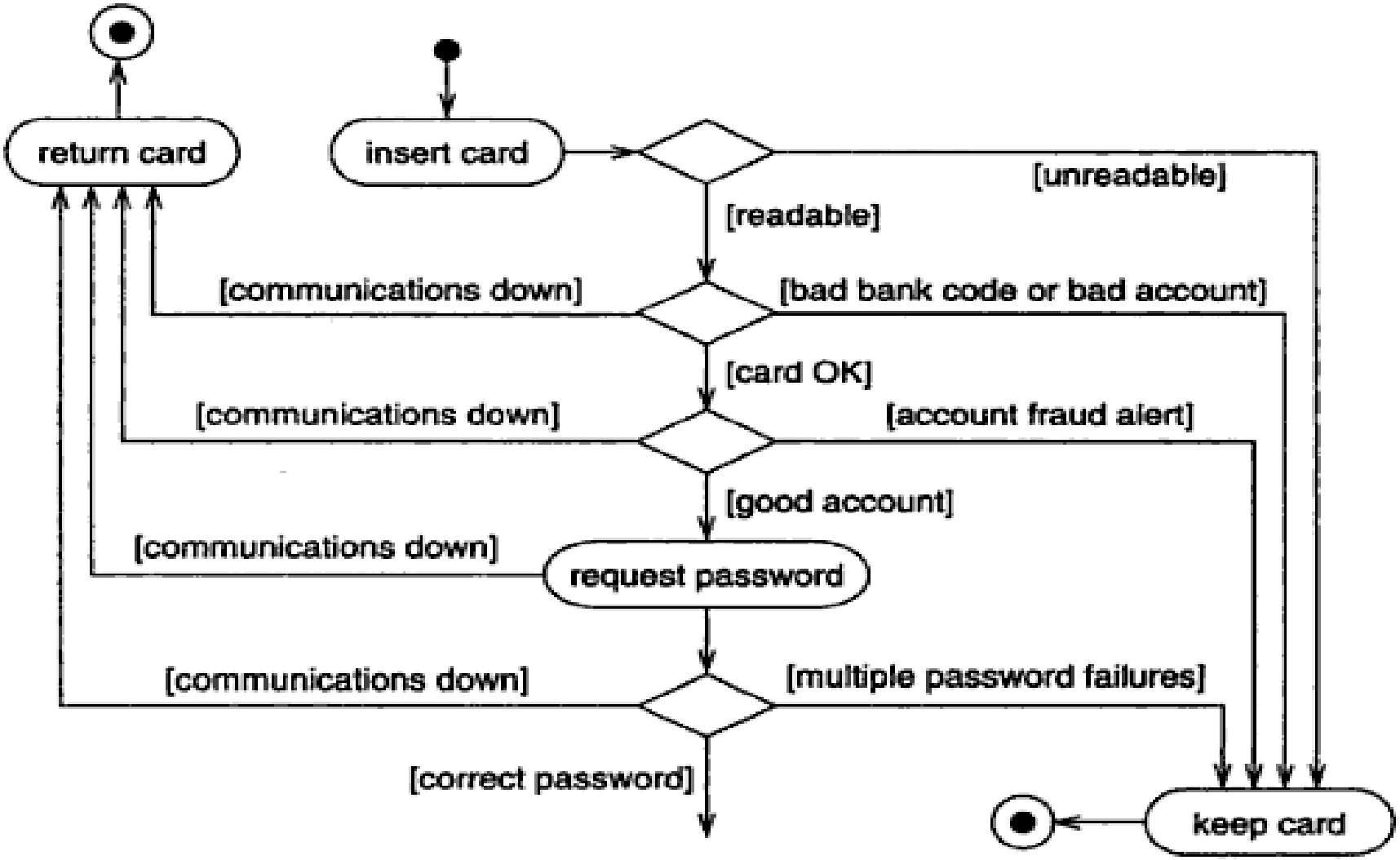
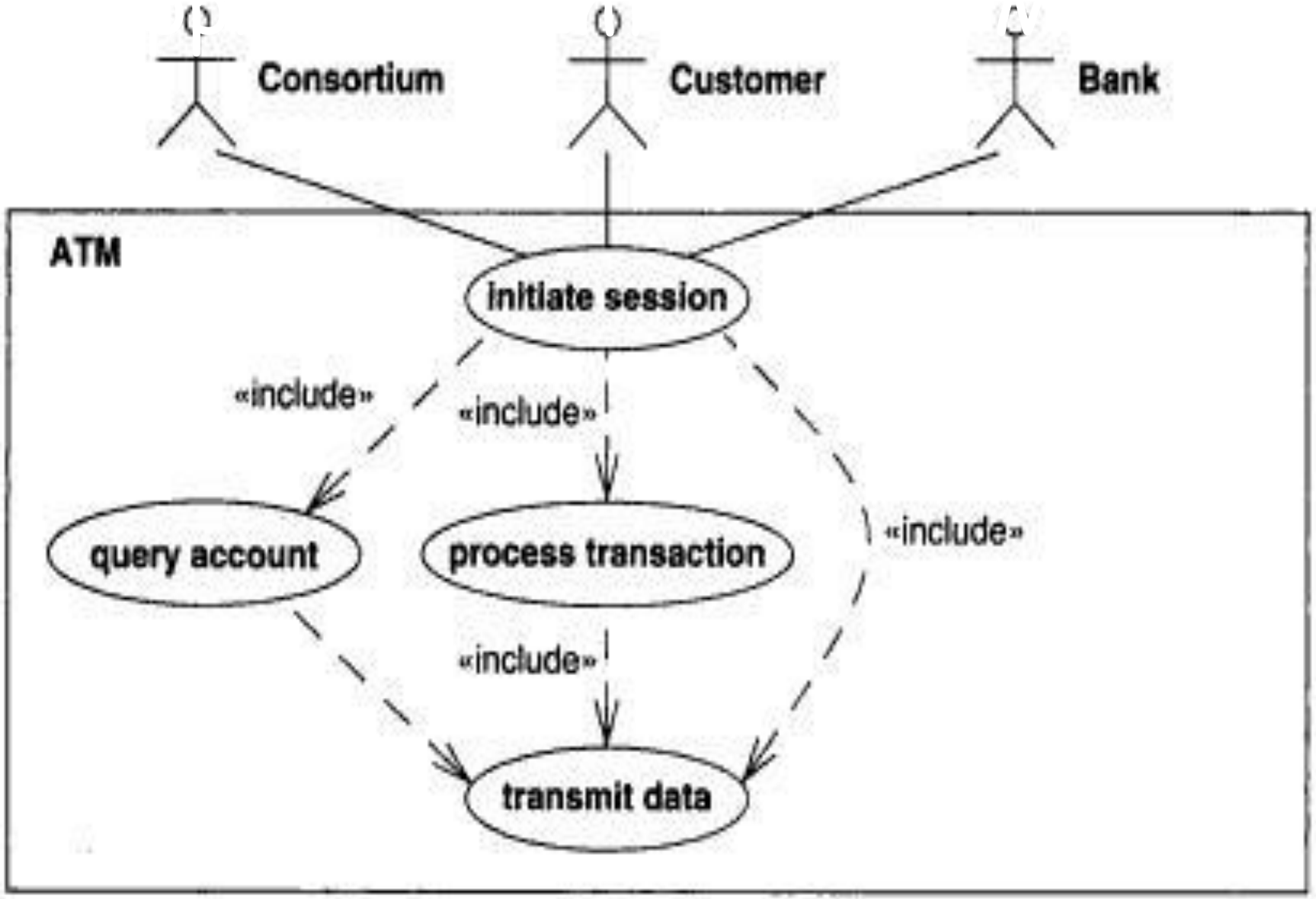


Figure 13.5 Activity diagram for card verification. You can use activity diagrams to document business logic, but do not use them as an excuse to begin premature implementation.

Organizing Actors and Use cases

- Next step to organize use cases with relationship (includes, extend and generalization)
- It will helpful for large and complex systems.
- For. Admin might be an operator with additional privileges.

Object-Oriented Analysis and Design
Process overview



Checking Against the Domain Class model

- Application and domain models should be mostly consistent.
- The actors, use cases and scenarios are all based on classes and concept from domain model.
- Cross check the application and domain models to ensure that there are no inconsistencies.

- **Construct an application class model with following steps:**
  - Specify User Interfaces
  - Define Boundary classes.
  - Determine controllers.
  - Check against the interaction model.

Specifying User Interface

- Most interaction divided into two parts
  - Application Logic
  - User Interface
- A *user interface* provides the user with way to access its objects, command (function/Features) and application options.
- Same *program logic* can accept input from command lines, files, mouse buttons, touch pane, physical push buttons, or remote links.

- It is acceptable to sketch out a sample interface to help you visualize the operation of an application.
- Might need mock up the interface so that user can try it.
- Dummy procedure can simulate application logic. It will help you to evaluate the “look and feel” of the user interface.

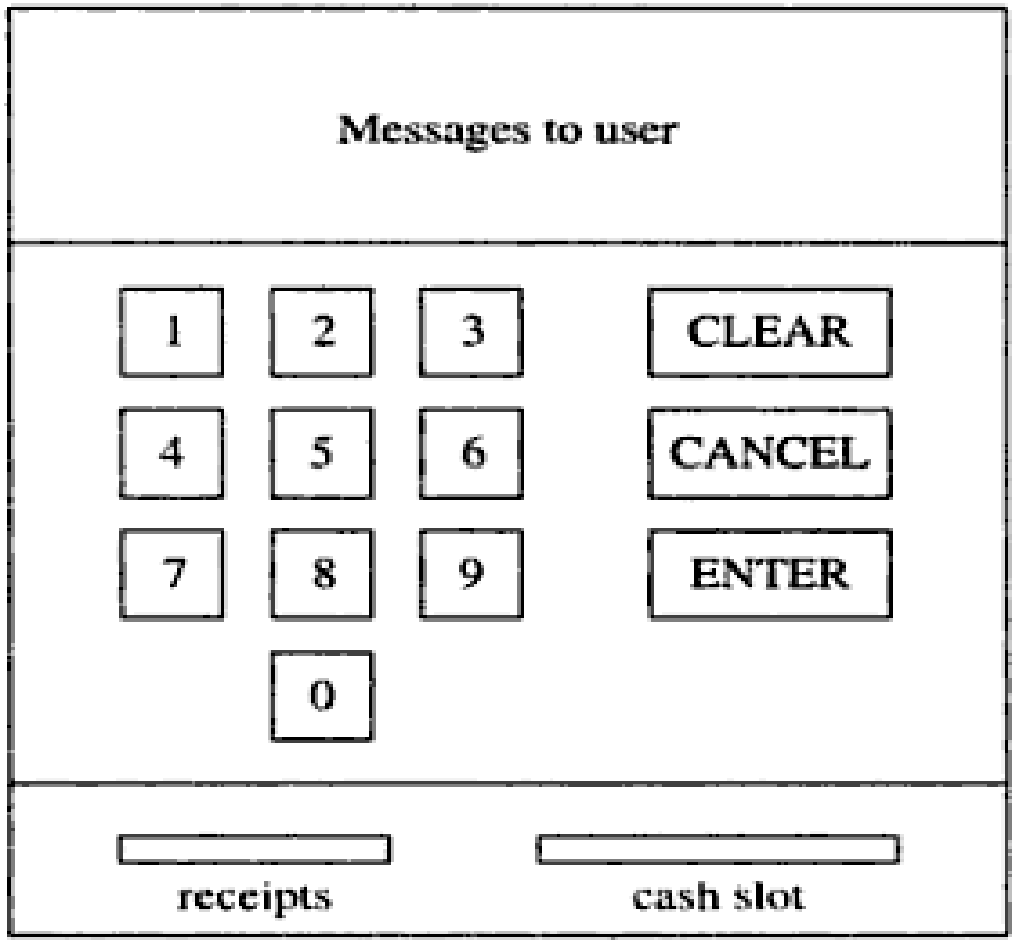


Figure 13.7 Format of ATM interface. Sometimes a sample interface can help you visualize the operation of an application.

Defining Boundary Classes

- It is always helpful to define boundary classes to isolate the inside of a system from the external world.
- A *boundary class* is a class that provides a staging area for communication between a system and an external source.
- It understand the format of one or more external sources and converts information for transmission to and from the internal system.

- For ex. To summarize the communication between ATM and consortium, we need to define boundary classes (CashCardBoundary, AccountBoundary).

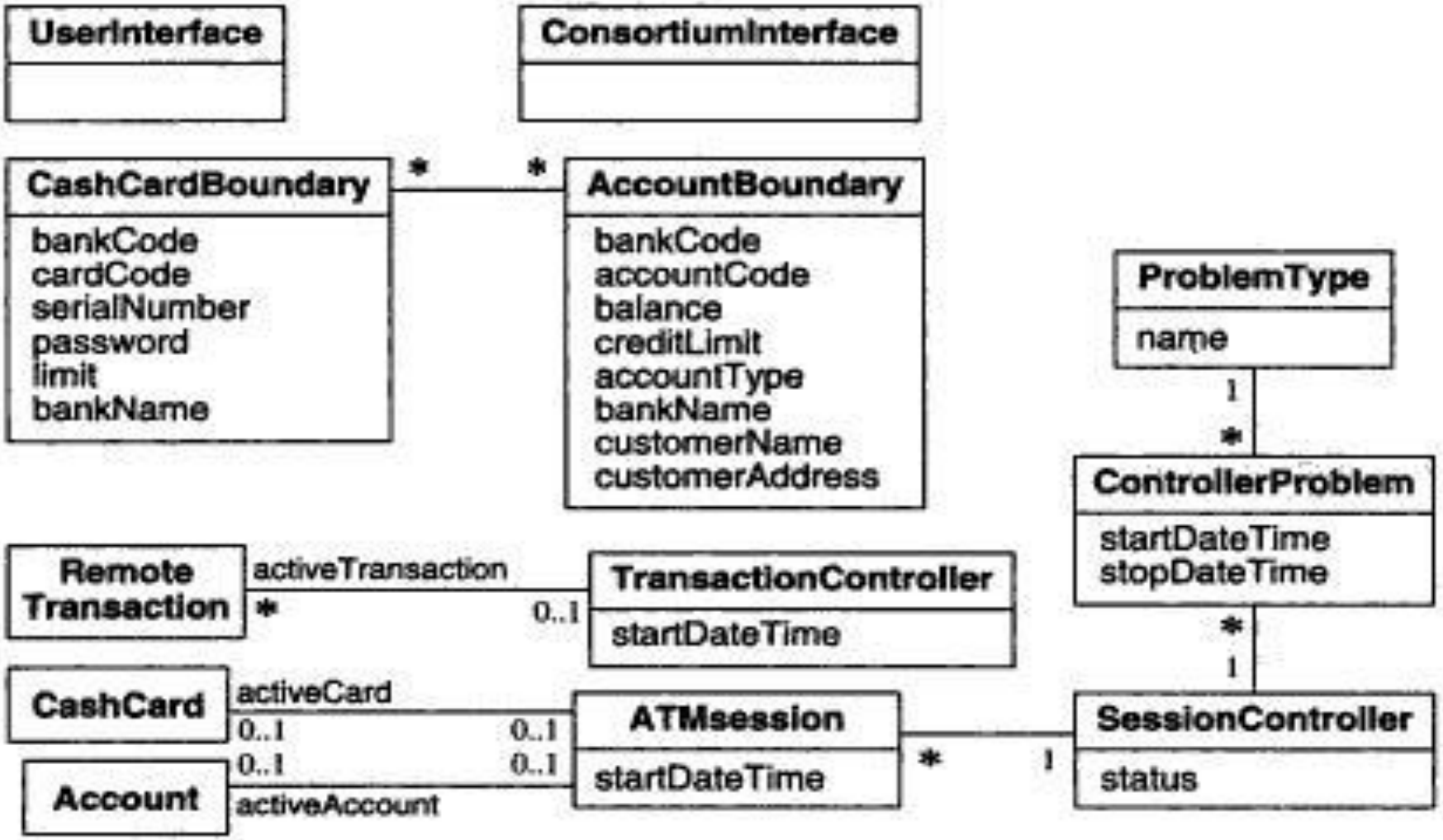
- Determining Controllers
- A controller is active object that manages control within the application.
  - It receives signal from outside world.
  - Reacts to them
  - Invokes operation on the objects in the system.
  - Send signals to outside world.
- A controller is piece of reified behavior captured in form of object.

- For Ex. ATM has two major control loops.
  - Outer loop verifies customer and account.
  - Inner loop services transactions.

Checking against the Interaction Model

- Once you build class model, go over the use cases and think about how they would work.
- For Examples
  - user sends a command
  - Parameters of command must come from UI object.
  - Requesting a command itself must come from some controller.
- Finally, simulate use case with the classes.

Object-Oriented Analysis and Design
Process overview



Overview of Class Design

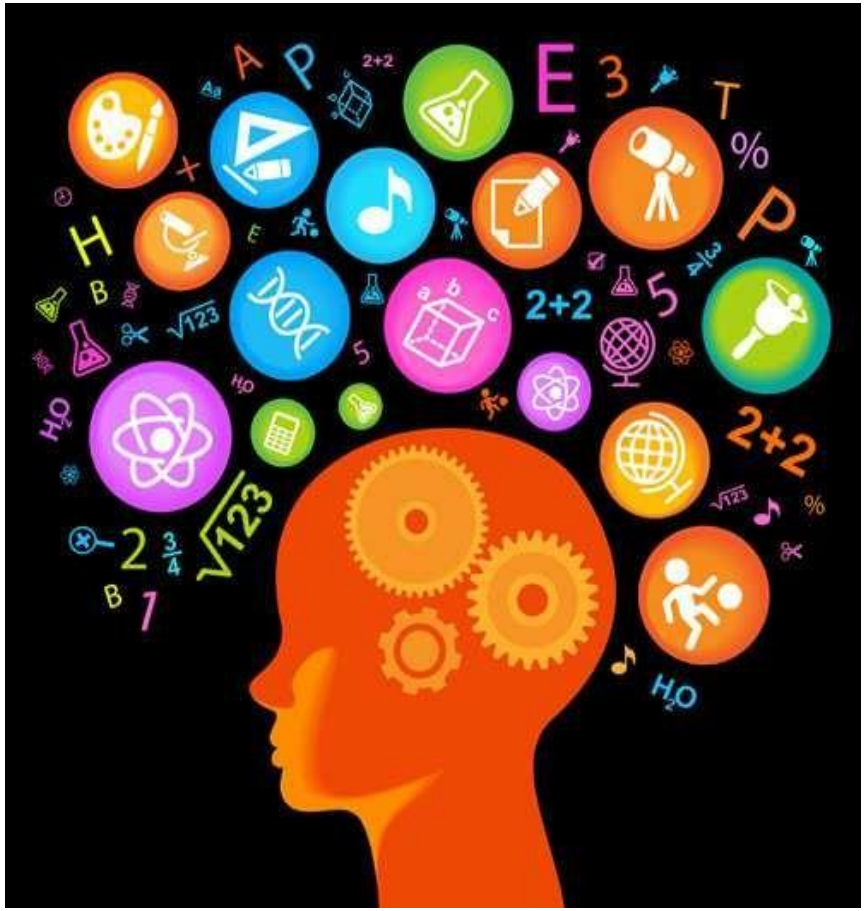
- In general class design is a process to add details to the class diagrams defined in the analysis phase and making fine decisions
- You choose how to implement your classes considering
  - Minimization of execution time, memory and other cost measures
  - Choice of algorithms implementing methods
  - Breaking complex operation into simpler operations
- OO design is an iterative process
  - When one level of abstraction is complete you should design the next lower level of abstraction
  - For each level you may
    - add new operations, attributes,and classes
    - Revise relations between classes

1.The Essence of Design is…

- To build a bridge across the gap between:
  - Desired features
    - Use cases
    - Application commands
    - System operations
    - System services
  - Available resources
    - Operating system infrastructure
    - Class libraries
    - Previous applications

Creativity

- Design is a creative task
  - You can only be provided with guidelines
- The tools you have to fill the gap between resources and features are
  - Classes
  - Operations
  - Other UML constructs
- You may have to compromise
  - The final goal is not to choose the best for single decisions but to optimize the entire system



2.Realizing Use Cases

- Use Cases define the functionalities of a system but not how to realize them
  - One goal of the design phase is to choose among different possible realizations (finding a balance between advantages and disadvantages)
- Implement the required behavior is not sufficient
  - You should take into consideration performance, reliability, facilitating possible future enhancements
- Use Cases define system level operations
  - During design you invent new objects and new operations (at a lower level of abstraction) to provide this behavior
  - Again: Bridge the gap!

Responsibilities

- First step for realizing a use case is to list its responsibilities
- A responsibility is something that a us case must do to be implemented
- Example: Online theater ticket system
  - Use case: Making Reservation
  - Responsibilities:
    - Finding unoccupied seats for the desired show
    - Marking the seats as occupied
    - Obtaining payment from the customer
    - Arranging delivery of the tickets
    - Crediting payment to the proper account

4.Designing Algorithms

- Choosing data structures
  - Data structures do not add information to the analysis model
  - Data structures organize information to permit efficient algorithms
  - Data structures include:Arrays, Lists, Trees, Sets, etc.
- Defining internal classes and operations
  - Expanding high level operation through algorithms may lead to create new (low-level) classes and operations to hold intermediate results

5.Recursing Downward

- Organize operations as layers
  - Operations in higher layers invoke operations in lower layers
- In general, the design process works top down
  - Start with the higher-level operations and proceed to define lower-level operations
  - Functionality Layers
    - High-level functionalities are decomposed into lesser operations
    - Implementation of the responsibilities
  - Mechanism Layers
    - Support mechanisms to make the system working
      - Store information, coordinate objects, sequence control, transmit information, perform computations, etc.

6.Refactoring

- The initial design always contains inconsistencies, redundancies,inefficiencies
- It is impossible to get a large correct design in one pass
- Refactoring is an essential part of any good engineering process
- It's not enough to deliver a functionality
  - If you expect to maintain a design, then you must keep the design clean, modular and understandable

7.Design Optimization: Provide efficient access paths

- Adjusting the structure of the class model to optimize frequent traversals

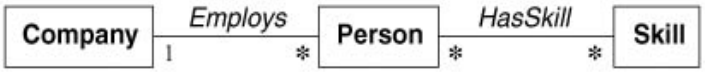


Figure 15.5 Analysis model for person skills. Derived data is undesirable during analysis because it does not add information.

- Company.findSkill(*speakJapanese*)
  - Frequency of access
  - Fan out (e.g., 1000 employees with on average 10 skills)
  - Selectivity (e.g., if only 5 employees actually speak Japanese)

8.Adjustment of Inheritance

- Through the following steps:
  - Rearrange classes and operations to increase inheritance
  - Abstract common behavior out of groups of classes
  - Use delegation to share behavior when inheritance is semantically invalid

9.Organizing a Class Design

- Information hiding
  - Separating external specification from internal implementation
- Coherence of Entities
  - An entity (a class, an operation or a package) should have a single major theme
    - A method should do only 1 thing
    - A class should not serve many purposes at once
- Fine-Tuning Packages
  - The interface between two packages (the associations that relate classes in one package to classes in the other and operations that access classes across package boundaries) should be minimal and well defined

Chapter 12

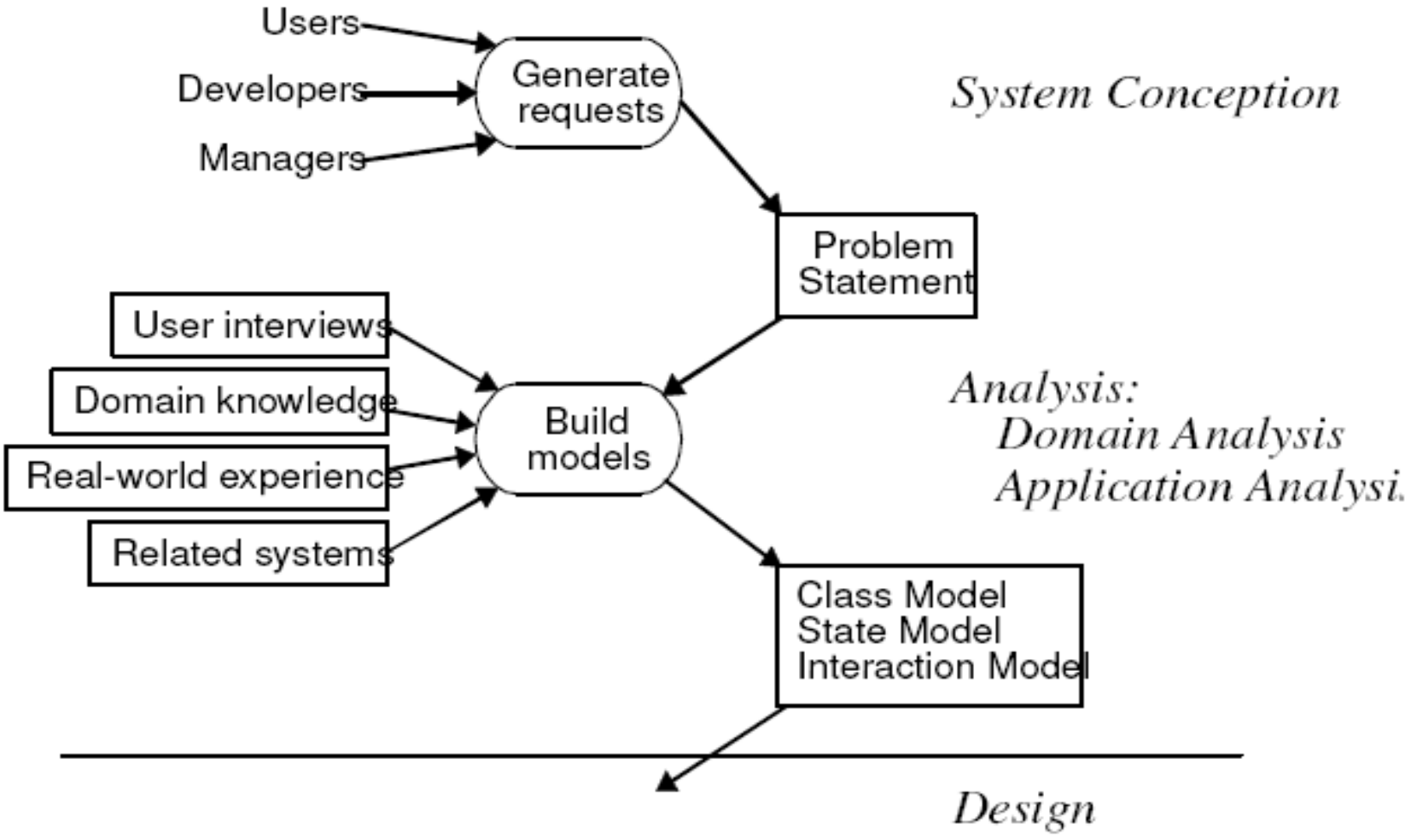
Domain Analysis

Object-Oriented Analysis and Design
Process overview

- Introduction
- Overview of Analysis
- Domain Class Model
- Domain State Model
- Domain Interaction Model

- During analysis, we build models and begin to understand the requirements deeply.
- To build a domain model, you must interview business expert, examine requirements statements, and study related requirements.
- Successful analysis model states what must be done, without restricting how it is done and avoid implementation decisions.

Object-Oriented Analysis and Design
Process overview



- As fig. shows, analysis begin with *problem statement* during system conception.
- Problem statement can be *incomplete or informal* but analysis makes it more *precise and expose ambiguities.*
- You must understand real-world system described by the problem statement, and *abstract its essential features into a model*.
- Sequence can be problem statement → build model (Domain) → build model(application).

- Analysis model addresses the three aspects of objects.
  - Static structure of objects (Class Model)
  - Interaction among objects (Interaction Model)
  - Life-cycle histories of objects (State 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:

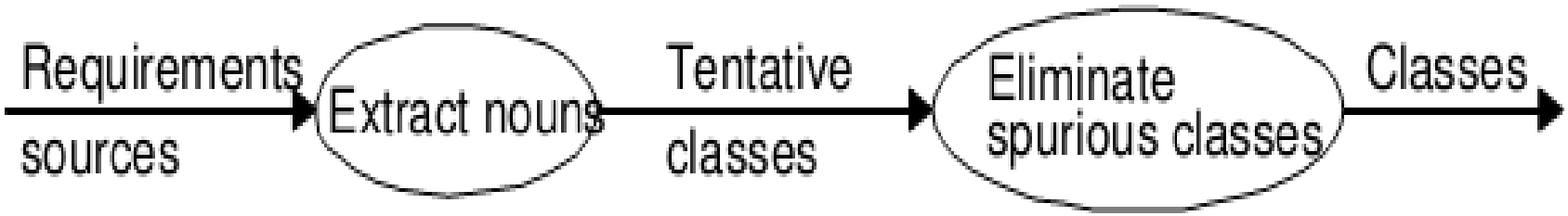
1. Find Classes.
2. Prepare a data dictionary.

3. Find associations.
4. Find attributes of objects and links.
5. Organize and simplify classes using inheritance.
6. Verify that access paths exist for likely queries.
7. Iterate and refine the model.
8. Reconsider the level of abstraction.
9. Group classes into packages

1. Finding Classes

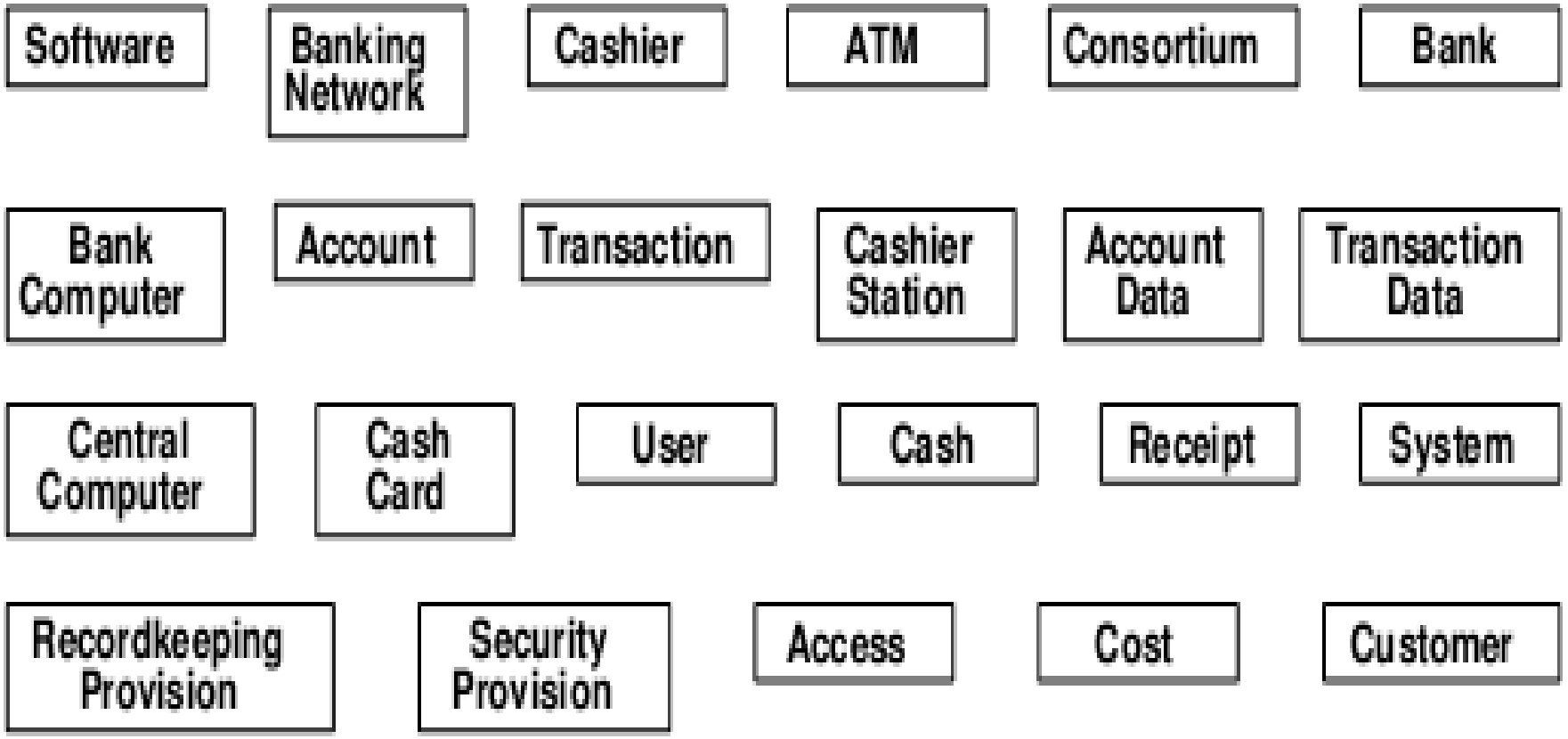
- First Step, find relevant classes for objects from application domain.
  - It includes houses, person, machines etc.
- Classes often correspond to nouns.
- Eg- ” a reservation system sell tickets to performances at various theater”-
  - Tentative classes would be Reservation, System, Tickets, Performance and Theaters.

- Idea is to capture concepts. not all nouns are concepts, and concepts are also expressed in other parts of speech.



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

Object-Oriented Analysis and Design
Process overview



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

ATM classes identified from knowledge of problem domain

2.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 class has little or nothing do with application, eliminate it.
  - ATM Ex. *cost* is outside the scope of the ATM software.
- **Vague classes**: class should be specific.
  - ATM Example, System, Security provision, Banking network etc are not specific thing.
- **Attributes:** Names that primarily describe individual objects should be restated as attributes.

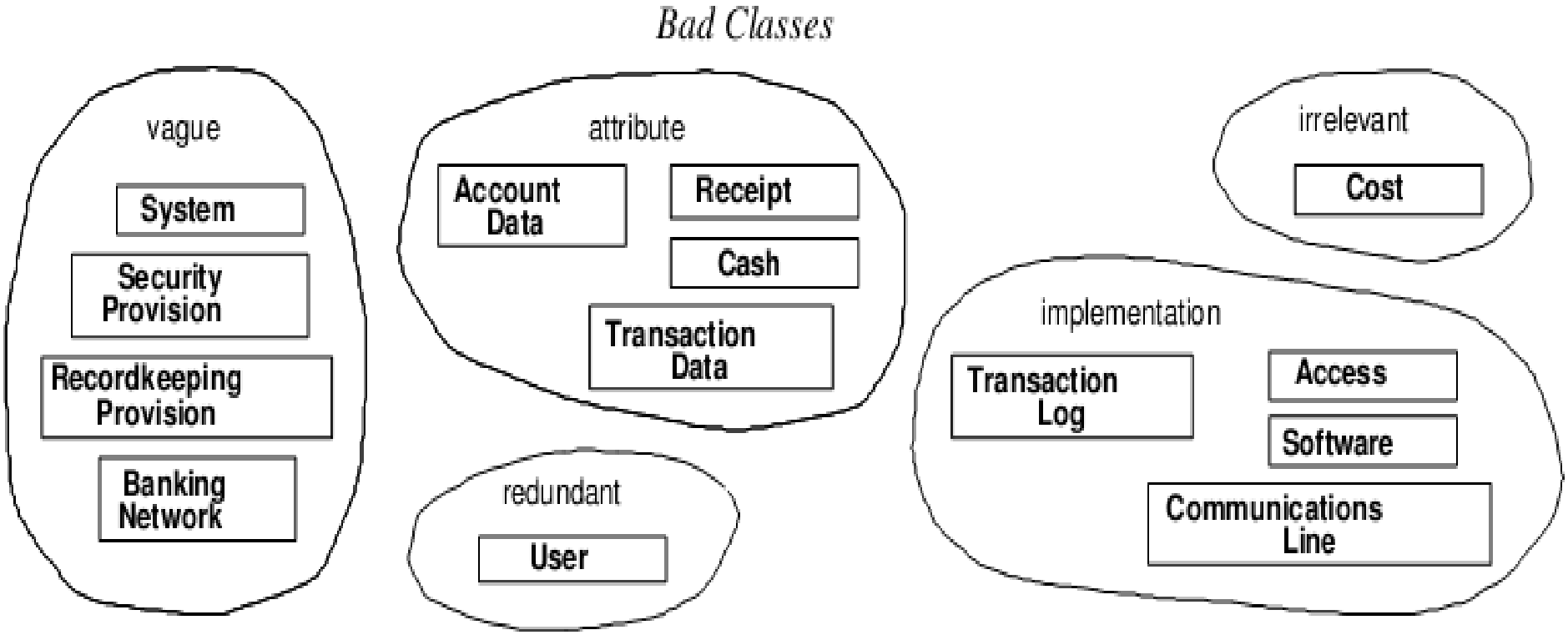
- ATM Example, Account Data, Cash, Transaction data are purely indicating attributes not a class.
- **Operations:** If a name describes an operation that is applied to objects and not manipulated in its own right, then it is not a class.
  - Eg-if we are simply building telephones, then call is part of the state model and not a class

  - But Billings system for telephone calls a Call would be important class with attributes 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.
  - Ex.Owner of a car in 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 irrelevant to the real world.
  - We may need them during design and not now.
  - Ex. Transaction Log class.
- **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.

Keeping the right classes:



3. 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.
- It also describes associations, attributes, operations and enumeration values.

- Account,
- ATM,
- Bank,
- BankComputer,
- CashCard,
- Cashier,
- CashierStation
- CentralComputer,
- Consortium,
- Customer,
- Transaction

4. Finding Associations

- Find 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.
  - Ex. Physical Location (part of, NextTo)
  - Directed Actions (Drives)
  - Communication (Talks To)

- Ownership (Has, Part of)
- Satisfaction of condition (WorksFor, Manages).

- Idea here is to capture relationships

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

5. Keeping the Right Associations

Discard unnecessary and incorrect associations, using the following criteria:

- *Associations between eliminated classes:* If you have eliminated one of classes then either you *eliminate association* or *restate* it.
  - *Ex. Banking Network includes cashier stations and ATMs.*
  - *ATM dispenses cash*

- *ATM prints receipts*
- *Bank provide software*
- *Cost apportioned to banks*
- *System provides record keeping and*
- *System provides security.*

- *Irrelevant or implementation associations*: Eliminate any association that deals with *implementation* or *outer problem statement.*
  - *Ex. System handles concurrent access (Implementation)*

- *Actions:* An association should describe a structural property of the application domain not a transient event.
  - Ex. *ATM accepts Cash card* (Interaction cycle)

  It is not a permanent relationship between ATM and Cash.
  - Eliminate *ATM interact with user*. *Central computer clears transactions with bank* .

- *Ternary associations*: You can decompose most association among three or more classes into binary associations.
- Always decomposed without losing information.
- Ex. *Bank computer processes transaction against account* can be convert into *Bank computer processes transaction* and *transaction concern accounts*.

- *Derived associations* : Omit association and attributes, they may be redundant.
  - Ex. GrandParentOf can be defined in terms of pair of ParentOf.
  - Ex. youngerThan expresses condition on the birthdate of two person, not additional information.
- Derived association don't add information, they useful for understanding.

Semantics of Association

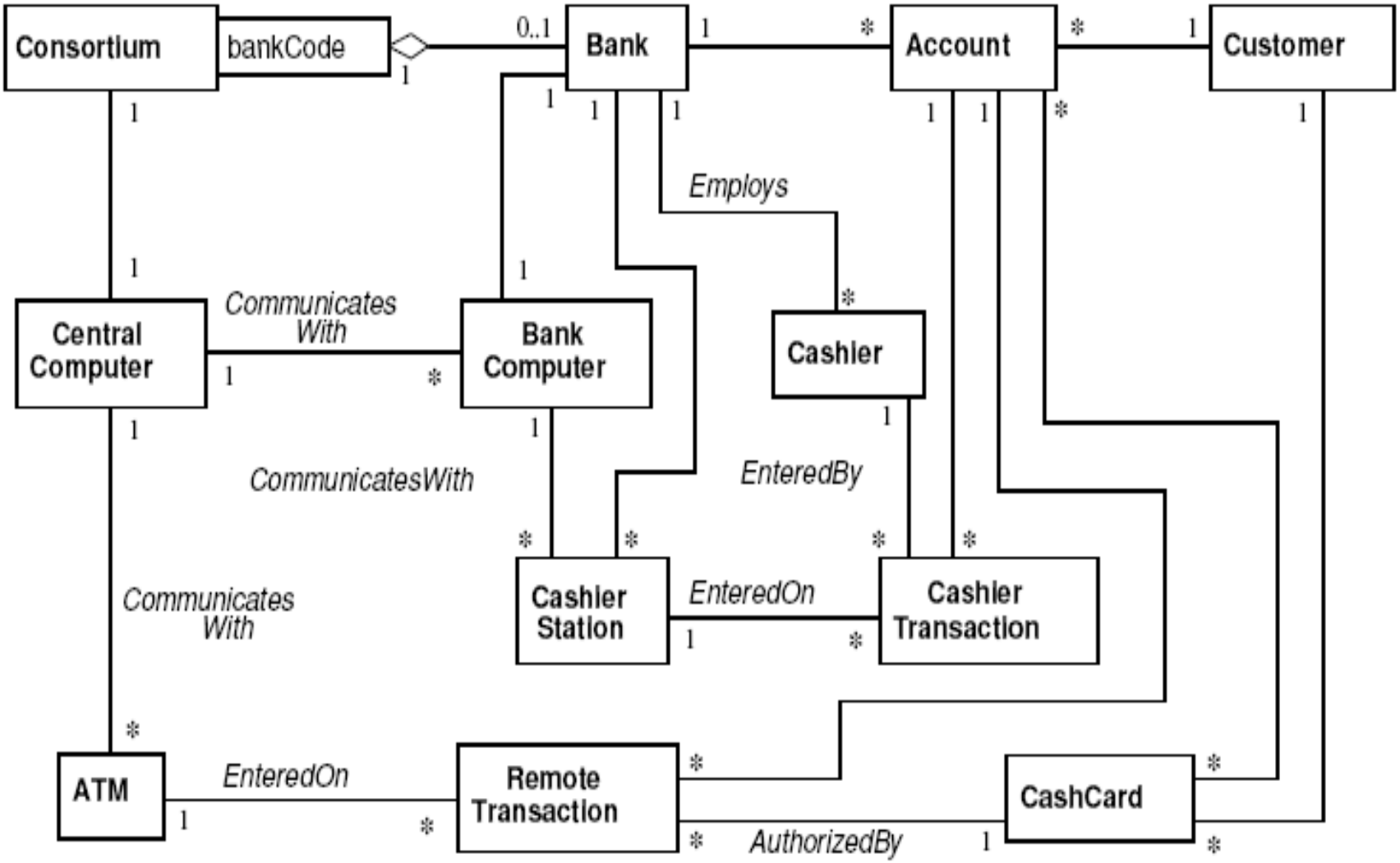
- *Misnamed Association:* Name are important to understanding and should be chosen with care.
  - Ex. *Bank computer maintain accounts*. Rephrase as *Bank hold account*.
- *Association End name*: Add association end name where appropriate.
  - Ex. *Person manages person*. It would be appropriate to give end names *boss* and *worker*

- *Qualified Associations*: Most names are not globally unique. So context combines with the name to uniquely identify the object.
  - Ex. Company name unique within one state but may be duplicated in other state.
  - So combining State + Company name will uniquely identify company.
  - Ex. bankCode differentiate bank in a consortium.

- *Multiplicity:* Don't put much effort as it is often changes during analysis.
- *Missing Association:* Add any missing association that are found during analysis.
  - *Transaction entered on cashier station, Customers have accounts and transaction authorized by cash cards*.
  - To perform above, we need to introduce relationship from Cashier to cashier station. So association *Cashier authorized on cashier station* needed

- *Aggregation*: it is specially for mechanical parts or bills of material.
  - Don't spend much time trying to defer between association and aggregation.
  - Ex. *Bank* is part of *Consortium* and indicate the relationship with *aggregation*.
- Now combining all things together, class diagram prepare.

Object-Oriented Analysis and Design
Process overview



6. Finding Attributes:

- Attributes are data properties of objects like colour, weight etc.
- Attributes usually correspond to nouns followed by possessive phrases, such as “the color of the car”
- Attributes are less likely to be fully described in problem statement.
- Only consider attributes *directly relevant to application*. Get important attributes then add details to it.

- Avoid derived attributes.
  - EX. Age is derived from birthdate and currentTime
- Looks for attributes on associations.
  - Ex. Workfor association attribute can be salary, title etc.

7. Keeping the Right Attributes

- Eliminate unnecessary and incorrect attributed with the following criteria:
- Objects: if element is important rather than just its value, then it is an object.
  - Ex. *Boss* refers to a class and *Salary* is an attributes.
- Name: Name often refer as Qualifier rather than attributes.
- Name is an attribute when its use does not depend on context,

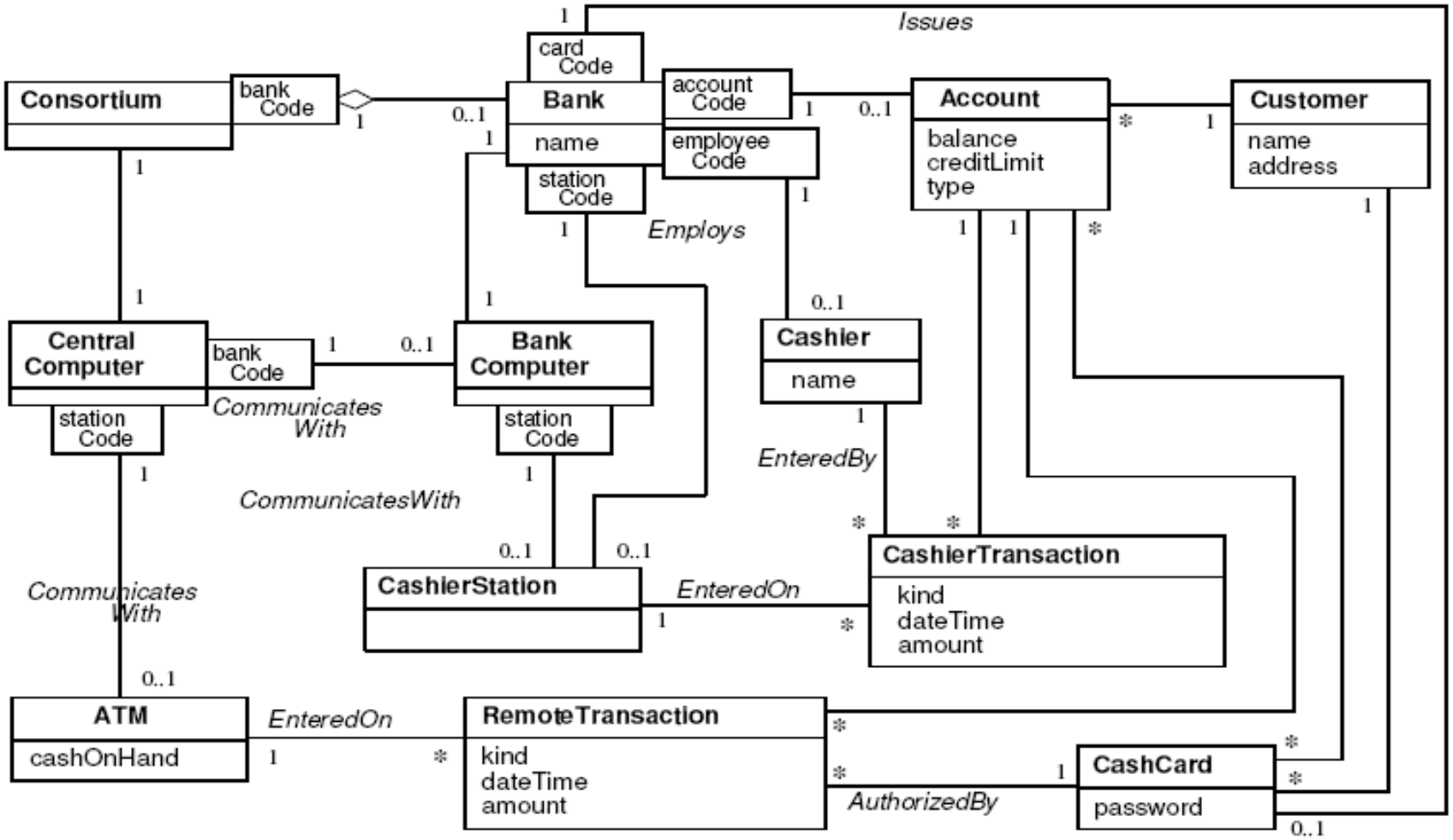
  - Ex. Names of person are not unique therefore its attributes.
- Identifiers: it mean referencing objects used for some work.
  - Ex. ATM transaction always generate *Transaction ID* for each operation. So you can count *Transaction ID* as attributes.
- Attributes on Association:
  - If value require the presence of link then attributes of the association should derived.

  - Ex. Workfor association attribute can be salary, title etc

- Internal Values: if an attributes describes internal state of an object then eliminate it.
- Fine Detail: Omit minor attributes that are unlikely to affect most operation.
- Discordant attributes :- an attribute that seems completely different from and unrelated to all other attributes then remove it.

- Boolean Attributes: Convert Boolean attributes into enumeration.

Object-Oriented Analysis and Design
Process overview



8. Refining with Inheritance

- Next step, to share common structure.
- Apply in two directions:
  - Bottom Up
    - By Generalizing common aspects of existing classes into a superclass.
  - Top Down
    - By Specializing existing classes into multiple classes

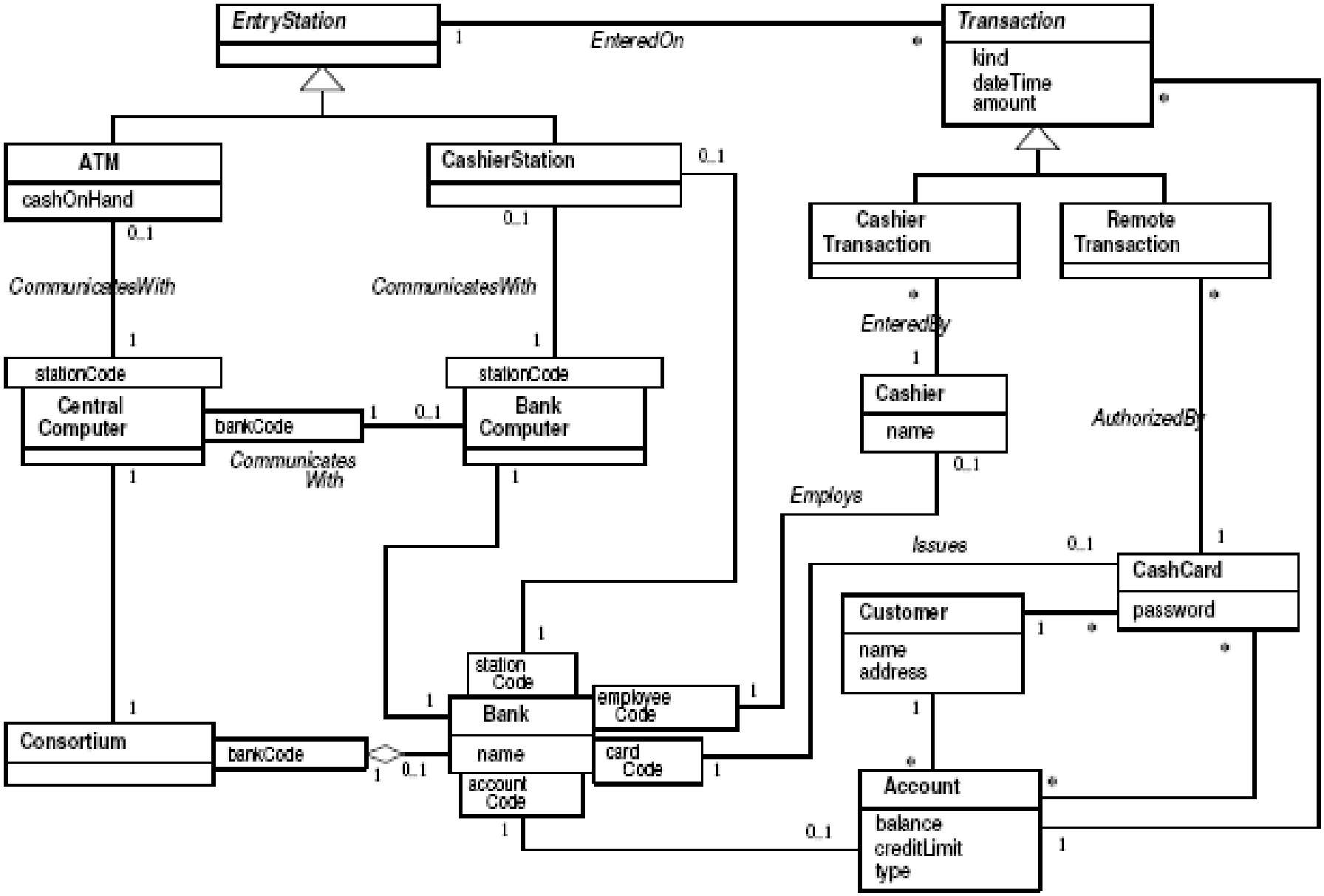
- **Bottom Up Generalization:**
  - Searching for classes(from bottom up) with similar attributes, associations and operation.
  - For each generalization, define a superclass to share common features.
  - May have to slightly redefine some attributes or classes to fit in. But don't push too hard it create wrong generalization.
  - Ex. *RemoteTransactiona* and *CashierTransaction* are similar and can be generalized by *Transaction*.

- Top-Down Specialization:
  - It main derived from application domain itself.
  - Look for noun phrases composed of adjectives:
    - Fixed menu, sliding menu and text menu.
  - Avoid excessive refinement.

- Generalization Vs. enumeration :-
  - Generalization is all about common structure
  - Enumeration is all about list of values.
  - Ex. *CurrentAccount* and *SavingAccount* share common structure but it does not affect behavior within the ATM application. So *Type* can introduces as attributes of account and enumerate it.
- Multiple Inheritance: if require then apply because it increasing both conceptual and implementation complexity

- **Similar association:**
  - when the same association name appears more than once with the same meaning, try to generalize the associated class.
  - Ex. EntryStation generalizes CashierStation and ATM.
- Adjusting inheritance Level
  - Assign attributes and association to specific classes in the class hierarchy.
  - You may need some adjustment to get everything right.

Object-Oriented Analysis and Design
Process overview



9.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.
- Make sure you have not overlooked any associations.

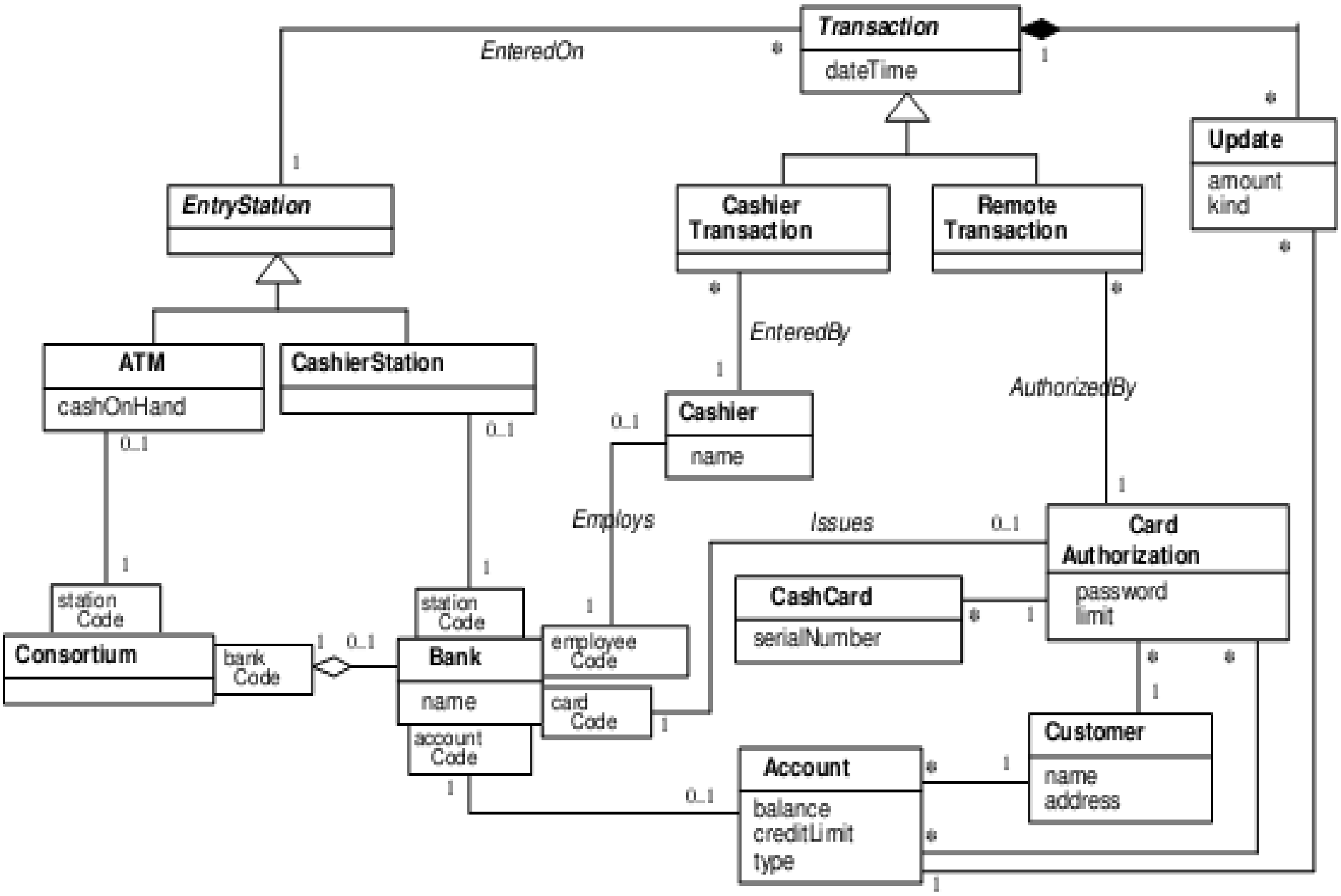
10. Iterating a Class model

- A class model is rarely correct after a single pass.
- If you find any deficiency, go back to an earlier stage if necessary to correct it. So iterate and refine the model.
- There are some sing of missing classes.
  - Asymmetries in association and generalization.
  - Disparate attributes and operation on a class.
  - Difficulty in generalizing cleanly.

  - Duplicate association with same name and purpose.
  - A role that substantially shapes the semantics of a class. Ex. It mean converting association into a class.
- Look out for missing associations
  - Missing access paths for operations
  - Lack of attributes, operations and association on a class.
  - Redundant information:
- Adjust the placement of attributes and associations

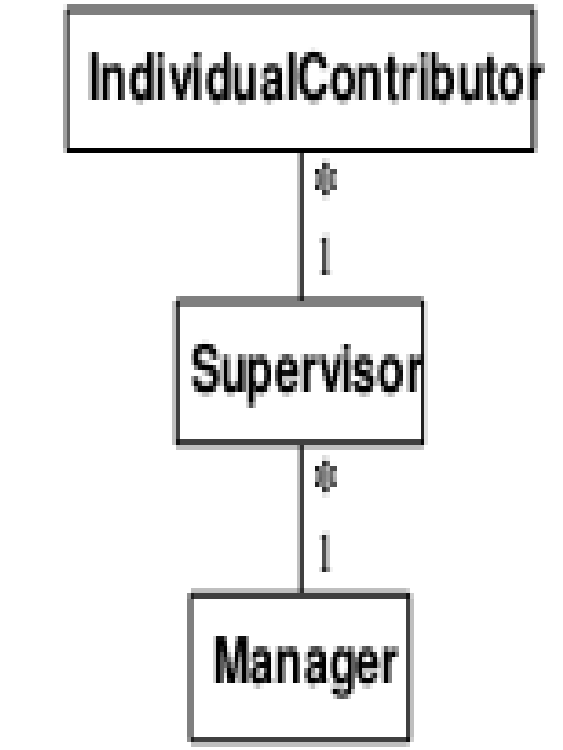
Object-Oriented Analysis and Design

Process overview

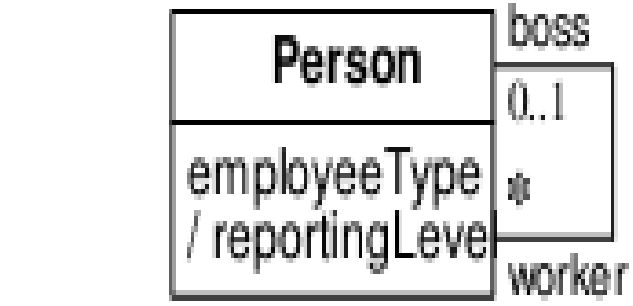


11.Shifting the level of abstraction

- Abstraction makes a model more complex but can increase flexibility and reduce the number of classes.
- In case of abstraction, we need to think in terms of pattern.
- A pattern distills the knowledge of experts and provide a proven solutions to a general problem.
  - Ex. Management hierarchy.



Original model



Improved model that is more abstract

12.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.
- When you place classes and association in a package, you are making semantic statement.

- Therefore, package might be:
  - Tellers – Cashier, Entry Station, Cashier Station, ATM
  - Accounts – Account, cash card, card authorization, customer, transactions, update, cashier transaction, remote transaction.
  - Bank- consortium, bank
- Each package can add details to it.

- 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

1. Identifying Classes with states

- Study list of domain classes.
- Look for classes that can be characterized by a *progressive history or represent cyclic behavior*.
- Identify significant states in the life cycle of an Object. Not every state occurs in every cycle.
  - ATM Example, *Account* is appropriate behavior for ATM. Life cycle of *Account* is progressive and cycling to and from problem states.

Finding States

- List the state for each class.
- Characterized the object by their
  - Attributes Values
  - Associations that may participants
  - Attributes and association that are meaningful in certain states only.
- Avoid names that indicate how the state came.

- By looking at events and considering transitions among states, missing states will become clear.
- Ex. Some states for Account
  - Normal (Normal access)
  - Closed (Closed by customer)
  - Overdrawn (withdrawal exceeds the balance)
  - Suspended (blocked for some reason)

Finding Events

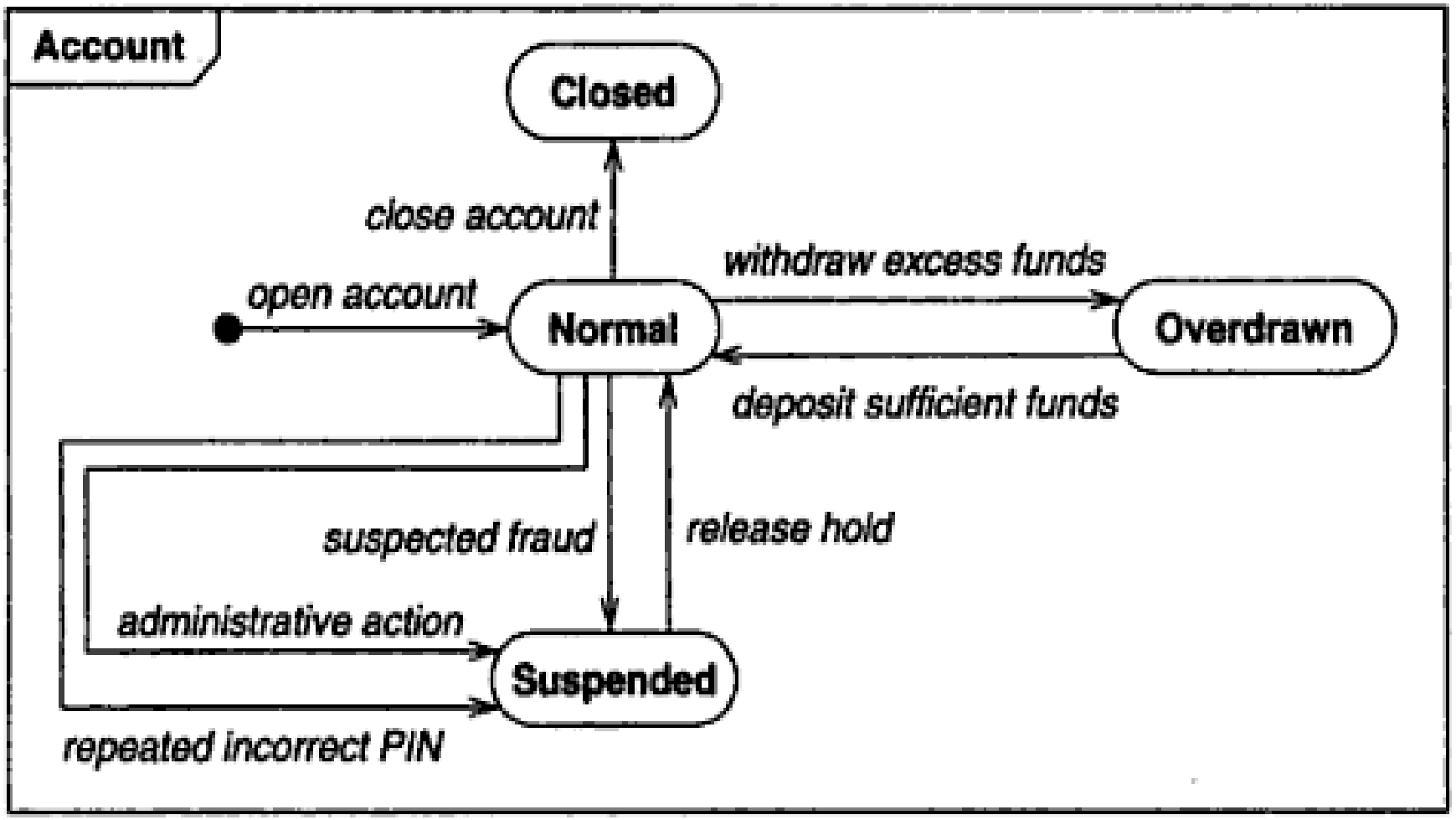
- Now find events that cause transitions among states.
- Think about stimuli (input) that cause a state to change.
- Find other events that takes object into a specific states
  - Ex. Pickup receiver on telephone, it enters into *Dialing* State.
  - But many telephone has pushbuttons that invoke specific functions.

- So there are some additional event that occur within a state and don't cause a transitions.
- For Domain State model, make *focus on events that cause transition among states.*
- Ex. Event includes: *close account, withdraw excess funds, repeated incorrect PIN, suspected fraud and Administrative action.*

Building State diagrams

- Determine for which state, each event applies.
- Add transitions to show the change in state caused by the occurrence of an event when an object is in particular state.
- Once you have specified the transitions, check does it represent an error or not? If yes then add transitions to error state.

Object-Oriented Analysis and Design
Process overview



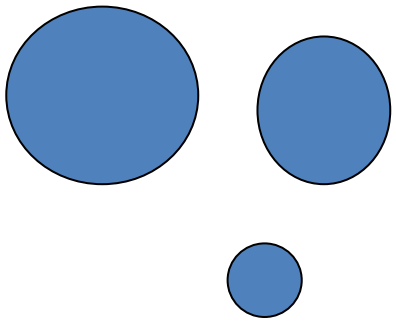
Evaluating State Diagram

- Examine each state model. Are all states connected?
- Path from initial state to the final state?
- Are the expected variations represent it?
- Are there any dead states that terminate the cycle?
- Find missing path and states from it.
- When complete, it should indicate life cycle of the class.

Characteristics of Objects - Classification

- Each objects is said to be *instance of its class.*
- Objects has its own value for each attributes but shares the attributes names & operations.

Example : Class Name : Circle



Attributes : radius, center

Operation : setCenter(), setRadius()

Characteristics of Objects - Inheritance

- Definition: Sharing of attributes & operations (features) among classes based on hierarchical relationship.
  - A ***superclass*** has general information that ***subclass*** refine and elaborate.
  - Each *subclass* incorporates all the features of its *superclass* and adds its own features.
  - In other words, defining new classes from the existing one.

Characteristics of Objects - Inheritance

Note: *subclasses* need not repeat the features of the *superclass*.

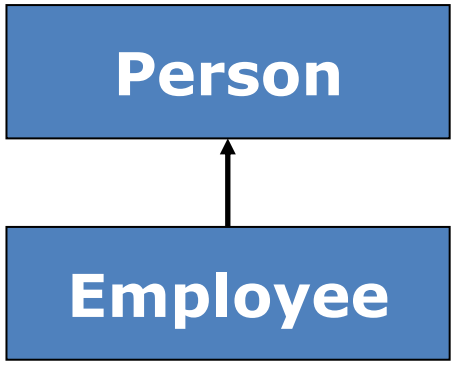
Advantage: common features of several classes into a superclass can reduce repetition within design and programs.

Characteristics of Objects – Example of Inheritance

Inheritance is implied by is-a or kind-of relationship.

```
class Person {
   String name;
   String age;
   void birthday () {
      age = age + 1;
   }
}
```

```
class Employee
        extends Person {
   double salary;
   void pay () { ...}
}
```



Every Employee has a name, age, and birthday method *as well as* a salary and a pay method.

Characteristics of Objects – Polymorphism

- Definition
  - Same *operation* may behave differently for different classes.
  - In simple words, " One name multiple form"
  - Here operation mean – it's a procedure or transformation that an object perform or is subject to.
  - For example , Class name is POLYGON
    - Attributes - vertices, border color, fill color.
    - Operations – Draw, erase, fill
  - An implementation of an operation by a specific class is called *Method*

What is OO Development?

- OO Development refers to the software life cycle.

  i.e. Planning, Analysis, Design & Implementation

Why OO Development?

- In essence of OO development is the identification & organization of application concepts, rather than in a programming language.

OO Dev – Modeling Concept, Not implementation

- Earlier, OO community focused on implementation part rather than analysis and design.
- It focuses excessively on implementation mechanisms rather than the underlying thought process that support.
- An OO development approach encourage software developers *to work & thinks in* terms of the application throughout software life cycle.

OO Dev – Modeling Concept, Not implementation

- OO development is a *conceptual process independent* of a programming language until the final stage.
- OO development is *fundamentally a way of thinking* & not a programming technique.
- It can serve as a medium for specification, analysis, documentation & interfacing as well as for programming

OO Dev – Modeling Concept, Not implementation

- OO Development & Graphical notation represents OO concept.
- OO process consists of building a model of an application & then adding details to it during design.
- Same notation is used from
  - Analysis → Design → Implementation.
- So information is not lost or translated into the next stage [Reusability].

OO Development Stages

- System Conception
- Analysis
- System Design
- Class Design
- Implementation

System Conception

- System Conception means origin of the system.
- S/W development begins with
  - business analyst or users conceiving an application & formulating tentative requirement.

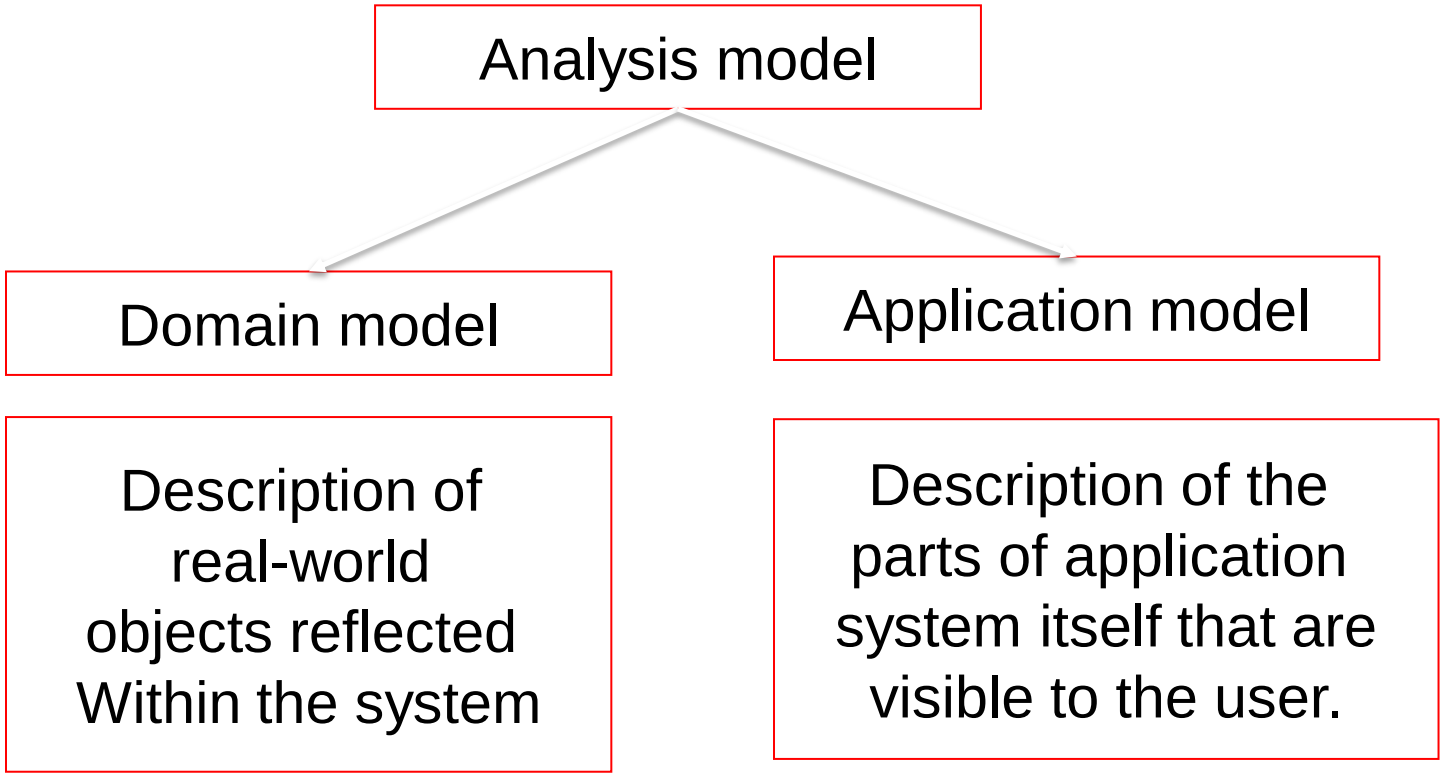
OO Development Stages

- System Conception
- Analysis
- System Design
- Class Design
- Implementation

Analysis

- Task of Analyst
  - Must work with the requester (client) to understand the problem, because problem statement are rarely complete or correct.
  - To design the Analysis model which demonstrates *what* the desired system must do, not *how* it will be done.
  - Analyst is not concerned about implementation decision.

Analysis



Ex. Bank account is domain model.
Application model includes Saving accounts, current account, demat account etc.

OO Development Stages

- System Conception
- Analysis
- System Design
- Class Design
- Implementation

System Design

- Task of system designer

  - must decide what performance characteristics to optimize.

  - choose strategy to attack the problem.

  - making tentative resource allocation.

- Ex. Designer might decide to *change* the window screen for *fast & smooth* working, even when windows are *moved or erased*.

OO Development Stages

- System Conception
- Analysis
- System Design
- Class Design
- Implementation

Class Design

- Task of class designer

  - add details to analysis model

  - They determine *data structures & algorithm* for each of the operation of window class.

  - They elaborate both domain & application objects using same OO concept & notation.

OO Development Stages

- System Conception
- Analysis
- System Design
- Class Design
- Implementation

Implementation

Task of Implementers :-

- Translates the classes & relationships developed during class design into particular programming language, database or hardware
- During implementation, follow good software engineering practice so that traceability to the design is apparent (i.e. clear).

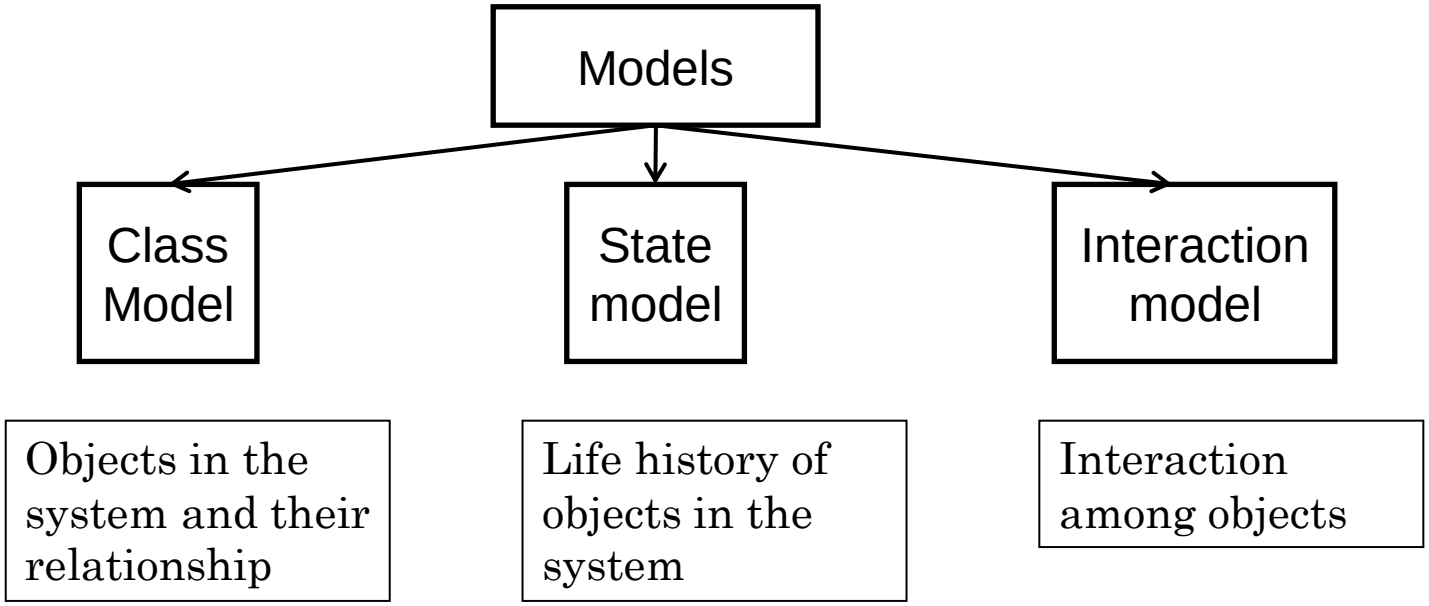
Summary of OO Development

- OO concepts apply throughout the System Development Life Cycle (SDLC)
  - i.e. Analysis → design → implementation
- Use same classes from *stage to stage without a change* of notation.
- Some classes *are not part of analysis* but are introduced during design or implementation.

Ex. Data structure such trees, hash table & linked list are not visible to users at the time of analysis.

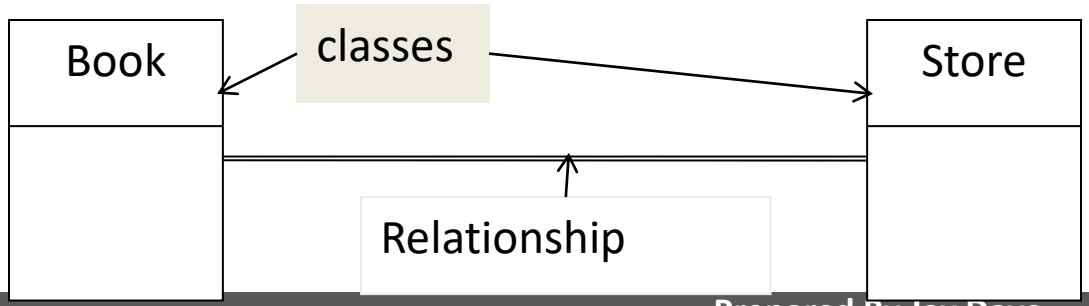
  - but designers introduce them to support particular algorithms.

Three Software Models



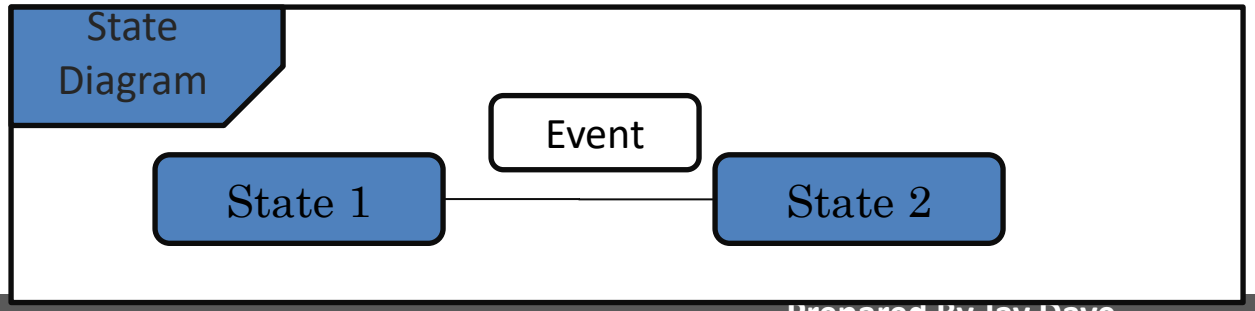
Class model

- Describe the static structure of the *objects in a system & their relationship*
- It define the context for software development.
- Class model contains class diagram to express it.
- A class diagram is graph phase.
  - Nodes are classes.
  - Arcs are relationship among classes.



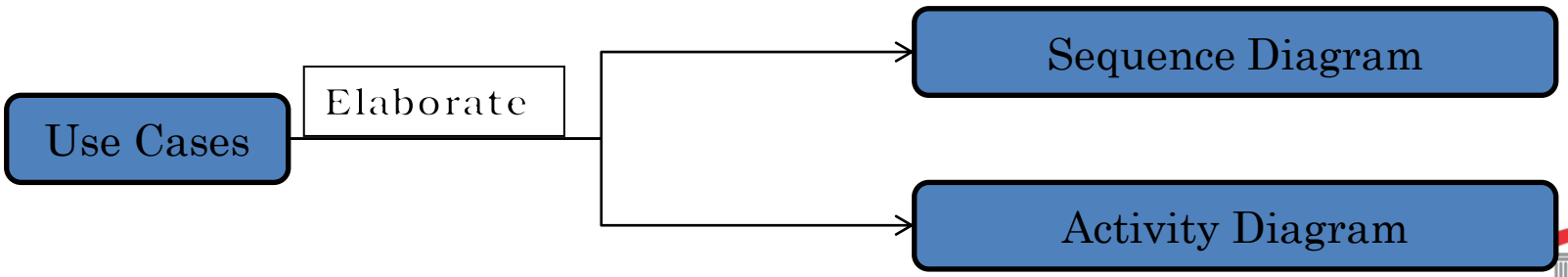
State Model

- Describe aspect of an object that change over time.
- State model specifies & implement control with state diagram
- A state diagram is a graph whose
  - Nodes are state
  - Arcs are transitions between state caused by events



Interaction model

- How the objects in a system co-operate to achieve broader results.
- Interaction model start with
  - Use case that are elaborate into with sequence and activity diagram.



Interaction Model

- Use cases:
  - Focus on functioning of system
  - Simple mean, ***what a system does for users***
- Sequence diagram
  - Shows the object that interact
  - Time sequence of their interactions
- Activity diagram
  - Elaborate important processing steps.
  - Activity diagrams can be used to describe the business and operational step-by-step workflows of components in a system

OO Modeling History

- **Grady Booch**
- **James Rumbaugh**
- **Ivar Jacobson**

  Are the man behind Invention of OO Modeling Technique.
- Object Modeling Technique (OMT) concept evolved in 1991.

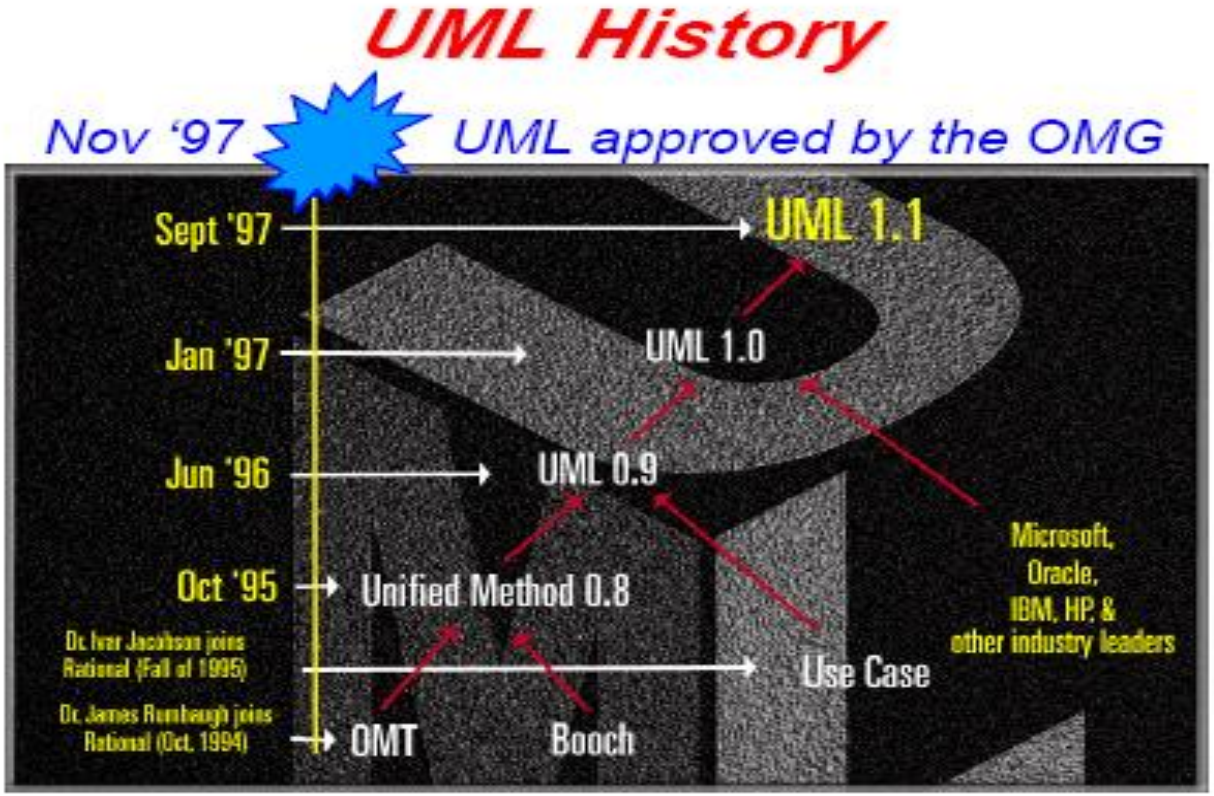
OO modeling History

- In 1994 James Rumbaugh joined Rational (now the part of IBM) in 1994 & began working with Grady Booch on UML Notations.
- In 1995, Ivar Jacobson also joined Rational & added his concept to the unification work.

OO modeling History

- In 1996 the Object Management Group issued a request for proposals for standard OO modeling notation.
- Later Rational led the final proposal team, with Booch, Rumbaugh & Jacobson deeply involved.

OO modeling History



OO THEME

- OO themes are not unique to OO systems, they are particularly well supported.
  - Abstraction
  - Encapsulation
  - Combining data and behavior
  - Sharing
  - Emphasis on the essence of an object
  - Synergy

OO THEME

- **Abstraction**:
  - Focus on *essential aspects of an application* while ignoring details.
    - i.e. focusing on *what an object is and does*, before deciding how to implement it.
  - Use of Abstraction:
    - Freedom to make decisions as long as possible by avoiding premature commitment to details.
  - Ability to abstract is probably the most important skill required for OO development.

OO THEME

- **Encapsulations** (Information Hiding):
  - It is separates the external (accessible to objects) aspects of an objects from the internal (hidden from other objects) implementation details.
  - It prevents portions of a program from becoming so interdependent that a small change has massive ripple effects.
  - For ex. You may want to change the objects to
    - Improve performance, Fix a bug, Consolidate code, Support porting

OO THEME

- Encapsulation is not unique to OO language but ability to combine data structure & behavior in a single entity makes encapsulation cleaner & powerful.

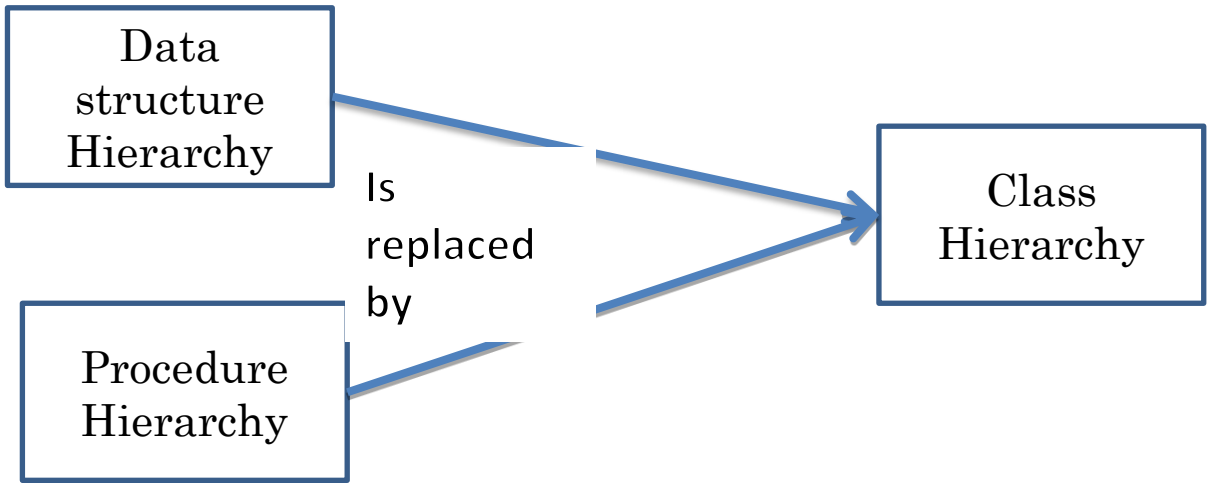
OO THEME

Combining data & Behavior

- In non-OO code, to display the content of a window must distinguish the type of each figure, such as circle, polygon etc & call the appropriate procedure to display it.
- In OO code, program invoke the “Draw” operations on each figure and each object implicitly decide which procedure to use, based on its class.
- So caller of an operation need not consider how many implementation exist.

OO THEME

- So, maintenance is easier, because the calling code need not be modified when a new class is added.
- In an OO system, the data structure hierarchy matches the operation inheritance hierarchy.



OO THEME

Sharing

- OO technique promote sharing at different levels.
- Sharing via inheritance is one of the main advantage of OO language.
- OO development not only lets you share information within an application, but also offers the aspects of reusing designs & code on future projects.
- OO provides the tools to build libraries (or collection) of reusable component.

OO THEME

Emphasis on the essence of an object:

- In OO technology, focus is on what an objects is rather than how it is used.
- Use of an object depend on the details of application and often change during development.
- OO development greater emphasis on data structure & lesser emphasis on procedure structure.

OO THEME

Synergy :

- OO concepts can be used in isolation but together they complement each other synergistically.