Object Oriented Modeling and Design Patterns
Advanced State Modeling

Nested State Diagrams:

- Problems with flat State Diagrams
- Expanding States
- Nested States
- Signal Generalization
- Concurrency

Problems with flat State Diagrams:

Conventional state diagrams are sufficient for describing simple systems but need additional power to handle large problems. E.g.: Consider an object with 'n' independent Boolean attributes that affect control. Representing such an object with a single flat state diagram would require 2^n states. For partitioning the state into n independent state diagrams, only 2n states are required.

Expanding States

One way to organize a model is by having a high-level diagram with subdiagrams expanding certain states. Figure 1 elaborates the dispense state with a lower-level state diagram called a submachine. A submachine is a state diagram that may be invoked as part of another state diagram.

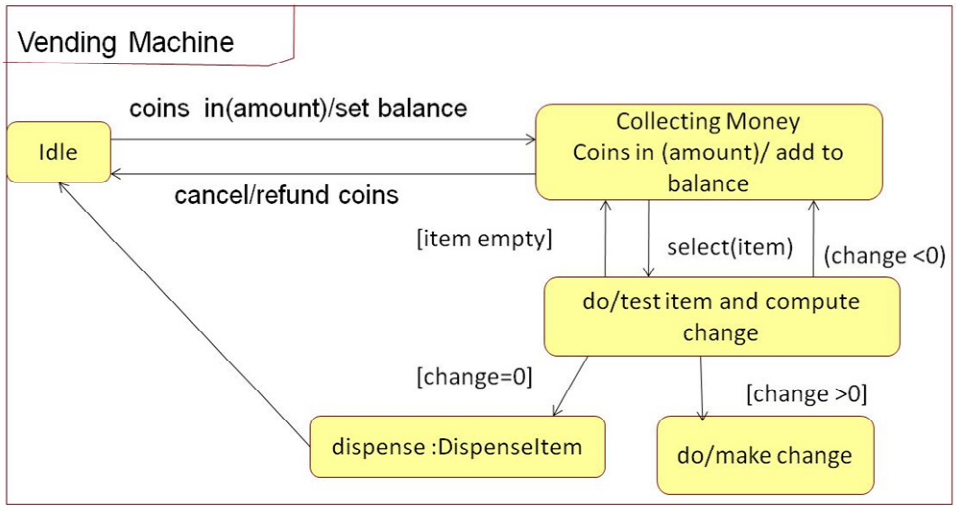


Figure-1: Vending Machine State Diagram

A submachine is a state diagram that may be invoked as part of another state diagram. E.g.: Vending Machine – High –level Diagram, DispenseItem - Subdiagram. The submachine replaces the local state.

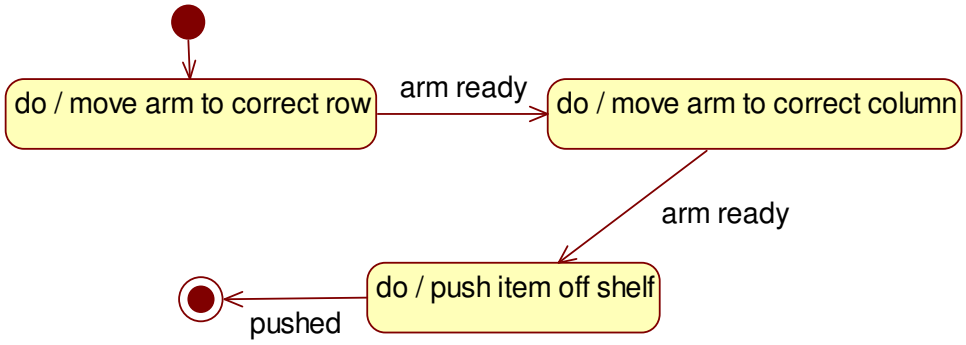


Figure-2: *DispenseItem* submachine of Vending Machine

Nested States

Second alternative for structuring states is using nested states. UML2 avoids using generalization in conjunction with states. Local state can be replaced with "nested states".

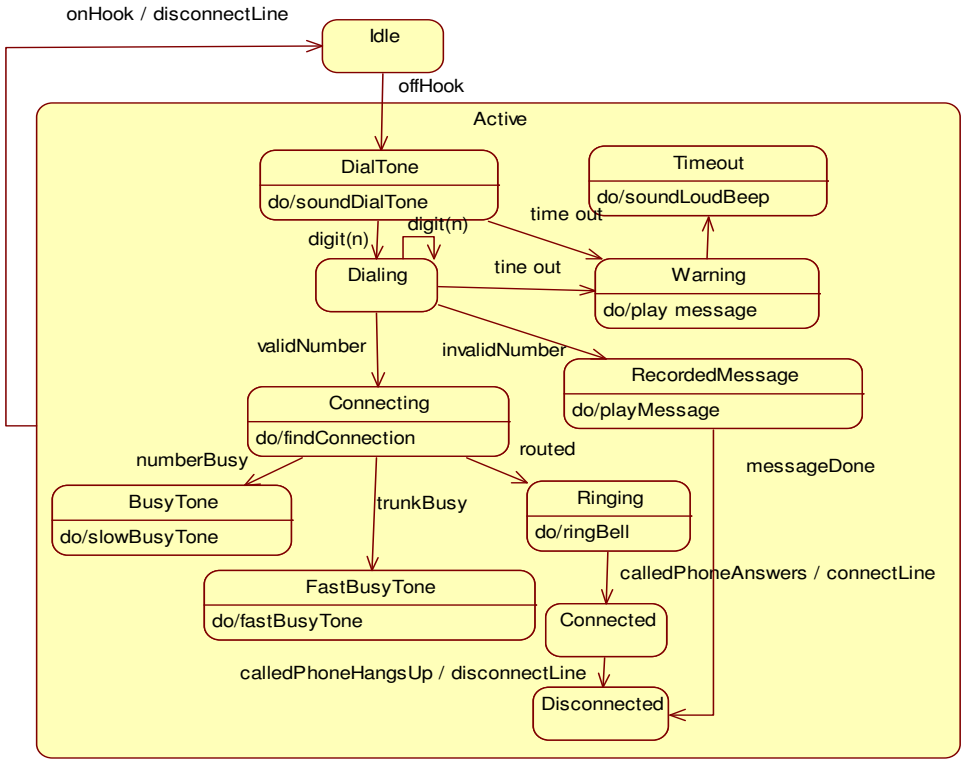


Figure-3: Nested states for phone line

Figure 3 simplifies the phone line model. The composite state name labels the outer contour that entirely encloses the nested states. A nested state receives the outgoing transitions of its composite state. States can be nested to an arbitrary depth (E. g: Car Transmission – Figure 4).

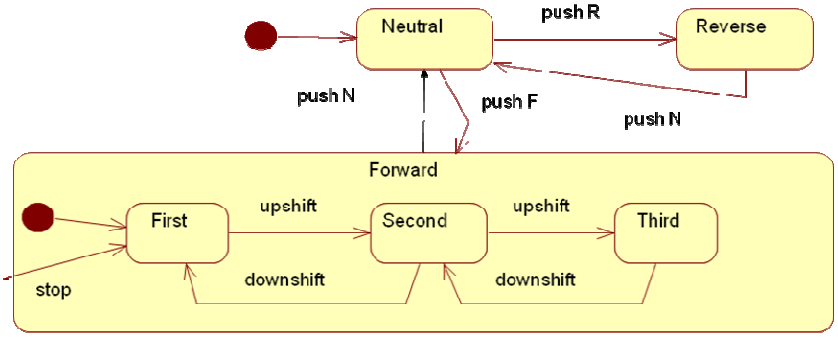


Figure-4: Nested states for Car Transmission

All three nested states share the transition on event *stop* from the *Forward* contour to state *First*.

- The nested states can be drawn as separate diagrams and reference them by including a submachine.
- Entry and exit activities are particularly useful in nested state diagrams because they permit a state.
- For a simple problem, implement nested states by degradation into "flat state" diagram.
- The entry activities are executed from the outside in and the exit activities from the inside out.

Signal Generalization

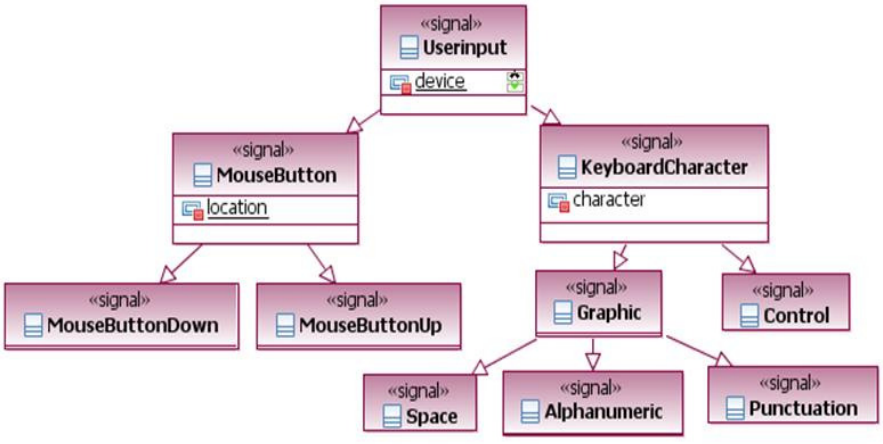


Figure-5: Partial hierarchy for keyboard signals

Signals can be organized into a generalization hierarchy with inheritance of signal attributes. It permits different levels of abstraction to be used in a model. E.g. Key board

signals - Partial hierarchy for keyboard signals. Figure 5 shows a part of a tree of input signals for a workstation.

- Two user inputs
  - MouseButton
  - KeyboardCharacter
- Inherited from MouseButton
  - MouseButtonDown
  - MouseButtonUp
- Inherited from KeyboardCharacter
  - Control
  - Graphic Character

Analogous to generalization of classes, all supersignals can be abstract.

Concurrency

The state model implicitly supports concurrency among objects.

- Aggregation Concurrency
- Concurrency within an Object
- Synchronization of Concurrent Activities
- Relation of Class and State Models

Aggregation Concurrency

The aggregate state corresponds to the combined states of all the parts. Aggregation is the "and-relationship". Transitions for one object can depend on another object being in a given state. Figure 6 shows the state of a *Car* as an aggregation of part states: *Ignition, Transmission, Accelerator,* and *Brake* (plus other unmentioned objects).

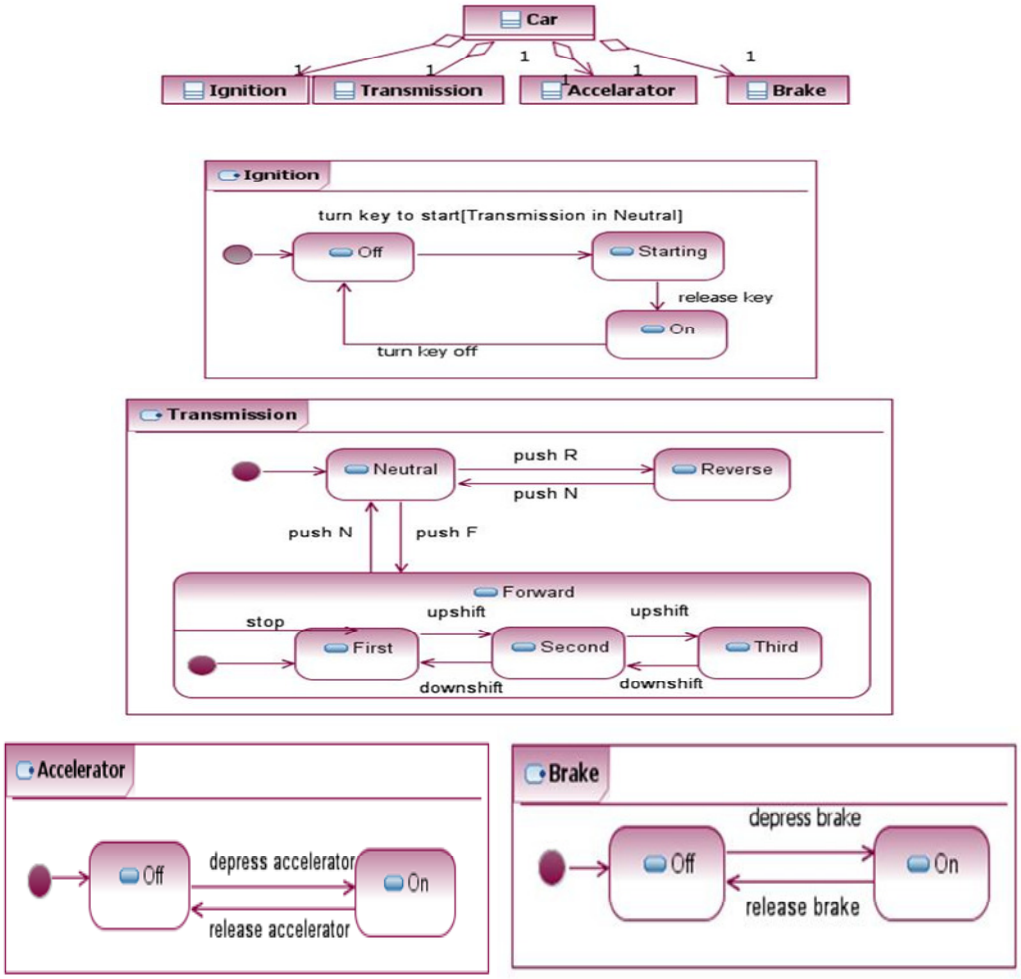


Figure-6: An aggregation and its concurrent state diagrams

Concurrency within an Object

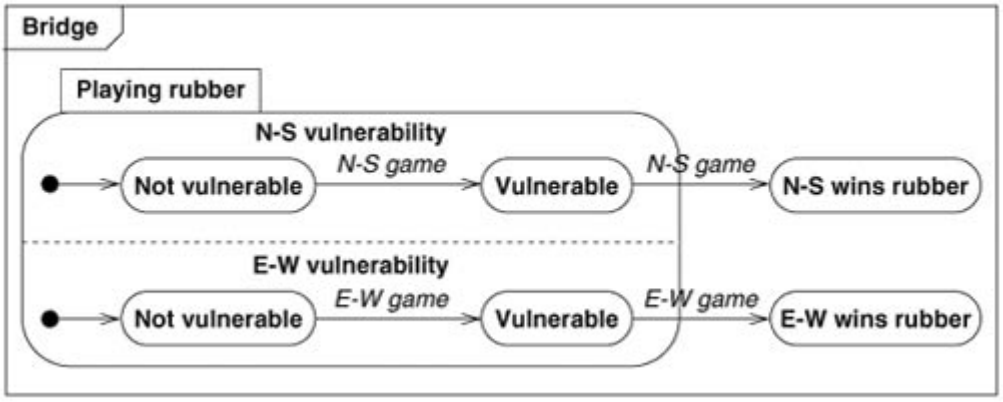


Figure-7: Bridge game with concurrent states

You can partition some objects into subsets of attributes or links, each of which has its own subdiagram. The state of the object comprises one state from each subdiagram. The UML shows concurrency within an object by partitioning the composite state diagram. Figure 7 shows the state diagram for the play of a bridge rubber. Most programming languages lack intrinsic support for concurrency.

Synchronization of Concurrent Activities

The object does not synchronize the internal steps of the activities but must complete both activities before it can progress to its next state.

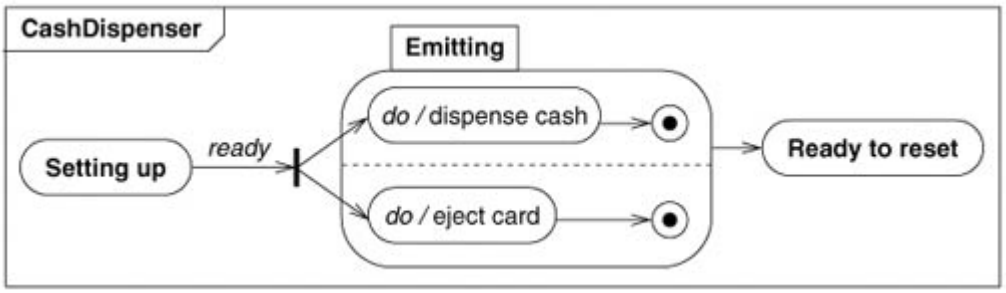


Figure-8: Synchronization of control

A cash dispensing machine dispenses cash and returns the user's card at the end of a transaction. Each region is a subdiagram that represents a concurrent activity within the composite activity.

If any subdiagrams in the composite state are not part of the merge, they automatically terminate when the merge transition fires. A completion transition fires when activity in the source state is complete. The firing of a merge transition causes a state diagram to perform the exit activities (if any) of all subdiagrams, in the case of both explicit and implicit merges.

A Sample State Model

The device controls a furnace and air conditioner according to time-dependent attributes that the owner enters using a pad of buttons.

While running, the thermostat operates the furnace of air conditioner to keep the current temperature equal to the target temperature. The target temperature is taken from a table of values at the beginning of each program period. The table specifies the target temperature and start time for eight different time periods, four on weekdays and four on weekends. The user can override the target temperature.

A pad of ten pushbuttons, three switches, sees parameters on an alphanumeric display. Each push button generates an event every time it is pushed. We assign one input event per button:

Button	**Input Event**
TEMP UP	Raises target temperature or program temperature
TEMP DOWN	Lowers target temperature or program temperature
TIME FWD	Advances clock time or program time
TIME BACK	Retards clock time or program time
SET CLOCK	Sets current time of day
SET DAY	Sets current day of the week
RUN PRGM	Leaves setup or program mode and runs the program
VIEW PRGAM	Enters program mode to examine and modify eight program time and program temperature settings
HOLD TEMP	Holds current target temperature in spite of the program
F-C BUTTON	Alternates temperature display between Fahrenheit and Celsius

Each switch supplies a parameter value chosen from two or three possibilities. The switches and their settings are:

Switch	**Setting**
NIGHT LIGHT	Lights the alphanumeric display. Values: light off, light on
SEASON	Specifies which device the thermostat controls. Values: heat (furnace), cool (air conditioner), off (none)
FAN	Specifies when the ventilation fan operates. Values: fan on (fan runs continuously), fan auto (fan runs only when furnace or air conditioner is operating)

The thermostat controls the furnace, air conditioner, and fan power relays. Run furnace, run air conditioner, run fan.

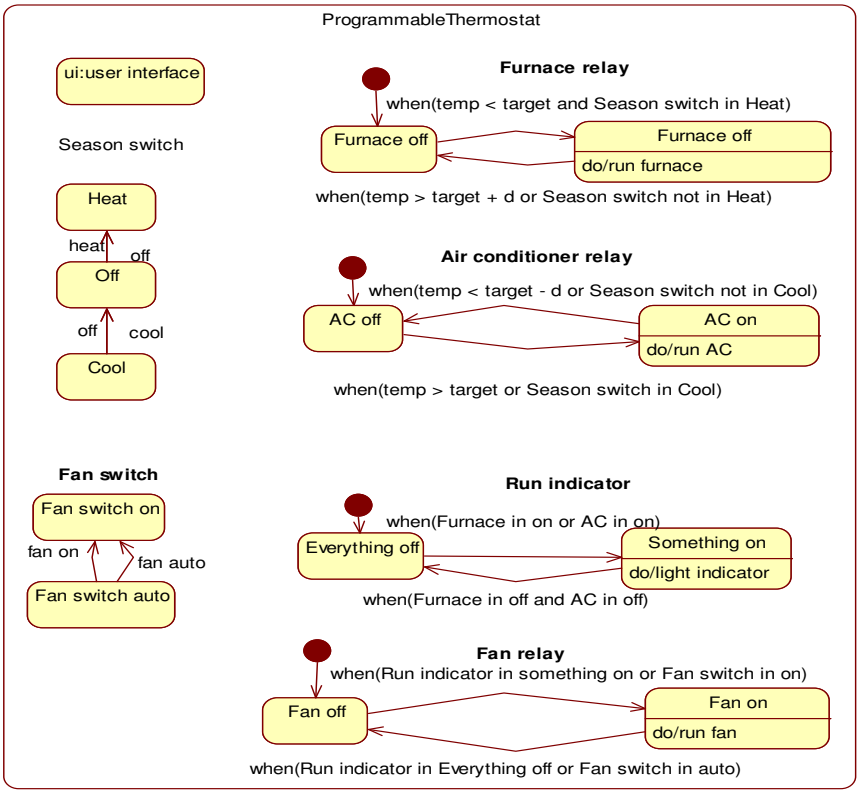


Figure-9: State diagram for programmable thermostat

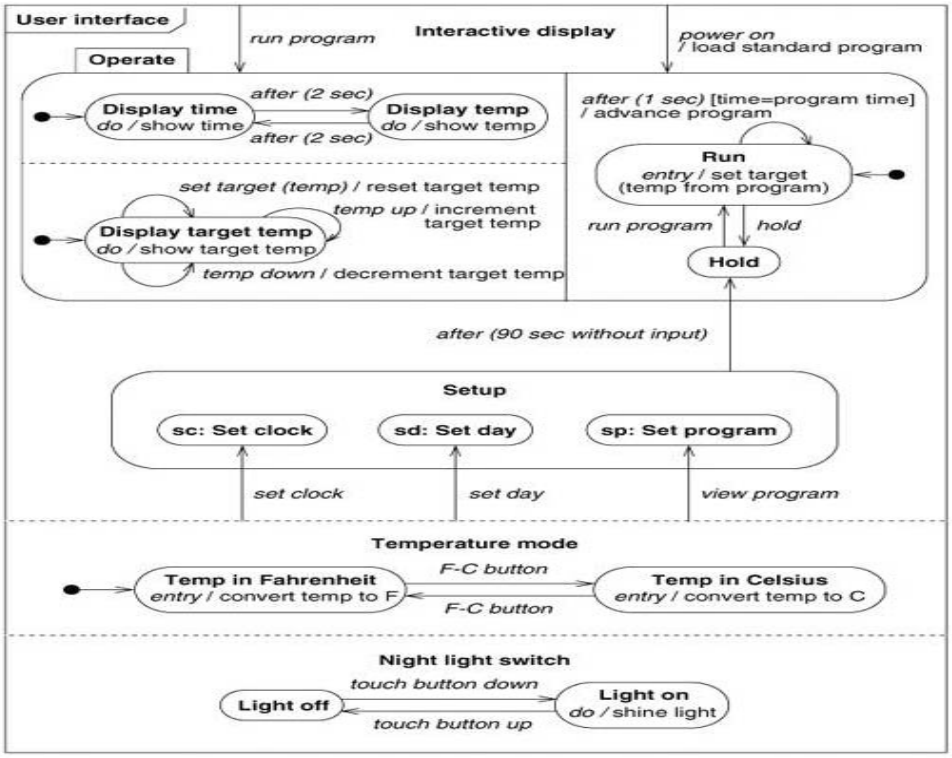


Figure-10: Subdiagram for thermostat user interface

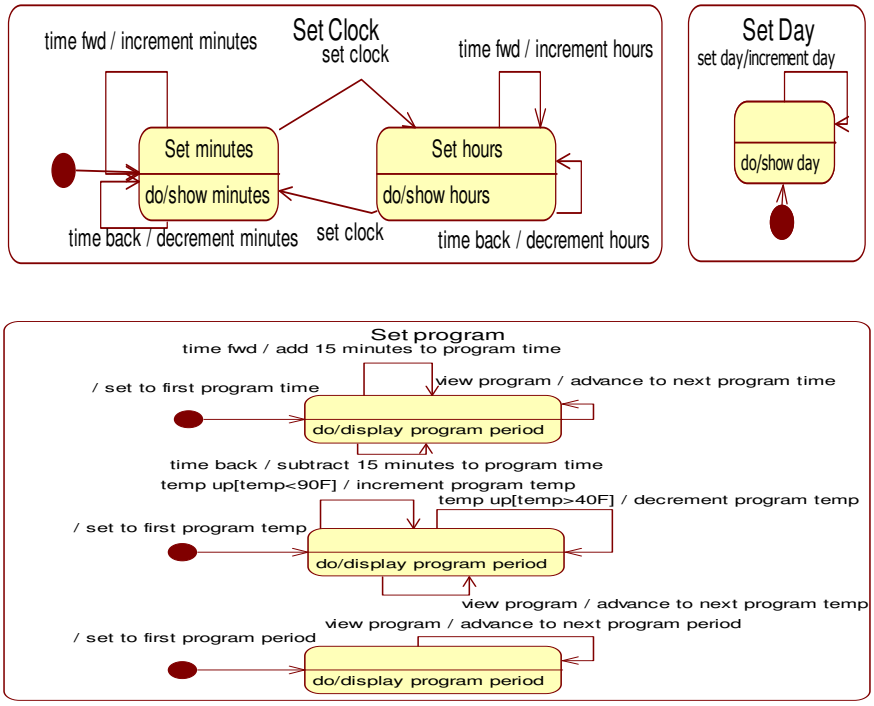


Figure-11: Subdiagram for thermostat user interface setup

Relation of Class and State Models

A state diagram describes all or part of the behavior of the objects of a given class. A single object can have different states over time – the object preserves its identity – but it cannot have different classes. There are three sources of concurrency within the class model:

- Aggregation of objects
- Aggregation within an object
- Concurrent behavior of an object

Aggregation within an object: the values and links of an object are its parts, and groups of them taken together define concurrent substates of the composite object state. The state model of a class is inherited by its subclasses. The state diagram of the subclass must be a refinement of the state diagram of the superclass. Transitions can often be implemented as operations on objects.

Interaction Modeling

The interaction model describes how the objects interact. Behavior of the system is needed to be described by State model and Interaction model. Use cases are helpful for capturing informal requirements. Sequence diagrams are good for showing the behavior sequences seen by users of a system. Activity diagrams can show data flows as well as control flows.

Use Case Models – Actors:

- An actor is a direct external user of a system. Each actor represents those objects that behave in a particular way toward the system.
  - E. g Travel agency system
    - Actors - traveler, agent, and airline
  - Vending machine
    - Actors - Customer, repair technician
- An object can be bound to multiple actors if it has different facets to its behavior.
  - Ex: Mary, Frank, Paul may be Customers of a vending machine – Paul may also be the repair technician interacts with the machine.
- An actor is directly connected to the system
  - E. g The dispatcher of repair technicians from a service bureau is not an actor of a vending machine
  - Only the repair technician interacts directly with the machine and he is an actor

Use Case Models - Use Cases:

- A use case is a coherent piece of functionality that a system can provide by interacting with actors
- Example
  - A *customer* actor can *buy a beverage* from a vending machine
  - *Repair technician* can perform *scheduled maintenance* on a vending machine.
- Use case summaries for a vending machine:
  - *Buy a beverage***.** The vending machine delivers a beverage after a customer selects and pays for it.
  - *Perform scheduled maintenance*. A repair technician performs the periodic service on the vending machine necessary to keep it in good working condition.

- *Make repairs***.** A repair technician performs the unexpected service on the vending machine necessary to repair a problem in its operation.
- *Load items***.** A stock clerk adds items into the vending machine to replenish its stock of beverages.

- Each use case involves one or more actors as well as the system itself
  - Example - In a telephone system, the use case *make a call* involves two actors, a *caller* and a *receiver*
- The actors need not be all persons
  - Example - The use case *make a trade* on an online stock broker involves a *customer* actor an a *stock exchange* actor
- A use case involves a sequence of messages among the system and its actors. This can be repeated several times
  - Example - In the *buy a beverage* use case, the *customer* first inserts a coin and the *vending machine* displays the amount deposited
- Define *a mainline behavior sequence* first, then define *optional subsequences*, *repetitions, and other variations*
  - Example - A customer can deposit a variable number of coins in the *buy a beverage* use case. Depending on the money inserted and the item selected the machine may or may not return change
- Error conditions are also part of a use case.
  - Example - If the customer selects a beverage whose supply is exhausted, the vending machine displays a warning message
- A use case brings together all of the behavior relevant to a slice of system functionality.
- Normal mainline behavior, variations on normal behavior, exception conditions, error conditions and cancellations of a request
- **Use case description - *buy a beverage*:**
  - **Use Case:** Buy a beverage
  - **Summary:** The vending machine delivers a beverage after a customer selects and pays for it
  - **Actors:** Customer
  - **Preconditions:** The machine is waiting for money to be inserted
  - **Description:** The machine starts in the waiting state in which it displays the message “Enter coins.” A customer inserts coin into the machine. The machine displays the total value of money entered and lights up the buttons for the items that can be purchased for the money inserted. The customer pushes a button. The machine dispenses the corresponding item and makes change, if the cost of the item is less than the money inserted.

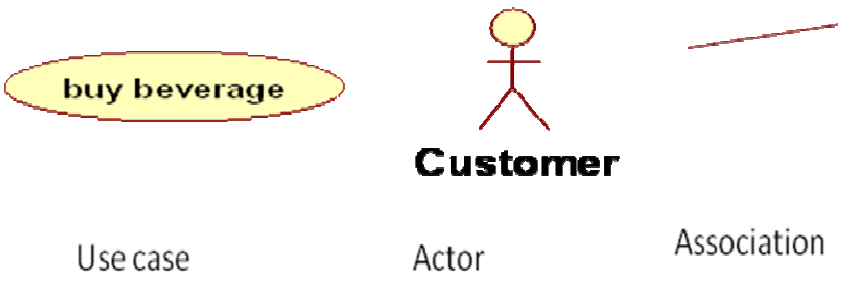
- **Exceptions:**
    - *Canceled:* If the customer presses the cancel button before an item has been selected, the customer's money is returned and the machine resets to the waiting state.
    - *Out of stock:* If the customer presses a button for an out-of-stock item, the message "The item is out of stock" is displayed. The machine continues to accept coins or a selection.
- **Exceptions:**
    - *Insufficient money:* If the customer presses a button for an item that costs more than the money inserted, the message "You must insert $nn.nn more for that item" is displayed, where nn.nn is the amount of additional money needed. The machine continues to accept coins or a selection
- **Exceptions:**
    - *No change:* If the customer has inserted enough money to buy the item but the machine cannot make the correct change, the message "Cannot make correct change" is displayed and the machine continues to accept coins or a selection.
- **Postconditions:** The machine is waiting for money to be inserted.

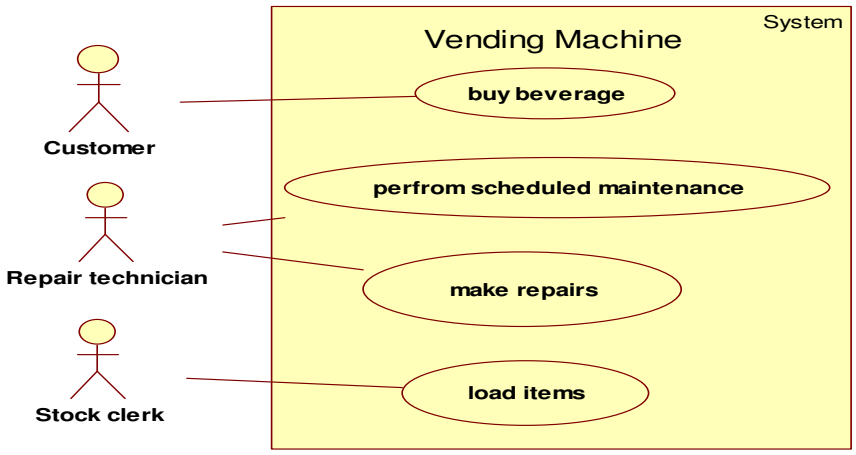
- In a complete model, the use cases partition the functionality of the system
- They should preferably all be at a comparable level of abstraction
- Example:
    - Use cases *make telephone call* & *record voice mail message* are at the same level of abstraction
    - The use case *set external speaker volume to high* is too narrow
    - *set speaker volume* or *set telephone parameters* – same level of abstraction as *make telephone call*

Use Case Diagrams:

- A system involves a set of use cases and a set of actors
- Set of use cases shows the complete functionality of the system at some level of detail
- Set of actors represents the complete set of objects that the system can serve

Use Case Diagrams – elements:





Guidelines for Use Case Models:

Use Cases identifies the functionality of a system and organize it according to the perspective of users. Use Cases describe complete transactions and are therefore less likely to omit necessary steps.

- First determine the system
- Ensure that actors are focused
- Each use case must provide value to users. For example, *dial a telephone number* is not a good use case for a telephone system
- Relate use cases and actors
- Remember that use cases are informal
- Use cases can be structured

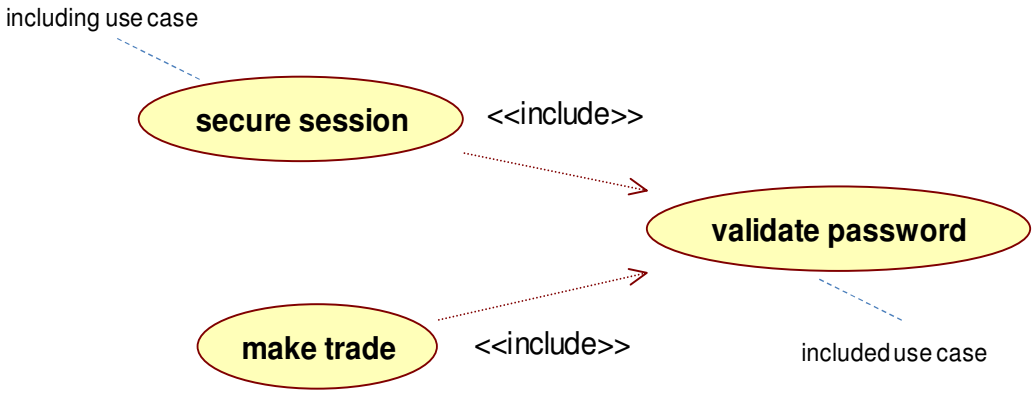
Use Case Relationships:

Use cases can be structured for large applications. Complex use cases can be built from smaller pieces with the following relationships:

- *include*
- *extend*
- *generalization*

***include* Relationship:**

- The *include* relationship incorporates one use case within the behavior sequence of another use case
- The UML notation for an *include* relationship is a dashed arrow from the source use case to the target (included) use case
- The keyword <<*include*>> annotates the arrow
- A use case can also be inserted within a textual description with the notation include use-case-name.
- Factoring a use case into pieces is appropriate when the pieces represent significant behavior units.

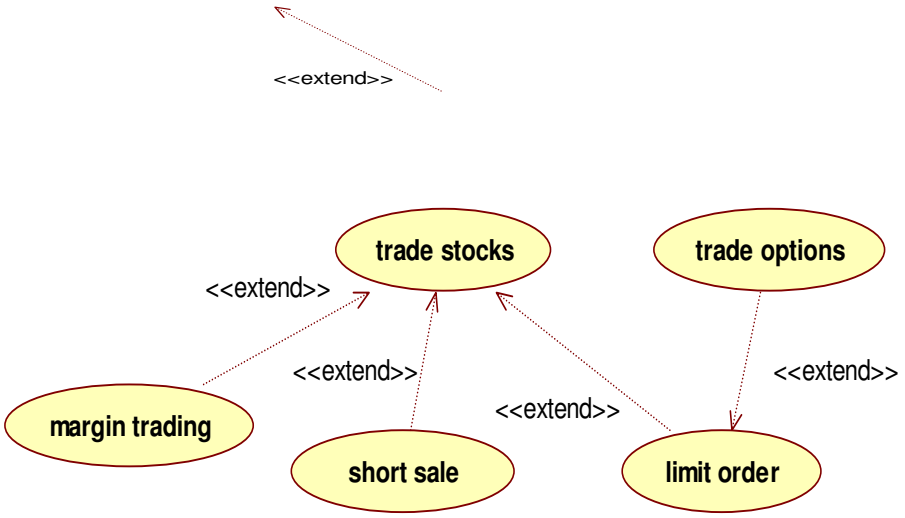


- A use case can also be inserted within a textual description with the notation include use-case-name.
- Factoring a use case into pieces is appropriate when the pieces represent significant behavior units.

***extend* Relationship:**

- The *extend* relationship adds incremental behavior to a use case
- It represents the frequent situation in which some initial capability is defined, and later features are added modularly
- Example
    - A stock brokerage system might have the base use case *trade stocks*, which permits a customer to purchase stocks for cash on hand in the account.
    - The additional behavior is inserted at the point where the purchase cost is checked against the account balance.
- The UML notation for extend relationship is a dashed arrow from the extension use case to the base use case

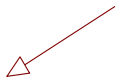
- The keyword <<*extend*>> annotates the arrow

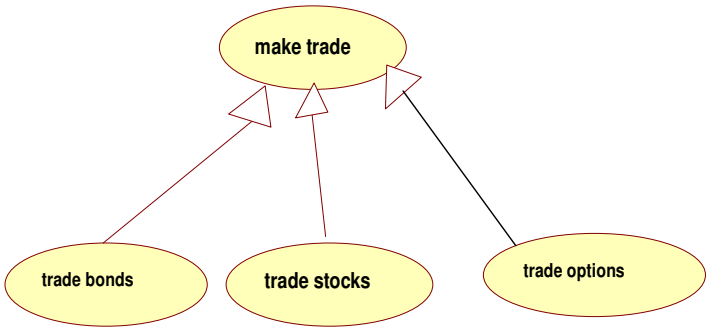


- The extend relationship connects an extension use case to a base use case
- The base use case must be a valid use case in the absence of any extensions.
- The behavior sequence of the extension use case occurs at the given point in the sequence.
- The extension behavior occurs only if the condition is true when control reaches the insert location.

Use cases – Generalization:

- Generalization can show specific variations on a general use case, analogous to generalization among classes
- The UML indicates generalization by an arrow with its tail on the child use case and a triangular arrowhead on the parent use case, the same notation that is used for classes



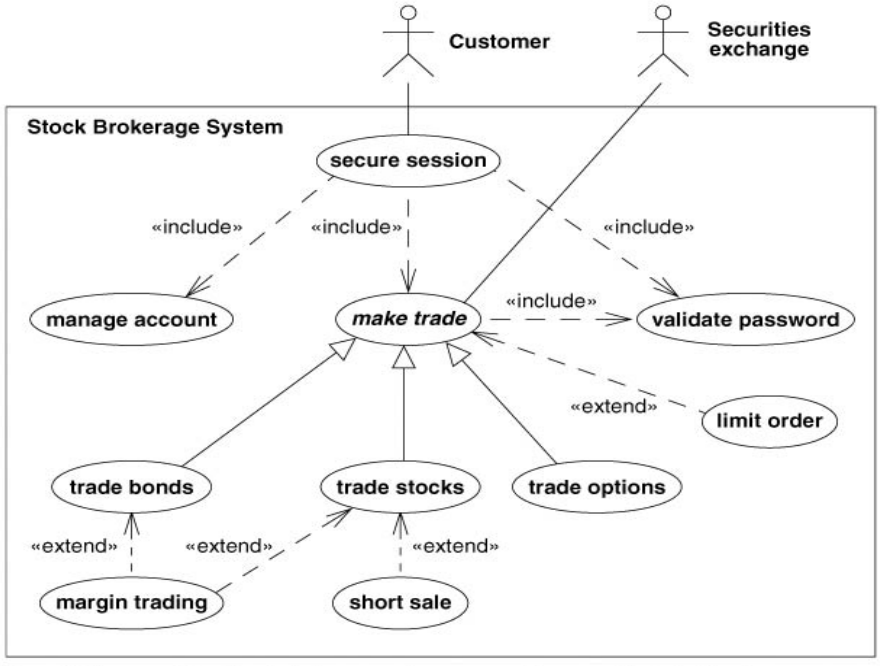


- A parent use case may be abstract or concrete
  - An abstract use case cannot be used directly
- Use cases also exhibit polymorphism
  - A child use case can freely substitute for a parent use case.
- Use case generalization is more complicated than class generalization.

Combinations of the Use Case Relationships:

- A single diagram may combine several kinds of use case relationships

Use case relationships:



- A parent use case may be abstract or concrete
  - An abstract use case cannot be used directly
- Use cases also exhibit polymorphism
  - A child use case can freely substitute for a parent use case.
- Use case generalization is more complicated than class generalization

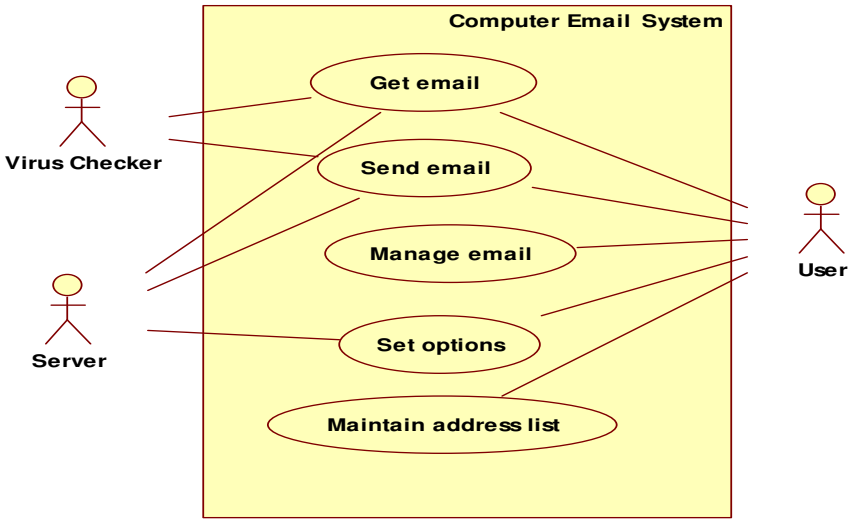
Guidelines for Use case Relationships:

- Use case generalization.
- Use case inclusion.
- Use case extension
- Include relationship vs. extend relationship

Use case diagram – Example1:

- Consider a computer email system.
  - List three actors. Explain the relevance of each actor.
  - One use case is to get email. List four additional use cases at a comparable level of abstraction. Summarize the purpose of each use case with a sentence.
  - Prepare a use case diagram for a computer email system.
- Actors:
  - **User:** A person who is the focus of services
  - **Server:** A computer that communicates with the email system and is the intermediate source and destination of email messages.
  - **Virus checker:** Software that protects against damage by a malicious user
- Use cases
  - **Get email:** Retrieve email from the server and display it for the user
  - **Send email:** Take a message that was composed by the user and send it to the server
  - **Manage email:** Perform tasks such as deleting email, managing folders, and moving email between folders
  - **Set options:** Do miscellaneous things to tailor preferences to the user's liking.
  - **Maintain address list:** Keep a list of email addresses that are important to the user

Use case diagram – Example1

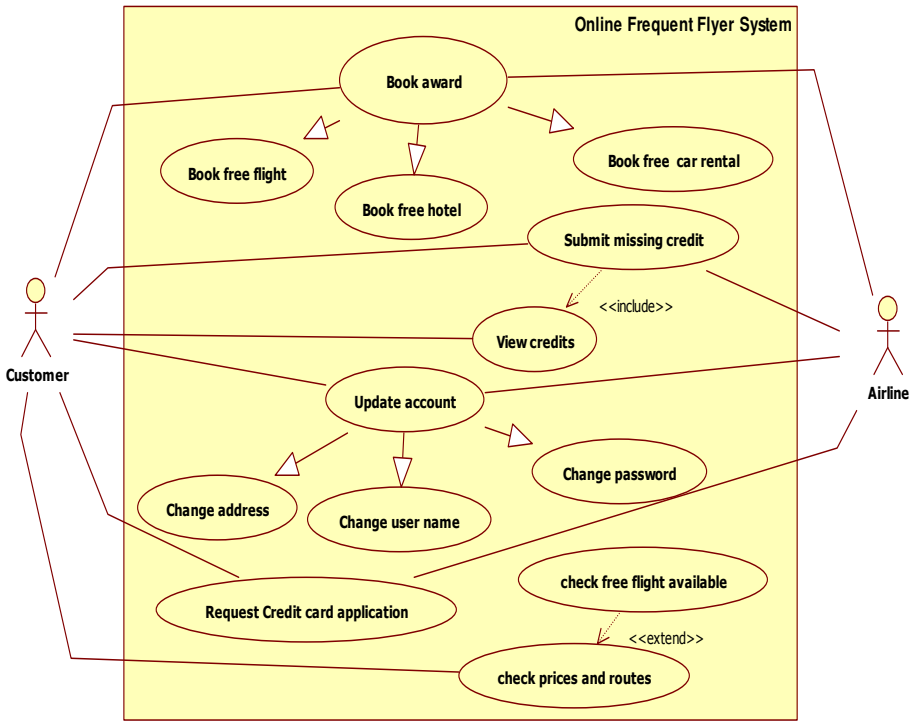


Use case diagram – Example2:

- Consider an online frequent flyer program. Prepare a use case diagram and include the appropriate relationships for the use cases listed below:
  - **View credits.** View the frequent flyer points currently available in the account
  - **Submit missing credit.** Request credit for an activity that was not credited
  - **Change address.** Submit a new mailing address
  - **Change user name.** Change the user name for the account
  - **Change password.** Change the password for the account
  - **Book a free flight**. Use frequent flyer credits to obtain a free flight
  - **Book a free hotel**. Use frequent flyer credits to obtain a free hotel
  - **Book a free rental car**. Use frequent flyer credits to obtain a free rental car
  - **Request frequent flyer credit card**. Fill out an application for a credit card that gives frequent flyer points as a bonus for purchases
  - **Check prices and routes**. Find possible routings and corresponding prices for a paid flight
  - **Check availability for a free flight**. Check availability of free travel for a specified flight

Hint

Finding a free flight is the same as finding a paid flight, except seats are limited for free flights (hence the *extend* relationship). When submitting a claim for missing credits, a user must first view their existing credits (hence the *include* relationship).



Use case diagram – Exercise 1:

- Consider a physical bookstore, such as in shopping mall.
  - List three actors that are involved in the design of a checkout system. Explain the relevance of each actor
  - One use case is the purchase of items. Take the perspective of a customer and list another use case at a comparable level of abstraction. Summarize the purpose of each use case with a sentence.
  - Prepare a use case diagram

  - One use case is the purchase of items. Take the perspective of a customer and list another use case at a comparable level of abstraction. Summarize the purpose of each use case with a sentence.
  - Prepare a use case diagram

Use case diagram – Exercise 2:

- You are interacting with an online travel agent and encounter the following use cases. Prepare a use case diagram, using the generalization and include relationships.

- **Purchase a flight.** Reserve a flight and provide payment and address information.
- **Provide payment information.** Provide a credit card to pay for the incurred changes.
- **Provide address.** Provide mailing and residence address.
- **Purchase car rental.** Reserve a rental car and provide payment and address information.
- **Purchase a hotel stay.** Reserve a hotel room and provide payment and address information.
- **Make a Purchase.** Make a travel purchase and provide payment and address information.

Sequence Models

The sequence model elaborates the themes of use cases. Two kinds of sequence models:

- scenarios and
- more structured format called *sequence diagrams*

Scenario:

A scenario is a sequence of events that occurs during one particular execution of a system, such as for a use case. A scenario can be displayed as a list of text statement.
Scenario for a session with an online stock broker:

- John Doe logs in
- System establishes secure communications
- System displays portfolio information
- John Doe enters a buy order for 100 shares of GE at the market price
- System verifies sufficient funds for purchase
- System displays confirmation screen with estimated cost
- John Doe confirms purchase
- System places order on securities exchange
- System displays transaction tracking number
- John Doe logs out
- System establishes insecure communications
- System displays good-bye screen
- System exchange reports results of trade

At early stages of development, scenarios are expressed at a high level. A scenario contains messages between objects as well as activities performed by objects.

- The first step of writing a scenario is to identify the objects exchanging messages.
- Determine the sender and receiver of each message and sequence of messages
- Add activities for internal computations

Sequence diagram:

A sequence diagram shows the participants in an interaction and the sequence of messages among them. Each actor as well as the system is represented by a vertical line called a lifeline and each message by a horizontal arrow from the sender to the receiver.
Note: sequence diagrams can show concurrent signals.

Each use case requires one or more sequence diagrams to describe its behavior.

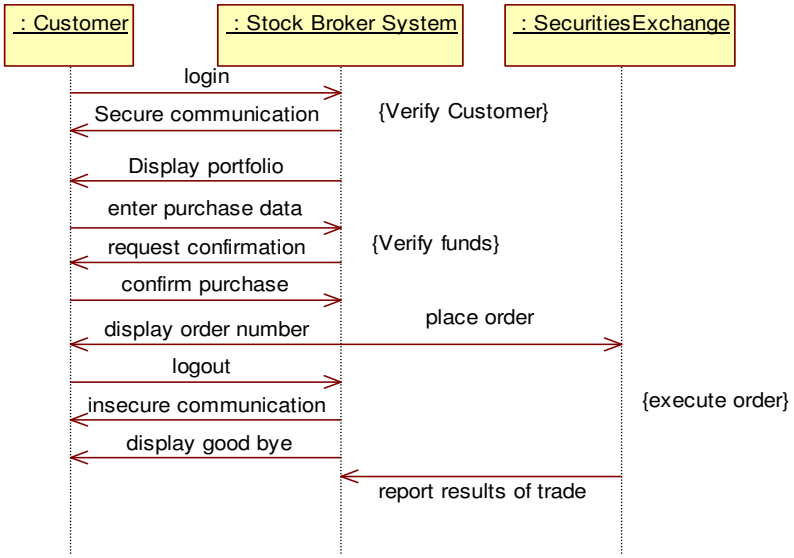
- Prepare one sequence diagram for each major flow of control
- Draw a separate sequence diagram for each task.
- Prepare a sequence diagram for each exception condition within the use case
- Try to elaborate all the use cases and cover the basic kinds of behavior with sequence diagrams

Guidelines for Sequence Models

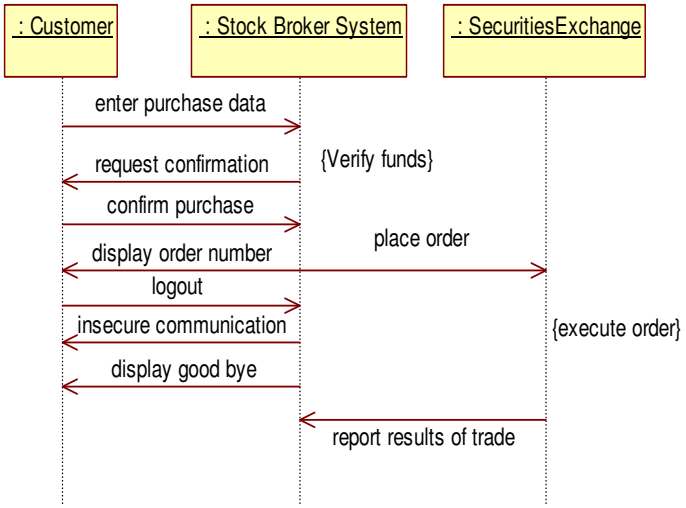
- Prepare at least one scenario per use case
- Abstract the scenarios into sequence diagrams

- Divide complex interactions
- Prepare a sequence diagram for each error condition

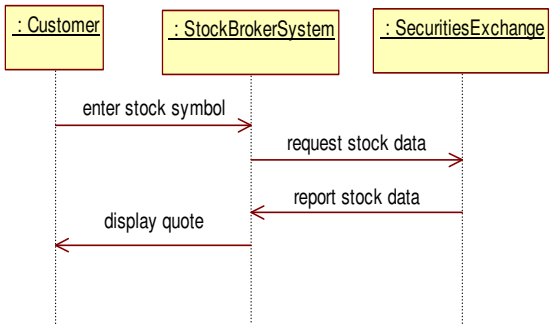
Sequence diagram for a session with online stock broker:



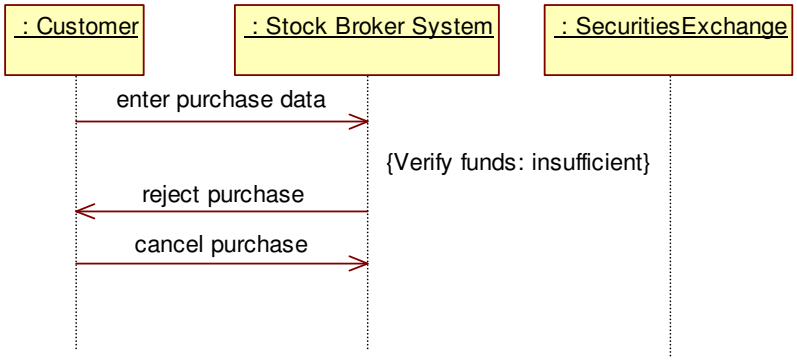
Sequence diagram for a stock purchase:



Sequence diagram for a stock quote:



Sequence diagram for a stock purchase that fails:



Sequence diagram – Example:

- Normal scenario for each use case
- **Get email**
  - User logs in to email system.
  - System displays the mail in the *Inbox* folder.
  - User requests that system get new email.
  - System requests new email from server.
  - Server returns new email to system.
  - System displays the mail in the *Inbox* folder and highlights unread messages.

Sequence Diagrams with Passive Objects:

Most objects are passive and do not have their own threads of control. Activation shows the time period during which a call of a method is being processed, including the time when the called method has invoked another operation.

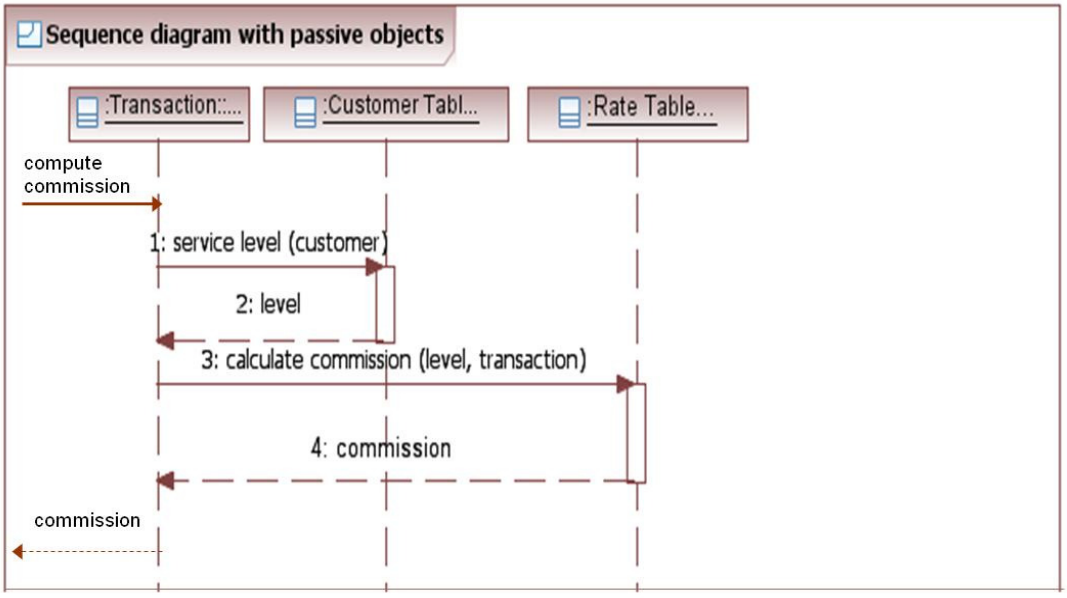
Sequence Diagrams with Transient Objects:

Object*A* is an active object that initiates an operation. The notation for a call is an arrow from the calling activation created by the call. Activation, therefore, has a call arrow coming into its top and a return arrow leaving its bottom. If an object does not exist at the beginning of a sequence diagram, then it must be created during the sequence diagram. The UML shows creation by placing the object symbol at the head of the arrow for the call that creates the object. Conditionals on a sequence diagram also can be shown.

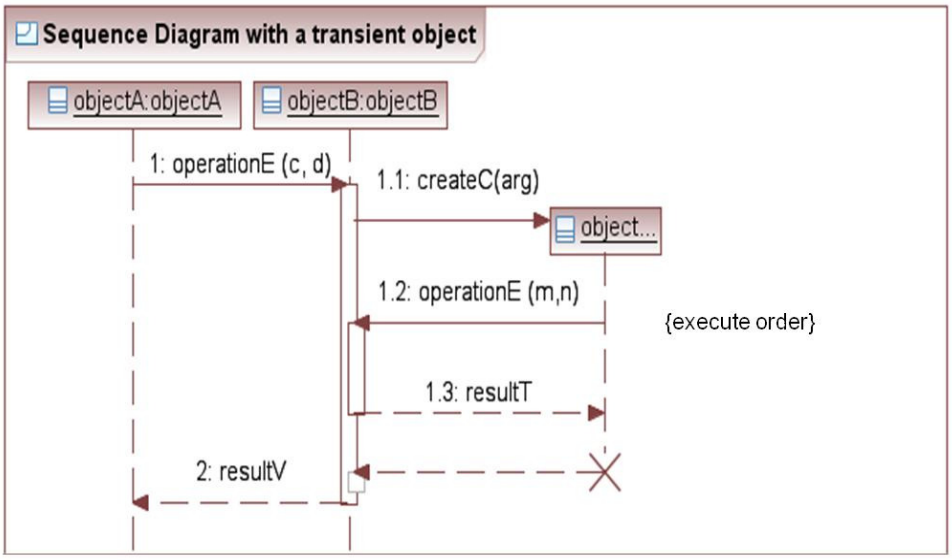
Guidelines for Procedural Sequence Models:

- Active vs. passive objects.
  - By definition, active objects are always activated and have their own focus of control.
- Advanced features.
  - Only show implementation details for difficult or especially important sequence diagrams.

Sequence diagram with passive objects:



Sequence diagram with a transient object:



Activity Models

An activity diagram shows the sequence of steps that make up a complex process, but focuses on operations rather than on objects. Activity diagrams are most useful during the early stages of designing algorithms and workflows.

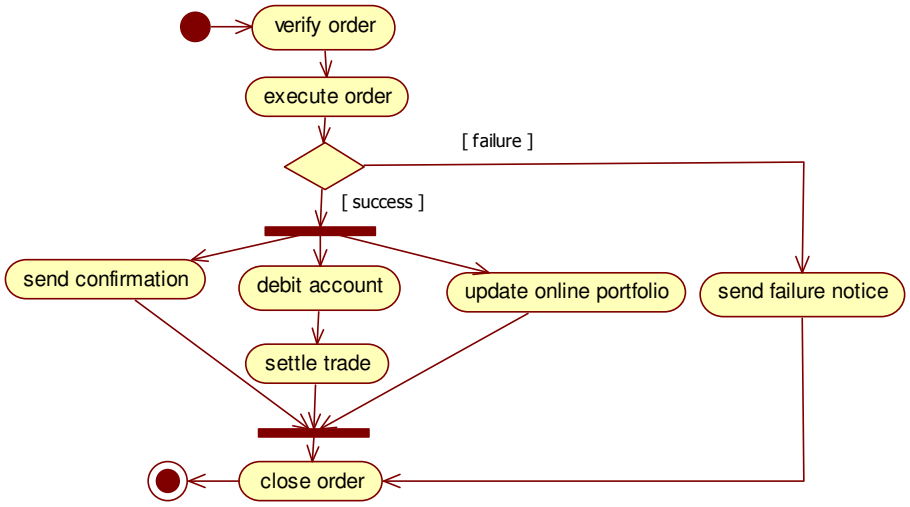
Activities:

The steps of an activity diagram are operations, specifically activities from the state model.

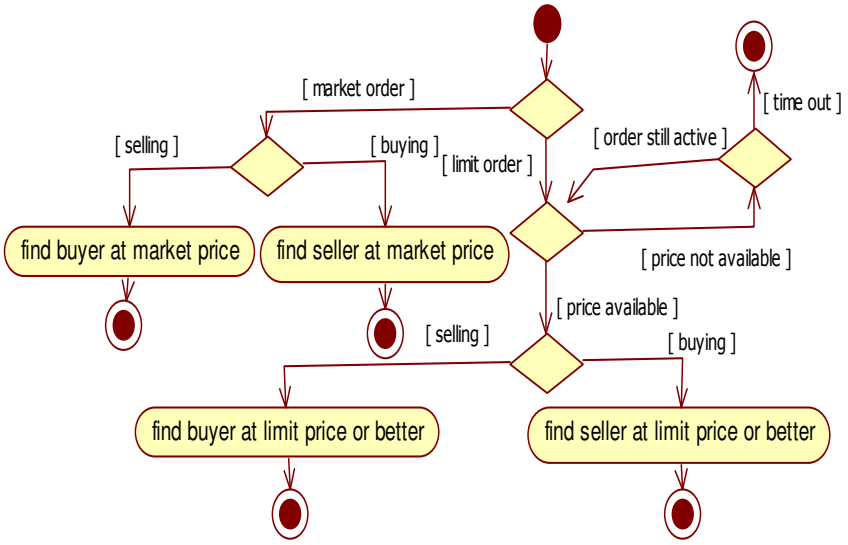
Executable Activity Diagrams:

An activity token can be placed on an activity symbol to indicate that it is executing. Multiple tokens can arise through concurrency.

Activity diagram for stock trade processing:



Activity diagram for *execute order:*



Executable Activity Diagrams

An activity token can be placed on an activity symbol to indicate that it is executing. Multiple tokens can arise through concurrency.

Guidelines for Activity Models:

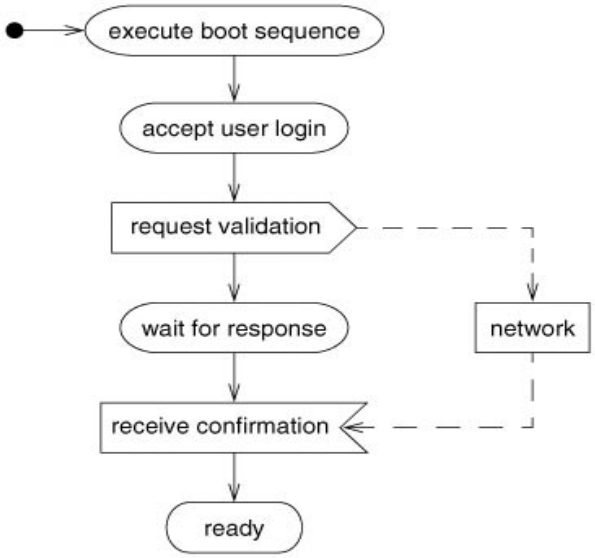
- Don't misuse activity diagrams.
  - Activity diagrams are intended to elaborate use case should not be used as an excuse to develop software via flowcharts.
- Level diagrams
- Consider executable activity diagrams

Sending and Receiving Signals

The UML shows the sending of a signal as a convex pentagon. The UML shows the receiving of a signal as a concave pentagon

- Example: Consider a workstation that is turned on. It goes through a boot sequence and then requests that the user login. After entry of a name and password, the workstation queries the network to validate the user. Upon validation, the workstation then finishes its startup process. Figure 8.7 shows the corresponding activity diagram

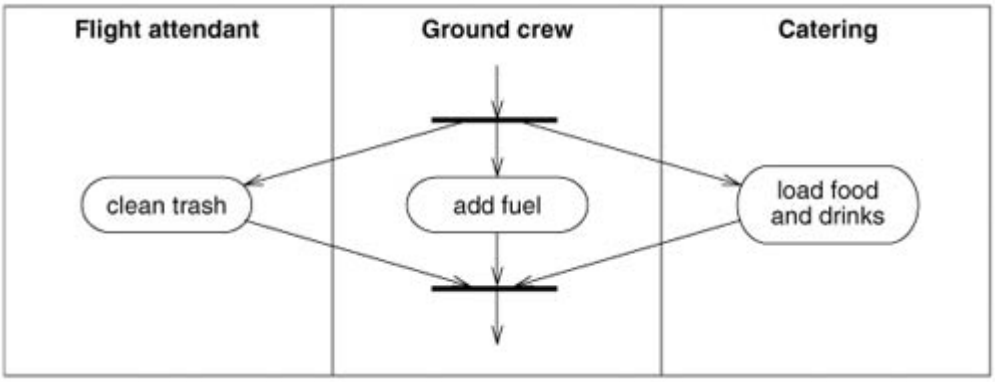
Activity diagram with signals:



Swimlanes:

- In a business model, it is often useful to know which human organization is responsible for an activity.
- Example: Sales, finance, marketing, and purchasing
- It is sufficient to partition the activities among organizations.
- Lines across swimlane boundaries indicate interactions among different organizations

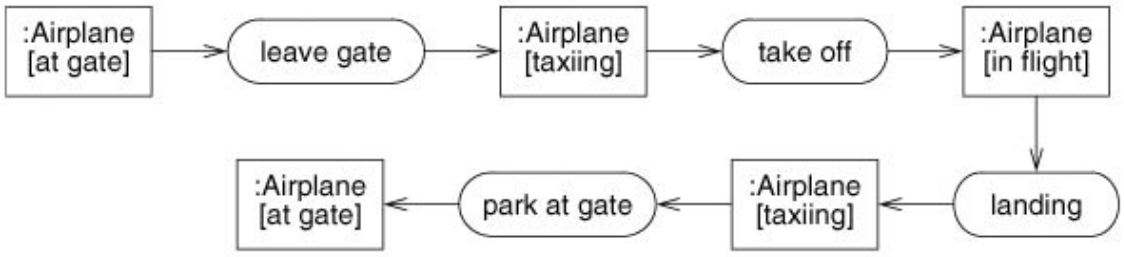
Activity diagram with swimlanes:



Object Flows

It is helpful to see the relationships between an operation and the objects that are its argument values or results. An input or output arrow implies a control flow Frequently the same object goes through several states during the execution of any activity diagram. UML shows an object value in a particular state by placing the state name in square brackets following the object name

Activity diagram with object flows:



Concepts Summary:

- Three different views: Class model, state model and interaction model
  - Class Model-describes objects in the system and their relationships
  - State model - life history of the objects
  - Interaction model-Interaction among objects

Class Design

Review of Analysis:

- Steps : Domain Analysis , Application Analysis
- Domain Class Model , Application class Model
- Domain Class Model:

  - Find Classes
  - Prepare data dictionary
  - Find Associations
  - Find attributes of objects and links
  - Organize and simplify classes using inheritance
  - Verify that access paths exists for likely queries
  - Iterate and refine the model
  - Reconsider the level of abstraction

- Group classes into packages

- Application Class Model

- Specify user Interface

- Define boundary classes

- Define controllers

- Check against the interaction model

Class Design:

The purpose of class design is to complete the definitions of the classes and associations and choose algorithms for operations. The OO paradigm applies equally well in describing the real-world specification and computer-based implementation.

Overview of Class Design:

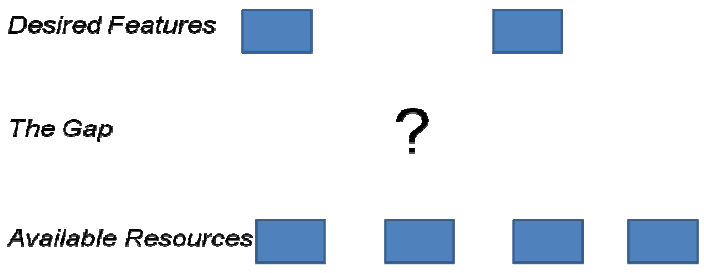
The analysis model describes the information that the system must contain and the high-level operations that it must perform. The simplest and best approach is to carry the analysis classes directly into design.

During design, choose among the different ways to realize the analysis classes for minimizing execution time, memory, and other cost measures. New classes may be introduced to store intermediate results during program execution and avoid recomputation. OO design is an iterative process.

Class design involves the following steps.

- Bridge the gap from high-level requirements to low-level services.
- Realize use cases with operations
- Formulate an algorithm for each operation.
- Recurse downward to design operations that support higher-level operations.
- Refactor the model for a cleaner design.
- Optimize access paths to data.
- Reify behavior that must be manipulated.
- Adjust class structure to increase inheritance.
- Organize classes and associations.

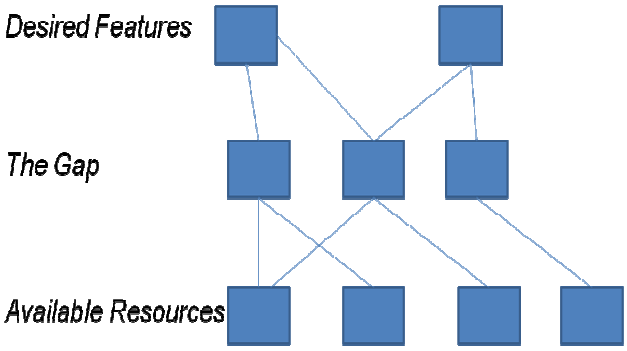
Bridging the Gap: The Design Gap



Resources include the operating system infrastructure, class libraries, and previous applications. If you can directly construct each feature from the resources, you are done.

- *A Web-based ordering system.*

  The system cannot be built readily from a spreadsheet or a programming language, because there is too big a gap between the features and the resources.



If the gap is large, the intermediate elements need to be organized into multiple levels. The intermediate elements may be operations, classes, or other UML constructs.

There can be many ways to decompose a high-level operation. If the intermediate elements have already been built, directly use them, but the principle of bridging the gap is the same. Design is difficult because it is not a pure analytic task.

Design requires synthesis: You have to invent new intermediate elements and try to fit them together. It is a creative task, like solving puzzles, proving theorems, playing chess, building bridges, or writing symphonies.

Example-Web Application:

- During analysis, we were content to leave the interface between the actor and the system—as expressed in the interaction diagrams—as user interface
- During design, the interface needs to be elaborated into a set of specific interfaces capable of handling the communication between the actors and the system, as well as supporting the flow of activity of business processes
- In addition to elaborating the classes and collaborations, design activities include
  - Partitioning objects into client, server, and other tiers
  - Separating and defining user interfaces, or Web pages
- Intermediate elements
  - A client page instance is an HTML-formatted Web page with a mix of data, presentation, and even logic. Client pages are rendered by client browsers and may contain scripts that are interpreted by the browser. Client page functions map to functions in tags in the page. Client page

attributes map to variables declared in the page's script tags that are accessible by any function in the page, or page scoped. Client pages can have associations with other client or server pages.

- A server page represents a dynamic Web page that contains content assembled on the server each time it is requested. Typically, a server page contains scripts that are executed by the server that interacts with server-side resources: databases, business logic components, external systems, and so on. The object's operations represent the functions in the script, and its attributes represent the variables that are visible in the page's scope, accessible by all functions in the page.
- Example intermediate elements for Web Application Design:
    - The Web Application Extension (WAE) for UML is a set of stereotypes, tagged values, and constraints that allows Web applications to be fully modeled with UML.
    - The core class elements of the WAE are «server page», «client page», and «HTML form».
    - The core association elements are «link», «build», «forward», «redirect», and «object».
    - The core component view elements are the «static page» and «dynamic page» components and the «physical root» package.
    - Properly partitioning the business objects in a Web application is critical and depends on the architecture.
    - Mappings between the User experience(UX) and design models are useful and help establish the contract between the UX and engineering teams

Realizing Use Cases:

During class design, elaborate the complex operations, most of which come from use cases. Design is the process of realizing functionality while balancing conflicting needs. During design invent new operations and new objects that provide this behavior. Inventing the right intermediate operations is what we have called “bridging the gap”. List the responsibilities of a use case or operation. A responsibility is something that an object knows or something it must do.

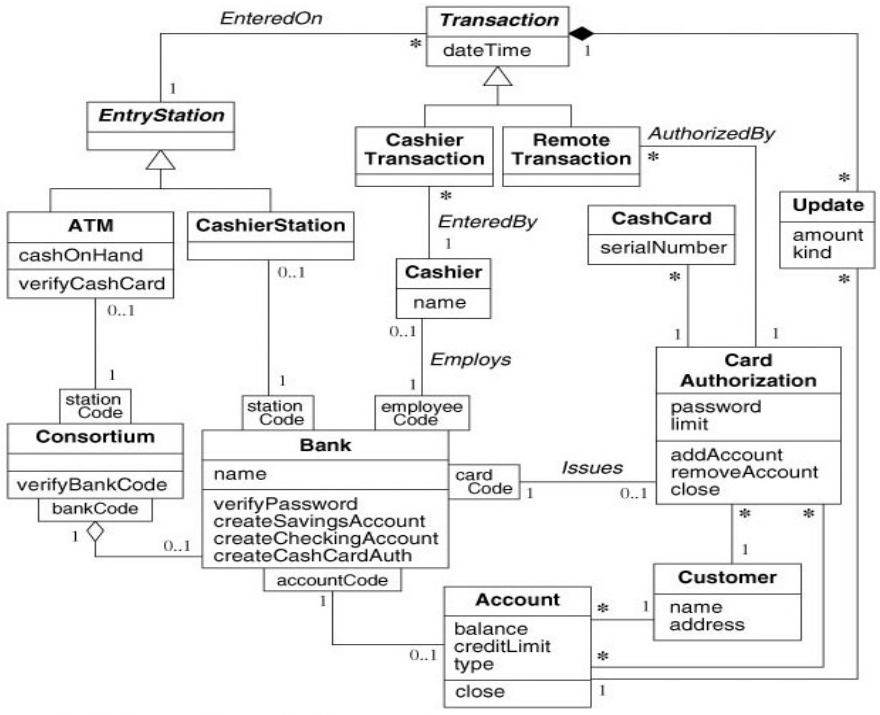
- E.g. An online theater ticket system
    - Use case - *Making a reservation*
    - Responsibilities
        - Finding unoccupied seats to the desired show
        - Marking the seats as occupied
        - Obtaining payment from the customer
        - Arranging delivery of the tickets and

- Crediting payment to the proper account

Each operation has various responsibilities. Group the responsibilities into clusters and try to make each cluster coherent. Define an operation for each responsibility cluster. If there is no good class to hold an operation, invent a lower-level class.

- E.g - *ATM*
  - *Use case – process transaction*
  - **Withdrawal** 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 disburse funds
    - debit bank account and
    - post entry on the customer's receipt
  - **Deposit -** responsibilities:
    - get amount from customer
    - accept funds envelope from customer
    - time-stamp envelope
    - credit bank account and
    - post entry on the customer's receipt
  - **Transfer -** responsibilities:
    - get source account
    - get target account
    - get amount verify that source account covers amount
    - verify that the amount is within the bank's policies
    - debit the source account
    - credit the target account and
    - post an entry on the customer's receipt

ATM Domain Class Model:



Exercises - EXCD 1;

Consider the following use cases for the simple diagram editor:

- Create drawing - Start a new, empty drawing in memory and overwrite any prior contents. Have the user confirm, if there is a prior drawing that has not been saved.
- Modify drawing - Change the contents of the drawing that is loaded into memory.
- Save to file - Save the drawing in memory to a file.
- Load from file. Read a file and load a drawing into memory overwriting any prior contents. Have the user confirm, if there is a prior drawing that has not been saved.
- Quit drawing. Abort the changes to a drawing and clear the contents of memory
- List at least four responsibilities for each one.

Answer EX-CD1:

- The responsibilities for the use cases are:
  - Create drawing
    - If there is an unsaved drawing, warn the user, and then clear the memory if the user confirms.
    - Create a single sheet and set the current sheet to this sheet.

- If there is an unsaved drawing, warn the user, and then clear the memory if the user confirms.
- Set the color to black and the line width to thin.
- Turn the automatic ruler on.

- **Modify drawing**
  - Make the change as indicated by the user on the screen.
  - Update the underlying memory representation.
  - Reset the undo buffer to recover from the modification.
  - Set the *hasBeenUpdated* flag in memory.
- **Load from file**
  - If there is an unsaved drawing, warn the user, and then clear the memory if the user confirms.
  - Find the file on the disc and complain if the file is not found.
  - Display an error message if the file is corrupted.
  - Show a drawing on the screen that corresponds to the contents in memory.
  - Set the current sheet to the first sheet.
- **Quit drawing**
  - If there is an unsaved drawing and the user declines to confirm, terminate *quit drawing*
  - Clear the contents of the screen.
  - Clear the contents of memory.
  - Clear the *undo* buffer.
-

Exercises – EXCD 2;

Consider the following use cases for Computerized Scoring System.

- register child
- schedule meet
- schedule season

List at least four responsibilities for each one.

Answer EX-CD2:

Responsibilities for the use cases

- *Register child*

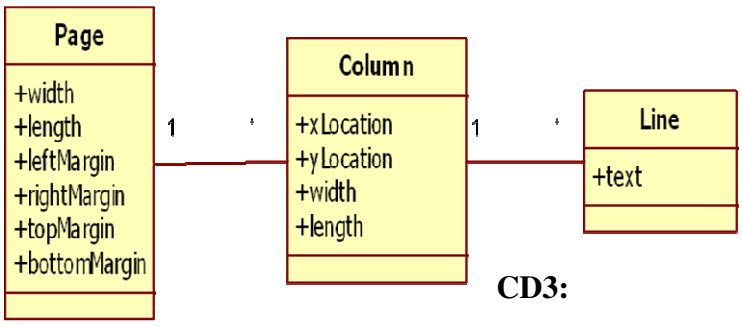
- Add a new child to the scoring system and record their data.
- Verify their eligibility for competition.
- Make sure that they are assigned to the correct team.
- Assign the child a number.

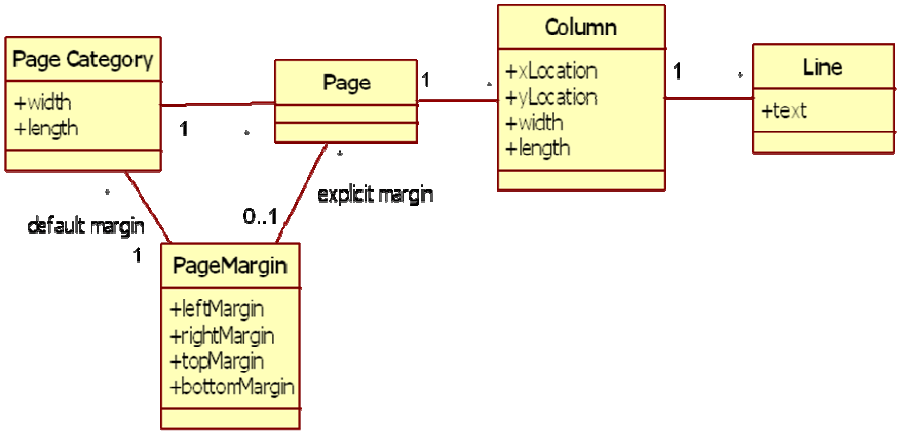
- *Schedule meet*
  - Choose a date for the meet within the bounds of the season.
  - Ensure that there are no conflicting meets on that date.
  - Notify the league and participating teams of the meet.
  - Arrange for judges to be there.
- *Schedule season*
  - Determine starting and ending dates.
  - Determine the leagues that will be involved.
  - Schedule a series of meets that comprise the season.
  - Check that each team in the participating leagues has a balanced schedule

Exercises – EXCD 3;

Modify the following class diagram, so that a separate class provides margins. Different categories of pages may have a default margin, and specific pages may override the default.

Answer EX-CD3:

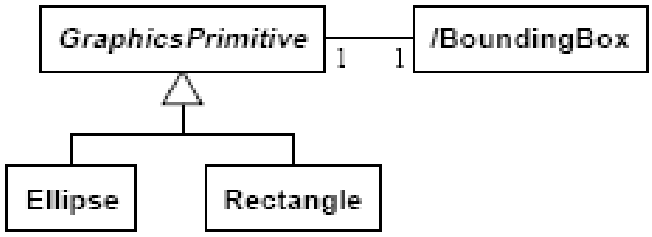




- Each page has an explicit *PageCategory* object that determines its size.
- Each *PageCategory* object has an associated *PageMargin* object that specifies the default margin settings.
- Any page can explicitly specify a *PageMargin* object to override the default settings.

Exercises – EXCD 4;

Which classes in the class diagram must supply a delete operation visible to the outside world? To delete means to destroy an object and remove it from the application. Explain your answer.



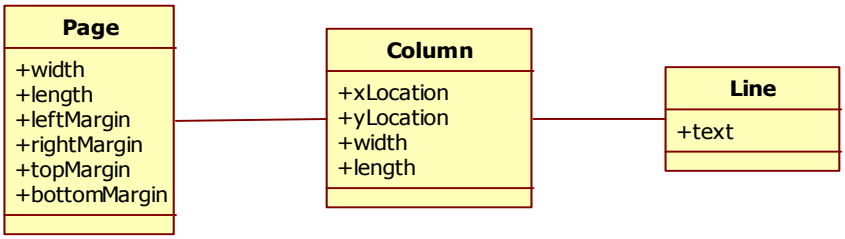
Answer EX-CD 4:

- All of the classes will probably implement a *delete* method, but the *GraphicsPrimitive* class must define *delete* as an externally-visible operation. A typical application would contain mixed sets of *GraphicsPrimitive* objects. A typical operation would be to delete a *GraphicsPrimitive* from a set.
- The client program need not know the *GraphicsPrimitive* subclass; all that matters is that it supplies a delete operation.
- *Ellipse* and *Rectangle* may have to implement *delete* as distinct methods, but they inherit the protocol from *GraphicsPrimitive*.
- Class *BoundingBox* must also define a *delete* operation, but this should not be visible to the outside world. A *Bounding-Box* is a derived object and may not be deleted independently of its *GraphicsPrimitive*.

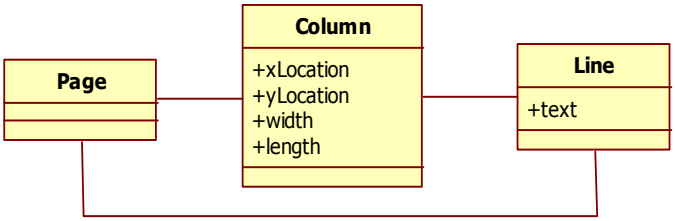
- *BoundingBox.delete* is visible only internally for the methods of *GraphicsPrimitive*

Exercises – EXCD 5;

Modify the following class diagram to make it possible to determine what *Page a Line* is on without first determining the *Column.*



Answer EX-CD 5:



Designing Algorithms

Steps:

- Choose algorithms that minimize the cost of implementing operations
- Select data structures appropriate to the algorithms
- Define new internal classes and operations as necessary
- Assign operations to appropriate classes

Choosing Algorithms

- Many operations traverse the class model to retrieve or change attributes or links
- OCL (Object Constraint Language) provides a convenient notation for expressing such traversals(next slide we brief about OCL)
- ATM Examples of OCL Expressions
- OCL to answer the credit card questions
  - What transactions occurred for a credit card account within a time interval?

    aCreditCardAccount.Statement.Transaction->

    select (aStartDate <= transactionDate and transactionDate <= anEndDate)

- What volume of transactions were handled by an institution in the last year?

  anInstitution.CreditCardAccount.Statement.Transaction ->

  select (aStartDate <= transactionDate and

  transactionDate <= anEndDate) .amount -> sum()

- What customers patronized a merchant in the last year by any kind of credit card?

  aMerchant.Purchase ->

  select (aStartDate <= transactionDate and

  transactionDate <= anEndDate) .Statement.

  CreditCardAccount.MailingAddress.Customer -> asset()

- How many credit card accounts does a customer currently have?

  aCustomer.MailingAddress.CreditCardAccount -> size()

- What is the total maximum credit for a customer, for all accounts?

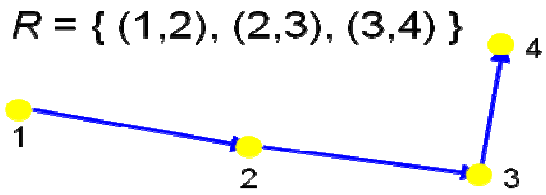
  aCustomer.MailingAddress.CreditCardAccount.

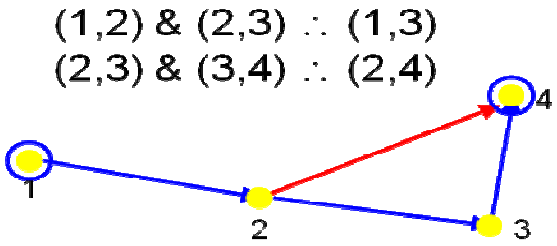
  maximumCredit -> sum()

- Pseudocode helps us think about the algorithm while deferring programming details(as traversal using OCL may not fully express some operations). For example, many applications involve graphs and the use of transitive closure(?)

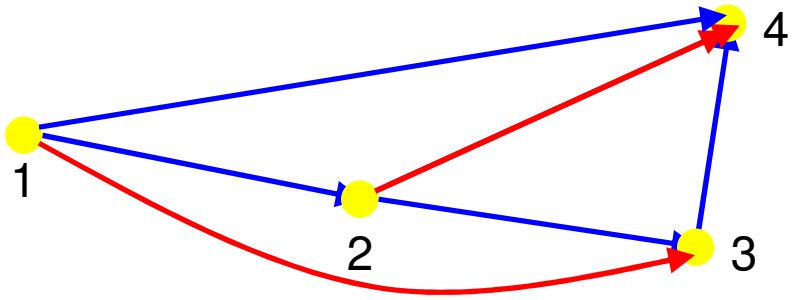
Transitive closure

- Informal definition: If there is a path from a to b, then there should be an edge from a to b in the transitive closure
- First take of a definition:
- In order to find the transitive closure of a relation R, we add an edge from a to c, when there are edges from a to b and b to c
- But there is a path from 1 to 4 with no edge!





- Informal definition: If there is a path from a to b, then there should be an edge from a to b in the transitive closure
- Second take of a definition:
    - In order to find the transitive closure of a relation R, we add an edge from a to c, when there are edges from a to b and b to c
    - Repeat this step until no new edges are added to the relation
- We will study different algorithms for determining the transitive closure
- red means added on the first repeat
- Teal means added on the second repeat



6 degrees of separation

- The idea that everybody in the world is connected by six degrees of separation
    - Where 1 degree of separation means you know (or have met) somebody else
- Let R be a relation on the set of all people in the world
    - (a,b) ∈ R if person a has met person b
- So six degrees of separation for any two people a and g means:
    - (a,b), (b,c), (c,d), (d,e), (e,f), (f,g) are all in R
- Or, (a,g) ∈ R6

Connectivity relation

- R contains edges between all the nodes reachable via 1 edge
- R◦R = R2 contains edges between nodes that are reachable via 2 edges in R
- R2◦R = R3 contains edges between nodes that are reachable via 3 edges in R
- Rn = contains edges between nodes that are reachable via n edges in R

- R* contains edges between nodes that are reachable via any number of edges (i.e. via any path) in R
    - Rephrased: R* contains all the edges between nodes a and b when is a path of length at least 1 between a and b in R

- R* is the transitive closure of R
    - The definition of a transitive closure is that there are edges between any nodes (a,b) that contain a path between them

Finding the transitive closure

- Let MR be the zero-one matrix of the relation R on a set with n elements. Then the zero-one matrix of the transitive closure R* is:

$$\mathbf{M}_{R^*} = \mathbf{M}_R \vee \mathbf{M}_R^{[2]} \vee \mathbf{M}_R^{[3]} \vee \cdots \vee \mathbf{M}_R^{[n]}$$

Nodes reachable with one application of the relation

Nodes reachable with two applications of the relation

Nodes reachable with *n* applications of the relation

Choosing Algorithms

Pseudocode example

Considerations for choosing among alternative algorithms

- Computational complexity
- Ease of implementation and understandability
- Flexibility

```
Node: :computeTransitiveClosure () returns NodeSet
        Nodes:= createEmptySet;
        return self.TCloop (nodes);
```

```
Node::TCloop (nodes:Nodeset) returns NodeSet
        add self to nodes;
        for each edge in self.Edge
                for each node in edge.Node
                /* 2 nodes are associated with an edge */
                if node is not in nodes then node.TCloop(nodes);
                end if
        end for each node
end for each edge
```

ATM example

- **Interactions between the consortium computer and bank computers could be complex.**
  - Distributed computing
  - The consortium computer to be scalable
  - The bank systems are separate applications from the ATM system

Choosing Data Structures

- During design data structures must be devised to permit efficient algorithms
- Many of these data structures are instances of container classes.
- Data structures include arrays, lists, queues, stacks, sets, bags, dictionaries, trees, and many variations, such as priority queues and binary trees.
- ATM example.
  - A Transaction should have an ordered list of Updates
  - By thinking about algorithms and working through the logic of an application, flaws can be found that will improve a class model

Defining Internal Classes and Operations

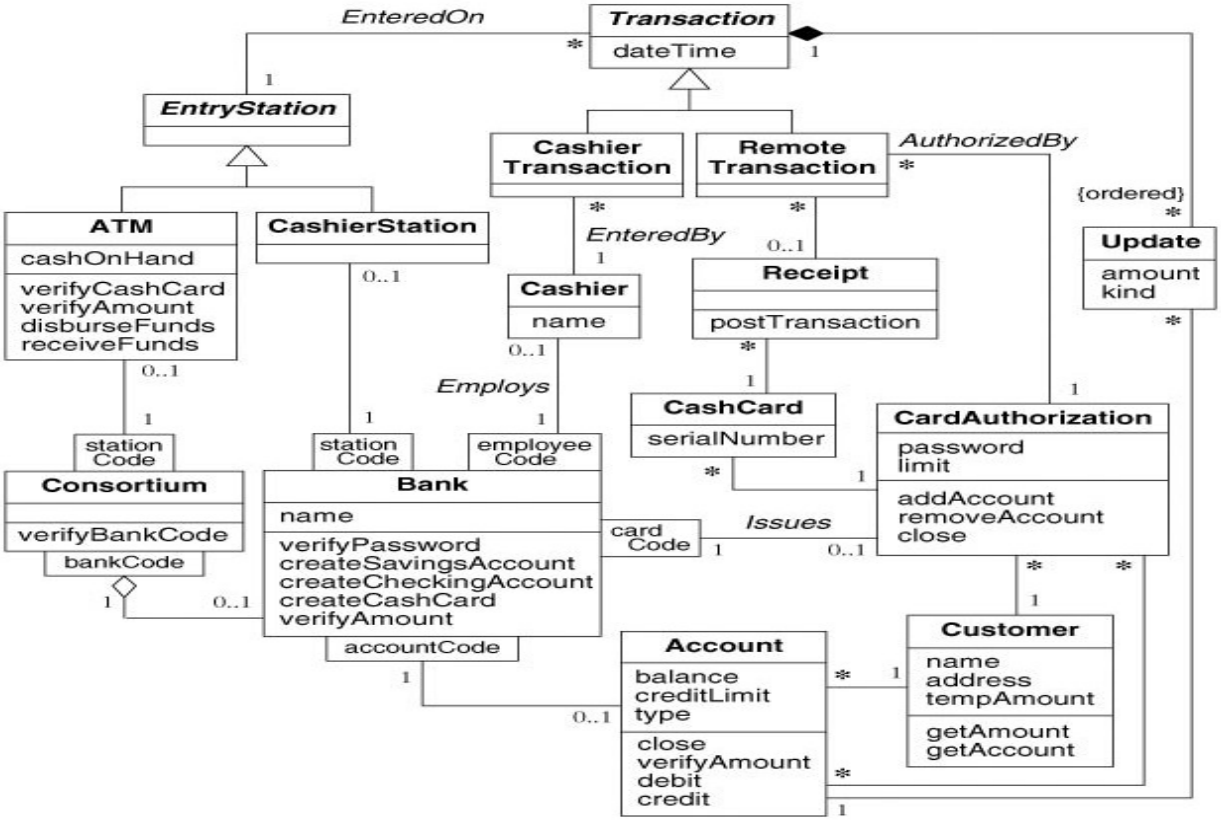
- New, low-level operations need to be invented during the decomposition of high-level operations.
- The expansion of algorithms may lead to create new classes of objects to hold intermediate results.

- ATM example
  - The design details for the *process transaction* involves a customer receipt.
  - The analysis class model did not include a *receipt* class, so it can be added during design

Assigning Operations to Classes

- During design, introduce internal classes that do not correspond to real-world objects but merely some aspect of them.
- Ask the following questions:
  - Receiver of Action
  - Query vs. Update
  - Focal class
  - Analogy to real world
- ATM Example: Internal operations for process transaction
  - Customer.get Amount()
  - Account.verifyAmount(amount)
  - ATM.verifyAmount(amount)
  - ATM.disburseFunds(amount)
- *ATM Example:* Internal operations for *process transaction*
  - Receipt.PostTranscation()
  - ATM.receive Funds(amount)
  - Account.creidet(amount)
  - Customer.getAccount()

ATM Domain Class model



Exercises: EXCD 6

Write algorithms to draw the following figures on a graphics terminal. The figures are not filled. Assume pixel-based graphics. State any assumptions that you make.

- Circle
- Ellipse
- Rectangle
- Square

Exercises: EXCD 6 - Answer

- **Circle - Solution 1:**
  - For a circle of radius R centered at the origin, we have x2+y2 =R2
  - Solving for y, we get y= ±(R2 – x2). We can generate a point for each x coordinate by scanning x from –R to R in steps of one pixel and computing y.

- The center point of the circle must be added to each generated point.

- **Solution 2:**
  - More sophisticated algorithms compute 8 points at once, taking advantage of the symmetry about the axes, and also space the points so that there are no gaps.
- **Ellipse:**
  - Centered at the origin with axes A and B parallel to the coordinate axes, we have (x/A)2+(y/B)2 =1
  - We can solve for *y* in terms of *x* as for the circle.
  - The equations are more complicated if the axes are not parallel to the coordinate axes.
- **Rectangle:**
  - A Rectangle of width *2A* and height *2B* with sides parallel to the coordinate axes centered at (*X, Y*)
  - Fill in all pixels with ordinates *Y-B* and
  - Y+B between abscissas X-A and X+A, and fill in all pixels with abscissas X-A and X+A between ordinates *Y-B*+1 and *Y+B*-1.
  - If the rectangle is not parallel to the coordinate axes, then line segments must be converted to pixel values.
- **Square:**
  - drawing a rectangle whose sides are equal

Exercises: EXCD 7

Discuss whether or not the algorithm that you wrote in the previous exercise to draw an ellipse is suitable for drawing circles and whether or not the rectangle algorithm is suitable for squares.

Exercises: EXCD 7 – Answer

- Any algorithm that draws ellipses must draw circles, since they are ellipses, and any algorithm that draws rectangles must draw squares.
- The real question is whether it is worthwhile providing special algorithms to draw circles and squares

- There is little or no advantage in an algorithm for squares, because both squares and rectangles are made of straight lines anyway.
- An algorithm for circles can be slightly faster than one for ellipses.
- This may be of value in applications where high speed is required, but is probably not worthwhile otherwise

Exercises: EXCD 8

- By careful ordering of multiplications and additions, the total number of arithmetic steps needed to evaluate a polynomial can be minimized. For example,. One way to evaluate the polynomial a4x4+a3x3+a2x2+a1x+a0 is to compute each term separately, adding each term to the total as it is computed, which requires 10 multiplications and 4 additions.
- Another way is to rearrange the order of the arithmetic to x.(x.(x.(x.a4+a3)+a2)+a1)+a0, which requires only 4 multiplications and 4 additions. How many multiplications and additions are required by each method for an nth-order polynomial? Discuss the relative merits of each approach.

Exercises: EXCD 8 – Answer

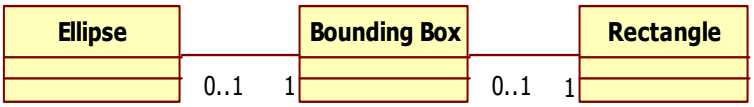
- A general n-th order polynomial has the form $\sum_{i=0}^{n} a_i x^i$. Each term requires i multiplications and

  one addition (except the 0-th term), so computing the sum of the individual terms requires $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ multiplications and n additions. Computing the

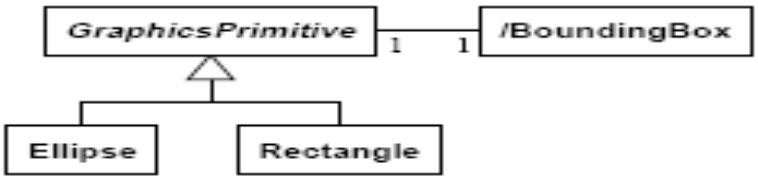
  sum by successive multiplication and addition requires one multiplication and one addition for each degree above zero, or a total of n multiplications and n additions.
- The second approach is not only more efficient than the first approach, it is better behaved numerically, because there is less likelihood of subtracting two large terms yielding a small difference.
- There is no merit at all to the first approach and every reason to use the second approach.

Exercises: EXCD 9:

- Improve the class diagram by generalizing the classes, Ellipse and Rectangle to the class *GraphicsPrimitive,* transforming the class diagram so that there is only a single one-to-one association to the object class *BoundingBox*. In effect, you will be changing the 0..1 multiplicity to exactly-one multiplicity. As it stands, the class *BoundingBox* is shared between Eclipse and Rectangle. A *BoundingBox* is the smallest rectangular region that will contain the associated Ellipse or Rectangle.



Exercises: EXCD 9 - Answer:



Recursing Downward:

- Organize operations as layers
- Downward recursion proceeds in two main ways:
  - By functionality and
  - By mechanism
- Functionality Layer
  - Take the required high-level functionality and break it into lesser operations
  - Combine similar operations and attach the operations to classes
  - Operations that are carefully attached to classes have much less risk than free-floating functionality
- **ATM Example: rework on the operations that are assigned to classes**

Mechanism Layers:

- Build the system out of layers of needed support Mechanisms.
- Various mechanisms are needed to store information, sequence control, coordinate objects, transmit information, perform computations, and to provide other kinds of computing infrastructure
- Any large system mixes functionality layers and mechanism layers. Reason is:
  - A system designed entirely with functionality recursion is brittle and supersensitive to changes in requirements
  - A system designed entirely with mechanisms doesn't actually do anything useful.

ATM Example:

- Need for communications and distribution infrastructure

- The bank and ATM computers are at different locations and must quickly and efficiently communicate with each other

Refactoring

- It is good to use an operation or class for multiple purpose
- Definition: changes to the internal structure of software to improve its design without altering its external functionality [Martin Fowler]
- ATM Example: operations of the process transaction use case
- Combine Account.credit(amount) and Account.debit(amount) into a single operation Account.post(amount)

Mathematics: Factor:

- fac·tor
  - One of two or more quantities that divides a given quantity without a remainder, e.g., 2 and 3 are factors of 6; a and b are factors of ab
- fac·tor·ing
  - To determine or indicate explicitly the factors of

SE: Factoring:

- fac·tor
  - The individual items that combined together form a complete software system:
  - identifiers
  - contents of function
  - contents of classes and place in inheritance hierarchy
- fac·tor·ing
  - Determining the items, at design time, that make up a software system

Refactoring:

- Process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure [Fowler'99]
- A program restructuring operation to support the design, evolution, and reuse of object oriented frameworks that preserve the behavioural aspects of the program [Opdyke'92]

Specifics:

- Source to source transformation
- Remain inside the same language, e.g., C++ to C++
- Does not change the programs behavior

- Originally designed for object-oriented languages, but can also be applied to non-object oriented language features, i.e., functions

Levels of Software Changes:

- High Level
  - Features to be added to a system
  - e.g., New feature
- Intermediate Level
  - Change design (factoring)
  - e.g., Move a member function
- Low Level
  - Change lines of code
  - e.g., Changes in (a least) two classes

Relationship to Design:

- Not the same as "cleaning up code"
  - May cause changes to behavioral aspects
  - Changes often made in a small context or to entire program
- Key element of entire process in agile methodologies
- Views design as an evolving process
- Strong testing support to preserve behavioral aspects

Quick Examples:

- Introduce Explaining Variable
- Rename Method
- Move Method
- Pullup Method
- Change Value to Reference
- Remove Parameter
- Extract Hierarchy

Why: Design Preservation:

- Code changes often lead to a loss of the original design
- Loss of design is cumulative:

- Difficulties in design comprehension ->
  Difficulties in preserving design ->
  More rapid decay of design

- Refactoring improves the design of existing code

Catalog:

- Collected by Fowler
- Refactoring entry composed of:
  - Name
  - Summary
  - Motivation
  - Mechanics
  - Examples
- Based on Java

Tools:

- Smalltalk Refactoring Browser
  - Development environment written in Smalltalk
  - Allows for Smalltalk source code to transform Smalltalk source code
  - Comments as a first-class part of the language
- XRefactory
  - Allows standard refactorings for C++

Challenges:

- Preservation of documentary structure (comments, white space etc.)
- Processed code (C, C++, etc.)
- Integration with test suite
- Discovery of possible refactorings
- Creation of task-specific refactorings

Limitations:

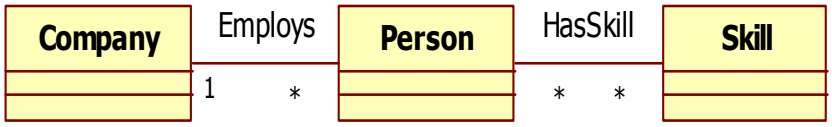
- Tentative list due to lack of experience
- Database

- Database schema must be isolated, or schema evolution must be allowed

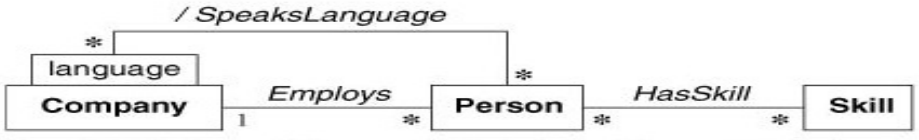
- Changing Published Interfaces
  - Interfaces where you do not control all of the source code that uses the interface
  - Must support both old and new interfaces
  - Don't publish interfaces unless you have to

Design Optimization

- **Good way to design a system: first get the logic correct and then optimize it**
- **The design model builds on the analysis model.**
- **Analysis model – captures the logic of a system**
- **Design model – adds development details**
- **Design Optimization involves the following tasks:**
    - Provide efficient access paths
    - Rearrange the computation for greater efficiency
    - Save intermediate results to avoid recomputation
- **Adding redundant associations for efficient access**
    - Design has different motivations and focuses on the viability of a model for implementation



- **Some improvements for the above analysis class model**
    - use hashed set for HasSkills
    - ex: japanese speaking
    - indexing
- **In cases 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.**



- **Indexes incur a cost:**
    - Require additional memory
    - Require to be updated whenever the base associations are updated

- Examine each operation and see what associations it must traverse to obtain its information
    - For each operation
        - Frequency of access
        - Fan-out
        - Selectivity
- Provide indexes for frequent operations with a low hit ratio, because such operations are inefficient when using nested loops to traverse a path in the network
- Elaborate the class model
- ATM Example:
- postTransaction() operation
    - Relate Receipt to CashCard (Assigning operations to classes)
    - Tracing from *CashCard* to *CardAuthorization* to *Customer* has no fan-out
    - Derived association from *Bank* to *Update* speed the process

Rearranging Execution Order for Efficiency:

- One key to algorithm optimization is to eliminate dead paths as early as possible
- ATM Example:
- Maintain two different derived associations between Bank and Update, one for individuals and the other for businesses

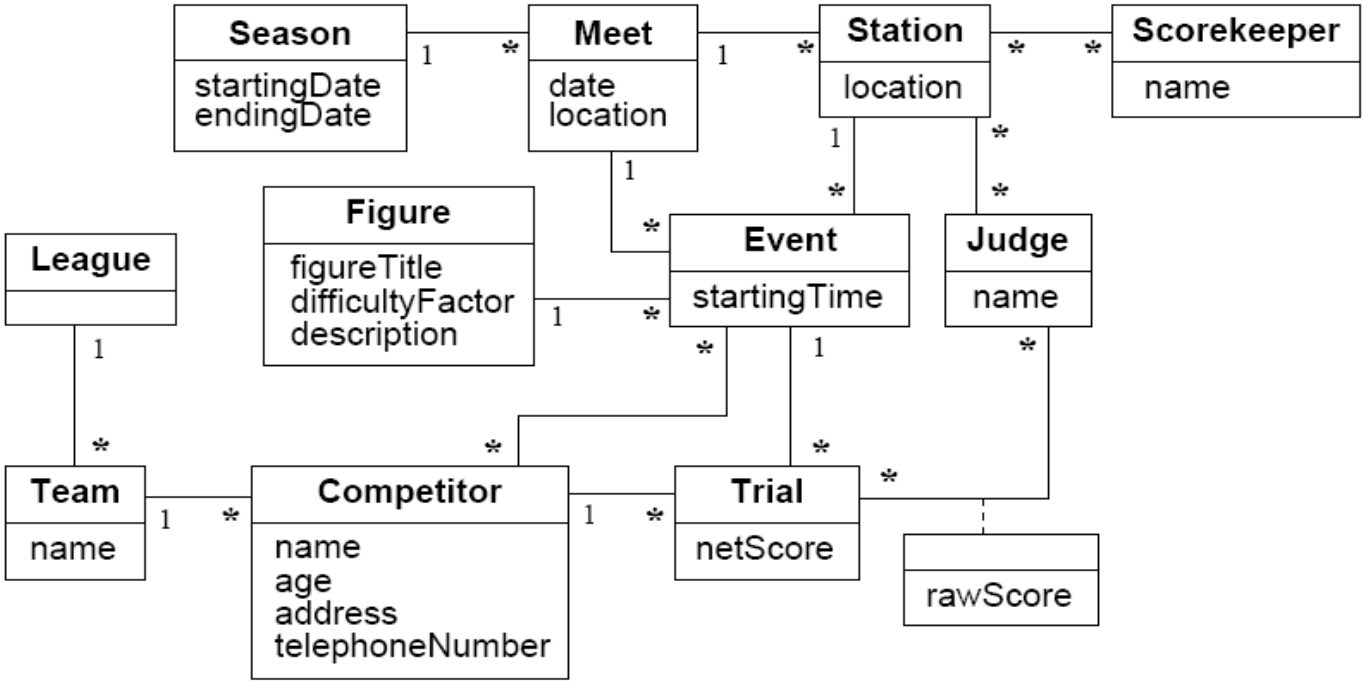
Saving Derived Values to Avoid Recomputation:

- Define new classes to cache derived attributes and avoid recomputation.
- Three ways to handle updates:
    - Explicit update
    - Periodic recomputation
    - Active values
- *ATM Example:*
    - For convenience, we might add the class *SuspiciousUpdateGroup*

Exercises: EXCD 10

- **Prepare pseudo code for the following operations to classes in the class diagram**
    - Find the event for a figure and meet.
    - Register a competitor for an event.

- Register a competitor for all events at a meet.
- Select and schedule events for a meet.
- Assign judges and scorekeepers to stations.



Exercises: EXCD 10 – Answer

- **Find the event for a figure and meet.**

Figure::findEvent (aMeet: Meet) : Event;

return self.Event INTERSECT aMeet.Event;

// Note that 0, 1, or more events could be returned.

- **Register a competitor for an event.**

Competitor::register (anEvent: Event);

If self.Event is not null then return;

// already registered

create new RegisteredFor link between self and anEvent

- **Register a competitor for all events at a meet.**

Competitor::registerAllEvents (aMeet: Meet)

For each anEvent in aMeet.Event

self.register(anEvent);

- **Select and schedule events for a meet.**

Meet::addFigure (aFigure: Figure)

Create an event object

Associate it with the meet

Associate it with the figure

- **Meet::scheduleAllEvents ()**

while events have not been scheduled do

for each aStation in self.Station do

anEvent := get next unscheduled event in

self.Event

if no more events then return;

associate anEvent to aStation;

- **Assign judges and scorekeepers to stations**

Meet::assignPersonnel (judges, scorekeepers)

Sort the judges and scorekeepers in priority order;

averageJudges := numberOfJudges / numberOfStations;

averageScorekeepers :=

numberOfScorekeepers / numberOfStations;

if averageJudges < minimumPermitted then error;

if averageScorekeepers < minimumPermitted then error;

neededJudges:= minimum (averageJudges, maximumPermitted);

neededScorekeepers :=

minimum (averageScorekeepers, maximumPermitted);

for each station:

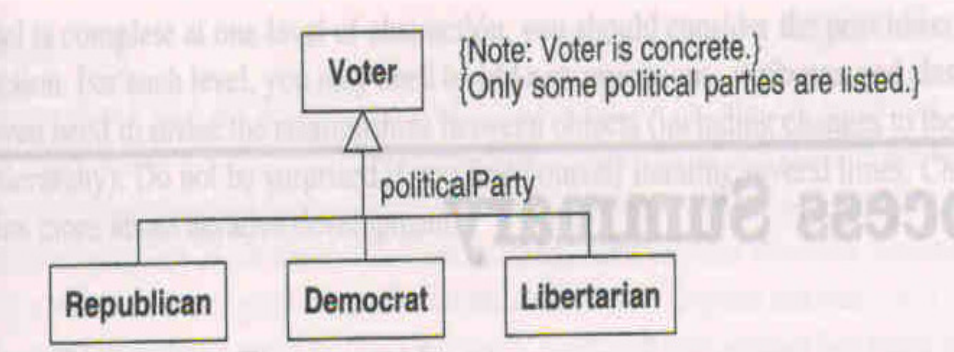
assign the next neededJudges judges and

neededScorekeepers scorekeepers.

Notify any remaining judges and scorekeepers that they are not needed.

Exercises: EXCD 11

- Improve the class diagram by transforming it, adding the class Political Party. Associate Voter with a party. Discuss why the transformation is an improvement.



Exercises: EXCD 11 – Answer

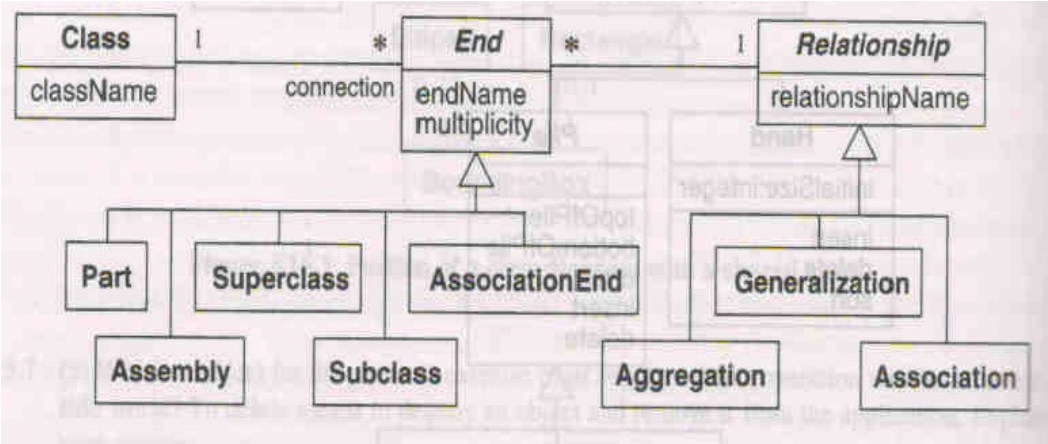
- Political party membership is not an inherent property of a voter but a changeable association.
- The revised model better represents voters with no party affiliation and permits changes in party membership.

Exercises: EXCD 11 – Answer

- If voters could belong to more than one party, then the multiplicity could easily be changed.
- Parties are instances, not subclasses, of class *PoliticalParty* and need not be explicitly listed in the model; new parties can be added without changing the model and attributes can be attached to parties.

Exercises: EXCD 12

- Refine the class diagram by eliminating the associations to the classes End and Relationship, replacing them with associations to the subclasses of End and Relationship.



Reification of Behavior

- Behavior written in code is rigid.
- To store, pass, or modify the behavior at run time, it should be reified
- *Reification is the promotion of something that is not an object into an object.*
- By reifying behavior, it can be stored, passed to other operations, and can be transformed.

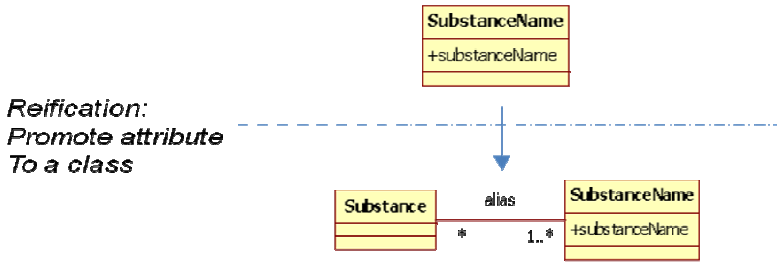
Reification of Behavior

- Example 1: Database Manager
  - A database manager has abstract functionality that provides a general-purpose solution to accessing data reliable and quickly for multiple users

Reification of Behavior

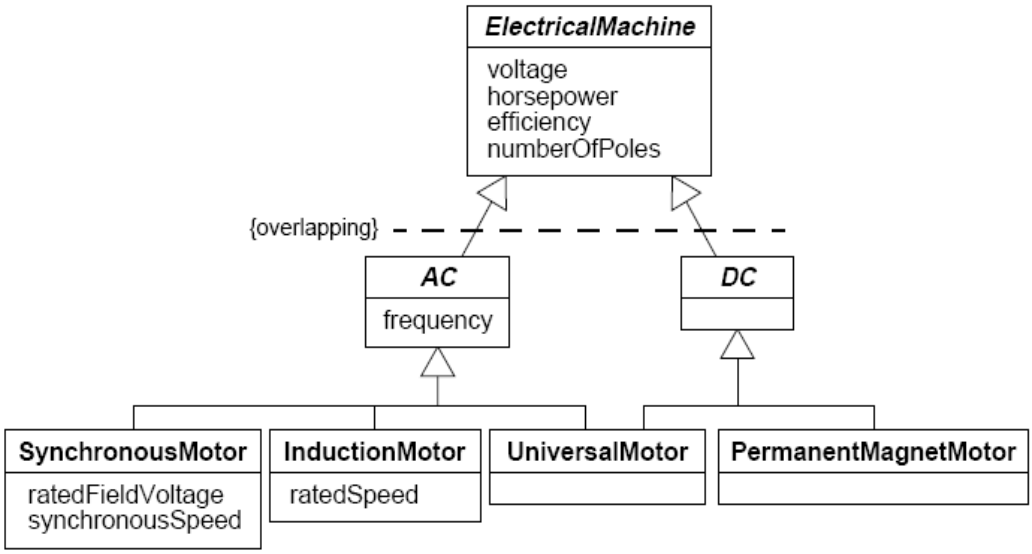
- Example 2: State-transition diagrams
  - Prepare a metamodel and store a stste-transition model as data. A general-purpose interpreter reads the contents f the metamodel and executes the intent

Reification of Behavior



Exercises-EXCD 13

- Revise the class diagram to eliminate use of multiple inheritance



Reification of Behavior

- In one sense you can regard the tasks of a recipe as operations
- In another sense they could be data in a class model

Reification of Behavior

- The behavioral patterns that reify behavior
  - Encoding a state machine as a table with a run-time interpreter – *State*
  - Encoding a sequence of requests as parameterized command objects – *Command*
  - Parameterizing a procedure in terms of an operation that it uses - *Strategy*

Adjustment of Inheritance

- Adjust the definitions of classes and operations to increase inheritance by:
  - 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

Rearrange Classes and Operations

- Adjustments to increase the chance of inheritance
  - Operations with optional arguments
  - Operations that are special cases
  - Inconsistent names
  - Irrelevant operations

Rearrange Classes and Operations

- ATM Example:
  - An ATM can post remote transactions on a receipt. It should be able to issue a receipt for cashier transactions.
  - A receipt for a *RemoteTransaction* involves *CashCard*, while a receipt for *CashierTransaction* directly involves a *Customer*.

Rearrange Classes and Operations

- ATM Example:
  - The cashier software is apart from the ATM software.
  - Two different kinds of receipts, a *RemoteReceipt* and a *CashierReceipt*.

Abstracting out common behavior

- Abstracting out a common superclass or common behavior.
- Make only abstract superclasses.
- It is worthwhile to abstract out a superclass even when the application has only one subclass that inherits from it.
- Consider the potentially reusable classes for future applications.

Abstracting out common behavior

- The splitting of a class into two classes that separate the specific aspects from the more general aspects is a form of modularity improves the extensibility of a software product
- Generating customized versions of the software to match each different configuration could be tedious

Abstracting out common behavior

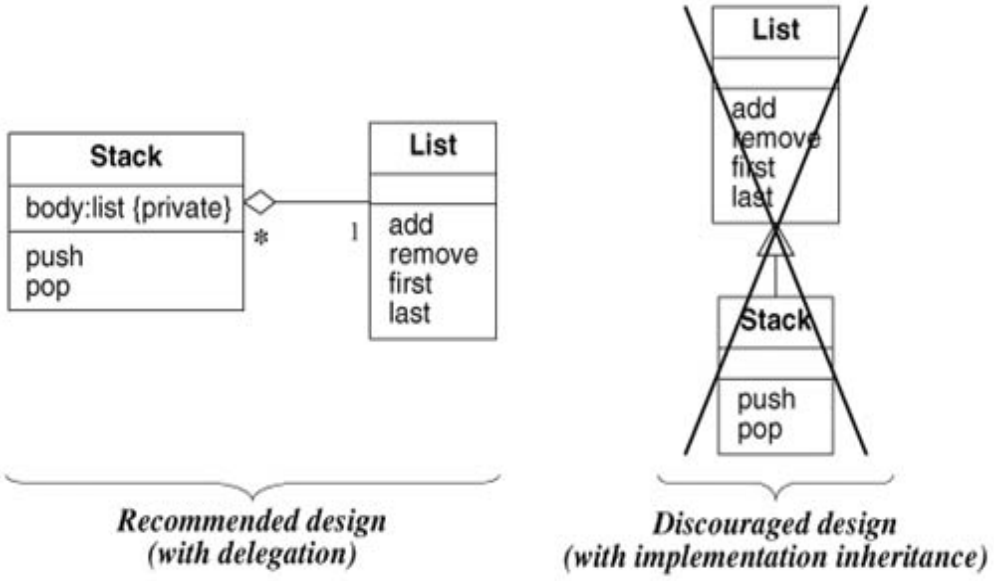
- ATM example:
  - We did pay attention to inheritance during analysis when we constructed the class model. We do not see any additional inheritance at this time. In a full-fledged application there would be much more design detail and increased opportunities for inheritance.

Using Delegation to Share Behavior

- Sharing of behavior is justifiable only when a true generalization relationship occurs.
- Discourage this **inheritance of implementation** because it can lead to incorrect behavior.
  - E.g. A stack class and a list class is available. One object can selectively invoke the desired operations of another class, using delegation rather than inheritance.

Using Delegation to Share Behavior

- **Delegation** consists of catching an operation on one object and sending it to a related object.



Organizing a Class Design

- Programs consist of discrete physical units that can be edited, compiled, imported, or otherwise manipulated.
- The organization of a class design can be improved with the following steps.

- Hide internal information from outside view.
- Maintain coherence of entities.
- Fine-Tune definition of packages.

Organizing a Class Design

- ***Information hiding:*** carefully separating external specification from internal implementation

Organizing a Class Design

- Ways to hide information
  - Limit the scope of class-model traversals.
  - Do not directly access foreign attributes.
  - Define interfaces at a high a level of abstraction.
  - Hide external objects.
  - Avoid cascading method calls.

Coherence of Entities

- Coherence is another important design principle.
- 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.

Coherence of Entities

- *Policy:* making of context-dependent decisions.
- *Implementation:* execution of fully- specified algorithms.
- Separating policy and implementation greatly increases the possibility of reuse.

Coherence of Entities

- E.g. consider an operation to credit interest on a checking account. Interest is compounded daily based on the balance, but all interest for a month is lost if the account is closed.
- A class should not serve too many purposes at once.

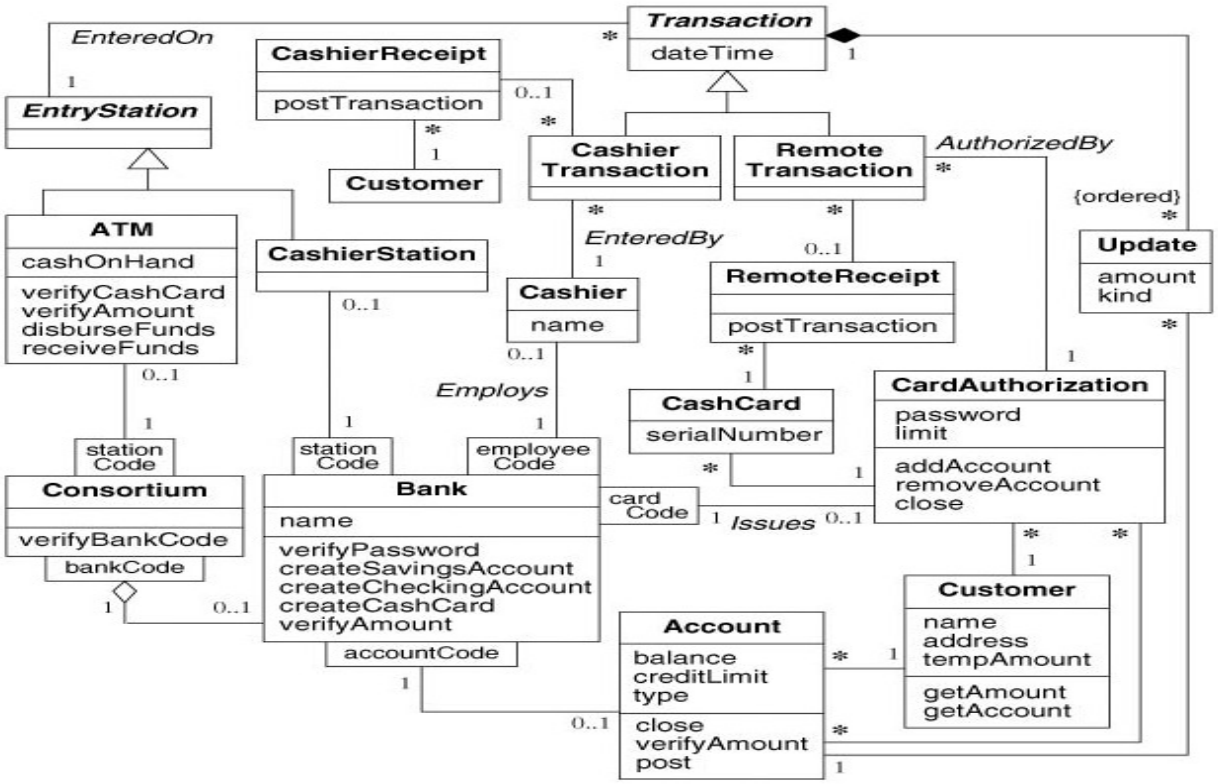
Fine- Tuning packages

- During analysis the class model is partitioned into packages
- The interface between two packages consists of the associations that relate classes in one package to classes in the other and operations that access classes across package boundaries

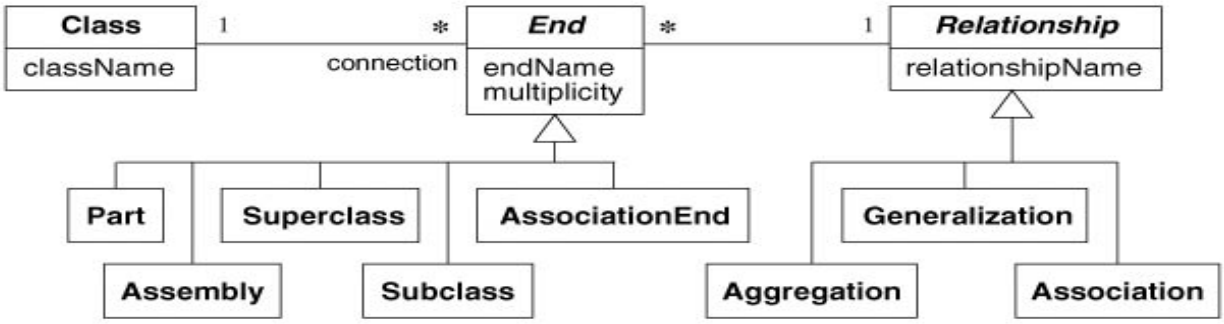
Fine- Tuning packages

- Packages should have some theme, functional cohesiveness, or unity of purpose
- Try to keep strong coupling within a single package

Final ATM domain class model after class design



Exercises: EXCD 14



In selecting an algorithm, it may be important to evaluate its resource requirements. How does the time required to execute the following algorithm depend on the following parameters: On the depth of the inheritance hierarchy.

Exercises: EXCD 14

Algorithm:

```
traceInheritancePath (class1, class2): Path
{ path := new Path;
 // try to find a path from class1 as descendent of class2
classx := class1;
while classx is not null do add classx to front of path;
        if classx = class2 then return path;
                classx := classx.getSuperclass();
```

Exercises: EXCD 14

```
// didn't find a path from class1 up to class2
 // try to find a path from class2 as descendent of
        class 1
        path.clear();
        classx := class2;
        while classx is not null do add classx to front of path;
        if classx = class1 then return path;
        classx := classx.getSuperclass();
```

Exercises: EXCD 14

```
// the two classes are not directly related
 // return an empty path path.clear(); return path; }
        Class::getSuperclass (): Class
        { for each end in self.connection do:
        if the end is a Subclass then:
        relationship := end.relationship;
         if relationship is a Generalization then:
        otherEnds := relationship.end;
        for each otherEnd in otherEnds do:
        if otherEnd is a Superclass then: return otherEnd.class
        return null;
```

Exercises: EXCD 14 – Answer

- In the worst case, if we search the wrong way first by chance, then the search will go to the top of the class hierarchy, so the cost is linear in the depth.

Exercises: EXCD 14 – Answer

- If we modify the algorithm so that we search up from both classes at the same time, then the search cost will be no more than twice the difference in depth of the two classes and will not depend at all on the depth of the class hierarchy (except as a limit on the difference in depths).
- If the class hierarchy is very deep, then the algorithm should be modified to limit the cost, otherwise the added complexity is probably not worth the bother.

Implementation Modeling

Implementation is the final development stage that addresses the specifics of programming languages.

Overview of Implementation

Implementation capitalizes on careful preparation from analysis and design. Implementation modeling involves the following steps:

- Fine-tune classes
- Fine-tine generalizations
- Realize associations
- Prepare for testing

Fine-tune classes and Fine-tine generalizations are motivated by theory of transformations. A transformation is a mapping from the domain of models to the range of models.

Fine-tuning classes

The purpose of implementation is to realize the models from analysis and design. To alter the design classes consider the possibilities:

- Partition a class
- Merge classes
- Partition/merge attributes
- Promote an attribute/demote a class

Partition a class: The information in a single class can be split into two classes. The class diagram in Figure 1 represents home and office information.

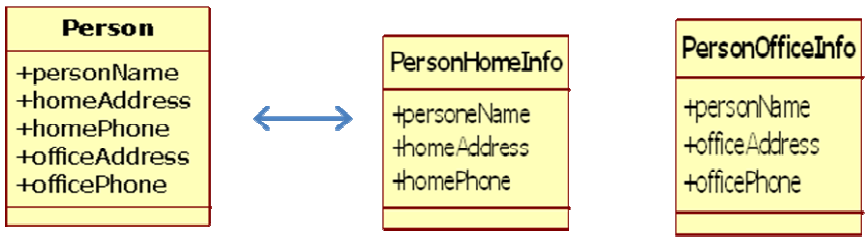


Figure-1: Partitioning a class

Partitioning of a class can be complicated by generalization and association. Figure 2 shows partitioning a class by generalization and association.

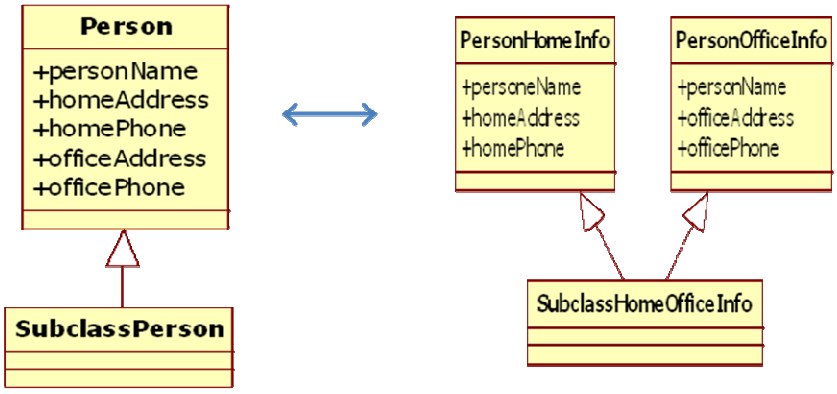


Figure-2: Partitioning a class by generalization and association

Merge classes: The converse to partitioning a class is to merge classes. Figure 3 shows an example with intervening associations.

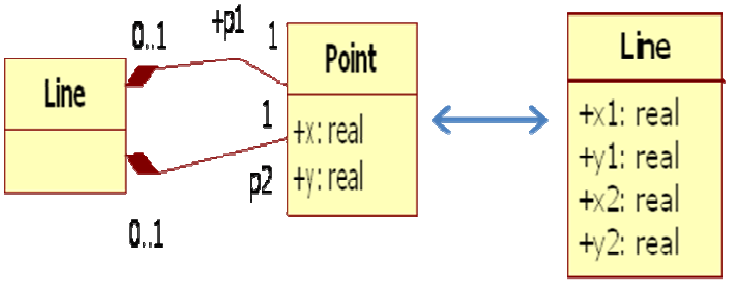


Figure-3: Merging classes

Partition/merge attributes: Attributes can be adjusted by partitioning and merging, as Figure 4 illustrates.

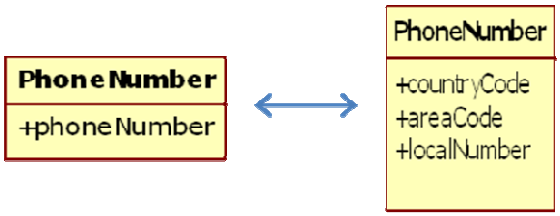


Figure-4: Partitioning/merging classes

Promote an attribute/demote a class: It is illustrated in Figure 5

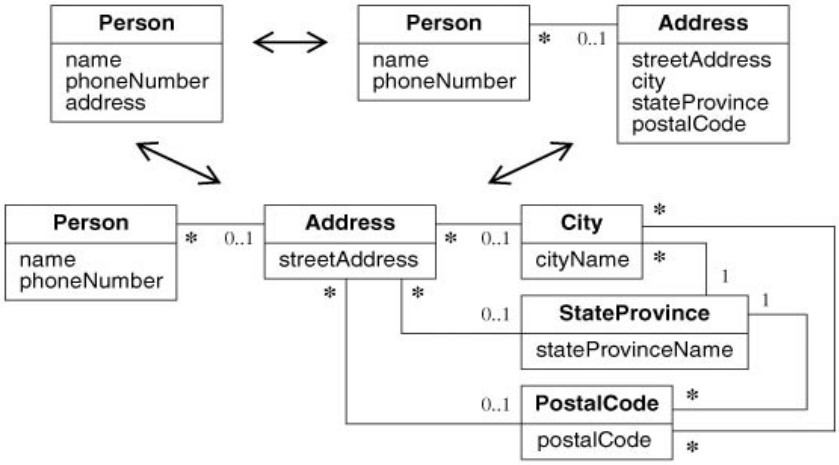


Figure-5: Promoting an attribute/demoting a class

ATM example:

If pre-populating address data, split Customer address into several related classes. E.g. preload *city*, *stateProvince*, and *postalCode* when creating a new Customer record.

Fine-tuning Generalizations:

It is helpful to remove or add a generalization prior to coding. Figure 6 shows a translation model.

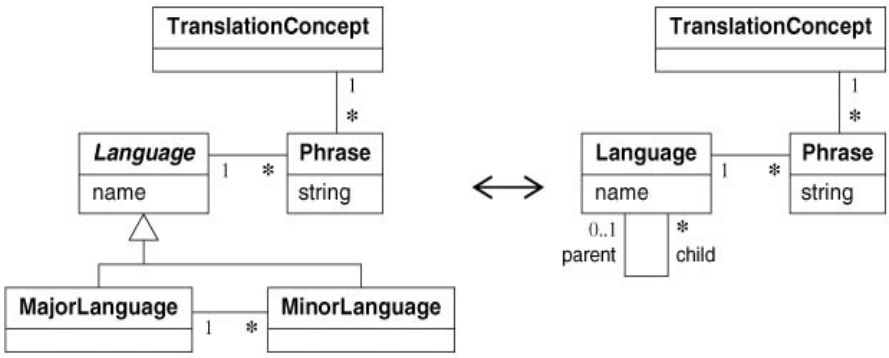


Figure-6: Removing/adding generalization

ATM example:

ATM domain class model encompassed two applications.

- ATM
- cashier

Figure 7 presents the ATM domain class model (Design).

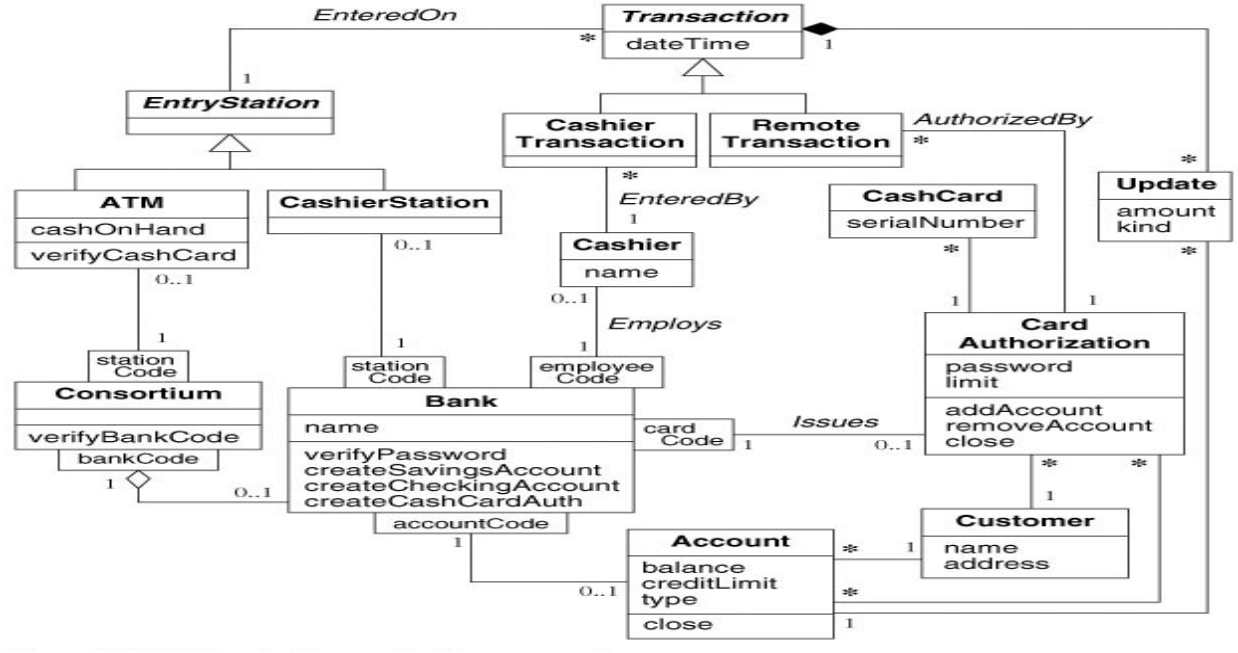


Figure-7: ATM domain class model (Design)

Deleting cashier information from the domain class model, leads to a removal of both generalizations.

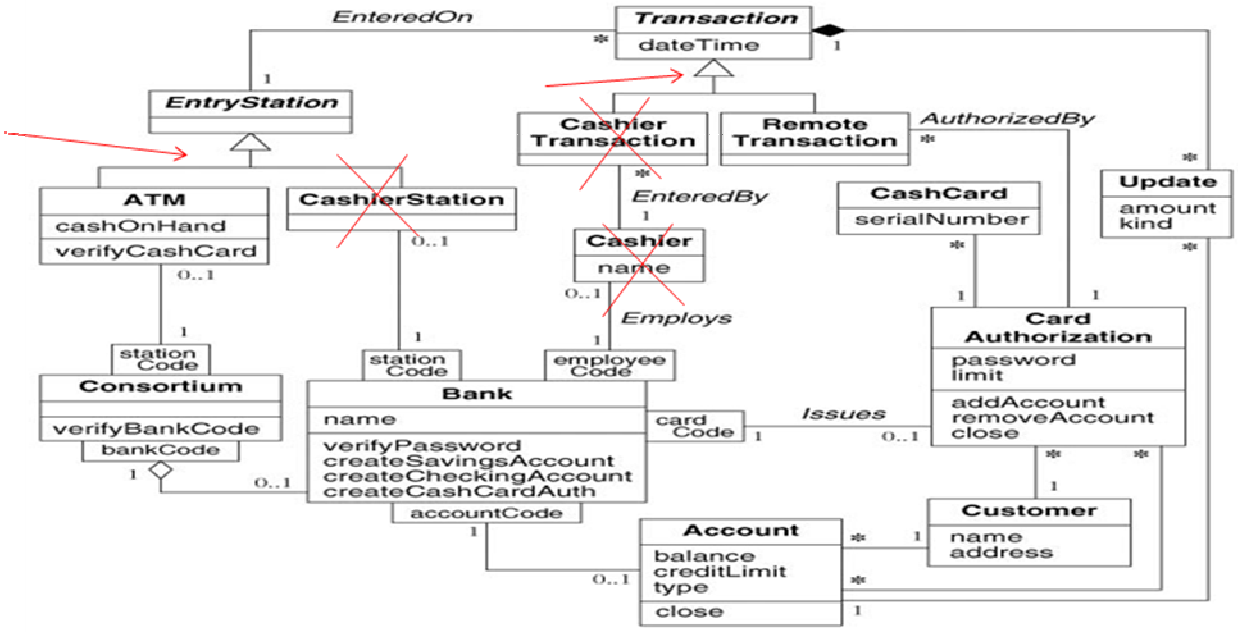


Figure-8: ATM domain class model (Design) showing removal of generalizations

Figure 9 shows ATM Implementation Model – domain class model and Figure 10 shows ATM Implementation Model – application class model

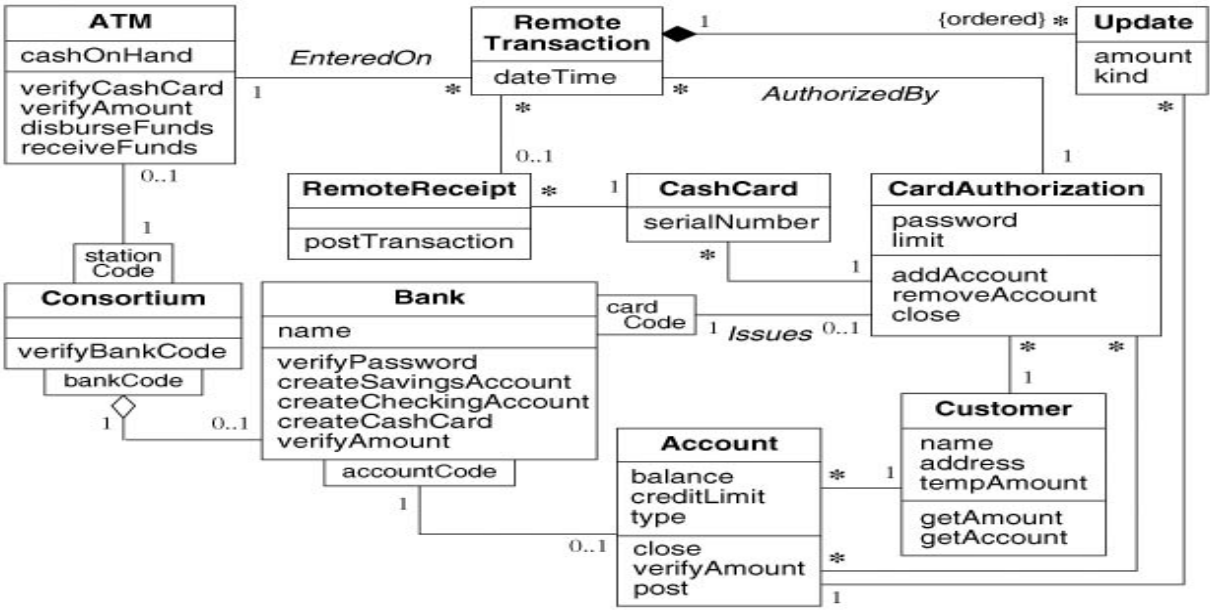


Figure 9 ATM Implementation Model – domain class model

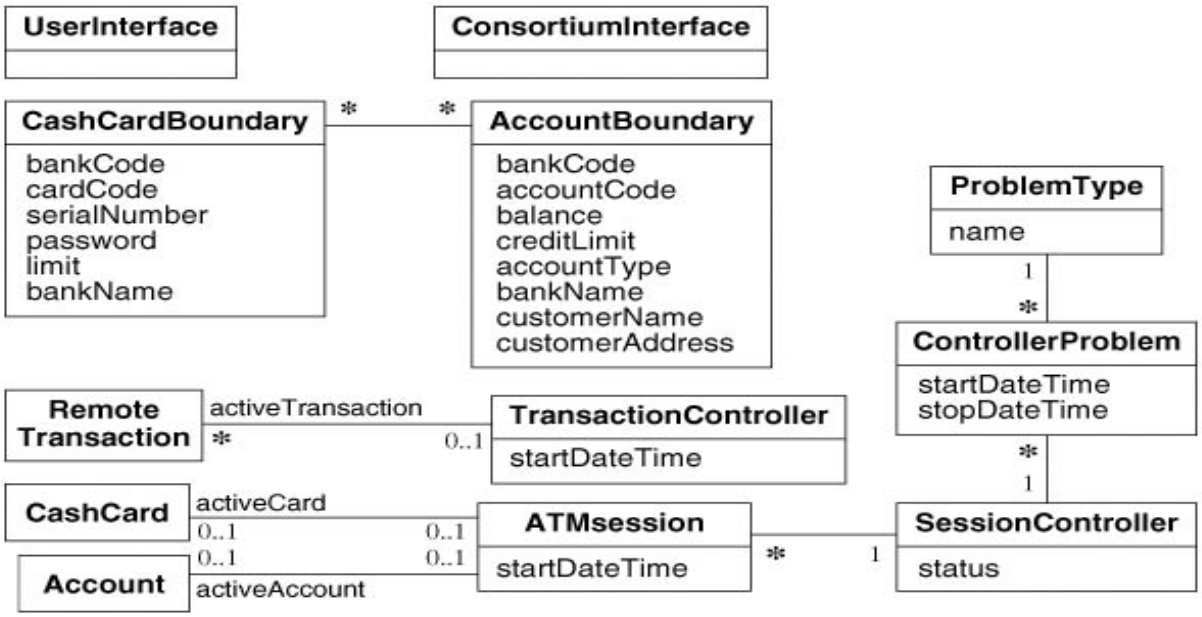


Figure 10 ATM Implementation Model – application class model

Realizing Associations:

Associations are the "glue" of the class model, providing access paths between objects. Strategies for analyzing association traversal

- Analyzing Association Traversal
- One-way Associations
- Two-way Associations
- Advanced Associations

Analyzing Association Traversal:

Implementation of associations, which are traversed in only one direction, can be simplified. For prototype work, use bidirectional associations, flexible to add new behavior and modify the application.

Analyzing Association Traversal:

If an association is traversed only in one direction, implement it as a pointer.

- Simple pointer - multiplicity is "one"
- Set of pointers - multiplicity is "many"

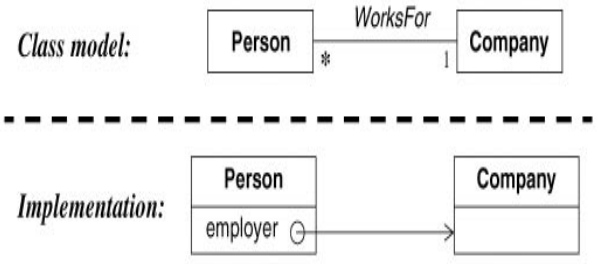


Figure-11: Implementing a one-way association with pointers

Two-way Associations:

Associations traversed in both directions. Approaches for implementation are:

- Implement one-way.
- Implement two-way: Implement with pointers in both directions as Figure 12.
- Implement with an association object: An association object is a set of pairs of associated objects. Stored in a single variable-size object.

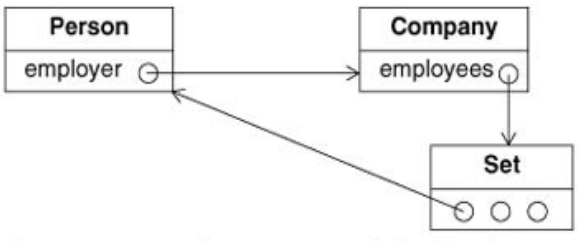


Figure-12: Implementing a two-way association with pointers

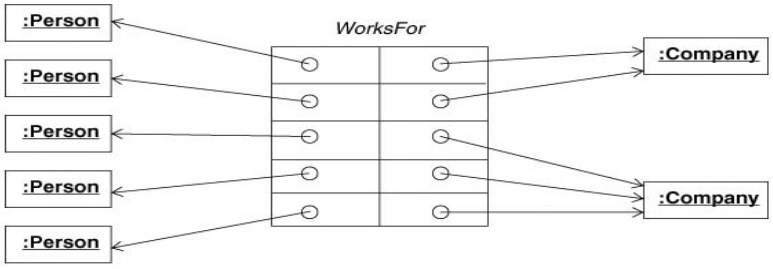


Figure-12: Implementing an association as an object

Advanced Associations:

Techniques for implementing advanced associations are:

- Association classes.
- Ordered associations.
- Sequences.
- Bags.
- Qualified associations.
- N-ary associations.
- Aggregation.

Testing:

Testing is a quality-assurance mechanism for catching residual errors. The number of bugs found for a given testing effort is an indicator of software quality.

- Analysis - test the model against user expectations by asking questions and seeing if the model answers them.
- Design - test the architecture and can simulate its performance.
- Implementation - test the actual code-the model serves as a guide for paths to traverse.

Testing should progress from small pieces to ultimately the entire application; Unit testing, Integration testing.

- Famous Problems – 1: F-16 : crossing equator using autopilot
  - Result: plane flipped over
  - Reason?
    - Reuse of autopilot software
- Famous Problems – 2: The Therac-25 accidents (1985-1987), quite possibly the most serious non-military computer-related failure ever in terms of human life (at least five died)

- Reason: Bad event handling in the GUI

- Famous Problems – 3: NASA Mars Climate Orbiter destroyed due to incorrect orbit insertion (September 23, 1999)
  - Reason: Unit conversion problem

Testing Fundamentals:

Software Testing is a critical element of Software Quality Assurance. It represents the ultimate review of the requirements specification, the design, and the code. It is the most widely used method to insure software quality. Many organizations spend 40-50% of development time in testing.

Testing is concerned with establishing the presence of program defects. Debugging is concerned with finding where defects occur (in code, design or requirements) and removing them. (Fault identification and removal). Even if the best review methods are used (throughout the entire development of software), testing is necessary. Testing is the one step in software engineering process that could be viewed as destructive rather than constructive. "A successful test is one that breaks the software." A successful test is one that uncovers an as yet undiscovered defect. Testing cannot show the absence of defects, it can only show that software defects are present. For most software exhaustive testing is not possible.

Testing Object-Oriented Applications: Why is it Different?

- No sequential procedural executions
- No functional decomposition
- No structure charts to design integration testing
- Iterative O-O development and its impact on testing and integration strategies

Testing Object-Oriented Applications – Issues:

- Implications of inheritance
  - New context of usage
  - Multiple Inheritance
  - True specialization -- reuse of test cases
  - Programming convenience -- must test subclasses and superclasses
- Implications of encapsulation
  - Makes reporting on state difficult
  - Correctness proof may help but difficult
- Implications of Polymorphism
  - Each possible binding requires separate test
  - Makes integration more difficult
- White-box testing

  - Appropriate only for methods not for classes
  - Control graph testing is not applicable to methods
- Black-box testing
  - Applicable to classes
  - The distance between OO specification and implementation is small compared to conventional systems
- Integration strategy
  - Thread-based: All classes needed to respond to certain external input
  - Uses-based: Starts with classes that do not use other classes; then continue with classes that use the first group and so on.
- Test process strategy
  - Design a little, code a little, and test a little cycle

Unit testing: Developers normally check their own code and do the unit and integration testing, because they understand the detailed logic and likely sources of error.

- What is a unit?
  - A single, cohesive function?
  - A function whose code fits on one page?
  - The amount of code that can be written in 4 to 40 hours?
  - Code that is assigned to one person?
- In object-oriented programs, a unit is a method within a class.

Methods for Generating Test Cases:

- Statement coverage
- Graph based
  - Branch coverage
  - Condition coverage
  - Path coverage
- All unit testing methods are also applicable to testing methods within a class.

Integration Testing:

- Strategies:
  - Bottom-up guided by functional decomposition tree
  - Top-down guided by functional decomposition tree
  - Big bang
  - Pairwise
- All focus on structure and interface
- Performed exclusively in terms of inputs and outputs of the system
- Performed mostly on the target platform
- Thread-based:
  - The behavior that results from a system level input
  - An interleaved sequence of system inputs (stimuli) and outputs (responses)

- A sequence of transitions in a state machine model of the system

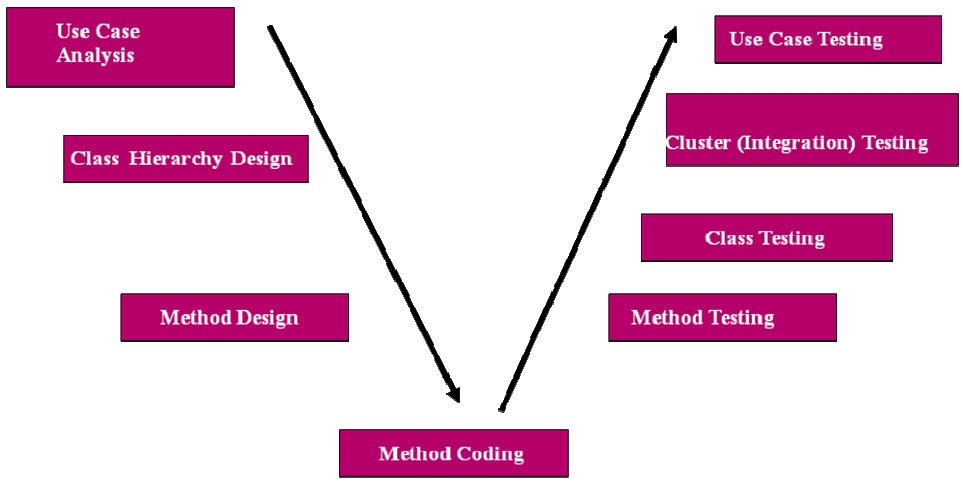
System Testing:

QA should derive their testing from the analysis model of the original requirements and prepare their test suite in parallel to other development activities. The scenarios of the interaction model define system-level test case. Pay attention to performance and stress the software with multiuser and distributed access, if that is appropriate. As much as possible use a test suite.

- Performed exclusively in terms of inputs and outputs of the system
- Performed mostly on the target platform
- Thread-based:
  - The behavior that results from a system level input
  - An interleaved sequence of system inputs (stimuli) and outputs (responses)
  - A sequence of transitions in a state machine model of the system

ATM example: We have carefully and methodically prepared the ATM model. Consequently we would be in a good position for testing, if we were to build a production application.

The OO Testing Pyramid:



Method Testing:

- Tests each individual method
- Performed by the programmer
- Glass box using method code
  - Statement Coverage
  - Decision Coverage
  - Path Coverage

Class Testing:

- Traditional black-box and white-box techniques still apply
  E.g. testing with boundary values

Inside each method: at least 100% branch coverage; also, DU-pairs inside a method

- Extension: DU pairs that cross method boundaries
  Example: inside method m1, field f is assigned a value; inside method m2, this value is read.

Legacy Systems

Most development does not involve new applications but rather evolves existing ones
It is difficult to modify an application if you don't understand its design

What is Reverse Engineering?

General definition: A systematic methodology for analyzing the design of an existing device or system, either as an approach to study the design or as a prerequisite for **re-**design.

Reverse Engineering helps you to:

Develop a systematic approach to thinking about the engineering design of devices and systems

Acquire a mental data bank of mechanical design solutions

Levels of Analysis in Reverse Engineering

System-Wide Analysis
Subsystem Dissection Analysis
Individual Component Analysis

System-Wide Analysis

Customer Requirements
Engineering Requirements
Functional Specifications
Prediction of Subsystems and Components

Subsystem Dissection Analysis

Document Disassembly

Define Subsystems

Determine Subsystem Functional Specifications

Determine Subsystem Physical/Mathematical Principles

Individual Component Analysis

Repeat Dissection Steps to Individual Component

Define Component Material Selection and Fabrication Process

Suggest Alternative Designs, Systems, Components, and Materials

What is Reverse Engineering ?

You have an unexpected case:
You finished one course project using Java

Your program runs OK
But, by accident, you delete the java file
How to hand in your project?

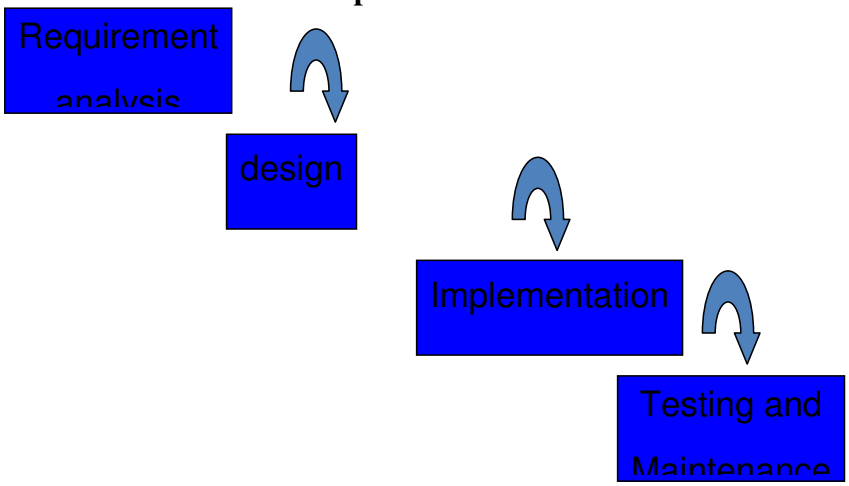
Reverse Engineering

What is Reverse Engineering?

RE encompasses any activity that is done to determine how a product works, to learn the ideas and technology that were used in developing that product.
RE can be done at many levels
RE generally belongs to Software Maintenance

What is Reverse Engineering ?

Waterfall Model of software development



Reverse Engineering-SE context

Trying to figure out the structure and behaviour of existing software by building general-level static and dynamic models'
Links:

http://www.rigi.csc.uvic.ca/UVicRevTut/F4rev.html
Compact information on reverse engineering
http://users.ece.gatech.edu/~linda/revengr/revrepos.html
Reengineering Resource Repository
Listings of tools, literature, …

The Early Days of RE

Law of Software Revolution (Lehman, 1980)
Fundamental strategies for program comprehension (Brooks, 1983)
Taxonomy of Reverse Engineering (Chikofsky&Cross, 1990)
WCRE (Working Conference on R.E., 1990)

- IWPC (Int. Workshop on Program Comprehension)

Why do we need RE ?

- Recovery of lost information
  - providing proper system documentation

- Assisting with maintenance
  - identification of side effects and anomalies
- Migration to another hw/sw platform
- Facilitating software reuse

- Benefits
  - maintenance cost savings
  - quality improvements
  - competitive advantages
  - software reuse facilitation

Difficulties of Reverse Engineering

- Gap between problem /solution domain
- Gap between concrete and abstract
- Gap between coherency/disintegration
- Gap between hierarchical/associational

Scope and Task of Reverse Engineering

- program understanding

- Reverse Engineering is focused on the challenging task of understanding legacy program code without having suitable documentation
- However, the major effort in software engineering organizations is spent after development,on maintaining the systems to remove existing errors and to adapt them to changed requirements.

Reverse Engineering

- mature software systems often have incomplete, incorrect or even nonexistent design documentation.
- Ideally, the system documentation describes the product and the complete design, including its rationale.
- The major purpose of tools essentially is to aid maintainers *understand the program*
- Pure plan recognition is unlikely to be powerful enough to drive reverse engineering for the following reasons:

-legacy code is written using rather tricky encodings to achieve better efficiency

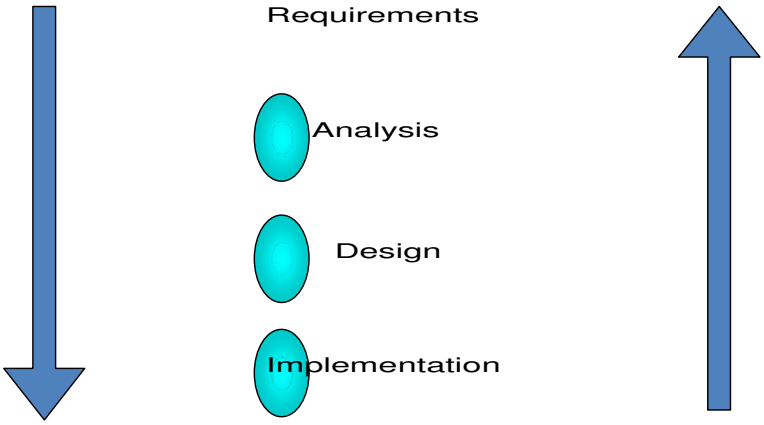
- all the necessary plan patterns in all the variations must be supplied in advance for a particular application,

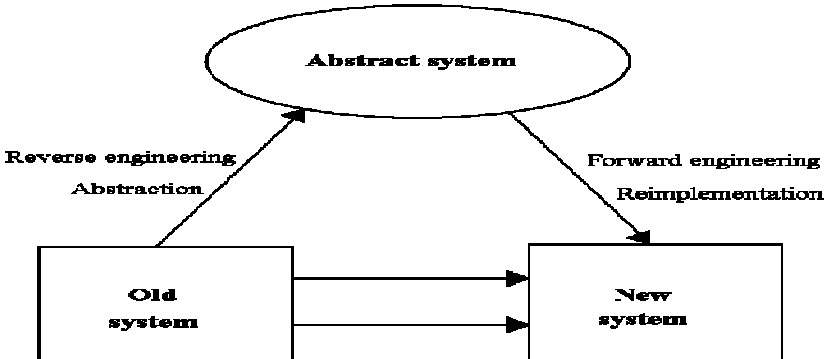
-different abstract concepts map to the same code within one application

- A huge amount of conventionally developed software exists and such systems will continue to be developed for a long time.
- These systems have errors and continual demand for the enhancement of their functional and performance requirements

- They are often badly designed and have an incomplete, nonexistent, or, even worse, wrong documentation without any design information, this is a challenging task.

Forward Engineering	**Reverse Engineering**
Given requirements, develop an application	Given an application, deduce tentative requirements
More certain	Less certain
Prescriptive	Adaptive
More mature	Less mature
Time consuming	10 to 100 times faster than forward engineering





Applications

- Modifying software
  - Change of environment (software migration)
  - Re-designing software (re-engineering)
    - E.g. Y2K, €, e-commerce
- Design and implementation in forward engineering, e.g. debugging
- Program understanding/comprehension
- Program visualisation
- Software re-use

Data reverse engineering

- ” Data reverse engineering focuses on data and data-relationships both among data structures within programs and data bases”
- For example: relational data bases (RDBs):

Reverse engineering

- Redocumentation
- Restructuring
  - transforming a system from one representation to another, while preserving its external functional behavior
- Retargeting
  - transforming and hosting or porting the existing system in a new configuration
- Business Process Reengineering
  - radical redesign of business processes to increase performance, such as cost, quality, service, and speed
  - reoptimization of organizational processes and structures
- Reverse specification
  - extracting a description of what the examined system does in terms of the application domain
  - a specification is abstracted from the source code or design description

Software reverse engineering

- Chikofsky & Cross: two-phase process
  - Collecting information
    - parsers, debuggers, profilers, event recorders
  - Abstracting information
    - Making understandable, high-level models
- *“Programmers have become part historian, part detective, and part clairvoyant” (T.A.Corbi 1989)*

- Inputs to Reverse Engineering
  - Programming code
  - Database structure
  - Data
  - Forms and reports
  - Documentation
  - Application understanding
  - Test cases

Reverse Engineering

- Outputs from Reverse Engineering
  - Models
  - Mappings
  - Logs

Building the Class Model

- Building the class model using three phases
  - Implementation Recovery
  - Design Recovery
  - Analysis Recovery

Building the Class Model

- Implementation Recovery
  - Learn about the application and create an initial class model
  - Study the data structures and operations and manually determine classes
  - Have an initial model focused on the implementation

- **Design Recovery**
  - The multiplicity in reverse direction is typically not declared and it must be determined from general knowledge or examination of the code
  - Many implementations use a collection of pointers to implement an association with "many multiplicity"
  - Pointers will implement both association directions

Building the Class Model

- **Design Recovery**
  - The multiplicity in reverse direction is typically not declared and it must be determined from general knowledge or examination of the code
  - Many implementations use a collection of pointers to implement an association with "many multiplicity"
  - Pointers will implement both association directions
- **Analysis Recovery**
  - If the source code is not object oriented, infer generalizations by recognizing similarities and differences in structure and behavior

Building the Interaction Model

- The purpose of each method is clear but the way that objects interact to carry out the purposes of the system is hard to understand from the code
- A *slice* is a subset of a program that preserves specified projection of its behavior

- The accumulated code lets the project an excerpt of behavior from the original program
- The purpose of each method is clear but the way that objects interact to carry out the purposes of the system is hard to understand from the code
- A *slice* is a subset of a program that preserves specified projection of its behavior
- The accumulated code lets the project an excerpt of behavior from the original program
- The power of slice comes from
    - They can be found automatically
    - Slices are generally smaller than the programs from which they originated
    - They execute independently of one another
    - Each reproduces exactly a projection of the original program's behavior

Building the State Model

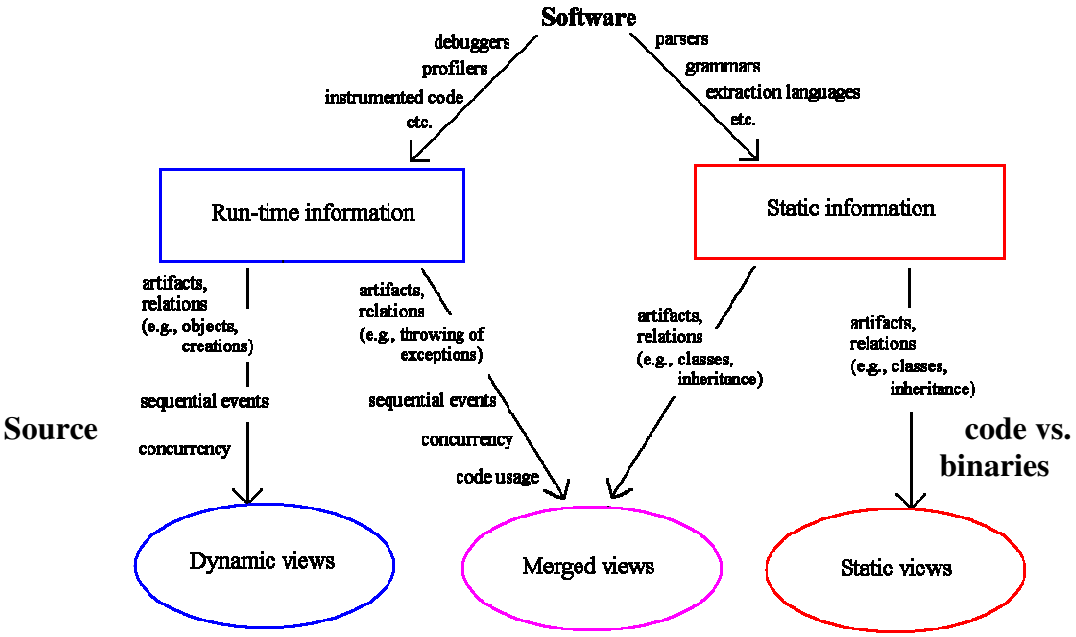
- To construct a state model
    - Fold the various sequence diagrams for a class together by sequencing events and adding conditions and loops
    - Augment the information in the sequence diagrams by studying the code and doing dynamic testing
    - Initiation and termination correspond to construction and destruction of objects

Reverse Engineering Tips

- Distinguish supposition from facts
- Use a flexible process
- Expect multiple interpretations
- Don't be discouraged by approximate results
- Expect odd constructs
- Watch for a consistent style

Wrapping

- A wrapper is a collection of interfaces that control access to a system
- Consists of a set of boundary classes that provide the interfaces
- Provides a clean interface for exposing the core functionality of existing applications
- New functionality can be added as a separate package
- Example
    - Web Application
- Wrapping is temporary solution
    - Constrained by the organization of the legacy systems
    - Original old code, the wrapper and code in a different format become so unwieldy and it must be rewritten
- The use of XML as a gateway for communication between the legacy systems and the modern world



- **Source code**
  - better form of representation
  - not always possible
  - result depends on the parser (notable differencies)
- **Binaries**
  - faster information collection (e.g. Java byte code)
  - legality issues

Usage of binaries
(reverse engineering, decompilation, disassembly)

- Recovery of lost source code
- Migration of applications to a new hardware platform
- Translation of code written in obsolete languages not supported by compiler tools nowadays
- Determination of the existence of viruses or malicious code in the program
- Recovery of someone else's source code (to determine an algorithm for example)

Binary copyrights
(decompilation, disassembly)

- Not all countries implement the same laws !
- Commonly allowed by law
  - for the purposes of interoperability
  - for the purposes of error correction where the owner of the copyright is not available to make the correction
  - to determine parts of the program that are not protected by copyright (e.g. algorithms), without breach of other forms of protection (e.g. patents or trade secrets)

- The decompilation page: **http://archive.csee.uq.edu.au/~csmweb/decompilation/home.html**

Copyrights cont.

- EU: 1991 EC Copyright Directive on Legal Protection of Computer Programs provided extensions to copyright to permit decompilation in limited circumstances
- An example: Sony sued Connectix Corp (1999) for developing of its Virtual Game Station emulator, and emulator of the Sony developed PlayStation (Mac) -> a long fight over emulation rights and extent of copyright protection on computer programs

Reverse Engineering Tools

- **Analysis Tools**
- **Browsers**
- **Object Server**
- **Task Oriented Tools**

Example--Java Decompiler

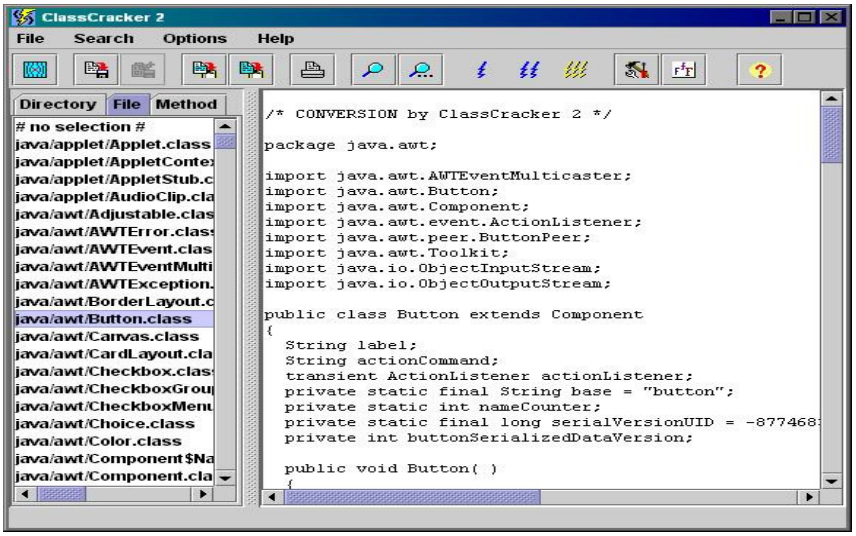
- How to recover bytecode from .class file under Unix/Win with JDK?

% javap -c <filename>
% javap -help (to see the options)

- Java Decompilers
  - "ClassCracker" http://www.pcug.org.au/~mayon/
  - "DeCafe Pro" from DeCafe, France at http://decafe.hypermart.net/index.htm
  - "SourceAgain" from Ahpah corp at http://www.ahpah.com

Example--Java Decompiler

- ClassCracker 2 Interface



- **Components of ClassCracker 2**
  - Java decompiler

        - retrieves Java source code from Java class files
    - Java disassembler
        - produces Java Assembly Code
    - A Java class file viewer
        - displays Java class file structures.
- **Features of ClassCracker 2**
    - User visual interface.
    - Can decompile class files within zip or jar files.
    - Conversion mode (JAVA, JASM or JDUMP) is selectable
    - A Batc Mode allows multiple class files to be decompiled simultaneously
    - more…...
- **ClassCracker 2.0--want to try it?**
    - Free download at http://www.pcug.org.au/~mayon/classcracker/ccgetdemo.html
    - Only first three methods are decoded.
- **Bridge 1.0---Free**
    - http://www.geocities.com/SiliconValley/Bridge/8617/jad.html

Exercises: EXIM 1:

Implement each association in ATM implementation model. Use one-way or two-way pointers as the semantics of the problem dictates. Explain your answers.

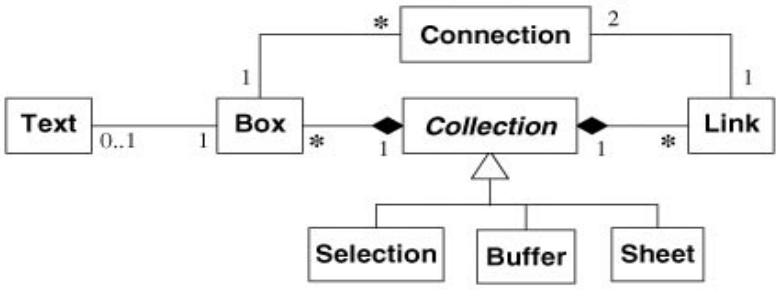
Exercises: EXIM 1: Answer

- An arrow indicates that the association is implemented in the given direction. Because of the large amounts of data for an ATM we used two-way pointers for associations that are traversed in both directions—we avoided one-way pointers combined with backward searching. Note that most of the associations in the domain model may be traversed either way, but associations for the application model are traversed only one way.
- **ATM <–> RemoteTransaction**
    - This association would be implicit for the ATM. However, it would be needed for the consortium and bank computers. Given a transaction we must be able to find the ATM. Given an ATM we might want to find all the transactions for a summary report.
- **RemoteTransaction <–> Update**
    - Obviously we must be able to find the updates for a transaction. However, we also must be able to go the other way. We might find the updates for an account and then need to find the transactions, such as to get the date and time.
- **CashCard <–> CardAuthorization**
    - A cash card must know its authorization. An authorization must also know about its cash cards.

- **Bank <–> CardAuthorization**
    - An authorization must be able to reference its bank and a bank must be able to find all authorizations (to facilitate issuing of card codes).
- **CashCardBoundary <– AccountBoundary**
    - This association is purely an artifact for the convenience of importing and exporting data.
- **TransactionController –> RemoteTransaction**
    - A transaction controller must know about its transactions. There is no need to go the other way.
- Consequently, there is no need to implement the association in both directions. We will choose to have pointers that retrieve cash card data for an account. (The decision on which way to implement is arbitrary and we could do it the other way.)
- **TransactionController –> RemoteTransaction**
    - A transaction controller must know about its transactions. There is no need to go the other way.

Exercises: EXIM 2:

Implement each association in the class diagram. Use one-way pointers wherever possible. Should any of the association ends be ordered? Explain your answers.



Exercises: EXIM 2: Answer

- An arrow indicates that the association is implemented in the given direction.
- **Text <–> Box**
    - The user can edit text and the box must resize, so there should be a pointer from text to box. Text is only allowed in boxes, so we presume that a user may grab a box and move it, causing the enclosed text to also move. So there should be a pointer from box to text.
- **Connection <–> Box**
    - A box can be dragged and move its connections, so there must be pointers from box to connections. Similarly, a link can be dragged and move its

connections to boxes, so there must also be a pointer from connection to box. There is no obvious ordering.

- **Connection <–> Link**
  - Same explanation as **Connection <–> Box**
- **Collection –> ordered Box**
  - Given a collection we must be able to find the boxes.
  - There does not seem to be a need to traverse the other way. There likely is an ordering of boxes, regarding their foreground / background hierarchy for visibility
- **Collection –> ordered Link**
  - Same explanation as

    Collection –> ordered Box