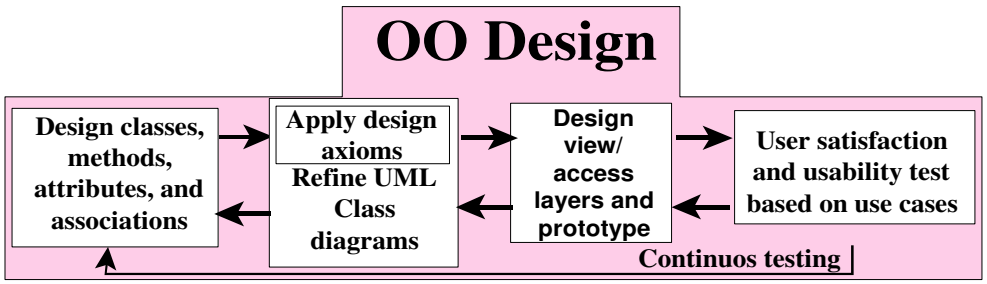
SCSX1010 OBJECT ORIENTED ANALYSIS AND DESIGN
Unit - IV



4.1 OO Design Process

1. Apply design axioms to design classes, their attributes, methods, associations, structures, and protocols.
   - 1.1. Refine and complete the static UML class diagram (object model) by adding details to the UML class diagram. This step consists of the following activities:
   - 1.1.1. Refine attributes.
     - 1.1.2. Design methods and protocols by utilizing a UML activity diagram to represent the method's algorithm.
     - 1.1.3. Refine associations between classes (if required).
     - 1.1.4. Refine class hierarchy and design with inheritance (if required).
   - 1.2. Iterate and refine again.

2. Design the access layer
   -2.1. Create mirror classes. For every business class identified and created, create one access class.
     - 2.2. define relationships among access layer classes.
     - 2.3. Simplify the class relationships. The main goal here is to eliminate redundant classes and structures.
       - 2.3.1. Redundant classes: Do not keep two classes that perform similar translate request and translate results activities. Simply select one and eliminate the other.
       - 2.3.2. Method classes: Revisit the classes that consist of only one or two methods to see if they can be eliminated or combined with existing classes.
     - 2.4. Iterate and refine again**.**

3. Design the view layer classes.
   - 3.1. Design the macro level user interface, identifying view layer objects.
   - 3.2. Design the micro level user interface, which includes these activities:
     - 3.2.1. Design the view layer objects by applying the design axioms and corollaries.
     - 3.2.2. Build a prototype of the view layer interface.
   - 3.3. Test usability and user satisfaction (Chapters 13 and 14).
   - 3.4. Iterate and refine.

4. Iterate and refine the preceding steps. Reapply the design axioms and, if needed, repeat the preceding steps.

4.1 Axioms, Theorems and Corollaries

- An axiom is a fundamental truth that always is observed to be valid and for which there is no counterexample or exception.
- A theorem is a proposition that may not be self-evident but can be proven from accepted axioms.
- A Corollary is a proposition that follows from an axiom or another proposition that has been proven.

Design Axioms

- Axiom 1 deals with relationships between system components (such as classes, requirements, software components).
- Axiom 2 deals with the complexity of design.

Axioms

- Axiom 1. The independence axiom. Maintain the independence of components.
- Axiom 2. The information axiom. Minimize the information content of the design.

Occam's Razor

The best theory explains the known facts with a minimum amount of complexity and maximum simplicity and straightforwardness.

Corollaries

- Corollary 1. Uncoupled design with less information content.
- Corollary 2. Single purpose. Each class must have single, clearly defined purpose.
- Corollary 3. Large number of simple classes. Keeping the classes simple allows reusability.
- Corollary 4. Strong mapping. There must be a strong association between the analysis's object and design's object.
- Corollary 5. Standardization. Promote standardization by designing interchangeable components and reusing existing classes or components.
- Corollary 6. Design with inheritance. Common behavior (methods) must be moved to superclasses.
- The superclass-subclass structure must make logical sense.

Coupling and Cohesion

- Coupling is a measure of the strength of association among objects.
- Cohesion is interactions within a single object or software component.
- Tightly Coupled Object
- **Corollary 1-** Uncoupled Design with Less Information Content

  The main goal here is to maximize objects (or software components) cohesiveness**.**
- **Corollary 2 -** Single Purpose

  Each class must have a purpose, as was explained !

  When you document a class, you should be able to easily explain its purpose in a sentence or two.
- **Corollary 3-** Large Number of Simpler Classes, Reusability

  A great benefit results from having a large number of simpler classes. The less specialized the classes are, the more likely they will be reused.

- **Corollary 4**. Strong Mapping

As the model progresses from analysis to implementation, more detail is added, but it remains essentially the same.A strong mapping links classes identified during analysis and classes designed during the design phase.

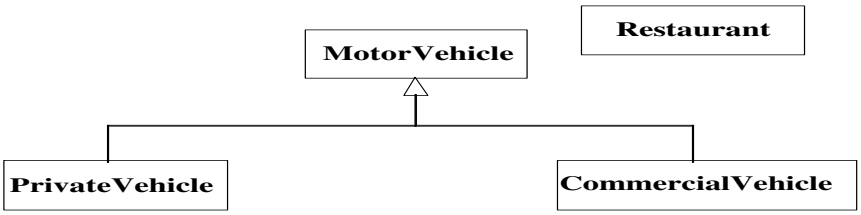
- **Corollary 5.** Standardization

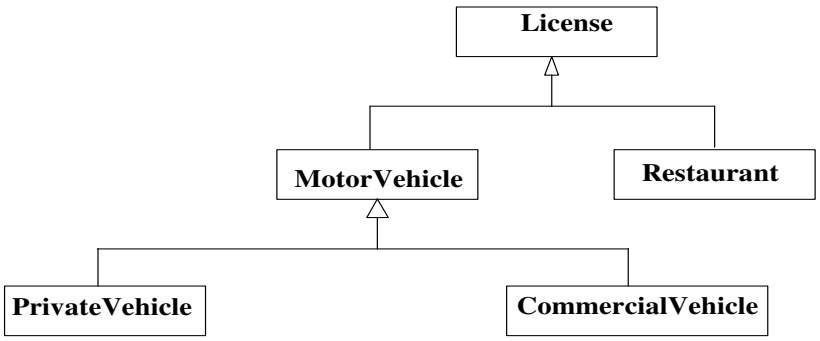
  The concept of design patterns might provide a way for standardization by capturing the design knowledge, documenting it, and storing it in a repository that can be shared and reused in different applications.

- **Corollary 6.** Designing with Inheritance

  Say we are developing an application for the government that manages the licensing procedure for a variety of regulated entities. Let us focus on just two types of entities: motor vehicles and restaurants.

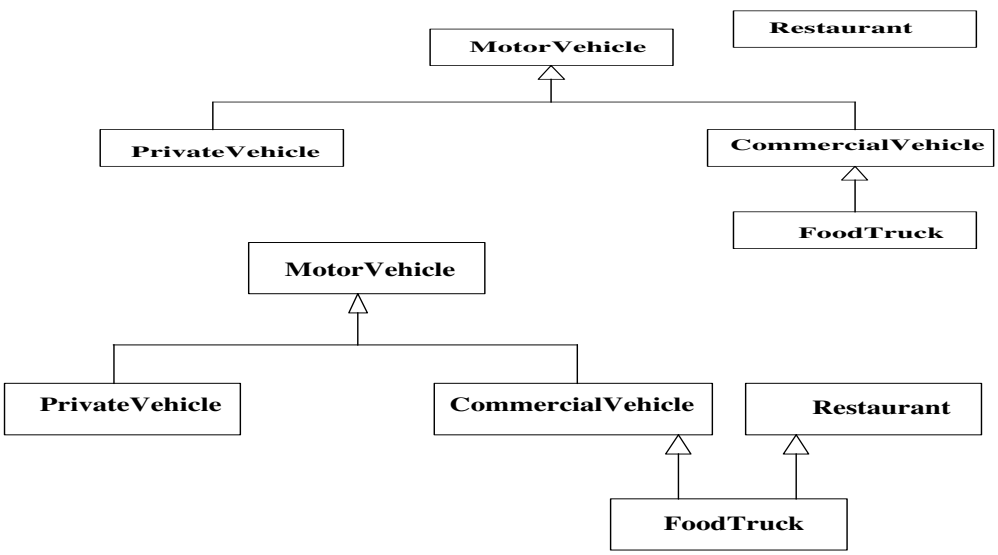
Designing With Inheritance



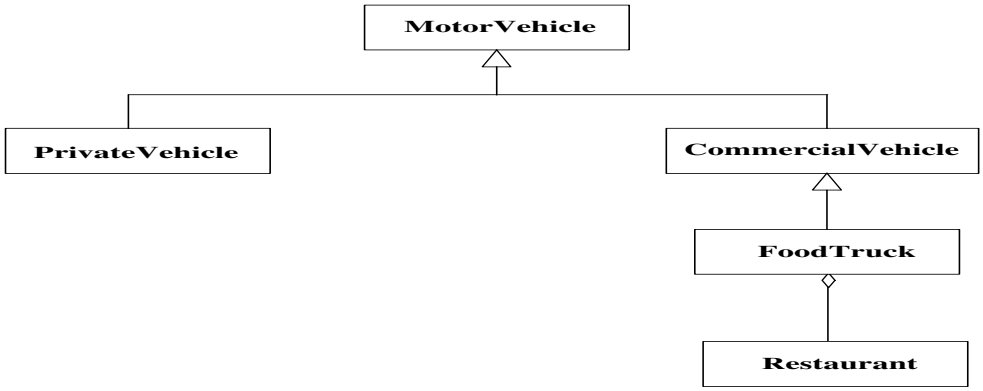


Designing With Inheritance Weak Formal Class

- MotorVehicle and Restaurant classes do not have much in common.
- For example, of what use is the gross weight of a diner or the address of a truck?



Achieving Multiple Inheritance using Single Inheritance Approach



Avoid Inheriting Inappropriate Behaviors

You should ask the following questions:

I. Is the subclass fundamentally similar to its superclass? or,

II. Is it entirely a new thing that simply wants to borrow some expertise from its superclass?

4.2 Designing Classes

OO Design Philosophy

- The first step in building an application should be to design a set of classes, each of which has a specific expertise and all of which can work together in useful ways.

Designing Class: the Process

1. Apply design axioms to design classes, their attributes, methods, associations, structures, and protocols.
   - 1.1. Refine and complete the static UML class diagram (object model) by adding details to that diagram.
     - 1.1.1. Refine attributes.
     - 1.1.2. Design methods and the protocols by utilizing a UML activity diagram to represent the method's algorithm..
     - 1.1.3. Refine the associations between classes (if required).
     - 1.1.4. Refine the class hierarchy and design with inheritance (if required).
   - 1.2. Iterate and refine again.

Classes

- In analysis modeling only the following needs to be shown:

Class name;
key attributes;
key operations
stereotypes(if they have any business significance)

Name compartment

- Class name is in UpperCamelcase
- Special symbols such as punctuation marks, dashes,underscores,ampersands,hashes, and slashes are always avoided andcan

lead to unexpected cnsequences when code or Html/Xml documentation is generated from the model.

- Avoid abbreviations at all costs
- Domain specific acronyms can be used(eg. CRM-Customer Relationship Management)

Attribute Component

- Attributes are named in lowerCamelCase-Starting with a lowercase letter and then mixed upper-and lowercase.

visibilityname:type[multiplicity]=initialValue

Visibility

- Visibility controls access to the features of a class.

+ ->Public visibility
- ->Private visibility
->protected visibility
~ -> package visibility

Object Construction and destruction

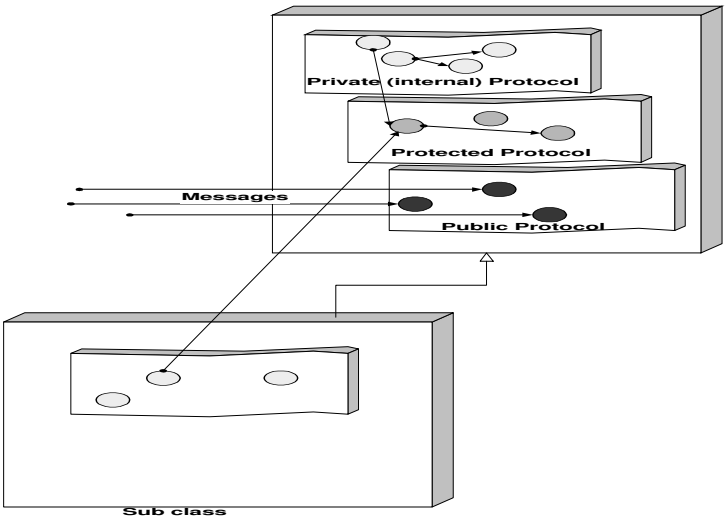
- Constructors are a design consideration and are generally not shown on analysis models.

Destructors

- Destructors are special operations that “clean up” when objects are destroyed.
- Different languages follow different algorithms for clean up.
- In java “Garbage Collection”

Class Visibility

- In designing methods or attributes for classes, you are confronted with two issues.
  - One is the protocol, or interface to the class operations and its visibility;
  - and how it should be implemented.
- Public protocols define the functionality and external messages of an object, while private protocols define the implementation of an object.



Private Protocol (Visibility)

- A set of methods that are used only internally.
- Object messages to itself.
- Define the implementation of the object (Internal).
- Issues are: deciding what should be private.

  - What attributes
  - What methods
- In a protected protocol, subclasses can use the method in addition to the class itself.
- In private protocols, only the class itself can use the method.

Public Protocol (Visibility)

- Defines the functionality of the object
- Decide what should be public (External).

Guidelines for Designing Protocols

- Good design allows for polymorphism**.**
- Not all protocols should be public, again apply design axioms and corollaries.
- The following key questions must be answered:
  - What are the class interfaces and protocols?
  - What public (external) protocol will be used or what external messages must the system understand?
  - What private or protected (internal) protocol will be used or what internal messages or messages from a subclass must the system understand?

Attribute Types

- The three basic types of attributes are:
  - 1. Single-value attributes.
  - 2. Multiplicity or multivalue attributes.
  - 3. Reference to another object, or instance connection.

Designing Methods and Protocols

- A class can provide several types of methods:
  - Constructor. Method that creates instances (objects) of the class.
  - Destructor. The method that destroys instances.
  - Conversion method. The method that converts a value from one unit of measure to another.
  - Copy method. The method that copies the contents of one instance to another instance.
  - Attribute set. The method that sets the values of one or more attributes.
  - Attribute get. The method that returns the values of one or more attributes
  - I/O methods. The methods that provide or receive data to or from a device.
  - Domain specific. The method specific to the application.

Five Rules For Identifying Bad Design

- If it looks messy then it's probably a bad design.
- If it is too complex then it's probably a bad design.
- If it is too big then it's probably a bad design.
- If people don't like it then it's probably a bad design.
- If it doesn't work then it's probably a bad design.

Avoiding Design Pitfalls

- Keep a careful eye on the class design and make sure that an object's role remains well defined.
- If an object loses focus, you need to modify the design.
- Apply Corollary 2 (single purpose).
- Move some functions into new classes that the object would use.
- Apply Corollary 1 (uncoupled design with less information content).
- Break up the class into two or more classes.
- Apply Corollary 3 (large number of simple classes).

Access Layer

Introduction

- A database management system (DBMS) is a collection of related data and associated programs that access, manipulate, protect and manage data.

What's the purpose of DBMS

- The main purpose of a DBMS is to provide a reliable, persistent data storage facility,and mechanism for efficient and convenient data access and retrieval.

Persistence (review)

- Persistence is defined as objects that outlive the programs which created them.
- Persistent object stores do not support query or interactive user interface facilities, as found in a fully supported DBMS or OODBMS.

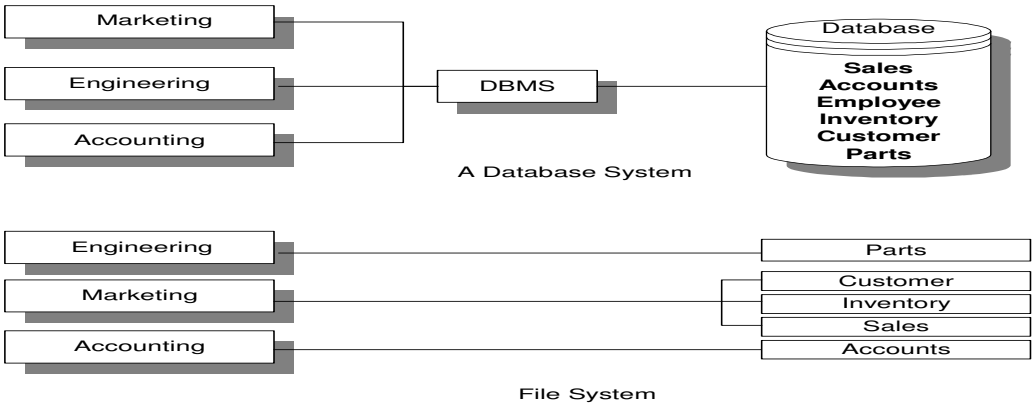
Object Storage and Persistence

- Atkinson et al. describe six broad categories for the lifetime of data:
  - 1. Transient results to the evaluation of expressions.
  - 2. Variables involved in procedure activation (parameters and variables with a localized scope).
  - 3. Global variables and variables that are dynamically allocated.
  - 4. Data that exist between the executions of a program.
  - 5. Data that exist between the versions of a program.
  - 6. Data that outlive a program.

Database Management Systems

- A DBMS is a set of programs that enable the creation and maintenance of a collection of related data.
- DBMS have a number of properties that distinguish them from the file-based data management approach.

Database system Vs. File System**.**



Database Models

- A database model is a collection of logical constructs used to represent the data structure and data relationships within the database.
  - Hierarchical Model
  - Network Model
  - Relational Model

Hierarchical model

The hierarchical model represents data as a single-rooted tree

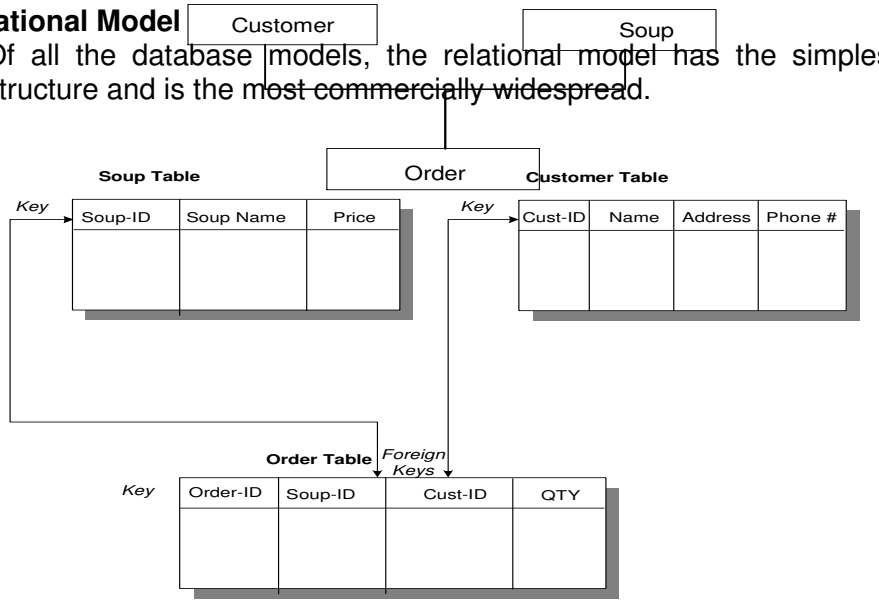
Network model

A network database model is similar to a hierarchical database, with one distinction.

Unlike the hierarchical model, a network model's record can have more than one parent.

Relational Model

- Of all the database models, the relational model has the simplest, most uniform structure and is the most commercially widespread.



What is a schema and metadata?

- The schema, or metadata, contains a complete definition of the data formats, such as the data structures, types, and constraints**.**
- In an object-oriented DBMS, the schema is the collection of class definitions.
- The relationships among classes (such as super/sub) are maintained as part of the schema.

Database Definition Language (DDL)

- A database definition language (DDL) is used to describe the structure of and relationships between objects stored in a database.

Data Manipulation Language (DML)

- Once data is stored in a database, there must be a way to get it, use it, and manipulate it.
- DML is a language that allows users to access and manipulate (such as: creation, saving and destruction of) data organization.
- The Structured Query Language (SQL) is the standard DML for relational DBMS.
- In a relational DBMS, the DML is independent from a host programming language.
- For example, a host language such as C or COBOL would be used to write the body of the application.
- SQL statements are then typically embedded in C or COBOL applications to manipulate data.

Sharability

- Data in the database often needs to be accessed and shared by different applications.
- The database then must detect and mediate the conflicts and promote the greatest amount of sharing possible without sacrificing the integrity of data.

Transaction

- A transaction is a unit of change, in which either all changes to objects within a transaction will be applied or not at all.

- A transaction is said to commit if all changes can be successfully made to the database and to abort if all changes cannot be successfully made to the database.

Concurrency Control

- Programs will attempt to read and write the same pieces of information simultaneously and, in doing so, create a contention for data.
- The concurrency control mechanism is thus established to mediate these conflicts.
- It does so by making policies that dictate how read and write conflicts will be handled.
- The most conservative way is to allow a user to lock all records or objects when they are accessed and to release the locks only after a transaction commits**.**
- By distinguishing between reading the data, and writing it (which is achieved by qualifying the type of lock placed in the data-read lock or write lock) somewhat greater concurrency can be achieved.
- This policy allows many readers of a record or an objective, but only one writer.

Distributed Databases

- In distributed databases portions of the database reside on different nodes (computers) in the network.

Client/Server Computing

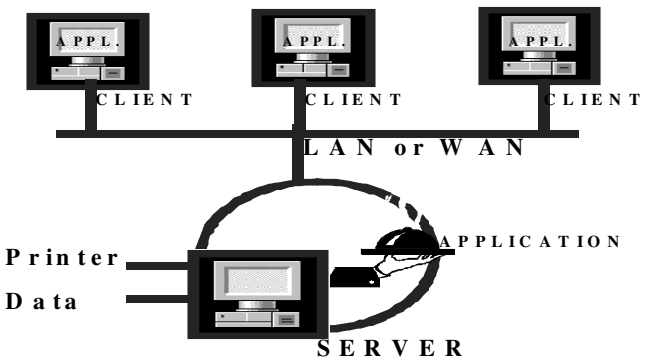
- Client/server computing allows objects to be executed in different memory spaces or even different machines.
- The calling module becomes the "client" (which requests a service), and the called module becomes the "server" (which provides the service).

Client programs usually manage:

- The user-interface
- Validate data entered by the user
- Dispatch requests to server programs, and sometimes
- Execute business logic.
- The Business layer contains all of the objects that represent the business such as:
  - Order
  - Customer
  - Line Item
  - Inventory, etc.
- A server process (program) fulfills the client request by performing the task requested.
- Server programs generally receive requests from client programs, execute database retrieval and updates, manage data integrity, and dispatch responses to client requests.

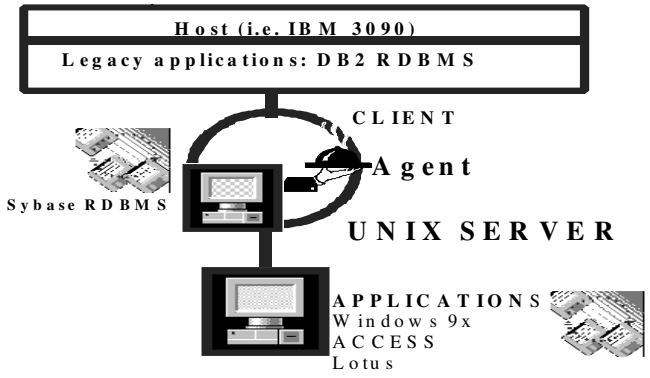
A Two-Tier Architecture

- A two-tier architecture is one where a client talks directly to a server, with no intervening server.
- This type of architecture is typically used in small environments with less than 50 users.



A Three-Tier Architecture

- A three-tier architecture introduces another server (or an "agent") between the client and the server.
- The role of the agent is many fold.
- It can provide translation services as in adapting a legacy application on a mainframe to a client/server environment.



Basic Characteristics of Client/Server Architectures

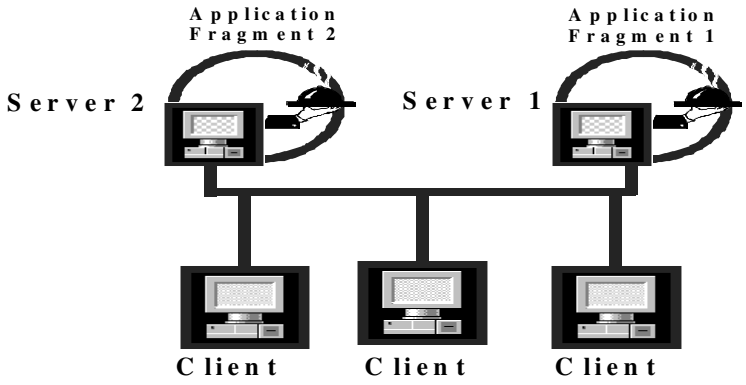
1. The client or front-end interacts with the user, and a server or back-end interacts with the shared resource.
2. The front-end task and back-end task have fundamentally different requirements for computing resources.
   Resources such as processor speeds, memory, disk speeds and capacities, and input/output devices.
- The environment is typically heterogeneous and multi-vendor.

4. Client-server systems can be scaled horizontally or vertically.

- Horizontal scaling means adding or removing client workstations with only a slight performance impact.
- Vertical scaling means migrating to a larger and faster server machine or multi servers.

Distributed Processing

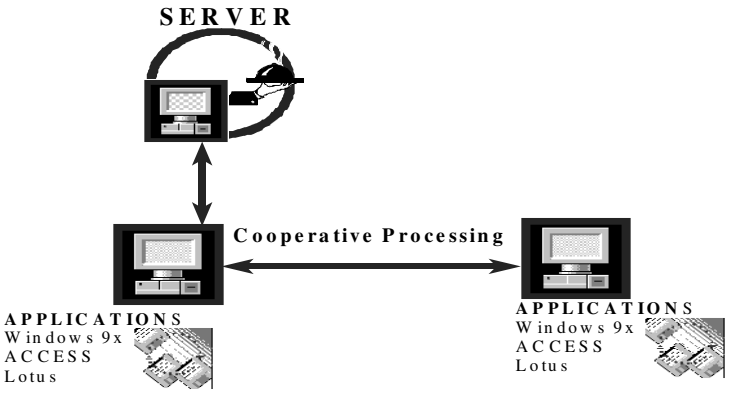
- Distributed processing implies that processing will occur on more than one processor in order for a transaction to be completed.



For example, in processing an order from our client, the client information may process at one machine and the account information will then be processed next on a different machine**.**

Cooperative processing

Cooperative processing is a form of distributed computing where two or more distinct processes are required to complete a single business transaction.



Client/Server Components

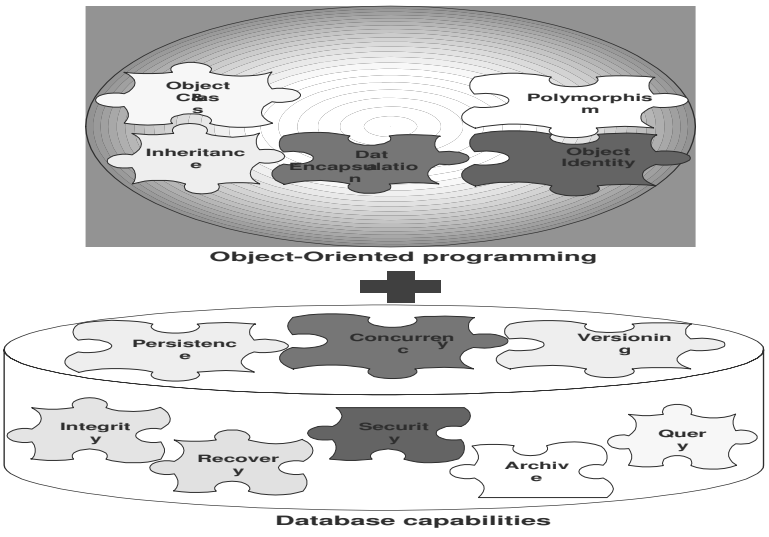
I. User Interface Layer: This is one of the major components of the client/server application.
It interacts with users, screens, Windows, Window management, keyboard, and mouse handling.

II. Business Processing Layer: This is a part of the application that uses the user interface data to perform business tasks.

III. Database Processing Layer: This is a part of the application code that manipulates data within the application.

Object-Oriented Database Systems

- The object-oriented database management system is a marriage of object-oriented programming and database technology to provide what we now call object-oriented databases.



Object-Oriented Database System Manifesto

- Malcolm Atkinson et al. described the necessary characteristics that a system must satisfy to be considered an object-oriented database.
- These categories can be broadly divided into object-oriented language properties and database requirements.
- First, the rules that make it an object-oriented system are as follows:
  - 1. The system must support complex objects.
  - 2. Object identity must be supported.
  - 3. Objects must be encapsulated.
  - 4. The system must support types or classes.
  - 5. The system must support inheritance.
  - 6. The system must avoid premature binding.
  - 7. The system must be computationally complete.
  - 8. The system must be extensible.
- Second, these rules make it a DBMS:
  - 9. It must be persistent, able to remember an object state.
  - 10. It must be able to manage very large databases.
  - 11. It must accept concurrent users.
  - 12. It must be able to recover from hardware and software failures.
  - 13. Data query must be simple.

Object-Oriented Databases versus Traditional Databases

- The objects are an "active" component in an object-oriented database.
- Relational database systems do not explicitly provide inheritance of attributes and methods.
- Each object has its own identity, or object-ID (as opposed to the purely value-oriented approach of traditional databases).
- Object identity allows objects to be related as well as shared within a distributed computing network.
- Object-Relational Systems: The Practical World
- In practice, even though many applications increasingly are developed in an object-oriented environment, chances are good that the data those applications need to access live in a very different universe—a relational database.

Object-Relation Mapping

- For a tool to be able to define how relational data maps to and from application objects, it must have at least the following mapping capabilities:
- Table-class mapping.
- Table-multiple classes mapping.
- Table-inherited classes mapping.
- Tables-inherited classes mapping.

Table-Class Mapping

- Each row in the table represents an object instance and each column in the table corresponds to an object attribute.

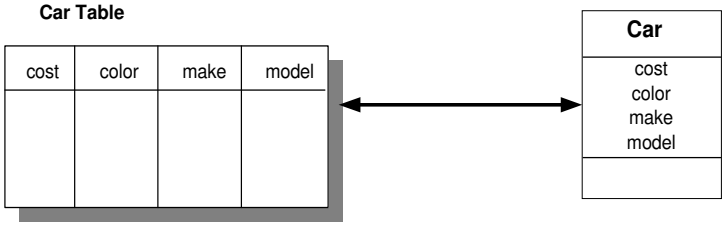


Table-Multiple Classes Mapping

- The custID column provides the discriminant. If the value for custID is null, an Employee instance is created at run time; otherwise, a Customer instance is created.

Table-Inherited Classes Mapping

- Instances of SalariedEmployee can be created for any row in the Person table that has a non null value for the salary column. If salary is null, the row is represented by an HourlyEmployee instance.

Tables-Inherited Classes Mapping

- Instances of Person are mapped directly from the Person table. However, instances of Employee can be created only for the rows in the Employee table (the joins of the Employee and Person tables on the ssn key). The ssn is used both as a primary key on the Person table and as a foreign key on the Person table and a primary key on the Employee table for activating instances of type Employee.

Keys for Instance Navigation

- The departmentID property of Employee uses the foreign key in column Employee.departmentID. Each Employee instance has a direct reference of class Department (association) to the department object to which it belongs.

Multidatabase System

- A different approach for integrating object-oriented applications with relational data environments is multidatabase systems, or heterogeneous database systems, which facilitate the integration of heterogeneous databases and other information sources.

Federated Multidatabase Systems

- Federated multidatabase systems provide a solution to the problem of interoperating heterogeneous data systems, provide uniform access to data stored in multiple databases that involve several different data models.

MDBS

- A multidatabase system (MDBS) is a database system that resides on top of, say existing relational and object databases and file systems (called local database systems) and presents a single database illusion to its users.

Characteristics of MDBS

- Automatic generation of a unified global database schema from local databases.
- Provision of cross-database functionality (global queries, updates, and transactions) by using unified schemata.
- Integration of a heterogeneous database system with multiple databases.
- Integration of data types other than relational data through the use of such tools as driver generators.
- Provision of a uniform but diverse set of interfaces (e.g., a SQL-style interface, browsing tools, and C++) to access and manipulate data stored in local databases.

Open Database Connectivity

- Open database connectivity (ODBC), provides a mechanism for creating a virtual DBMS.

Designing Access Layer Classes

- The main idea behind creating an access layer is to create a set of classes that know how to communicate with data source, whether it be a file, relational database, mainframe, Internet, DCOM, or via ORB.

- The access classes must be able to translate any data-related requests from the business layer into the appropriate protocol for data access.
- The business layer objects and view layer objects should not directly access the database. Instead, they should consult with the access layer for all external system connectivity.

4.3 Designing Access Layer Classes

- The main idea behind creating an access layer is to create a set of classes that know how to communicate with data source, whether it be a file, relational database, mainframe, Internet, DCOM, or via ORB.
- The access classes must be able to translate any data-related requests from the business layer into the appropriate protocol for data access.
- The business layer objects and view layer objects should not directly access the database. Instead, they should consult with the access layer for all external system connectivity.

The access layer performs two major tasks:

1.Translate the request.

2. Translate the results.

Benefits of Access Layer Classes

- Access layer classes provide easy migration to emerging distributed object technology, such as CORBA and DCOM.
- These classes should be able to address the (relatively) modest needs of two-tier client/server architectures as well as the difficult demands of fine-grained, peer-to-peer distributed object architectures.

Process

- The access layer design process consists of these following activities:

If a class interacts with a nonhuman actor, such as another system, database or the web

1. For every business class identified, mirror the business class package.
2. Define relationships. The same rule as applies among business class objects also applies among access classes.
3. Simplify classes and relationships. The main goal here is to eliminate redundant or unnecessary classes or structures.
   3.1. Redundant classes. If you have more than one class that provides similar services, simply select one and eliminate the other(s).
   3.2. Method classes. Revisit the classes that consist of only one or two methods to see if they can be eliminated or combined with existing classes.
4. Iterate and refine.

Another approach is to let the methods be stored in a program (ex: A compiled c++ program stored on a file)

1. For every business class identified, determine if the class has persistent data.
2. Mirror the business class package.
3. Define relationships. The same rule as applies among business class objects also applies among access classes.
4. Simplify classes and relationships. The main goal here is to eliminate redundant or unnecessary classes or structures.
   4.1. Redundant classes. If you have more than one class that provides similar services, simply select one and eliminate the other(s).
   4.2. Method classes. Revisit the classes that consist of only one or two methods to see if they can be eliminated or combined with existing classes.
5. Iterate and refine

4.5 Designing View Layer Classes

- interface objects
  - The only exposed objects of an application with which users can interact
  - Represent the set of operations in the business that users must perform to complete their tasks

The view layer classes are responsible for two major aspects of the applications:

- Input-Responding to user interaction
  - User interface must be designed to translate an action by the user
- Output-Displaying business objects

Design of the view layer classes are divided into the following activities:

I. Macro Level UI Design Process- Identifying View Layer Objects.
II. Micro Level UI Design Activities.
  - Designing the view layer objects by applying design axioms
  - Prototyping the view layer interface

III. Usability and User Satisfaction Testing.
IV. Refine and Iterate.

Macro-Level Process – By analyzing usecases

1. For Every Class Identified
   1.1 Determine If the Class Interacts With Human Actor: If yes, do next step otherwise move to next class.
   1.1.1 Identify the View (Interface) Objects for The Class.
   1.1.2 Define Relationships Among the View (Interface) Objects.
2. Iterate and refine.

Relationships Among Business, Access and View Classes

- In some situations the view class can become a direct aggregate of the access object, as when designing a web interface that must communicate with application/Web server through access objects.

View Layer Micro Level

- Better to design the view layer objects user driven or user centered
- Process of designing view objects
  1. For Every Interface Object Identified in the Macro UI Design Process.
     1.1 Apply Micro Level UI Design Rules and Corollaries to Develop the UI.
  2. Iterate and refine.

UI Design Rules

- Rule 1- Making the Interface Simple
- Rule 2- Making the Interface Transparent and Natural
- Rule 3- Allowing Users to Be in Control of the Software

UI Design Rule 1

- Making the interface simple: application of corollary 2.
- Keep It Simple.
- Simplicity is different from being simplistic.
- Making something simple requires a good deal of work and code.
- Every additional feature potentially affects performance, complexity, stability, maintenance, and support costs of an application.
- A design problem is harder to fix after the release of a product because users may adapt, or even become dependent on, a peculiarity in the design.

UI Design Rule 2

- Making the interface transparent and Natural: application of corollary 4.
- Corollary 4 implies that there should be strong mapping between the user's view of doing things and UI classes.

Making The Interface Natural

- The user interface should be intuitive so users can anticipate what to do next by applying their previous knowledge of doing tasks without a computer.

Using Metaphors

- Metaphore, relates two unrelated things by using one to denote the other
- Metaphors can assist the users to transfer their previous knowledge from their work environment to your application interface.
- For example, question mark to label a Help button.

UI Design Rule 3

- Allowing users to be in control of the software: application of corollary 1.
- Users should always feel in control of the software, rather than feeling controlled by the software.

Allowing Users Control of the Software

- Some of the ways to put users in control are:
  - Making the interface forgiving.
  - Making the interface visual.
  - Providing immediate feedback.
  - Making the interface consistent.

Making the Interface Forgiving

- Users should be able to back up or undo their previous action.
- They should be able to explore without fear of causing an irreversible mistake.

Making the Interface Visual

- You should make your interface highly visual so users can see, rather than recall, how to proceed.
- Whenever possible, provide users with a list of items from which they can choose.

Providing Immediate Feedback

- Users should never press a key or select an action without receiving immediate visual feedback, audible feedback, or both.

Making the Interface Consistent

- User Interfaces should be consistent throughout the applications.
- For example, keeping button locations consistent make users feel in control.

Purpose of a View Layer Interface

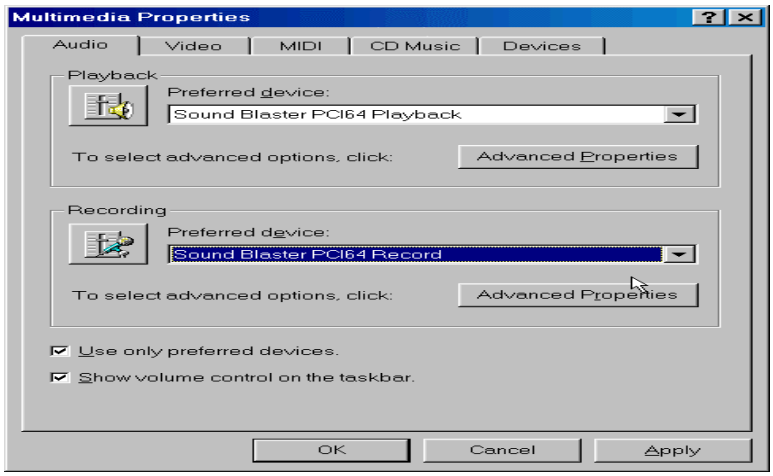
- Data Entry Windows: Provide access to data that users can retrieve, display, and change in the application.
- Dialog Boxes: Display status information or ask users to supply information.
- Application Windows (Main Windows): Contain an entire application that users can launch.

Guidelines For Designing Data Entry Windows

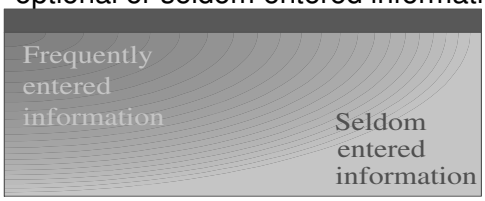
- You can use an existing paper form such as a printed invoice form as the starting point for your design.

If the printed form contains too much information to fit on a screen:

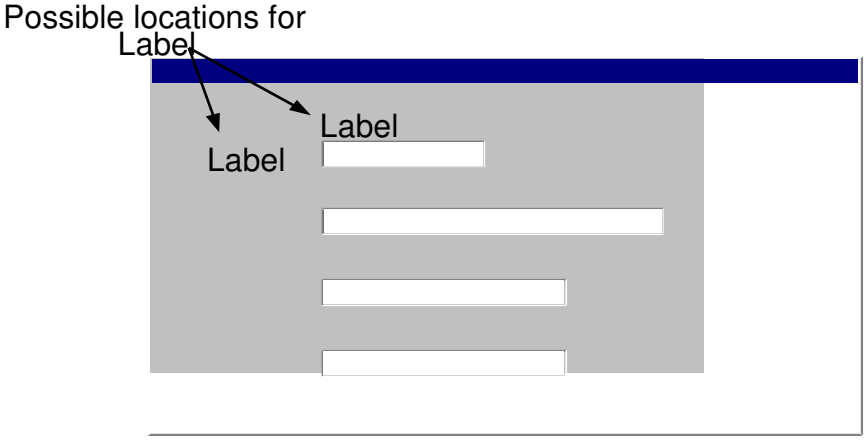
- Use main window with optional smaller Windows that users can display on demand, or
- Use a window with multiple pages.
- Users scan a screen in the same way they read a page of a book, from left to right, and top to bottom.
- An example of a dialog box with multiple pages in the Microsoft multimedia setup.



- Orient the controls in the dialog box in the direction people read.
- Usually left to right, top to bottom.
- Required information should be put toward the top and left side of the form, entering optional or seldom entered information toward the bottom.



- Place text labels to the left of text box controls, align the height of the text with text displayed in the text box.



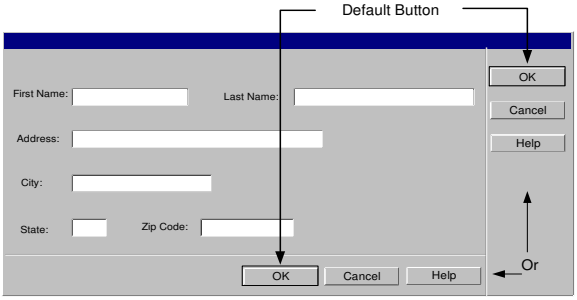
Guidelines For Designing Dialog Boxes

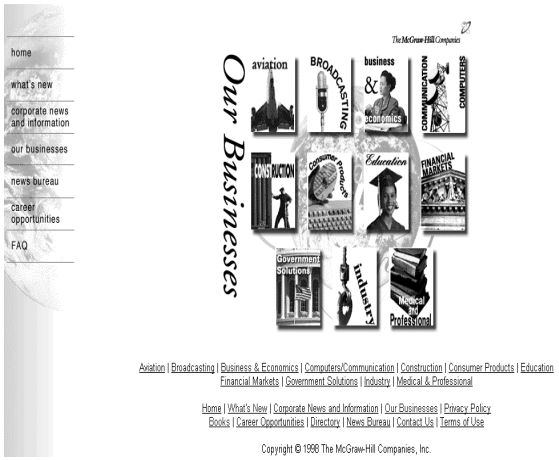
- If the dialog box is for an error message, use the following guidelines:
- Your error message should be positive.
- For example instead of displaying “You have typed an illegal date format,” display this message “Enter date format mm/dd/yyyy.”
- Your error message should be constructive, brief and meaningful.
- For example, avoid messages such as
  “You should know better! Use the OK button”

- instead display
  "Press the Undo button and try again."

Guidelines For The Command Buttons Layout

- Arrange the command buttons either along the upper-right border of the form or dialog box or lined up across the bottom.
- Positioning buttons on the left or center is popular in Web interfaces.





Guidelines For Designing Application Windows

A typical application window consists of a frame (or border) which defines its extent: title bar, scroll bars,menu bars, toolbars, and status bars.

Guidelines For Using Colors

- Use identical or similar colors to indicate related information.
- Use different colors to distinguish groups of information from each other.
- For example, checkout and in-stock tapes could appear in different colors.
- For an object background, use a contrasting but complementary color.
- For example, in an entry field, make sure that the background color contrasts with the data color
- Use bright colors to call attention to certain elements on the screen.
- Use dim colors to make other elements less noticeable.
- For example, you might want to display the required field in a brighter color than optional fields.
- . Use colors consistently within each window and among all Windows in your application.
- For example the colors for Pushbuttons should be the same throughout.

- Using too many colors can be visually distracting, and will make your application less interesting.
- Allow the user to modify the color configuration of your application.

Guidelines For Using Fonts

- Use commonly installed fonts, not specialized fonts that users might not have on their machines.
- Use bold for control labels so they will remain legible when the object is dimmed.
- Use fonts consistently within each form and among all forms in your application.
- For example, the fonts for check box controls should be the same throughout.
- Consistency is reassuring to users, and psychologically makes users feel in control.
- Using too many font styles, sizes and colors can be visually distracting and should be avoided.

Prototyping the User Interface

- Rapid prototyping encourages the incremental development approach, “grow, don’t build.”

Three General Steps

- Create the user interface objects visually.
- Link or assign the appropriate behaviors or actions to these user interface objects and their events.
- Test, debug, then add more by going back to step 1.