PART B
Unit: 5 APPLICATION ANALYSIS, SYSTEM DESIGN

7 Hours

Syllabus:

- ***Application Analysis: Application interaction model; Application class model; Application state model;***
- ***Adding operations. Overview of system design; Estimating performance;***
- ***Making a reuse plan; Breaking a system in to sub-systems;***
- ***Identifying concurrency; Allocation of sub-systems; Management of data***
- ***storage; Handling global resources; Choosing a software control strategy;***
- ***Handling boundary conditions; Setting the trade-off priorities; Common***
- ***Architectural styles; Architecture of the ATM system as the example.***

Application Analysis

Application Interaction Model - steps to construct model

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

1. Determine the system boundary

- Determine what the system includes.
- What should be omitted?
- Treat the system as a black box.
- ATM example:
  - For this chapter,
    - Focus on ATM behavior and ignore cashier details.

2. Find actors

- The external objects that interact directly with the system.
- They are not under control of the application.
- Not individuals but archetypical behavior.
- ATM Example:
  - Customer, Bank, Consortium

3. Find use cases

- For each actor, list the different ways in which the actor uses the system.
- Try to keep all of the uses cases at a similar level of detail.
  - apply for loan
  - withdraw the cash from savings account
  - make withdrawal

Use Case for the ATM

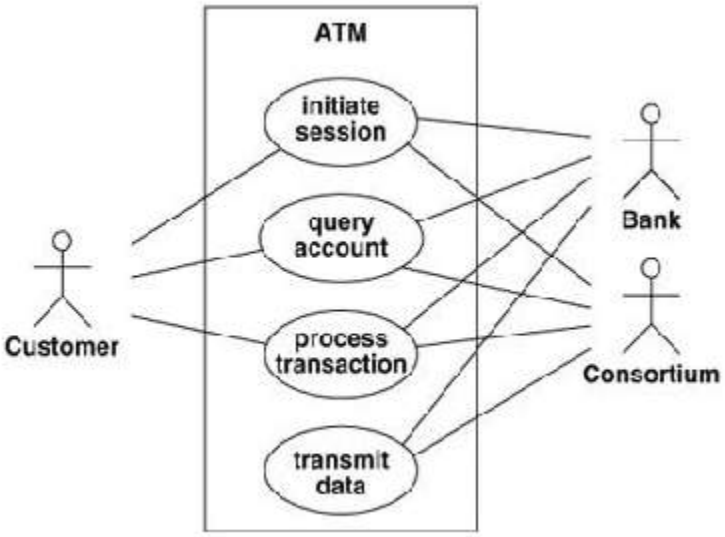


Figure 13.1 Use case diagram for the ATM. Use cases partition the

- **Initial session**
  - The ATM establishes the *identity* of the user and makes available *a list of accounts and actions.*
- **Query account**
  - The system provides general data for an account, such as the *current balance, date of last transaction, and date of mailing for last statement.*

- **Process transaction**
  - The ATM system performs an action that affects an account's balance, such as *deposit, withdraw, and transfer*. The ATM ensures that all completed transactions are ultimately written to the bank's database.
- **Transmit data**
  - The ATM uses the consortium's facilities to communicate with the appropriate bank computer.

4. Find initial and final events

Finding the initial and final events for each use case

To understand the **behavior** clearly of system

Execution sequences that cover each use case

Initial events may be

a. A request for the service that the use case provides
b. An occurrence that triggers a chain of activity

ATM example

- Initial session
  - Initial event
    - The customer's insertion of a cash card.
  - final event
    - The system keeps the cash card, or
    - The system returns the cash card.

  ATM example
- Query account
  - Initial event
    - A customer's request for account data.
  - final event
    - The system's delivery of account data to the customer.

  ATM example
- Process transaction
  - Initial event
    - The customer's initiation of a transaction.
  - final event
    - Committing or
    - Aborting the transaction

  ATM example
- Transmit data
  - Initial event
    - Triggered by a customer's request for account data, or
    - Recovery from a network, power, or another kind of failure.
  - final event
    - Successful transmission of data.

5. Prepare normal scenarios

For each use case, prepare one or more *typical dialogs*.

A scenario is a *sequence of events* among a set of ***interacting*** *objects*.
Sometimes the problem statement describes the full interaction sequence

6. Normal ATM scenarios

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.

7. Add variation and exception scenarios

Special cases

Omitted input E.g., maximum values, minimum value

Error cases

E.g. Invalid values, failures to respond

Other cases

E.g. Help requests, status queries

ATM example

- Variations and exceptions:
  - The ATM can't read the card.
  - The card has expired.
  - The ATM times out waiting for a response.

- The amount is invalid.
- The machine is out of cash or paper.
- The communication lines are down
- The transaction is rejected because of suspicious pattern of card usage.

8. Find external events

- The external events include
  - All inputs,
  - decisions,
  - interrupts, and
  - Interactions to or from users or external devices.
- An event can trigger effects for a target object.
- Use scenarios for normal events
  Sequence diagram
- Prepare a sequence diagram for each scenario.
- The sequence diagram captures the dialog and interplay between actors.
  The sequence diagram clearly shows the sender and receiver of each event
  ATM Example

Sequence diagram of the process transaction

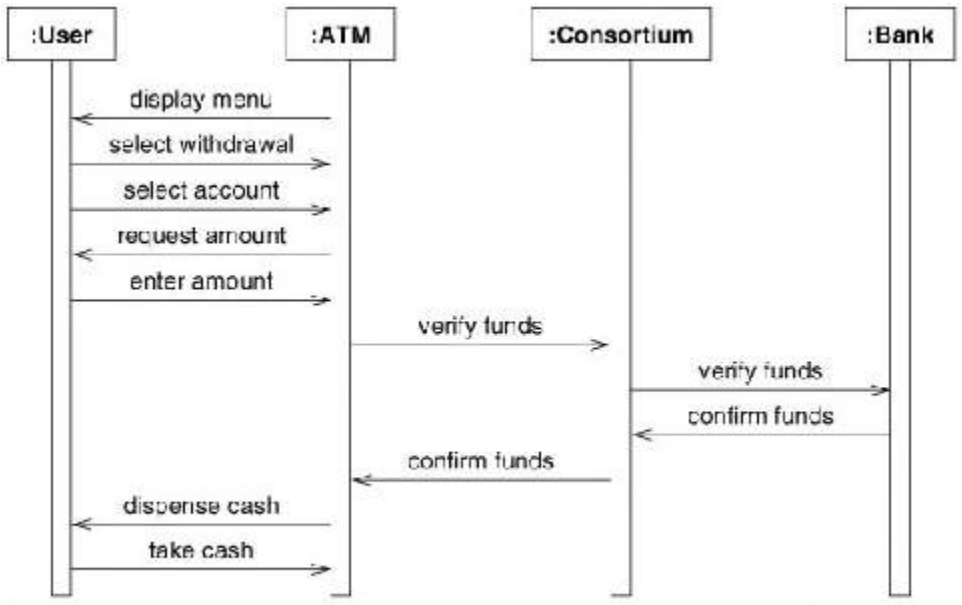


Figure 13.3 Sequence diagram for the *process transaction* scenario. A sequence diagram clearly shows the sender and receiver of each event.

- Events for the ATM case study

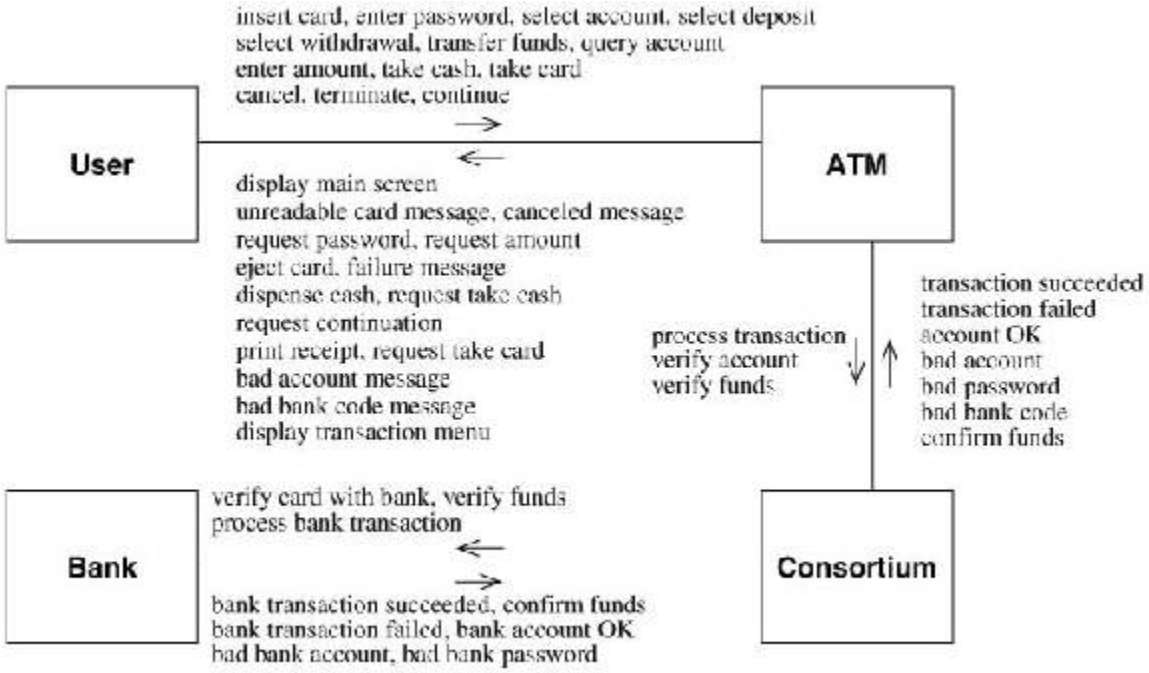


Figure 13.4 Events for the ATM case study. Tally the events in the scenarios and note the classes that send and receive each event.

9. Activity Diagram

- Activity diagram shows behaviors like alternatives and decisions.
- Prepare activity diagrams for complex use cases.
- Appropriate to document business logic during analysis
- Do not use activity diagram as an excuse to begin implementation.

ATM Example

Activity diagram for card verification

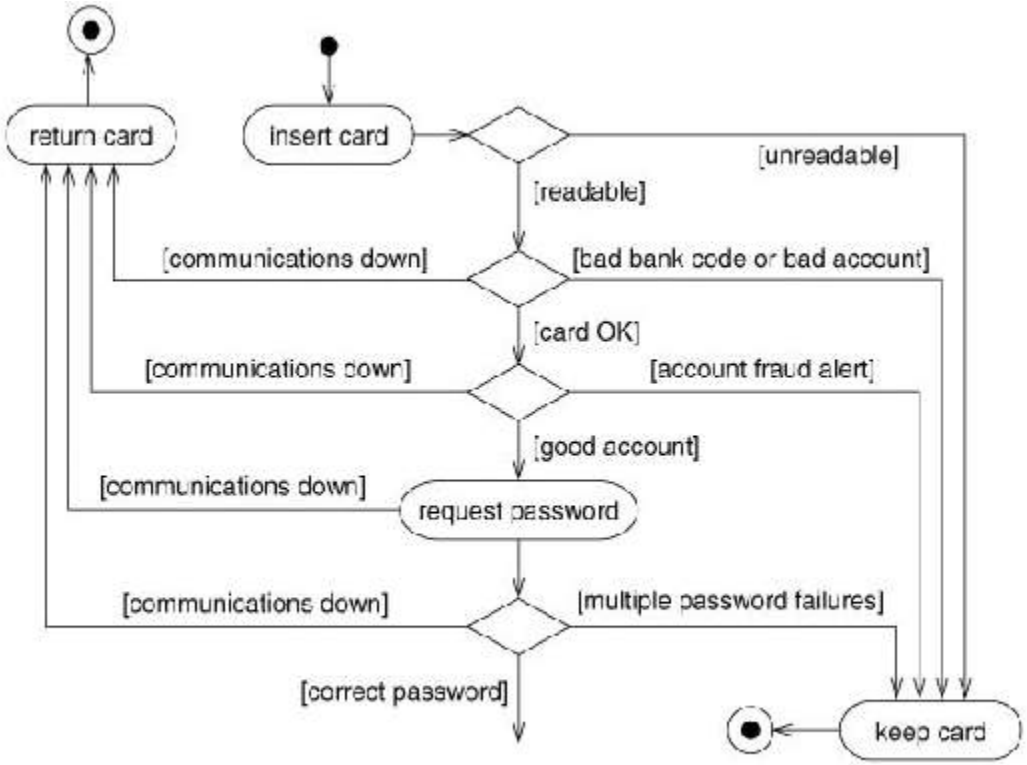


Figure 13.5 Activity diagram for card verification. You can use activity diagrams to document business logic but do not use them as an excuse to begin premature implementation.

10. Organize actors and use cases

- Organize use cases with relationships
  - Include, extend, and generalization
- Organize actors with generalization.

ATM Example

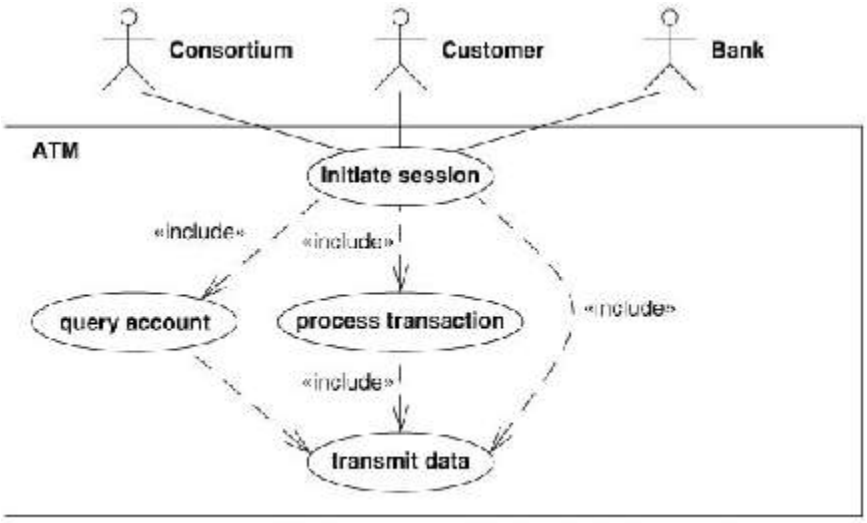
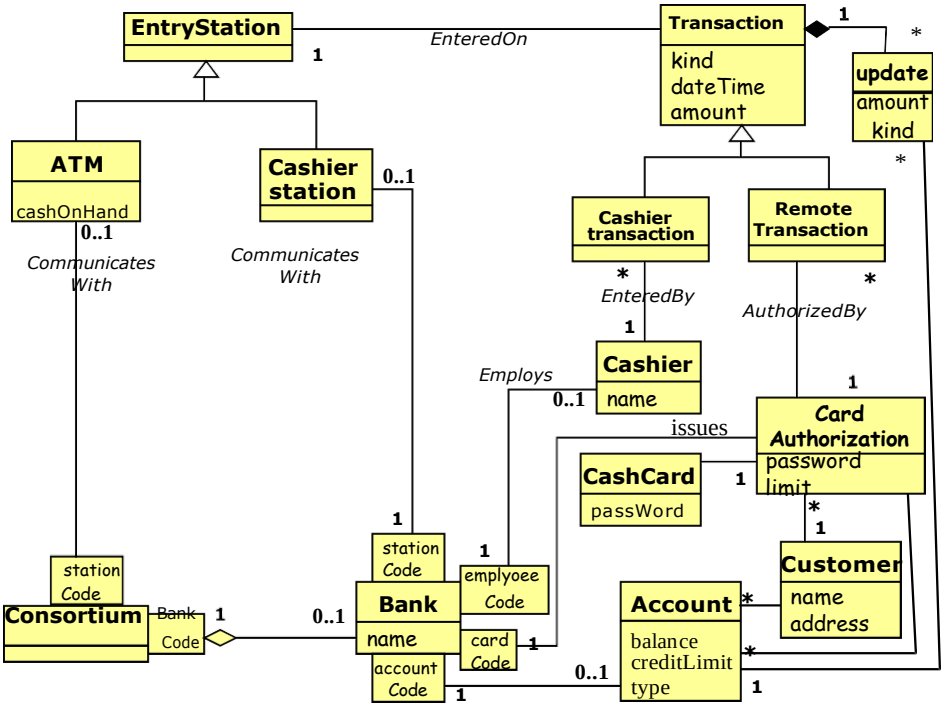


Figure 13.6 **Organizing use cases.** Once the basic use cases are identified, you can organize them with relationships.

11. Checking Against the Domain Class Model

- The application and domain models should be mostly consistent.
- The actors, use cases, and scenarios are all based on classes and concepts from the domain model.
- Examine the scenarios and make sure that the domain model has all the necessary data.
- Make sure that the domain model covers all event parameters.



Application Class Model

- Application classes define the application itself, rather than the real-world objects that the application acts on
- Most application classes are *computer-oriented* and define the way that users perceive the applications

Application Class Model – steps

1. Specify user interfaces
2. Define boundary classes
3. Determine controllers
4. Check against the interaction model

1. Specify user interfaces

User interface

a. Is an object or group of objects
b. Provide user a way to access system's
   i. domain objects,
   ii. commands, and
   iii. Application options.

Try to determine the commands that the user can perform.

A command is a large-scale request for a service,

c. E.g.
   i. Make a flight reservation
   ii. Find matches for a phrase in a database

Decoupling application logic from the user interface.

ATM example - The details are not important at this point.

- The important thing is the information exchanged.

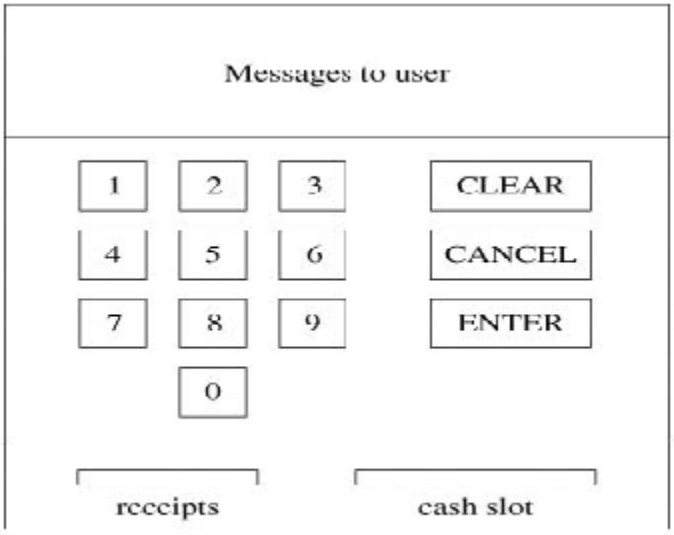


Figure 13.7 Format of ATM Interface. Sometimes a sample interface can help you visualize the operation of an application.

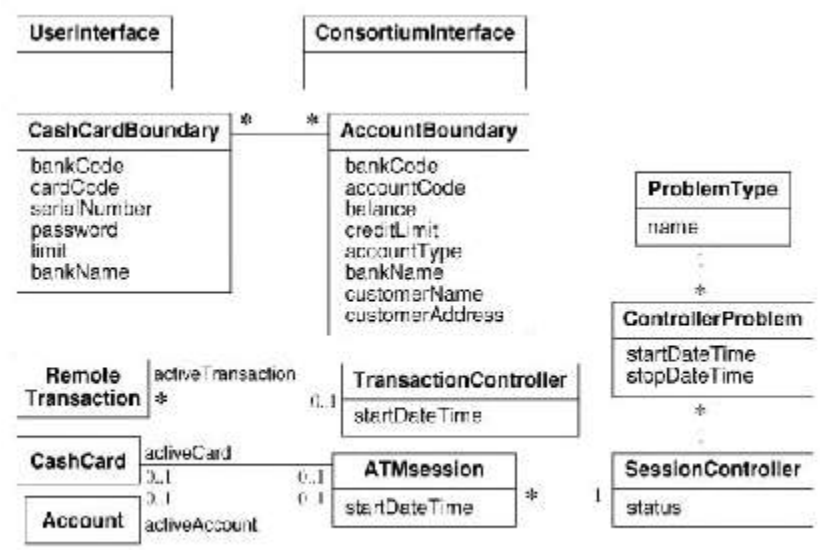


Figure 13.8 ATM application class model. Application classes augment the domain classes and are necessary for development.

2. Defining Boundary Classes

- A boundary class
  - Is an area for communications between a system and external source.
  - Converts information for transmission to and from the internal system.
- ATM example
    - *CashCardBoundary*
    - *AccountBoundary*
  - Between the ATM and the consortium

ATM Example

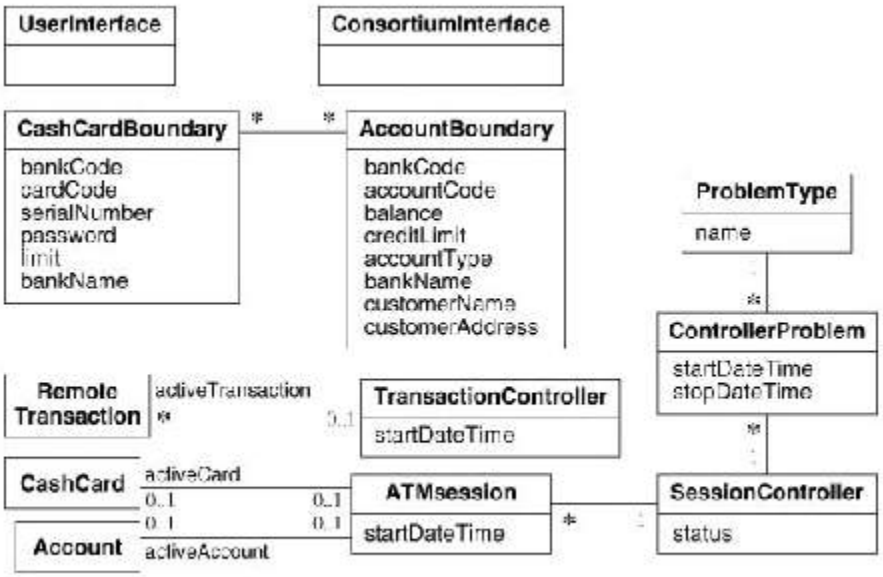


Figure 13.6 ATM application class model. Application classes augment the domain classes and are necessary for development.

3. Determining Controllers

- **Controller** is an active object that manages control within an application.
- Controller
  - *Receives* signals from the outside world or
  - Receives signals from objects within the system,
  - *Reacts* to them,
  - *Invokes* operation on the objects in the system, and
  - *Sends* signals to the outside world.

ATM Example

- There are two controllers
  - The outer loop verifies customers and accounts.
  - The inner loop services transactions.

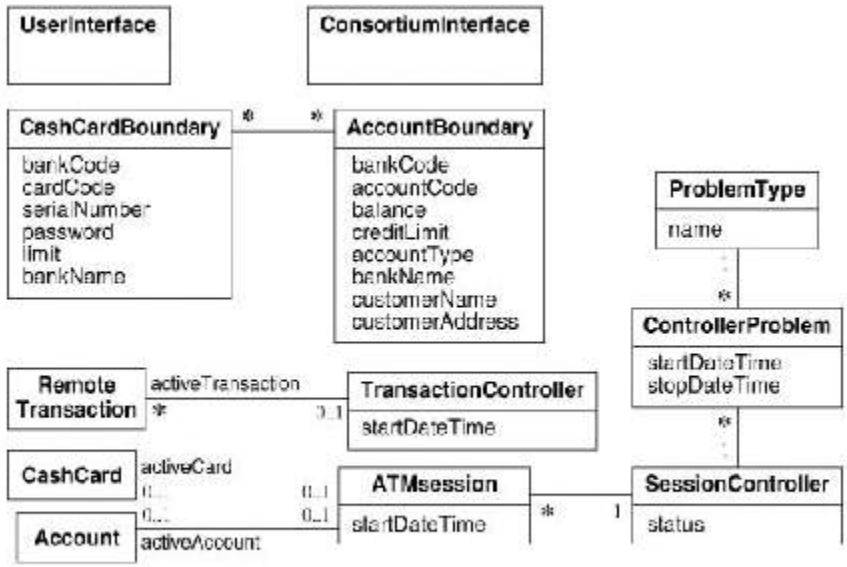
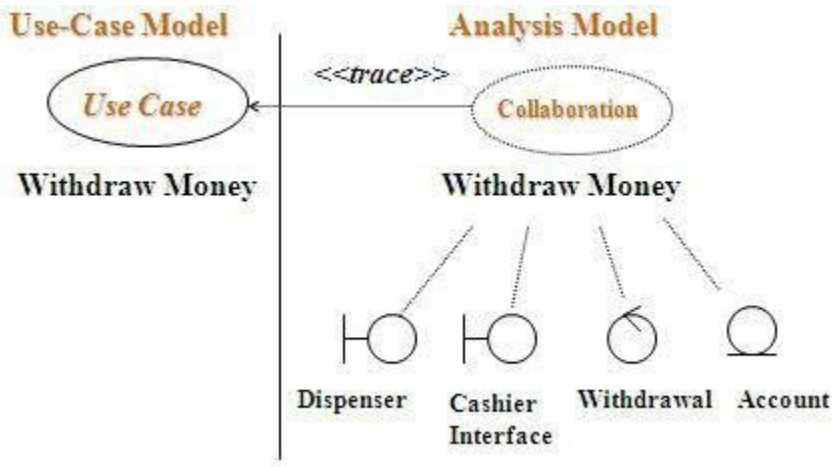


Figure 13.8 ATM application class model. Application classes augment the domain classes and are necessary for development.

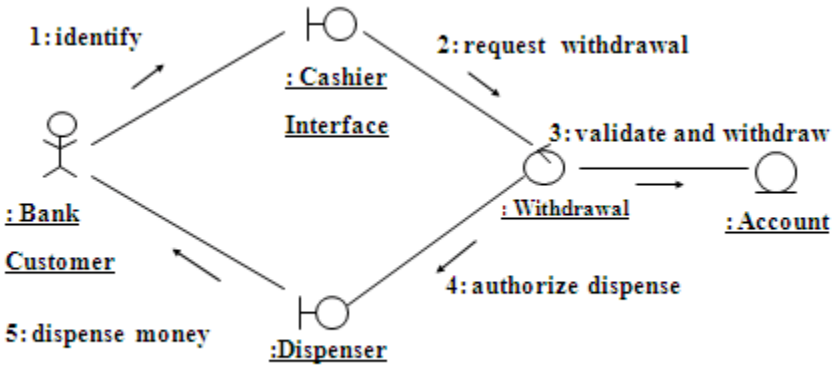
Analysis Stereotypes

- <<boundary>> classes in general are used to model interaction between the system and its actors.
- <<entity>> classes in general are used to model information that is long-lived and often persistent.
- <<control>> classes are generally used to represent coordination, sequencing, transactions, and control of other objects. And it is often used to encapsulate control related to a specific use case.

The Realization of a Use Case in the Analysis Model

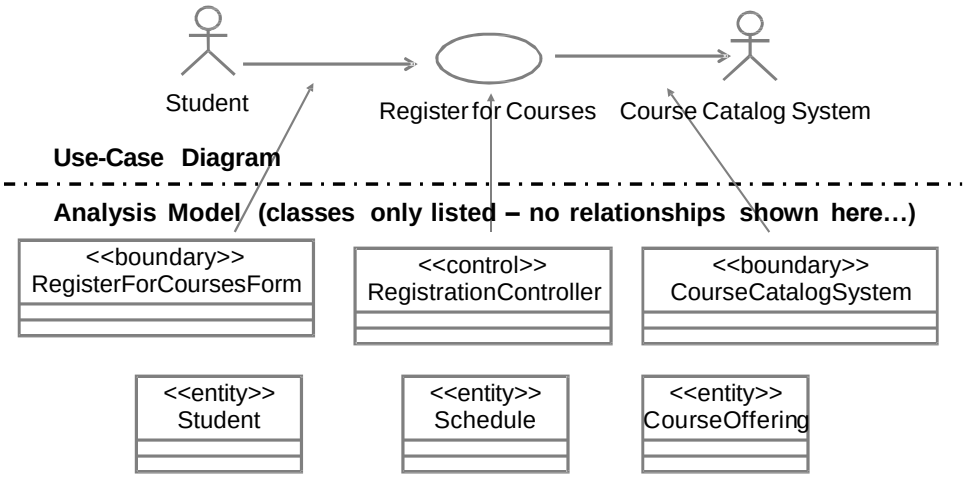


A collaboration diagram for the Withdraw Money use-case realization in the analysis model



Example: Analysis Classes

- The diagram shows the **classes participating** in the Register for Courses use case



4. Checking Against the Interaction Model

- Go over the use cases and think about how they would work.
- When the domain and application class models are in place, you should be able to simulate a use case with the classes.

ATM Example

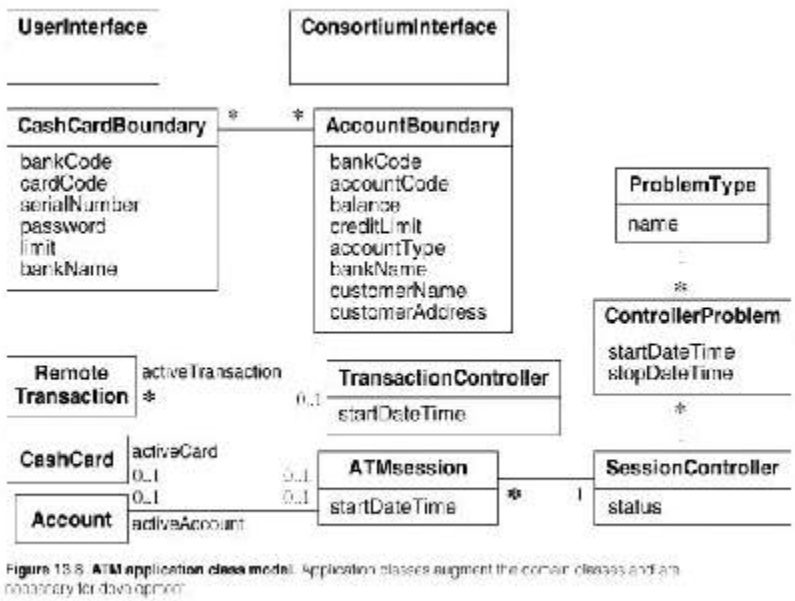


Figure 13.8 ATM application class model. Application classes augment the control classes and are necessary for data storage.

Application State Model

- The application state model focuses on application classes
- Augments the domain state model

Application State Model- steps

1. Determine Application Classes with States
2. Find events
3. Build state diagrams
4. Check against other state diagrams
5. Check against the class model
6. Check against the interaction model

1. Determine Application Classes with States

- Good candidates for state models
  - User interface classes
  - Controller classes
- ATM example
  - The controllers have states that will elaborate.

2. Find events

- Study scenarios and extract events.
- In domain model
  - Find states and then find events
- In application model
  - Find events first, and then find states
- ATM example
  - Revisit the scenarios, some events are:
  - *Insert card, enter password, end session and take card.*

3. Building State Diagrams

- To build a state diagram for each application class with temporal behavior.

- Initial state diagram
  - Choose one of these classes and consider a sequence diagram.
  - The initial state diagram will be a sequence of events and states.
  - Every scenario or sequence diagram corresponds to a path through the state diagram.
- Find loops
  - If a sequence of events can be repeated indefinitely, then they form a loop.
- Merge other sequence diagrams into the state diagram.
- After normal events have been considered, add variation and exception cases.
- The state diagram of a class is finished when the diagram covers all scenarios and the diagram handles all events that can affect a state.
- Identify the classes with multiple states
- Study the interaction scenarios to find events for these classes
- Reconcile the various scenarios
- Detect overlap and closure of loops

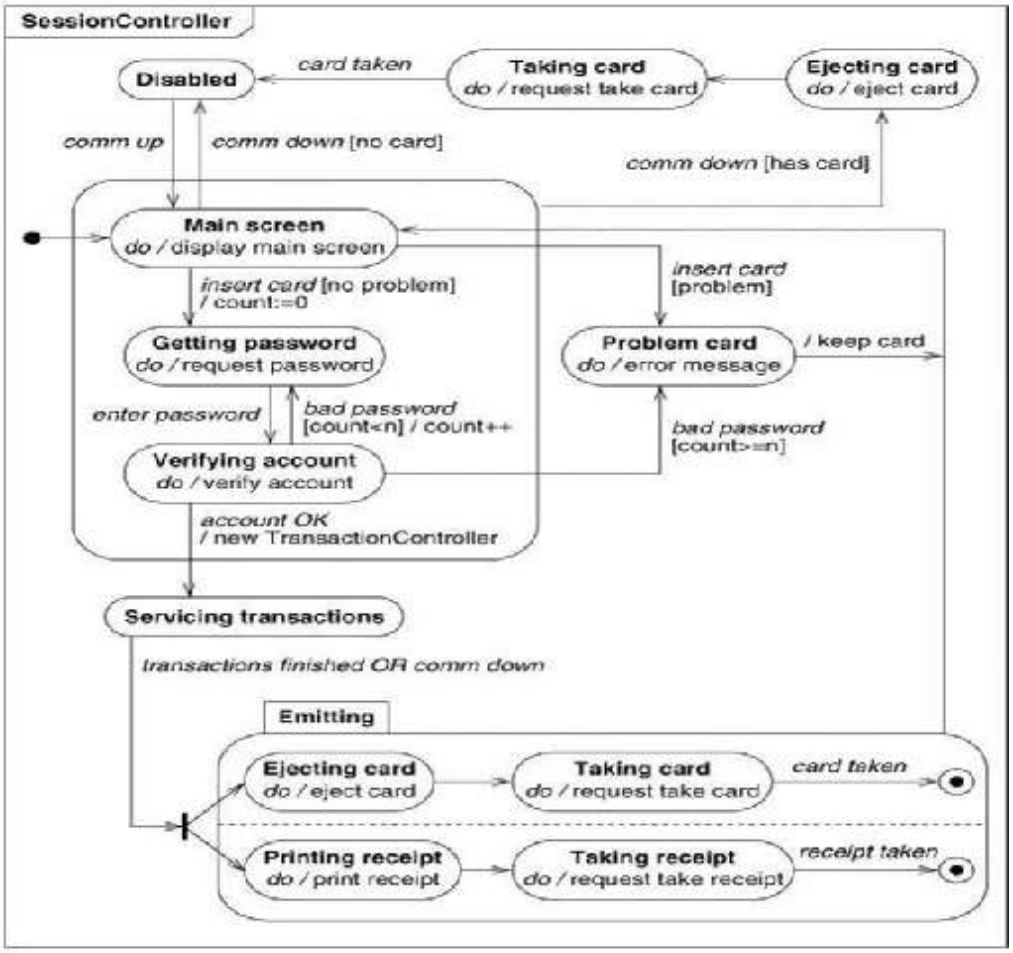


Figure 13.9 State diagram for *SessionController*. Build a state diagram for each application class with temporal behavior.

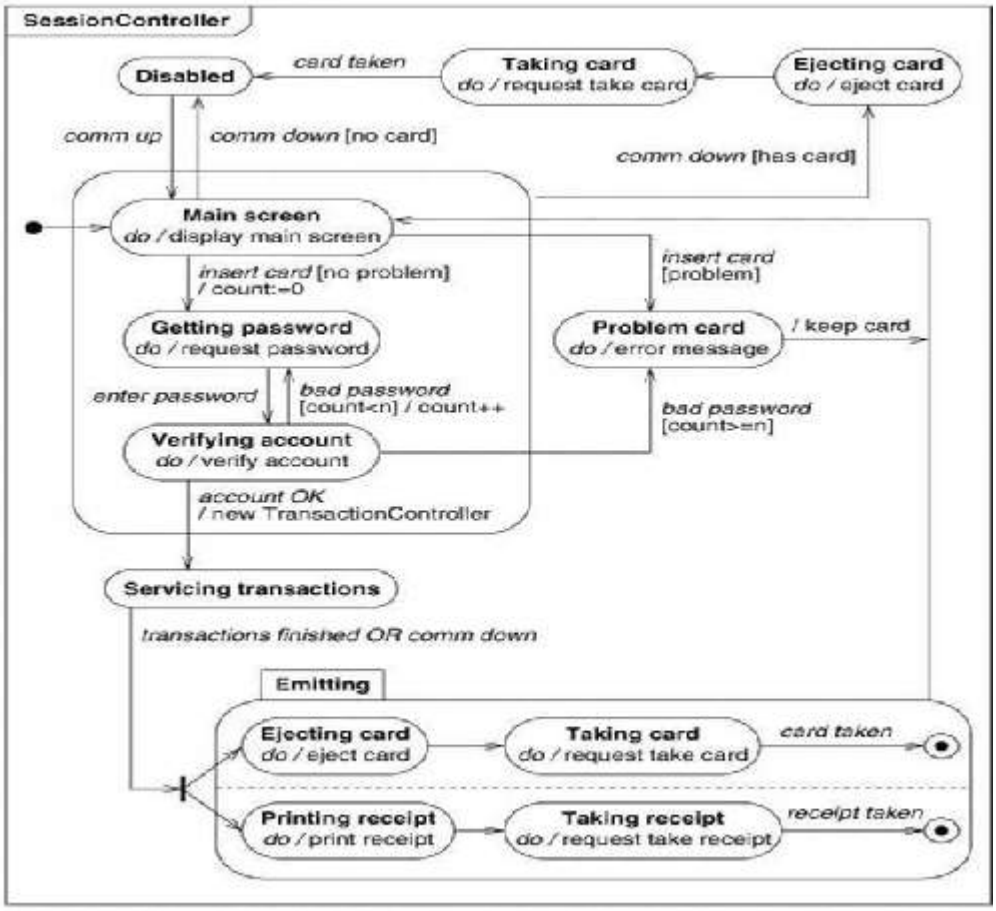


Figure 13.9 State diagram for *SessionController*. Build a state diagram for each application class with temporal behavior.

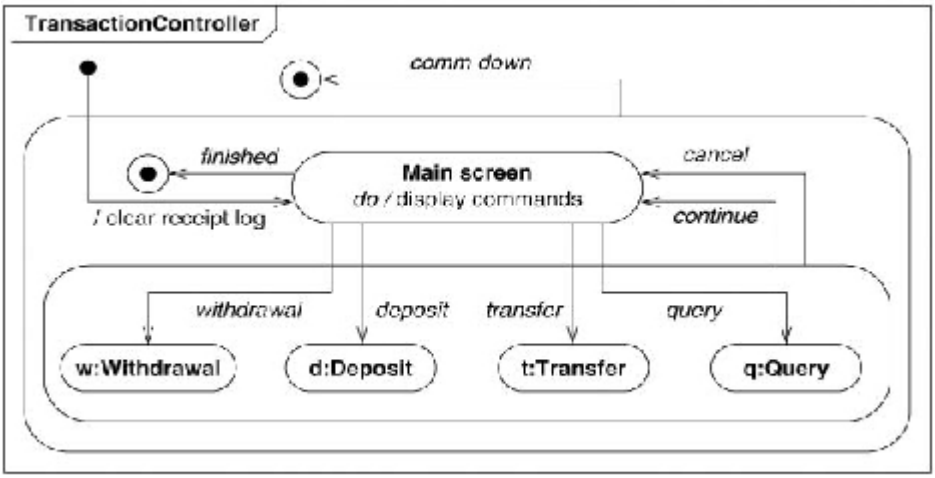


Figure 13.10 State diagram for *TransactionController*. Obtain information from the scenarios of the interaction model.

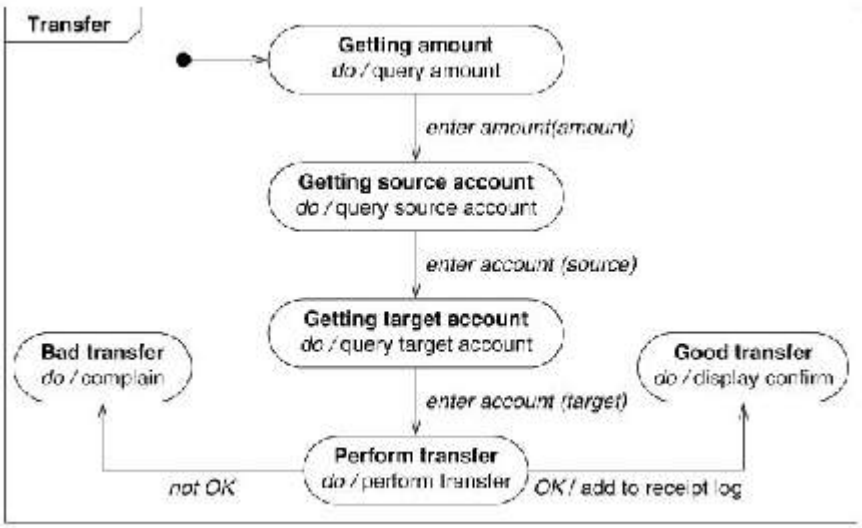


Figure 13.11 State diagram for *transfer*. This diagram elaborates the *transfer* state in Figure 13.10.

4. check against other state diagrams

- Every event should have a sender and a receiver.
- Follow the effects of an input event from object to object through the system to make sure that they match the scenarios.
- Objects are inherently concurrent.
- Make sure that corresponding events on different state diagrams are consistent.
- ATM example
  - The *SessionController* initiates the *TransactionController*,
  - The termination of the *TransactionController* causes the *SessionController* to resume.

5. Check against the class model

- ATM example
  - Multiple ATMs can potentially concurrently access an account.
  - Account access needs to be controlled to ensure that only one update at a time is applied.

6. Check against the interaction model

- Check the state model against the scenarios of the interaction model.
- Simulate each behavior sequence by hand and verify the state diagrams.
- Take the state model and trace out legitimate paths.

Adding Operations

- Operations from the class model
- Operations from use cases
- Shopping-list operations
- Simplifying operations

Operations from the class model

- The reading and writing of attribute values and association links.
- Need not show them explicitly.

Operations from use cases

- Use cases lead to activities.
- Many of these activities correspond to operations on the class model.
- ATM example
  - *Consortium* → *verifyBankCode.*
  - *Bank* → *verifyPassword.*
  - *ATM* → *verifyCashCard*

Shopping-List Operations

- The real-world behavior of classes suggests operations.
- Shopping-list operations provide an opportunity to broaden a class definition.

ATM example

- Account.close()
- Bank.createSavingsAccount(customer):account
- Bank.createCheckingAccount(customer):account
- Bank.createCashCardAuth(customer);cashCardAuthorization

Simplifying Operations

- Try to broaden the definition of an operation to encompass similar operations.
- Use inheritance to reduce the number of distinct operations.

ATM domain class model

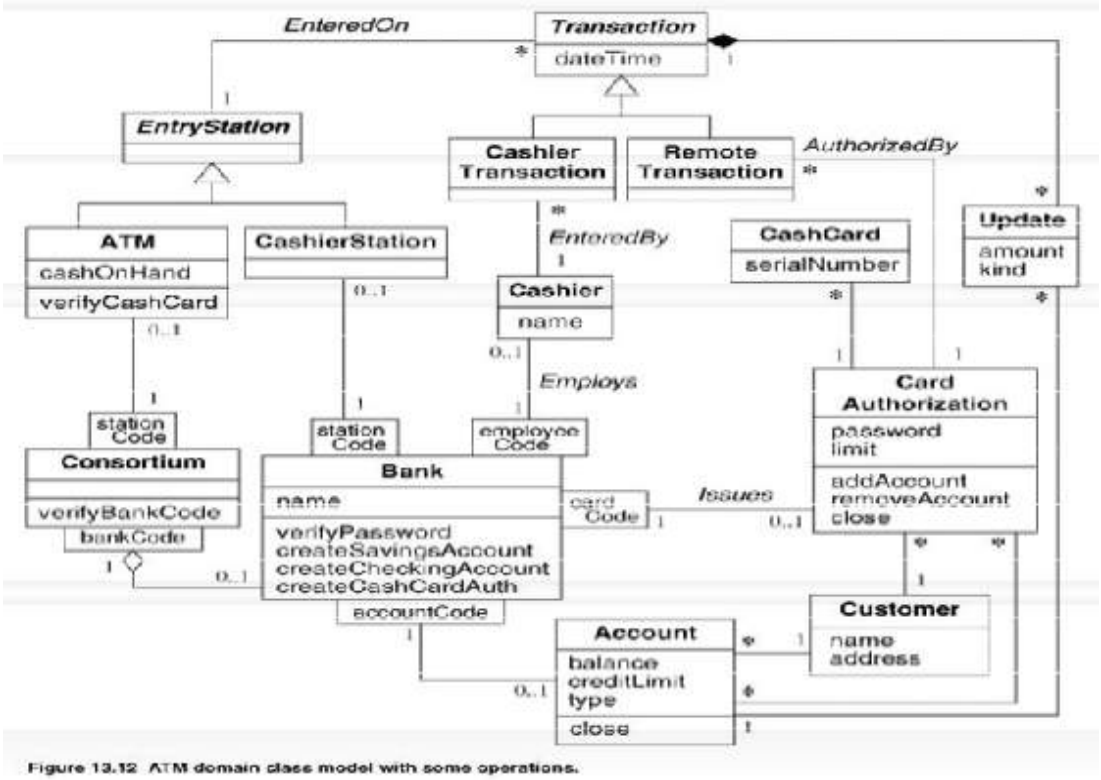
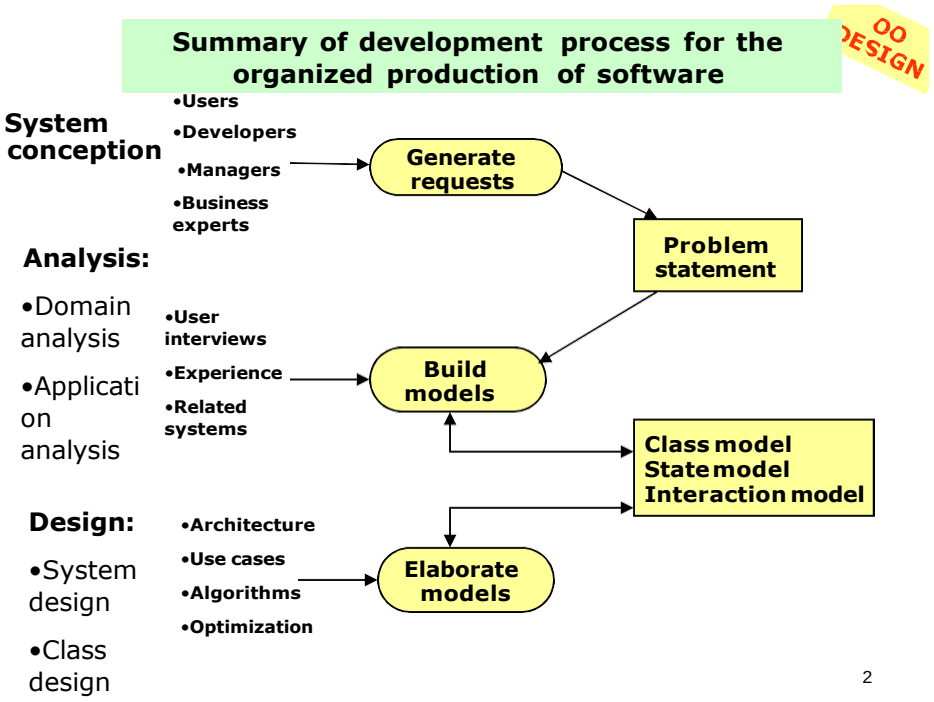
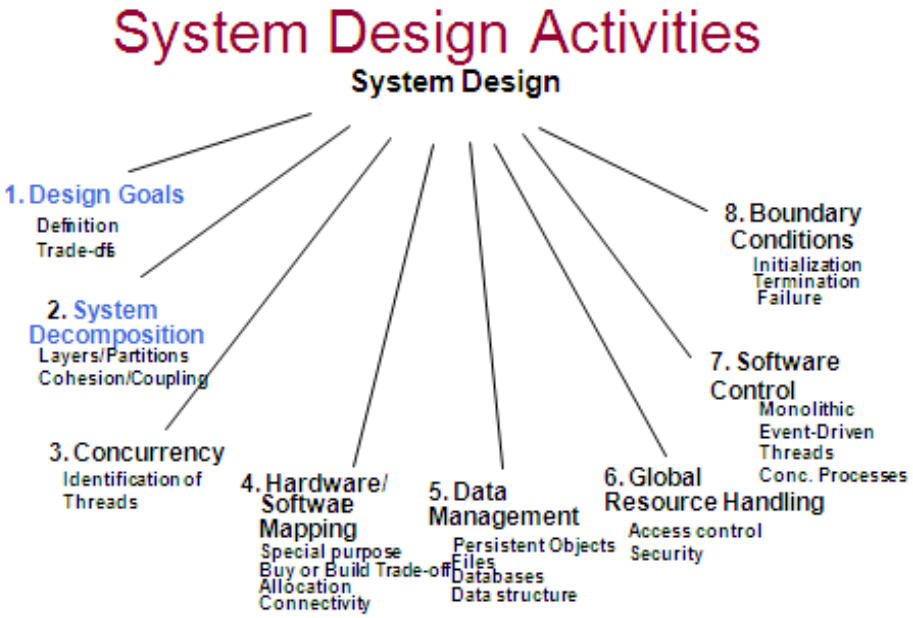


Figure 13.12 ATM domain class model with some operations.

Overview of System Design



- Analysis – focus is on *what* needs to be done; independent of *how it* is done
- Design – focus is on decisions about how the problem will be solved
  - First at high level
  - Then with more detail
- *System Design* –
  - first design stage
  - Overall structure and style
  - Determines the organization of the system into subsystems
  - Context for detailed decisions about how the problem will be solved



Estimate system performance

- To determine if the system is feasible
- To make simplifying assumptions

ATM Example

- Suppose
  - The bank has 40 branches, also 40 terminals.
  - On a busy day half the terminals are busy at once.
  - Each customer takes one minute to perform a session.
  - A peak requirement of about 40 transactions a minute.
  - *storage*
  - Count the number of customers.
  - Estimate the amount of data for each customer.
  - :
  - :

Make a reuse plan

- Two aspects of reuse:
  - Using existing things
  - Creating reusable new things
- Reusable things include:
  - Models
  - Libraries
  - Frameworks
  - Patterns

Reusable Libraries

- A library is a collection of classes that are useful in many contexts.
- Qualities of "Good" class libraries:
  - *Coherence* – well focused themes
  - *Completeness* – provide complete behavior
  - *Consistency* - polymorphic operations should have consistent names and signatures across classes
  - *Efficiency* – provide alternative implementations of algorithms
  - *Extensibility* – define subclasses for library classes
  - *Genericity* – parameterized class definitions
- Problems limit the reuse ability:
  - Argument validation
    - Validate arguments by collection or by individual
  - Error Handling
    - Error codes or errors
  - Control paradigms
    - Event-driven or procedure-driven control
  - Group operations
  - Garbage collection

- Name collisions

Reusable Frameworks

- A framework is a skeletal structure of a program that must be elaborated to build a complete application.
- Frameworks class libraries are typically application specific and not suitable for general use.

Reusable Patterns

- A pattern is a proven solution to a general problem.
- There are patterns for analysis, architecture, design, and implementation.
- A pattern is more likely to be correct and robust than an untested, custom solution.
- Patterns are prototypical model fragments that distill some of the knowledge of experts.

Pattern vs. Framework

- A pattern is typically a small number of classes and relationships.
- A framework is much broader in scope and covers an entire subsystem or application.

ATM example

- Transaction
- Communication line

Breaking a System into Subsystem

- Each subsystem is based on some common theme, such as
  - Similar functionality
  - The same physical location, or
  - Execution on the same kind of hardware.

Software Architecture

Breaking a System into Subsystem

- A subsystem is a group of classes, associations, operations, events, and constrains.
- A subsystem is usually identified by the services it provides.
- Each subsystem has a well-defined interface to the rest of the system.

- The relation between two subsystems can be
  - Client-server relationship
  - Peer-to-peer relationship

The decomposition of systems

- Subsystems is organized as a sequence of
  - Horizontal layers,
  - Vertical partitions, or
  - Combination of layers and partitions.

Layered system

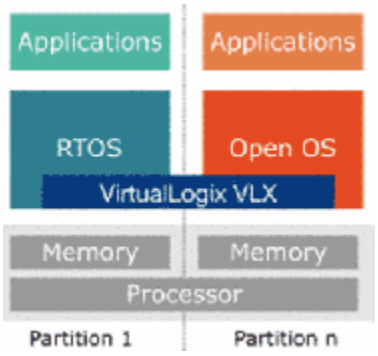
- Each built in terms of the ones below it.
- The objects in each layer can be independent.
- E.g.
  - A client-server relationship
- Problem statement specifies only the top and bottom layers:
  - The top is the desired system.
  - The bottom is the available resources.
- The intermediate layers is than introduced.
- Two forms of layered architectures:
  - Closed architecture
    - Each layer is built only in terms of the immediate lower layer.
  - Open architecture
    - A layer can use features on any lower layer to any depth.
    - Do not observe the principle of information hiding.

Partitioned System

- Vertically divided into several subsystems
- Independent or weakly coupled
- Each providing one kind of service.
- E.g. A computer operating system includes
  - File system
  - Process control
  - Virtual memory management
  - Device control

Partitions vs. Layers

- Layers vary in their level of abstraction.
- Layers depend on each other.
- Partitions divide a system into pieces.
- Partitions are peers that are independent or mutually dependent. (peer-to-peer relationship)



Combining Layers and Partitions

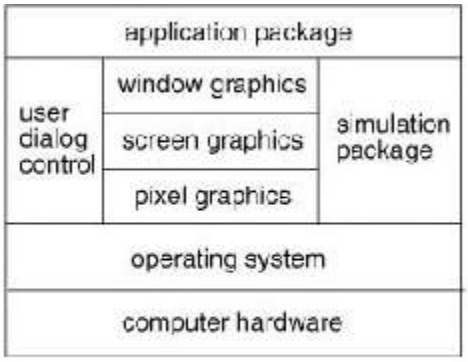


Figure 14.1 Block diagram of a typical application.
Most large systems mix layers and partitions.

ATM Example

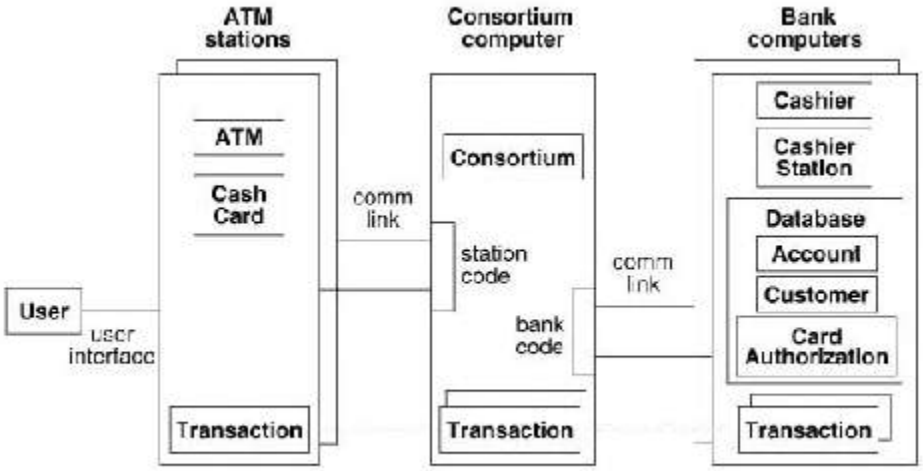


Figure 14.2 Architecture of ATM system. It is often helpful to make an informal diagram showing the organization of a system into subsystems.

Identifying Concurrency

- To identify
  - The objects that must be active concurrently.
  - The objects that have mutually exclusive activity

Inherent Concurrency

- By exam the state model
- Two objects are inherently concurrent if they can receive events at the same time without interacting.
- If the events are unsynchronized, you cannot fold the objects onto a single thread of control.

Defining Concurrent Tasks

- By examining the state diagrams, you can fold many objects onto a single thread of control.
- A ***thread of control*** is a path through a set of state diagrams on which only a single object at a time is active.
- ATM example:
  - Combine the ATM object with the bank transaction object as a single task.

Allocation of Subsystems

- Allocate each concurrent subsystem to a hardware unit by
  - Estimating hardware resource requirements
  - Making hardware-software trade-offs
  - Allocating tasks to processors
  - Determining physical connectivity

Estimating hardware resource requirements

- The number of processors required depends on the volume of computations and the speed of the machine
- Example: military radar system generates too much data in too short a time to handle in single CPU, many parallel machines must digest the data
- Both steady-state load and peak load are important

Making hardware-software trade-offs

- You must decide which subsystems will be implemented in hardware or software
- Main reasons for implementing subsystems in hardware
  - Cost -
  - Performance – most efficient hardware available

Allocating tasks to processors

- Allocating software subsystems to processors
- Several reasons for assigning tasks to processors.
  - Logistics – certain tasks are required at specified physical locations, to control hardware or permit independent operation
  - Communication limits
  - Computation limits – assigning highly interactive systems to the same processor, independent systems to separate processors

Determining physical connectivity

- Determine the arrangement and form of the connections among the physical units
  - Connection topology- choose an topology for connecting the physical units
  - Repeated units-choose a topology of repeated units
  - Communications- choose the form of communication channels and communication protocols

Management of Data Storage

- Alternatives for data storage:
  - Data structures,
  - Files,
  - Databases

Data Suitable for Files

- Files are cheap, simple, and permanent, but operations are low level.

- Data with high volume and low information density (such as archival files or historical records).
- Modest quantities of data with simple structure.
- Data that are accessed sequentially.
- Data that can be fully read into memory.

Data Suitable for Databases

- Database make applications easier to port, but interface is complex.

- Data that require updates at fine levels of detail by multiple users.
- Data that must be accessed by multiple application programs.
- Data that require coordinated updates via transactions.
- Large quantities of data that must be handled efficiently.
- Data that are long-lived and highly valuable to an organization.
- Data that must be secured against unauthorized and malicious access.

Figure 14.4 Data suitable for databases. Databases provide heavyweight data management and are used for most important business applications.

Handling Global Resources

- The system designer must identify global resources and determine *mechanisms for controlling access* to them.
- Kinds of global resources:
  - Physical units

- Processors, tape drivers…
- Spaces
  - Disk spaces, workstation screen…
- Logical name
  - Object ID, filename, class name…
- Access to shared data
  - Database

- Some common mechanisms are:
  - Establishing "**guardian**" object that serializes all access
  - **Partitioning** global resources into disjoint subsets which are managed at a lower level, and
  - **Locking**

ATM example

- Bank codes and account numbers are global resources.
- Bank codes must unique within the context of a consortium.
- Account codes must be unique within the context of a bank.

Choosing a Software Control Strategy

- To choose a single control style for the whole system.
- Two kinds of control flows:
  - External control
  - Internal control

Software External Control

- Concerns the flow of externally visible events among the objects in the system.
- Three kinds:
  - Procedure-driven sequential
  - Event-driven sequential
  - Concurrent

Procedure-driven Control

- Control resides within the program code
- Procedure request external input and then wait for it
- When input arrives, control resumes with in the procedure that made the call.
- Advantage:
  - Easy to implement with conventional languages
- Disadvantage:
  - The concurrency inherent in objects are to mapped into a sequential flow of control.
- Suitable only if the state model shows a regular alternation of input and output events.
- C++ and Java are procedural languages.

- They fail to support the concurrency inherent in objects.

Event-driven Control

- Control resides within a dispatcher or monitor that the language, subsystem, or operating system provides.
- The dispatcher calls the procedures when the corresponding events occur.

Software Internal Control

- Refer to the flow of control within a process.
- To decompose a process into several tasks for logical clarity of for performance.
- Three kinds:
  - Procedure calls,
  - Quasi-concurrent intertask call,
    - Multiple address spaces or call stacks exist but only a single thread of control can be active at once.
  - Current intertask calls

Handling Boundary Conditions

- Most of system design is concerned with steady-state behavior, but boundary conditions are also important
- Boundary conditions are
  - Initialization
  - Termination, and
  - Failure
- Initialization
  - The system must initialize constant data, parameters, global variables, …
- Termination
  - Release any external resources that it had reserved.
- Failure
  - Unplanned termination of a system. The good system designer plans for orderly failure

Setting Trade-off Priorities

- The priorities reconcile desirable but incompatible goals.
  - E.g memory vs. cost
- Design trade-offs affect the entire character of a system.
- The success of failure of the final product may depend on how well its goal s are chosen.
- Essential aspect of system architecture is making trade-offs between
  - time and space
  - Hardware and software
  - Simplicity and generality, and
  - Efficiency and maintainability
- The system design must state the priorities

Common Architectural Styles

- Several prototypical architectural styles are common in existing system.
- Some kinds of systems:
  - Batch transformation } Functional transformations

- Continuous transformation
- Interactive interface
- Dynamic simulation } Time-dependent systems
- Real-time system
- Transaction manager -> Database system

Batch transformation

- Perform sequential computation.

- The application receives the inputs, and the goal is to compute an answer.
- Does not interact with the outside world
- E.g.
  - Compiler
  - Payroll processing
  - VLSI automatic layout
  - :
- The most important aspect is to define a clean series of steps
- Sequence of steps for a compiler

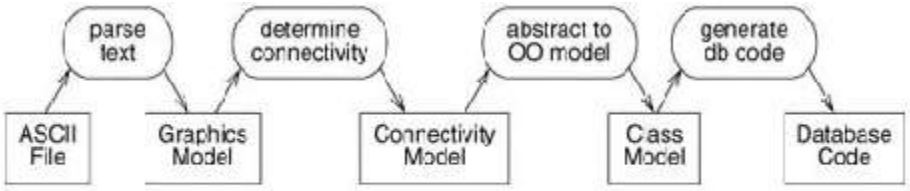


Figure 14.5 Sequence of steps for a compiler. A batch transformation is a sequential input-to-output transformation that does not interact with the outside world.

The steps in designing a batch transformation are as follows

- Break the overall transformation into stages, with each stage performing one part of the transformation.
- Prepare class models for the input, output and between each pair of successive stages. Each stage knows only about the models on either side of it.
- Expand each stage in turn until the operations are straightforward to implement.
- Restructure the final pipeline for optimization.

Continuous transformation

- The outputs actively depend on changing inputs.
- Continuously updates the outputs (in practice discretely)
- E.g.
  - Signal processing
  - Windowing systems
  - Incremental compilers
  - Process monitoring system
- Sequence of steps for a graphics application

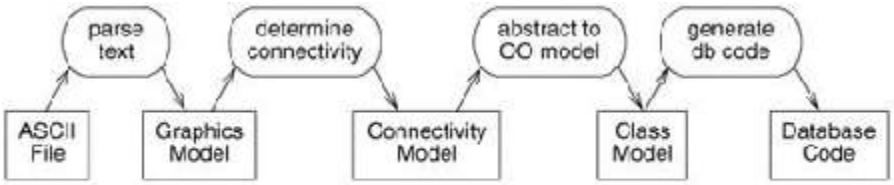


Figure 14.5 Sequence of steps for a compiler. A batch transformation is a sequential input-to-output transformation that does not interact with the outside world.

- Steps in designing a pipeline for a continuous transformation are as follows
  - Break the overall transformation into stages, with each stage performing one part of the transformation.
  - Define input, output and intermediate models between each pair of successive stages as for the batch transformation
  - Differentiate each operation to obtain incremental charges to each stage.
  - Add additional intermediate objects for optimization.

Interactive interface

- Dominated by interactions between the system and external agents.

Steps in designing an interactive interface are as follows

- ✓ Isolate interface classes from the application classes
- ✓ Use predefined classes to interact with external agents
- ✓ Use the state model as the structure of program
- ✓ Isolate physical events from logical events.
- ✓ Fully specify the application functions that are invoked by the interface

Dynamic simulation

- Models or tracks real-world objects.
- Steps in designing a dynamic simulation
  - Identify active real-world objects from the class model.
  - Identify discrete events
  - Identify continuous dependencies
  - Generally simulation is driven by a timing loop at a fine time scale

Real-time system

- An interactive system with tight time constraints on actions.

Transaction manager

- Main function is to store and retrieve data.
- Steps in designing an information system are as follows
  - Map the class model to database structures.
  - Determine the units of concurrency
  - Determine the unit of transaction
  - Design concurrency control for transactions

Architecture of the ATM system

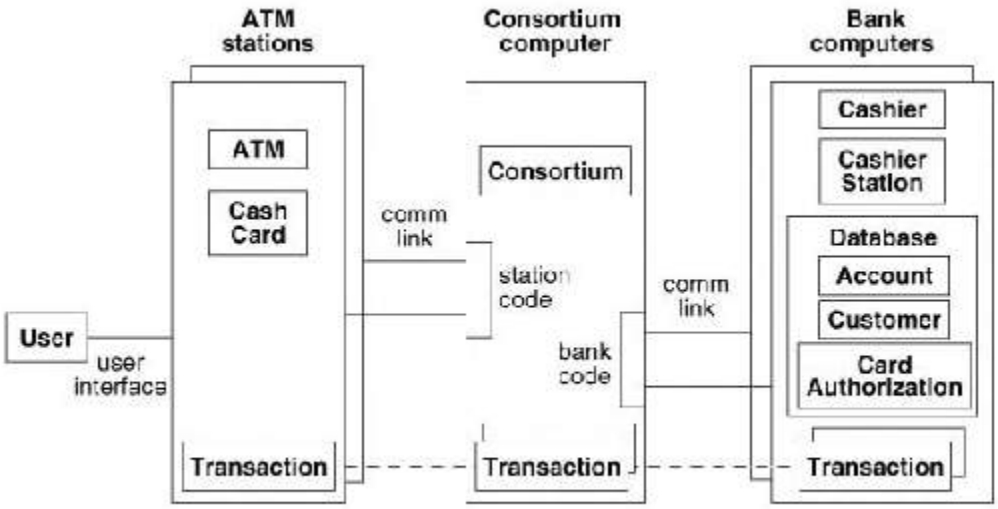


Figure 14.2 Architecture of ATM system. It is often helpful to make an informal diagram showing the organization of a system into subsystems.

Unit-6: Class Design, Implementation modeling

7 Hours

Syllabus:

- *Class Design: Overview of class design;*
- *Bridging the gap; Realizing use cases; Designing algorithms; Recursing downwards, Refactoring;*
- *Design optimization; Reification of behavior; Adjustment of inheritance; Organizing a class design;*
- *ATM example.*
- *Implementation Modeling: Overview of implementation; Fine-tuning classes; Fine-tuning generalizations; realizing associations; Testing.*
- *Legacy Systems: Reverse engineering;*
- *Building the class models; Building the interaction model;*
- *Building the state model; Reverse engineering tips; Wrapping; Maintenance.*

Class design

- The analysis phase determines *what* the implementation must do
- The system design phase *determines the plan of attack*
- The purpose of the class design is to complete the *definitions of the classes* and *associations* and choose *algorithms* for operations

Overview of Class Design – steps

1. Bridging the gap
2. Realizing Use Cases
3. Designing Algorithms
4. Recursing Downward
5. Refactoring
6. Design Optimization
7. Reification of Behavior
8. Adjustment of Inheritance
9. Organizing a Class Design

1. <u>Bridging the gap</u>

Bridge the gap from high-level requirements to low-level services

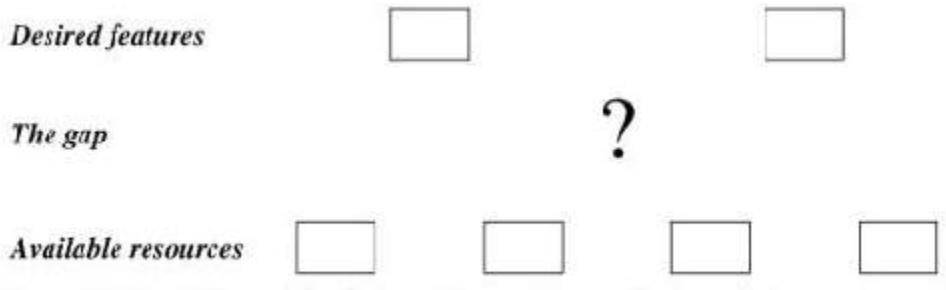
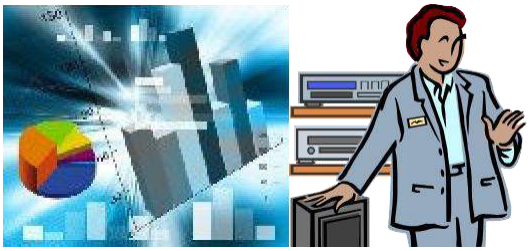


Figure 15.1 The design gap. There is often a disconnect between the desired features and the available resources.

- Salesman can use a spreadsheet to construct formula for his commission – readily build the system
- Web-based ordering system – cannot readily build the system because too big gap between the resources and features



- The intermediate elements may be operations, classes or other UML constructs.
- You must invent intermediate elements to bridge the gap.

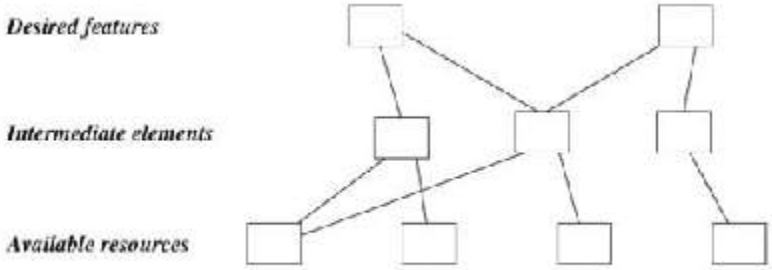


Figure 15.2 Bridging the gap. You must invent intermediate elements to bridge the gap between the desired features and the available resources.

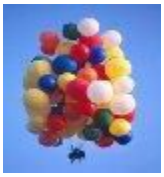
2. Realizing Use Cases

- Realize use cases with operations.
- The cases define system-level behavior.
- During design you must invent new operations and new objects that provide this behavior.

- Step1: List the **responsibilities** of a use case or operation.

- A ***responsibility*** is something that an object knows or something it must do.
- For Example:
  - An **online theater ticket system**
  - Making a reservation has the **responsibility** of
    - Finding unoccupied seats to the desired show,
    - Marking the seats as occupied,
    - Obtaining payment from the customer,
    - Arranging delivery of the tickets, and
    - Crediting payment to the proper account.

- Step2: Each operation will have various responsibilities.
  - Group the responsibilities into **clusters** and try to make each cluster coherent.
- Step3: Define an operation for each responsibility cluster.
- Step4: Assign the new lower-level operations to classes.



ATM Example

- *Process transaction* includes:
  - Withdrawal includes responsibilities:
    - Get amount from customer, verify that amount is covered by the account balance, verify that amount is within the bank's policies, verify that ATM has sufficient cash, ….
    - A database transaction ensures all-or-nothing behavior.
  - Deposit
  - Transfer

Use Case for the ATM

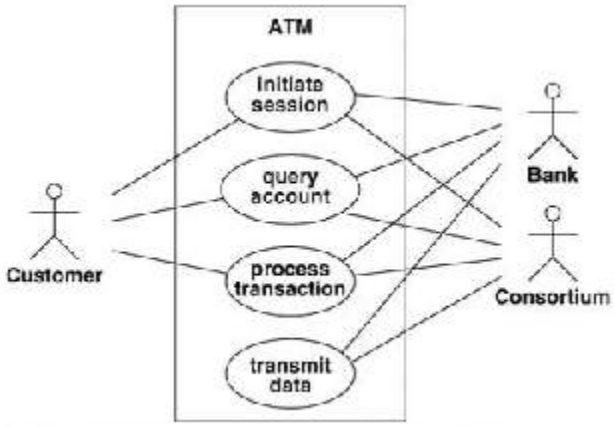


Figure 13.1 Use case diagram for the ATM. Use cases participate

- Process transaction includes:
  - Deposit includes responsibilities:
    - Get amount from customer, accept funds envelope from customer, …
  - Transfer includes responsibilities:
    - Get source account, get target account, get amount, verify that source account covers amount, …
- There is some overlap between the operations.
- A reasonable design would coalesce this behavior and build it once.

3. Designing Algorithms

- Formulate an ***algorithm*** for each operation
- The analysis specification tells *what* the operation does for its clients
- The algorithm show *how* it is done

Designing Algorithms- steps

i. Choose *algorithms* that minimize the cost of implementing operations.
ii. Select *data structures* appropriate to the algorithms
iii. Define new internal classes and operations as necessary.
iv. Assign operations to appropriate classes.

i. Choosing algorithms (Choose algorithms that minimize the cost of implementing operations)

- When efficiency is not an issue, you should use simple algorithms.
- Typically, 20% of the operations consume 80% of execution time.
- Considerations for choosing alternative algorithms
  - Computational complexity
  - Ease of implementation and understandability
  - Flexibility

- Simple but inefficient
- Complex efficient

➢ ATM Example
  - Interactions between the consortium computer and bank computers could be complex.
  - Considerations:
    - Distributed computing
    - The scale of consortium computer (scalability)
    - The inevitable conversions and compromises in coordinating the various data formats.
  - All these issues make the choice of algorithms for coordinating the consortium and the banks important

The ATM Case Study

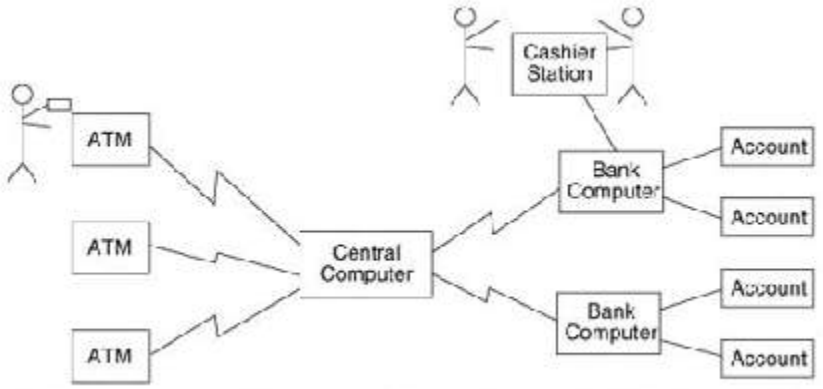


Figure 11.3 ATM network. The ATM case study threads throughout the remainder of this book.

ii. Choosing Data Structures (select data structures appropriate to the algorithm)
  a. Algorithms require data structures on which to work.
  b. They organize information in a form convenient for algorithms.
  c. Many of these data structures are instances of ***container classes***.
  d. Such as arrays, lists, queues, stacks, set…etc.

iii. Defining New Internal Classes and Operations
  a. To invent new, low-level operations during the decomposition of high-level operations.
  b. The expansion of algorithms may lead you to create new classes of objects to hold intermediate results.
  c. ATM Example:
    i. *Process transaction* uses case involves a customer receipt.
    ii. A *Receipt* class is added.

iv. Assigning Operations to Classes (assign operations to appropriate classes)
  a. How do you decide what class owns an operation?
    i. Receiver of action

1. To associate the operation with the *target* of operation, rather than the *initiator*.

b. Query vs. update
   i. The object that is changed is the target of the operation
c. Focal class
   i. Class centrally located in a star is the operation's target
d. Analogy to real world

ATM Example

- *Process transaction* includes:
  - Withdrawal includes responsibilities:
    - Get amount from customer, verify that amount is covered by the account balance, verify that amount is within the bank's policies, verify that ATM has sufficient cash, ….
    - A database transaction ensures all-or-nothing behavior.
  - Deposit
  - Transfer
- Customer.getAccount(), account.verifyAmount(amount), bank.verifyAmount(amount), ATM.verifyAmount(amount)

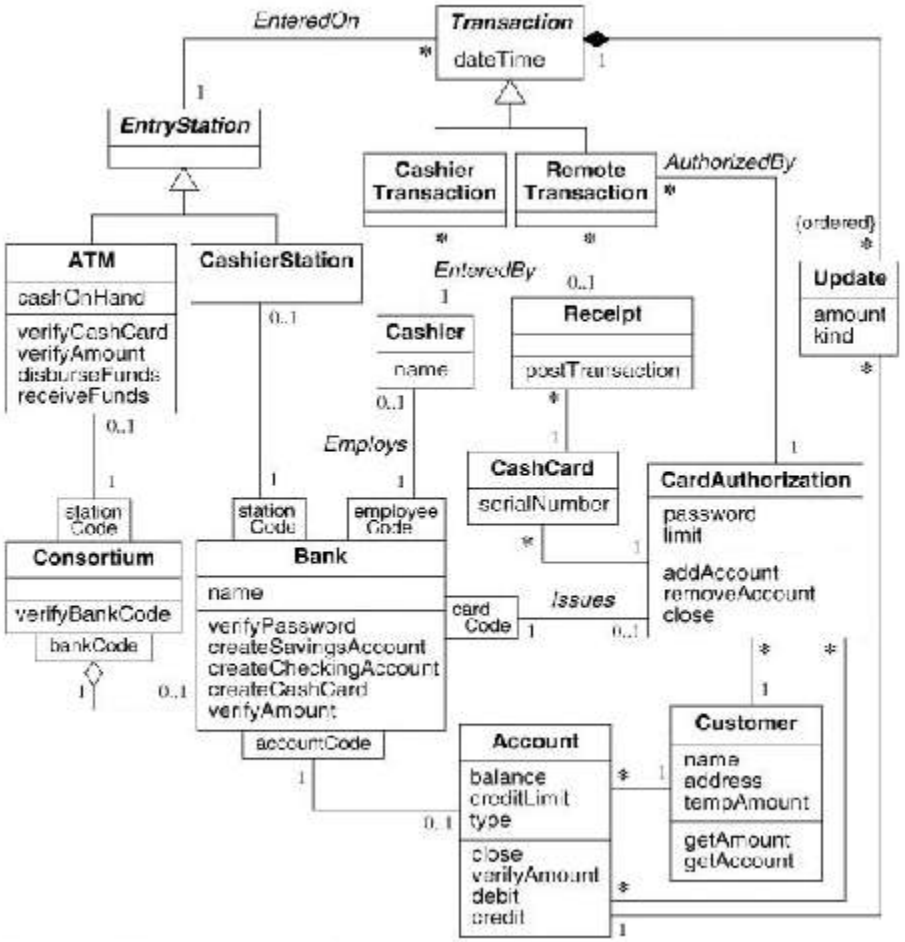


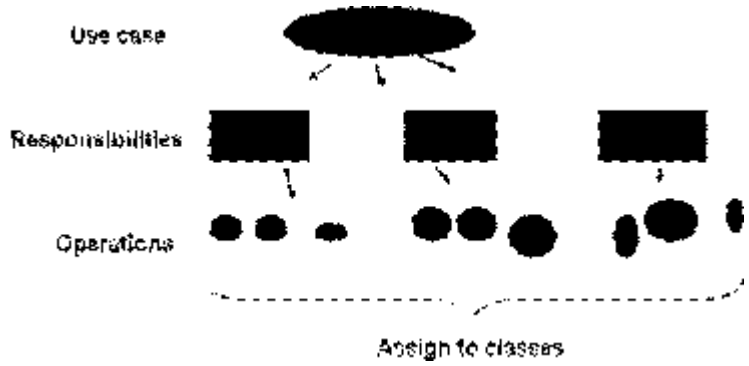
Figure 16.4 ATM domain class model with some class design elaborations.

4. Recursing Downward

- To organize operations as layers.
  - Operations in higher layers invoke operations in lower layers.
- Two ways of downward recursion:
  - By functionality
  - By mechanism
- Any large system mixes functionality layers and mechanism layers.

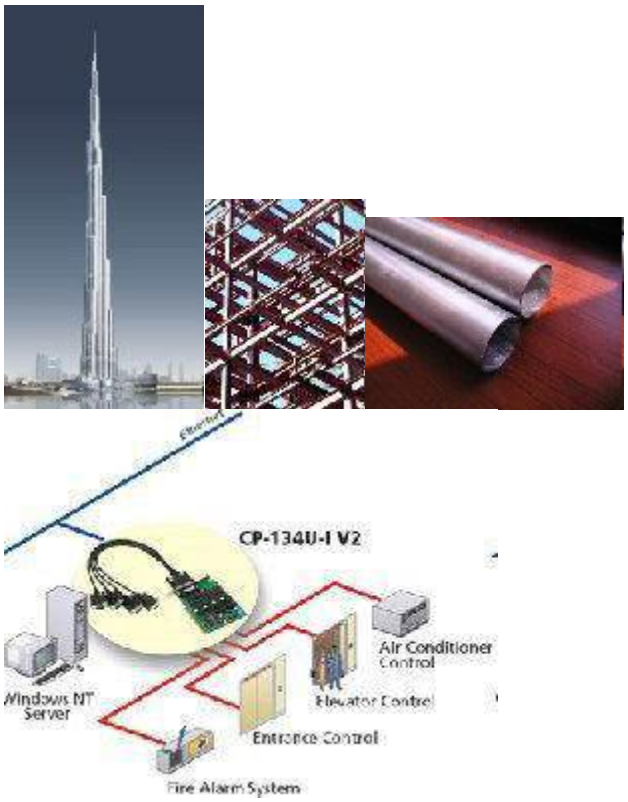
Functionality Layers

- Take the required high-level functionality and break it into lesser operations.
- Make sure you combine similar operations and attach the operations to classes.
- An operation should be coherent meaningful, and not an arbitrary portion of code.
- ATM eg., **use case** decomposed into **responsibilities** (see sec 15.3). Resulting **operations** are assigned to classes (see sec 15.4.4). If it is not satisfied rework them



Mechanism Layers

- Build the system out of layers of needed support mechanisms.
- These mechanisms don't show up explicitly in the high-level responsibilities of a system, but they are needed to make it all work.
- E.g. Computing architecture includes
- Data structures, algorithms, and control patterns.
- A piece of software is built in terms of other, more mechanisms than itself.



5. Refactoring

- Refactoring
  - Changes to the internal structure of software to improve its design without altering its external functionality.
- You must revisit your design and rework the classes and operations so that they clean satisfy all their uses and are conceptually sound.

ATM Example

- Operations of process transaction
  - *Account.credit(amount)*
  - *Account.debit(amount)*
- Combine into
  - *Account.post(amount)*

6. Design Optimization

- To design a system is to first get the logic correct and then optimize it.
- Often a small part of the code is responsible for most of the time or space costs.
- It is better to focus optimization on the **critical areas,** than to spread effort evenly.

Design Optimization

- Optimized system is **more obscure** and less **likely to be reusable**.
- You must strike an appropriate **balance between efficiency and clarity.**
- **Tasks** to optimization:
  i. Provide efficient access paths.
  ii. Rearrange the computation for greater efficiency.
  iii. Save intermediate results to avoid recomputation.

i. Adding Redundant Associations for Efficient Access

- ✓ Rearrange the associations to optimize critical aspects of the system.
- ✓ Consider employee skills database

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

- ✓ *Company.findSkill*( ) returns a set of persons in the company with a given skill.
- ✓ Suppose the company has 1000 employees,.
- ✓ In case where the number of hits from a query is low because few objects satisfy the test, an *index* can improve access to frequently retrieved objects.

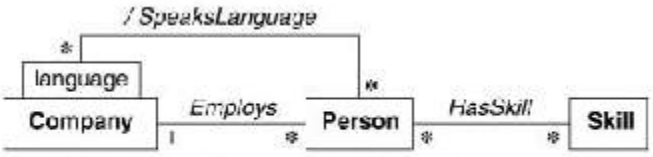


Figure 15.6 Design model for person skills. Derived data is acceptable during design for operations that are significant performance bottlenecks.

- ✓ Examine each operations and see what associations it must traverse to obtain its information.
- ✓ Next, for each operation, note the following,
  - Frequency of access
  - Fan-out
  - Selectivity

ATM Example

- Banks must report cash deposits and withdrawals greater than $10,000 to the government.

- Trace from
  - *Bank* to *Account*,
  - *Account* to *Update*,
  - Then filter out the updates that are cash and greater than $10,000
- A derived association from *Bank* to *Update* would speed this operation.

ii. Rearranging Execution Order for Efficiency

- ✓ After adjusting the structure of class model to optimize frequent traversals, the next thing is
- ✓ To optimize the algorithm
  i. To eliminate dead paths as early as possible
  ii. To narrow the search as soon as possible
  iii. Sometimes, invert the execution order of a loop

iii. Saving Derived Values to Avoid Recomputation

- ✓ There are three ways to handle updates
  i. Explicit update
  ii. Periodic recomputation
  iii. Active values

Reification behavior

- Behavior written in code is rigid; you can execute but cannot manipulate it at run time
- If you need to store, pass, or modify the behavior at run time, you should reify it

Adjustment of Inheritance

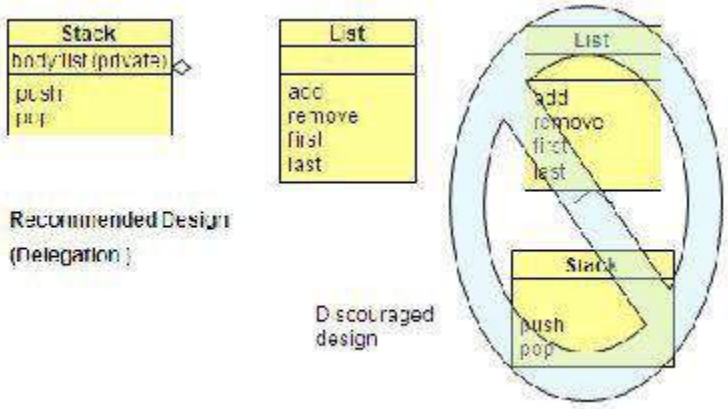
- To increase inheritance perform the following steps
  - Rearrange classes and operations to increase inheritance
  - Abstract common behavior out of groups of clusters
  - Use delegation to share behavior when inheritance is semantically invalid

Rearrange classes and operations to increase inheritance

- Use the following kinds of adjustments to increase the chance of inheritance
  - Operations with optional arguments
  - Operations that are special cases
  - Inconsistent names
  - Irrelevant operations

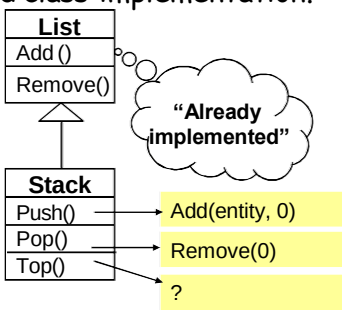
Use delegation to share behavior when inheritance is semantically invalid

- When class B inherits the specification of class A, you can assume that every instance of class B is an instance of class A because it behaves the same
- ***Inheritance of implementation*** – discourage this
- One object can selectively invoke the desired operations of another class, using delegation rather than inheritance
- Delegation consists of catching operation on one object and sending it to a related object
- Delegate only meaningful operations, so there is no danger of inheriting meaningless operations by accident



Implementation Inheritance

- A very similar class is already implemented that does almost the same as the desired class implementation.
  - Example: I have a **List** class, I need a **Stack** class. How about subclassing the **Stack** class from the **List** class and providing three methods, **Push()** and **Pop(), Top()**?



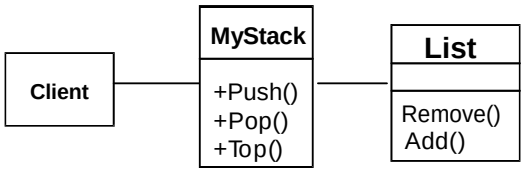
- Problem with implementation inheritance:
  - Some of the inherited operations might exhibit unwanted behavior. What happens if the Stack user calls Remove() instead of Pop()?
  - Close coupling – what happens if the Add() method is changed?

Problem with implementation inheritance

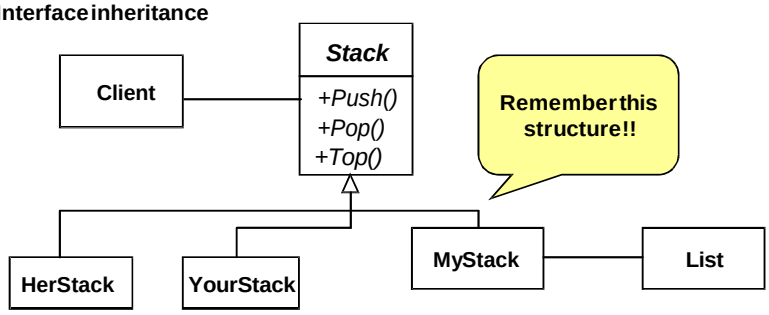
- How to avoid the following problem?

Some of the inherited operations might exhibit unwanted behavior.
What happens if the Stack user calls Remove() instead of Pop()?

1. Delegation



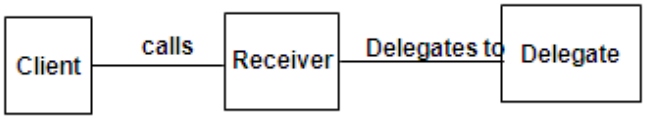
2. Interface inheritance



Delegation as alternative to Implementation Inheritance

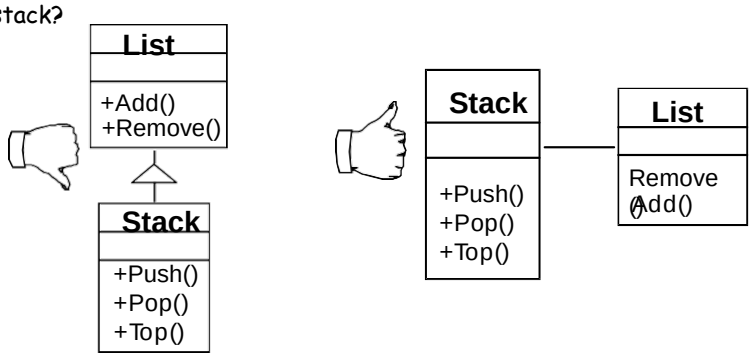
- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance
- In Delegation two objects are involved in handling a request
  - A receiving object delegates operations to its delegate.

The developer can make sure that the receiving object does not allow the client to misuse the delegate object



Delegation instead of Implementation Inheritance

- **Inheritance**: Extending a Base class by a new operation or overwriting an operation.
- **Delegation**: Catching an operation and sending it to another object.
- Which of the following models is better for implementing a stack?



Organization of Class Design

- We can improve the organization of a class design with the following steps:
  - Information hiding
  - Coherence of Entities
  - Fine-tuning packages

Information hiding

- Carefully separating external specification from internal specification
- There are several ways to hide information:
  - Limit the scope of class-model traversals
  - Do not directly access foreign attributes
  - Define interfaces at a high level of abstraction
  - Hide external objects
  - Avoiding cascading method calls

Coherence of Entities

➔ An entity, such as a class, an operation or a package is coherent if it is organized on a consistent plan and all its parts fit together toward a common goal.

➔ An entity should habve a single major theme

➔ It should not be a collection of unrelated parts.

Fine – Tuning Packages

- **Overview of Implementation**
- **Fine-tuning Classes**
- **Fine-tuning Generalization**
- **Realizing Associations**
- **Testing**

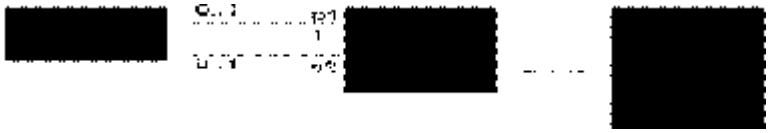
Fine-tuning classes

- Fine tune classes before writing code in order to simplify development or to improve performance
- Partition a class
- Merge classes
- Partition / merge attributes
- Promote an attribute / demote a class

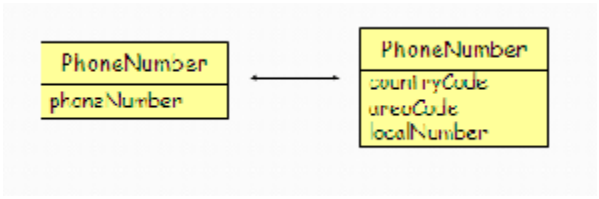
Fine-tuning classes – partition a class

- Sometimes it is helpful to fine-tune a model by partitioning or merging classes
- partitioning of a class can be complicated by generalization and association

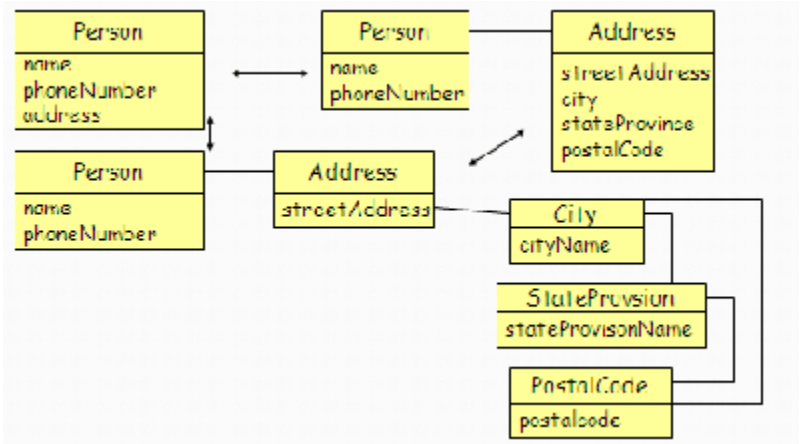
Fine-tuning classes – merge classes



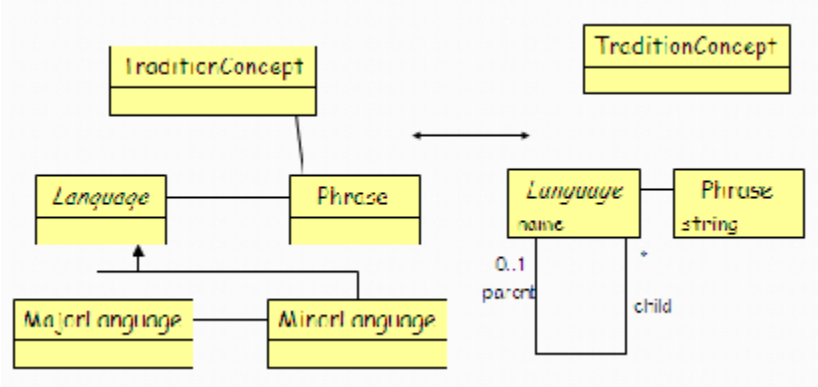
Fine-tuning classes – partition / merge attributes



Fine-tuning classes – promoting an attribute / demote a class

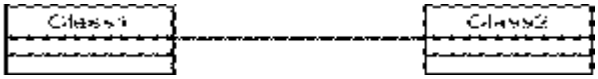


Fine-tuning generalizations



Realizing associations

- Associations are "glue" of the class model, providing access paths between objects
- Analyzing associations by traversing associations

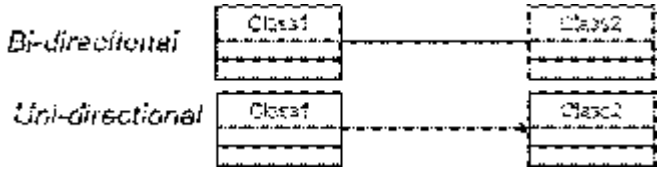


Analyzing Association Traversal

- Until now we assumed that associations are bidirectional
- But some applications are traversed in only one direction
- We may add another operation that make traversal in reverse direction

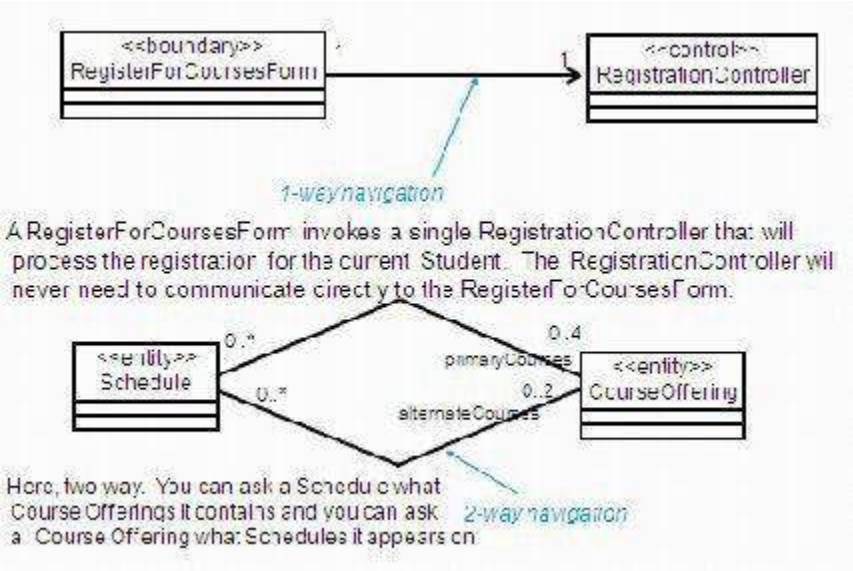
Navigability

- Possible to navigate from an associating class to the target class – indicated by arrow which is placed on the target end of the association line next to the target class (the one being navigated to).
- Associations are bi-directional by default – suppress arrows.
- Arrows only drawn for associations with one-way navigability.



- ✓ Navigability is inherently a design and implementation property.
- ✓ Can be specified in Analysis, but with expectation of refining in Class Design.
- ✓ In analysis, associations are usually bi-directional; design, we really check this.

Example: Navigability



One-way Associations

- Implement one-way associations using ***pointer-*** an attribute that contains the object reference
- Actual implementation of pointer using
  - Programming language pointer or
  - Database foreign key
- If the multiplicity is "one" then it is a *simple pointer*
- If the multiplicity is "many" then it is a *set of pointers*

Two-way Association

- Many associations are traversed in both directions, not usually with equal frequencies
- Three approaches for implementation
  - Implement one-way
  - Implement two-way
  - Implement with an association object

Testing

- Unit testing
- System testing

UNIT - 7 DESIGN PATTERNS – 1:

Syllabus : - 6hrs

- **What is a pattern**
- **what makes a pattern?**
- **Pattern categories;**
- **Relationships between patterns;**
- **Pattern description.**
- **Communication Patterns:**
- **Forwarder-Receiver;**
- **Client-Dispatcher-Server;**
- **Publisher-Subscriber.**

Patterns:-

❖ Patterns help you build on the collective experience of skilled software engineers.

❖ They capture existing, well-proven experience in software development and help to promote good design practice.

❖ Every pattern deals with a specific, recurring problem in the design or implementation of a software system.

❖ Patterns can be used to construct software architectures with specific properties

What is a Pattern?

- Abstracting from specific problem-solution pairs and distilling out common factors leads to patterns.
- These problem-solution pairs tend to fall into families of similar problems and solutions with each family exhibiting a pattern in both the problems and the solutions.

Definition :

The architect Christopher Alexander defines the term pattern as

❖ Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.