Module-2

OBJECT ORIENTED ANALYSIS

- MODULE-2
- COURSE OUTCOMES:
- Classify the various object oriented analysis techniques

- # MODULE II

OBJECT ORIENTED ANALYSIS

- Identifying use cases-Object Analysis-Classification: Theory-Approaches for Identifying Classes: Noun Phrase approach, Common Class pattern approach, Use case driven approach, Classes, Responsibilities and Collaborators-Identifying Object relationships: Associations, Super–sub class relationships, Aggregation.

Object Oriented Analysis Process: Identifying Use Cases

Objectives

At the end of this chapter, students should be able to

1. Define and understand the object-oriented analysis process
2. Explain the use case modelling process
3. Identify actors
4. Identify use cases
5. Developing Effective Documentation

Introduction

- Analysis is the **process of** **extracting the needs of a system** **and** **what the system must do to satisfy the users' requirements**
- Analysis focus on understanding the problem and its domain
- Analysis involves a great deal of interaction with the people who will be affected by the system.
- By constructing models of the system that concentrate on describing **WHAT the system does, rather than HOW it does**

Introduction

- **Analysis** = “process of transforming a problem definition from a fuzzy set of facts and myths (into a) → coherent statement of system’s requirements”
- **Objective:-**
    1. To *capture a complete, unambiguous and consistent picture of the requirements of the system* &
    2. *What the system must do* to satisfy the user’s requirements.

Tools to extract information about a system

1) Examination of existing system documentation
2) Interviews
3) Questionnaire
4) Observation
5) And Also, Literature Survey

Why Analysis is a difficult activity?

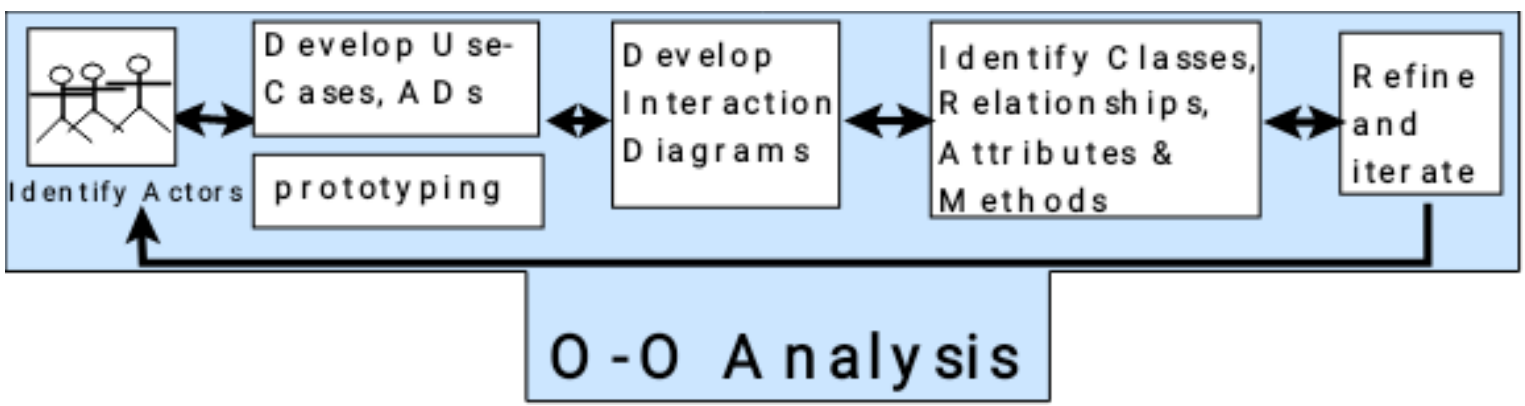
- Analysis is a **creative process**, that involves understanding problem domain, its associated constraints, and methods to overcome those constraints
- Three most common **sources of requirements difficulties** :-
    1. **Fuzzy Description**
    2. **Incomplete Requirements**
    3. **Unnecessary Features**

Business object analysis

- Business object analysis is a process of,
  - ➢ **understanding the system's requirements and**
  - ➢ **establishing the goals of an application**
- The outcome of the business object analysis is to,
  - ➢ identify classes that make up the business layer and
  - ➢ the relationships that play a role in achieving system goals
  - **To understand users' requirements :**

    Find out how they "use" the system, by developing use cases

OOA phase of the unified approach uses actors and use cases to describe the system from users' perspective



OOA process consists of the following steps :

1. Identify the actors :
    - Who is using the system
    - Who will be using the system (in case of a new system)
2. Develop a simple business process model using UML activity diagram
3. Develop the use case :
    - What are the users doing with the system
    - What will users be doing with the new system (in case of a new system)
    - Use cases provide comprehensive documentation of the system under study
4. Prepare interaction diagrams
    - Determine the sequence
    - Develop collaboration diagrams
5. Classification – develop a static UML class diagram :
    - Identify classes
    - Identify relationships
    - Identify attributes
    - Identify methods
6. Iterate and refine : if needed, repeat the preceding steps

Business process modelling

- Not necessary for all project
- When required business process and requirements can be modelled and recorded to any level of detail
- Activity diagram support this modelling
- Disadvantages:-
  - Time consuming process
- Advantages:-
  - familiarity

Use case model

1) Scenario for understanding the system
2) Interaction between user and system
3) Captures users goal and systems responsibility
4) Used to discover classes and relationship
5) Developed by talking to users

- Use case model – Provides external view of the system
- Object model (UML class diagram) – Provides internal view

Use cases and microscope

" A use case is a sequence of transaction in a system whose task is to yield results of measurable value to an individual actor of the system "

Use-Case model

- A use-case model is a model of the system's intended functions (use cases) and its surroundings (actors)
- The same use-case model is used in requirement analysis, design, & test
- Primary purpose is to communicate the system's functionality and behavior to the customer or end user
- Benefits of a use-case model : To communicate with the end users and domain experts

Actor & Use-Case

Actor

– Role played by the user with respect to the system

– Single actor may perform many use cases

– Can be external system

– Can be one get value from the system, or just participate in the use case

“an actor represents anything that interacts with the system ”

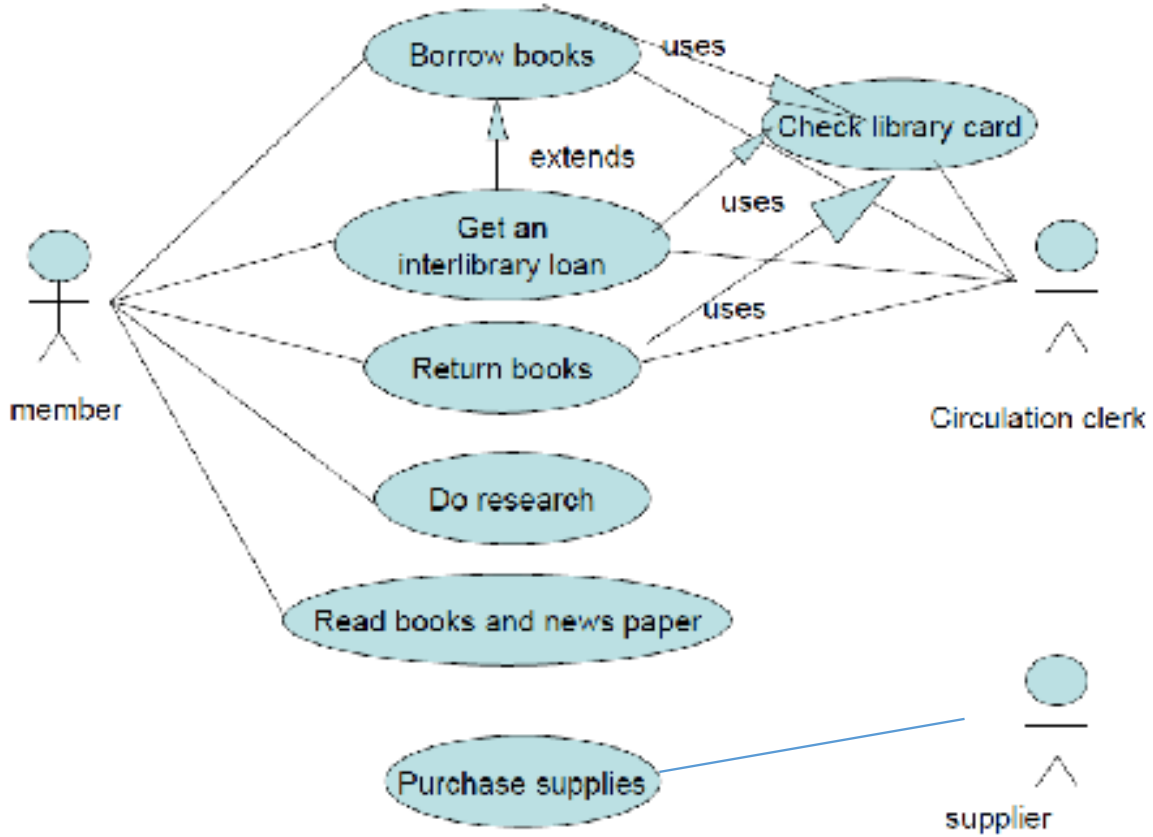
“ a user may play more than one role“

“a use case is a sequence of actions a system performs that yields an observable result of value to a particular actor ”

- Uses Association
  - common sub flows are extracted and separate use case is created
  - Relationship between use-case and extracted one is called uses relationships
- Extends Association
  - Used when use case is similar to other, but do bit more or more Specialized
- Abstract use case
  - No initiating actor
  - Used by concrete use cases
- Concrete use cases
  - Interacts with actors

Identifying actor (cont..)

- Two-three rule
  - Used to identify the actors
  - Start with naming at least 2 or 3 , people who could serve as the actor in the system.other actor can be identified in the subsequent iteration


Borrow books
uses
Check library card
extends
uses
Get an
interlibrary loan
uses
Return books
member
Circulation clerk
Do research
Read books and news paper
Purchase supplies
supplier

How detailed must a use case be? When to stop decomposing it and when to continue

- Develop system use case diagram
- Draw package
- Prepare at least one scenario for each use case
- When the lowest use case level is arrived, which can’t be broken further, sequence and collaboration diagram is drawn

Identifying actors

- Who is using the system? Or,
- Who affects the system? Or
- which user groups are needed by the system to perform its function?
- Which external hardware or other systems use the system to perform tasks?
- Which problem does this application solve?
- How do users use the system ? What are they doing with the system?

Guidelines for Finding Use Cases

- For each actor, find the tasks and functions that the actor should be able to perform or that the system needs the actor to perform.
- Name the use cases
- Describe the use cases briefly by applying terms with which the user is familiar.

Dividing Use Case into Packages

- Whole system is divided into many packages
- Each package encompasses multiple use cases

Naming a Use Case

- Provide a general description of the use-case function
- Name should express what happens when an instance of use case is performed

Developing effective documentation

- An effective document can serve as a communication vehicle among the project's team members, or it can serve as initial understanding of the requirements.
- Important factor in making a decision about committing (assigning, handover, giving) resources
- Mainly depends on organization’s rules and regulations.

Guidelines for developing effective documentation:

According to Bell and Evans:

- Common cover *[common cover sheet]*
- 80-20 rule *[80% work – 20% document]*
- Familiar vocabulary *[don't use buzz words]*
- Make the document as short as possible *[eliminate all repetition]*
- Organize the document *[use rules of good organization]*

objectives

- Classification
- Noun Phrase Approach
- Common class pattern approach
- Use-Case Driven Approach
- Class, Responsibilities and collaborators

Classification and Noun phrase approach

Classification

- Classification is the process of checking to see if an object belongs to a category or a class and it is regarded as a basic attribute of human nature.
- A class is a specification of structure, behaviour, and the description of an object.

... Intelligent classification is intellectually hard work, and it best comes about through an incremental and iterative process

Booch

..There is no such thing as the perfect class structure, nor the right set of objects. As in any engineering discipline, our design choice is compromisingly shaped by many competing factors.

Booch

The Challenge of Classification

- Intelligent classification is intellectually hard work and may seem rather arbitrary.
- Martin and Odell have observed in object-oriented analysis and design, that

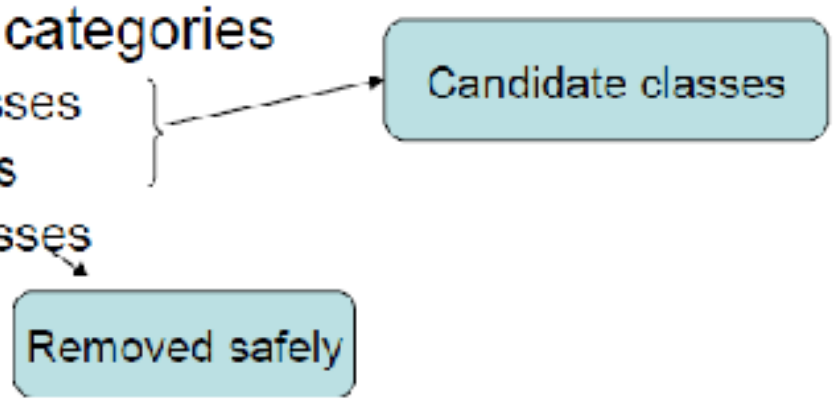
  "In fact, an object can be categorized in more than one way."

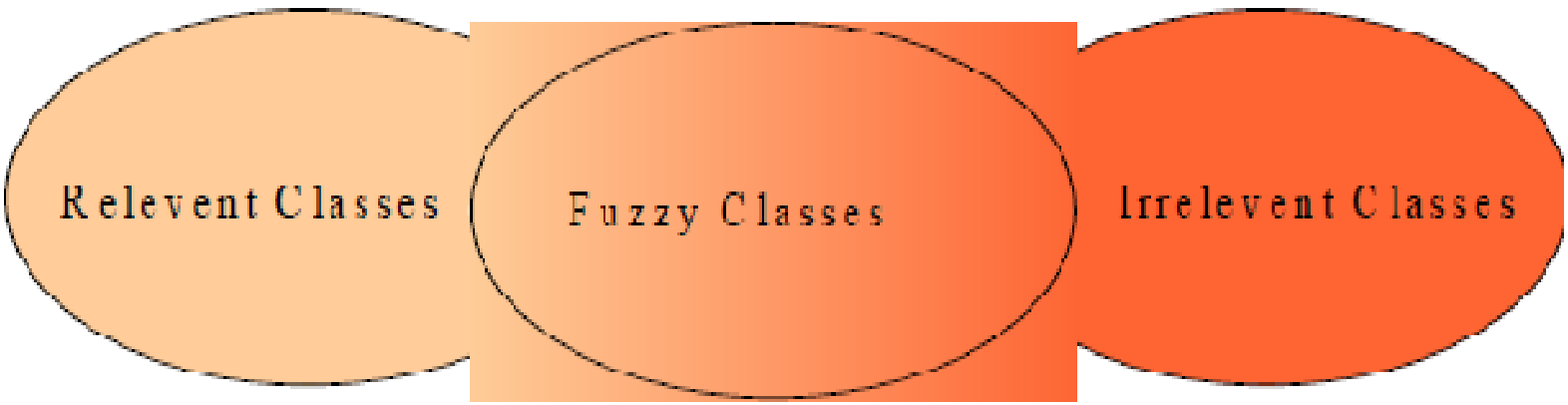
Approaches for Identifying Classes

- The noun phrase approach.
- The common class patterns approach.
- The use-case driven approach.
- Class, Responsibilities and Collaborators (CRC)

Noun phrase approach

- Identify Noun phrases from requirements or use cases
- Nouns - classes
- Verbs - methods
- All plurals ‣ singular
- Create a List of nouns
  - Divided into 3 categories
    - Relevant classes
    - Fuzzy classes
    - Irrelevant classes




Relevent Classes
Fuzzy Classes
Irrelevent Classes

- Identifying tentative classes
- Guidelines
  - Look for nouns and noun phrases in the use case
  - Some classes are implicit and taken from knowledge
  - Avoid computer implementation classes (defer them to design phase). application domain related classes makes sense
  - Carefully choose and define class name

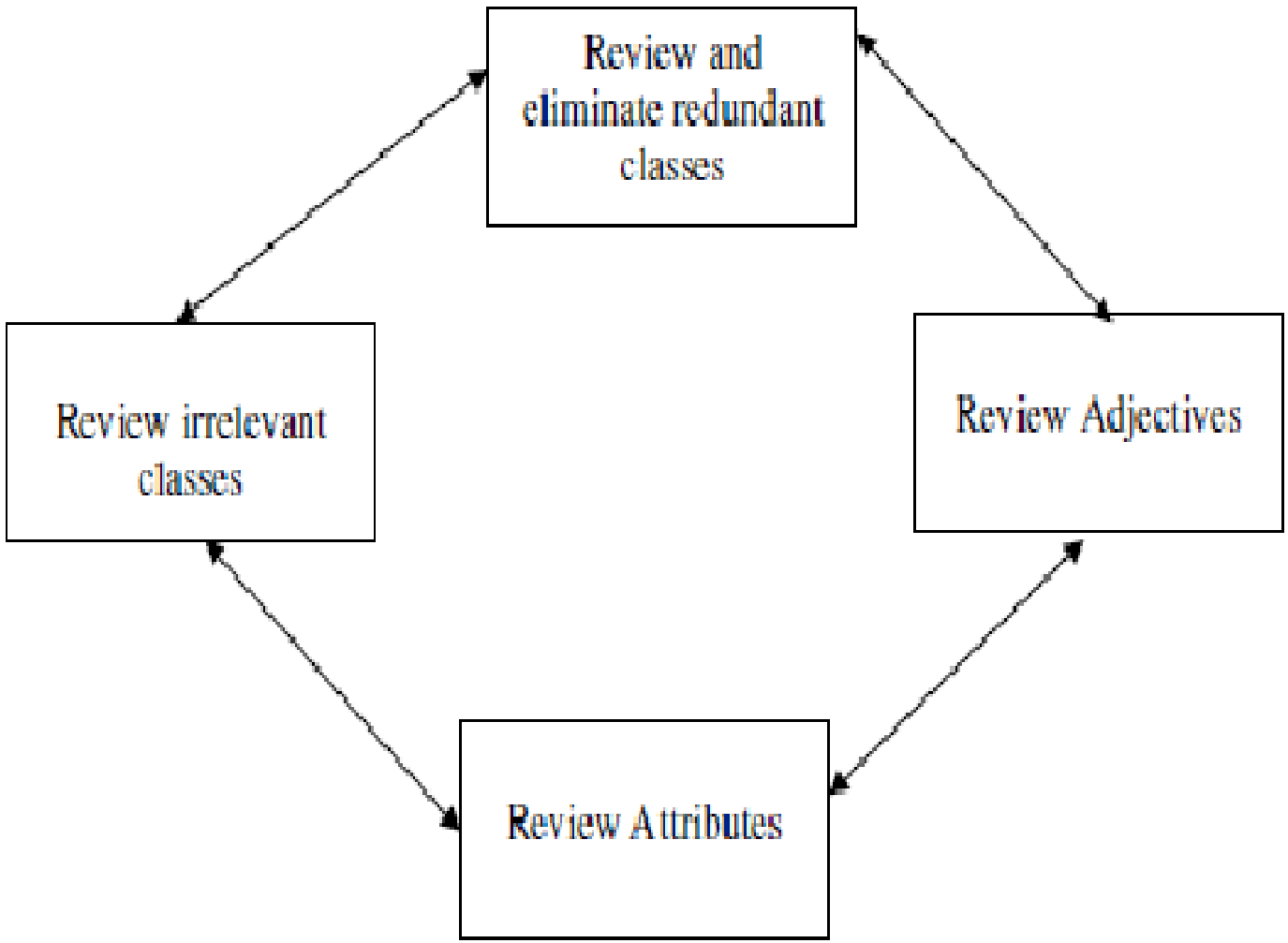
Guidelines :Selecting classes from relevant and fuzzy category

- **Redundant classes**
  - Avoid
  - Choose more meaningful name and name used by user
- **Adjective classes**
  - Adjective can suggest
    - Different kind of object
    - Different use of same object
    - Utterly irrelevant

  eg: adult member and youth member
- **Attrlbute classes**
  - Objects used only as value can be treated as attribute instead of classes
- **Irrelevant classes**
  - Relevant class have statement of purpose.
  - Irrelevant classes - have no statement of purpose

Eliminating redundant classes and refining

1) Review redundant class
2) Review irrelevant class
3) Review adjectives
4) Review attributes


Review and eliminate redundant classes
Review irrelevant classes
Review Adjectives
Review Attributes

Initial list of noun classes : in vianet bank

- Account
- Account balance
- Amount
- Approval process
- Atm card
- Atm machine
- Bank
- Bank client
- Card
- Cash
- Check
- Checking
- Checking account
- Client
- Client's account
- Currency
- Dollar
- Envelope
- Four digits
- Fund
- Invalid pin
- Message
- Money
- Password
- PIN
- Pin code
- Record
- Savings
- Savings account
- Step
- System
- Transaction
- transaction history

Removing irrelevant classes

- Account
- Account balance
- Amount
- Approval process
- Atm card
- Atm machine
- Bank
- Bank client
- Card
- Cash
- Check
- Checking
- Checking account
- Client
- Client's account
- Currency
- Dollar
- ~~Envelope~~
- ~~Four digits~~
- Fund
- Invalid pin
- Message
- Money
- Password
- PIN
- Pin code
- Record
- Savings
- Savings account
- ~~Step~~
- System
- Transaction
- transaction history

Removing redundant classe and building common vocabulary

- Account
- Account balance
- Amount
- Approval process
- Atm card
- Atm machine
- Bank
- Bank client
- ~~Card~~
- Cash
- Check
- ~~Checking~~
- Checking account
- ~~Client~~
- ~~Client's~~ account
- Currency
- Dollar
- ~~Envelope~~
- ~~Four digits~~
- Fund
- Invalid pin
- Message
- ~~Money~~
- Password
- PIN
- ~~Pin code~~
- Record
- ~~Savings~~
- Savings account
- ~~Stop~~
- System
- Transaction
- transaction history

Reviewing the classes containing adjectives

- When class represented by noun behaves differently when adjective is applied to it, then separate class has to be created
- In this ex no such classes

Reviewing the possible attributes

- Noun phrases used only as values should be treated as attributes

Reviewing possible attributes

- Account
- ~~Account balance~~
- ~~Amount~~
- Approval process
- Atm card
- Atm machine
- Bank
- Bank client
- ~~Card~~
- Cash
- Check
- ~~Checking~~
- Checking account
- ~~Client~~
- ~~Client's account~~
- Currency
- Dollar
- ~~Envelope~~
- ~~Four digits~~
- Fund
- ~~Invalid pin~~
- Message
- ~~Money~~
- ~~Password~~
- ~~PIN~~
- Pin code
- Record
- ~~Savings~~
- Savings account
- ~~Stop~~
- System
- Transaction
- ~~transaction history~~

Reviewing the class purpose

- Include classes with
  - Purpose
  - Clear definition
  - Necessary in achieving system goal
- Eliminate classes with no purpose
- Ex: Candidate class with purpose are
  - ATM machine class
  - ATM card class
  - Bankclient class
  - Bank class
  - Account class
  - Checking account class
  - Saving account class
  - Transaction class

Home work

- Apply noun phrase approach for Grocery store problem.

- A store wants to automate its inventory. It has point-of-sale terminals that can record all of the items and quantities that a customer purchases. Another terminal is also available for the customer service desk to handle returns. It has a similar terminal in the loading dock to handle arriving shipments from suppliers. The meat department and produce department have terminals to enter losses/discounts due to spoilage.

- **Step.1**:
- **Identify nouns**
- Store
- Inventory
- Point-of-sale terminal
- Terminals
- Items
- Quantity
- Purchase
- Customer
- Customer service desk
- Handle returns
- Returns
- Loading dock
- Shipment
- Handle shipment
- Suppliers
- Meat department
- Produce department
- Department
- Enter losses
- Enter discount
- Spoilage

- **Step.2:**
- **Eliminate irrelevant nouns**
- Store
- Point-of-sale terminal
- inventory
- Item
- Customer
- Customer service desk
- Handle returns
- Returns
- Handle shipment
- Shipment
- Enter losses
- Enter discount
- Meat department
- Produce department
- Department

- **Step.3**
- **Eliminate redundancies**
- Store
- Point-of-sale terminal
- Item
- Customer service desk
- Handle returns
- Handle shipment
- Meat department
- Produce department
- Enter losses
- Enter discount

- **Step.4:**
- The final set of classes and objects after the elimination process.
- Point-of-sale terminal
- Item
- Handling return
- Handling shipment
- Enter losses
- Enter discount
- Meat department
- Produce department
- Store

Hospital Management System
a comprehensive example

- Using Noun approach

System Domain

- The hospital has several wards divided into male wards & female

wards.

- Each ward has a fixed number of beds. When a patient is admitted

they are placed onto an appropriate ward.

- The Doctors in the hospital are organized into teams such as

Orthopedics A, or Pediatrics, and each team is headed by a consultant doctor. There must be at least one grade 1 junior doctor in

each team.

System Domain

- The Administration department keeps a record of these teams and the doctors allocated to each team.

- A patient on award is under the care of a single team of doctors, with the consultant being the person who is responsible for patient.

- A record is maintained of who has treated the patient, and is must be possible to list the patients on award and the patients treated by a Particular team.

Noun/ Noun phrase:

Hospital → System general

Ward → class

Male wards

Female wards → Property (attribute)

Fixed number of beds → Att(fixed → constraint)

Admitted → relation

Patient → class

Appropriate ward

They (patient)

Doctors → class

Organized → relation

Teams → class

Orthopedics A

Prediatrics → language

Headed by → relation

Consultant doctor → subclass, Property

One grade → attribute

1 junior → subclass

Administrator

department

Record → relation

Under the care

Single team of doctors → class

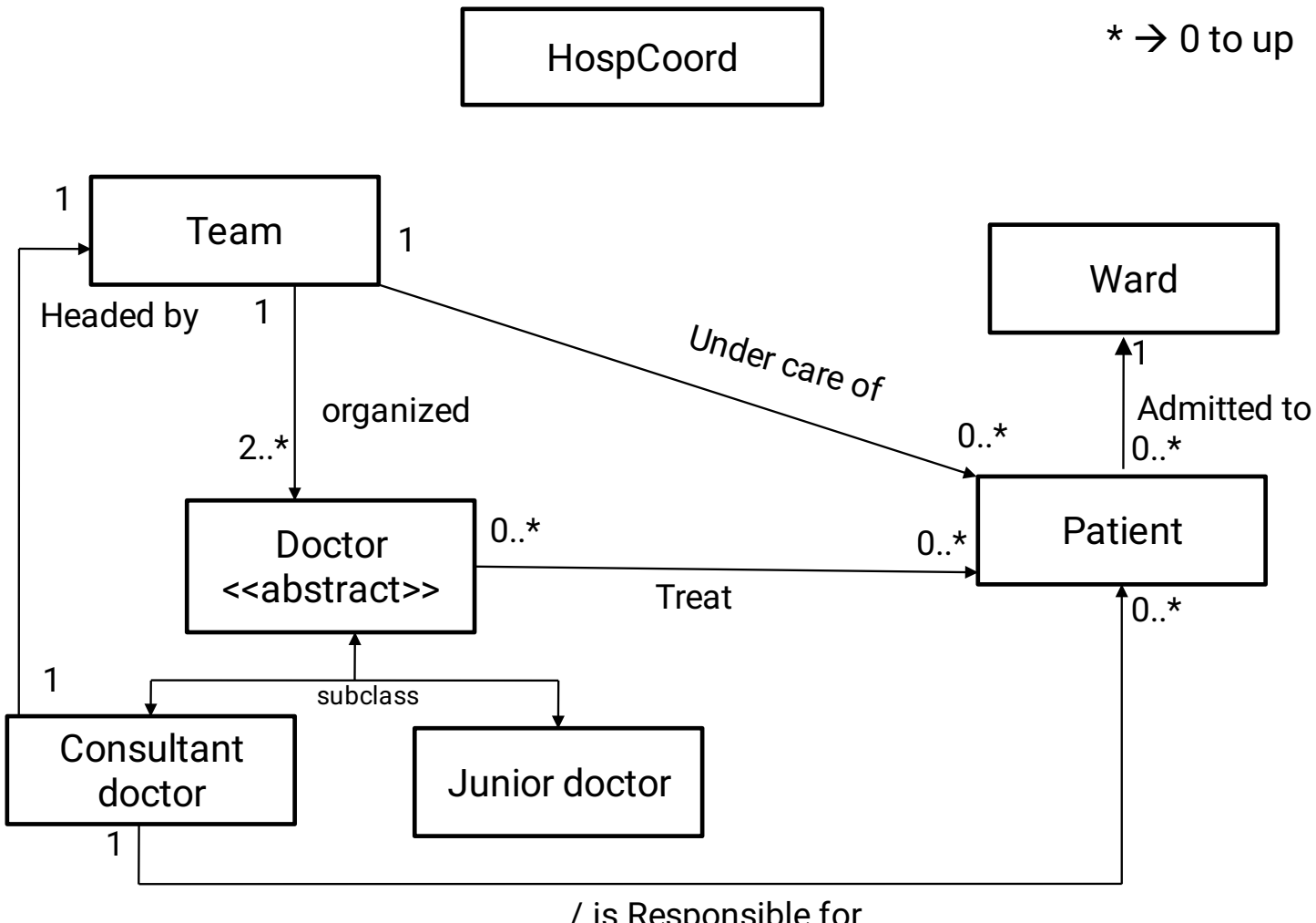
person

Responsible for → relation

List the patient → relation

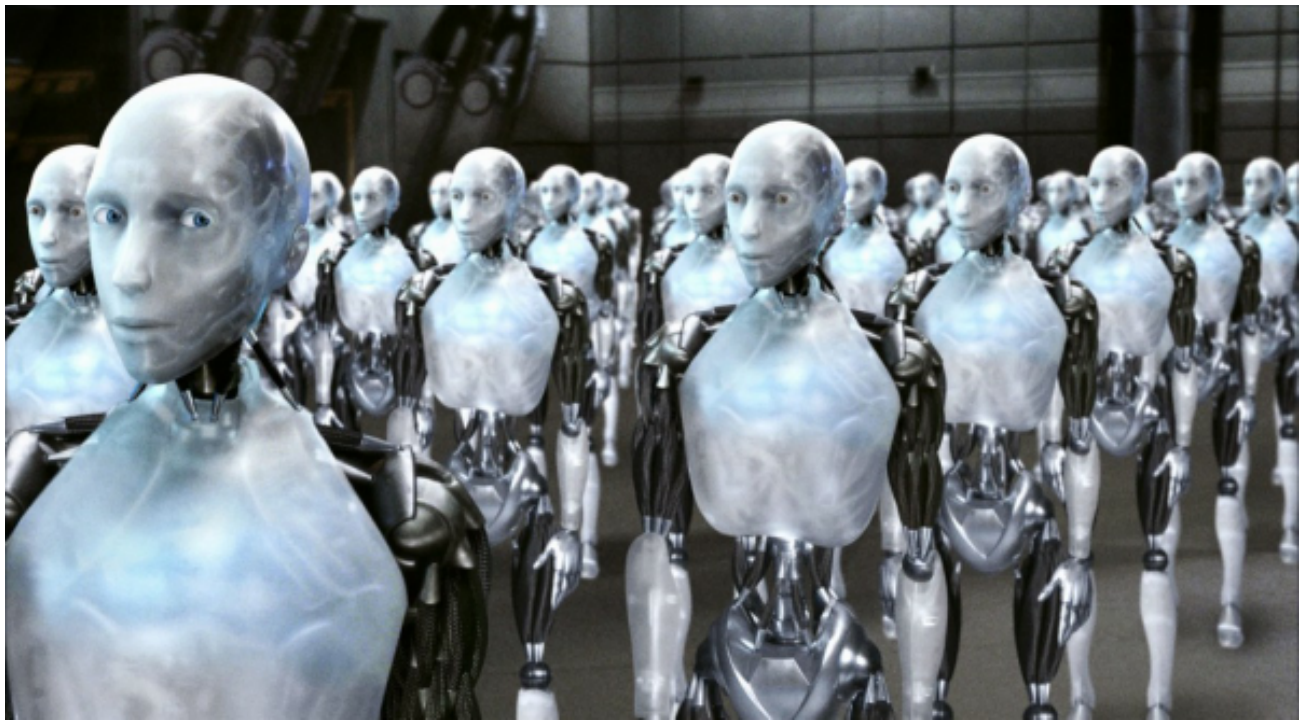
Patient treated by particular

Class Diagram



COMMON CLASS PATTERNS APPROACH

Pattern



Common Class Pattern Approach

- This approach is based on the knowledge – base of the common classes that have been proposed by the various researchers.

Patterns for finding the candidate class and objects :-

- Concept Class [Idea or Understanding]
- Event Class [points in time to be recorded]
- Organization Class [collection of people, resources, groups]
- People Class [roles user play to interact with the system]
- Places Class [physical locations the system has info about]
- Tangible things and devices [physical objects]

Candidate Classes-Events

- There are points in time that must be recorded and remembered.
- Things happen, usually to do something else, at a given date and time, or as step in an ordered in an sequence.
- For example
  - Order placed time which is an event must be remembered

In ATM SYSTEM

Events?

List of event in ATM SYSTEM

- Account class: is formal class it defines common behavior inherited

  -**checkingAccount** class- model checking clients accounts

  -**savingAccount** class-models for clients savings accounts

  -**Transaction** Class-tracking transaction,date,time

Candidate Classes- Organization

- The organizational units that peoples belongs to,
- For example accounting department might be considered as a potential class.

Organization in ATM?

Organization

- Bank class: Bank clients belong to Bank class
- Its repository of **accounts** and processes the accounts transaction

Candidate Classes-people and person(Roles and roles played)

- **The different roles users play in interacting with the application.**

Candidate Classes-People (Can't)

- It can be divided into two types (Coad & Yourdon):
- Those representing users of the system, such as an operator, or a clerk;

Candidate Classes-People (Can't)

- **Those people who do not use the system but about whom information is kept.**
  - **Some examples are Client, Employee, Teacher, Manager.**

People

- Bank clients
- ATM OPERATORS

For ATM system

Event
- Account

Organization
- Bank

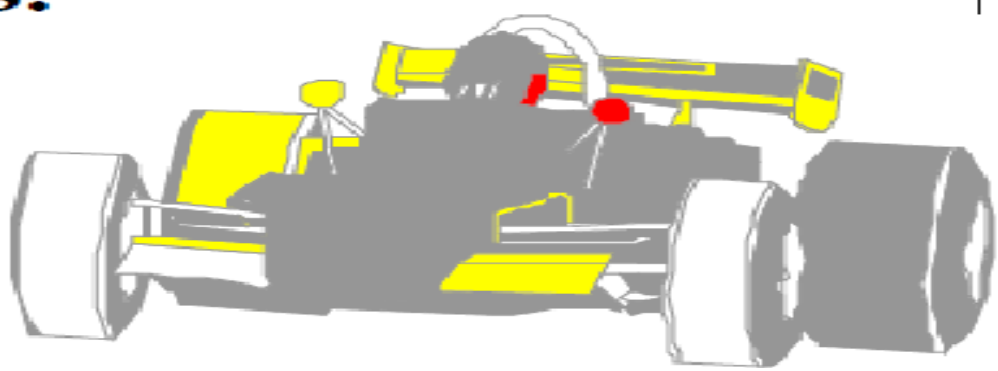
People
- Bank clients
- ATM OPERATORS

Candidate Classes-Places

- These are physical locations, such as buildings, stores, sites or offices that the system must keep information about.

Candidate Classes-Tangible Things and Devices

- **Physical objects, or group of objects, that are tangible, and devices with which the application interacts.**
- **For example, cars, pressure sensors.**



Candidate Classes-Concepts

- Concepts are principles or ideas not tangible but used to organize or keep track of business activities and/or communications.

Work out Example

- Apply common class pattern approach for Elevator system

Identifying Object relationships

Goals

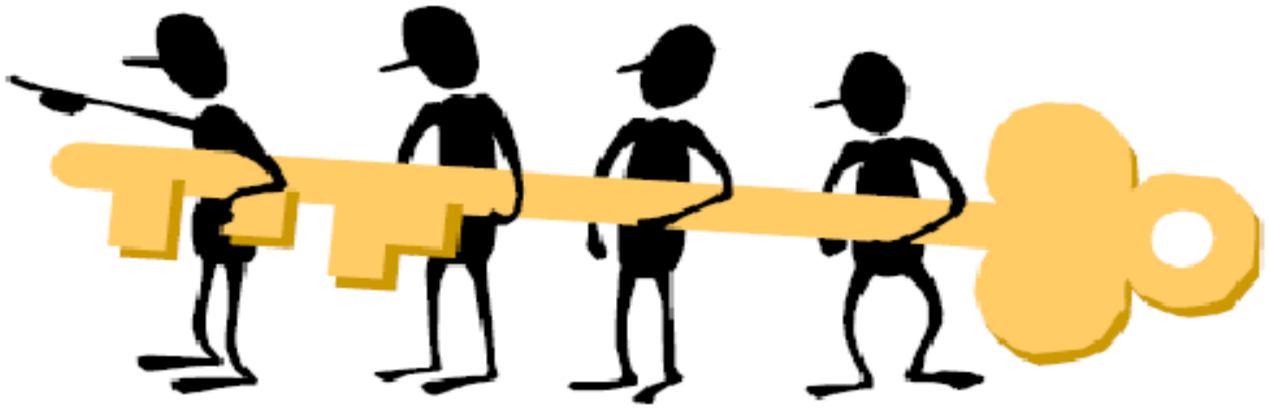
- Analyzing relationships among classes.
- Identifying association.
- Association patterns.
- Identifying super- and subclass hierarchies.

Introduction

- Identifying aggregation or a-part-of compositions.
- Class responsibilities.
- Identifying attributes and methods by analyzing use cases and other UML diagrams.

Objects contribute to the behavior of the system by collaborating with one another.

—Grady Booch



In OO environment, an application is the interactions and relationships among its domain objects.

All objects stand in relationship to others, on whom they rely for services and controls.

Objects Relationships

- **Three types of relationships among objects are:**
  - *Association.*
  - *Super-sub structure* (also known as *generalization hierarchy*).
  - Aggregation and a-part-of structure.

Associations

- A reference from one class to another is an association.
- Basically a dependency between two or more classes is an association.
- For example, Jackie *works for* John.

Associations (Con't)

- **Some associations are implicit or taken from general knowledge.**

Guidelines For Identifying Associations

- **Association often appears as a verb in a problem statement and represents relationships between classes.**
- **For example a pilot *can fly* planes.**

Guidelines For Identifying Associations (Con't)

- **Association often corresponds to verb or prepositional phrases such as *part of*, *next to*, *works for*, *contained in*, etc.**

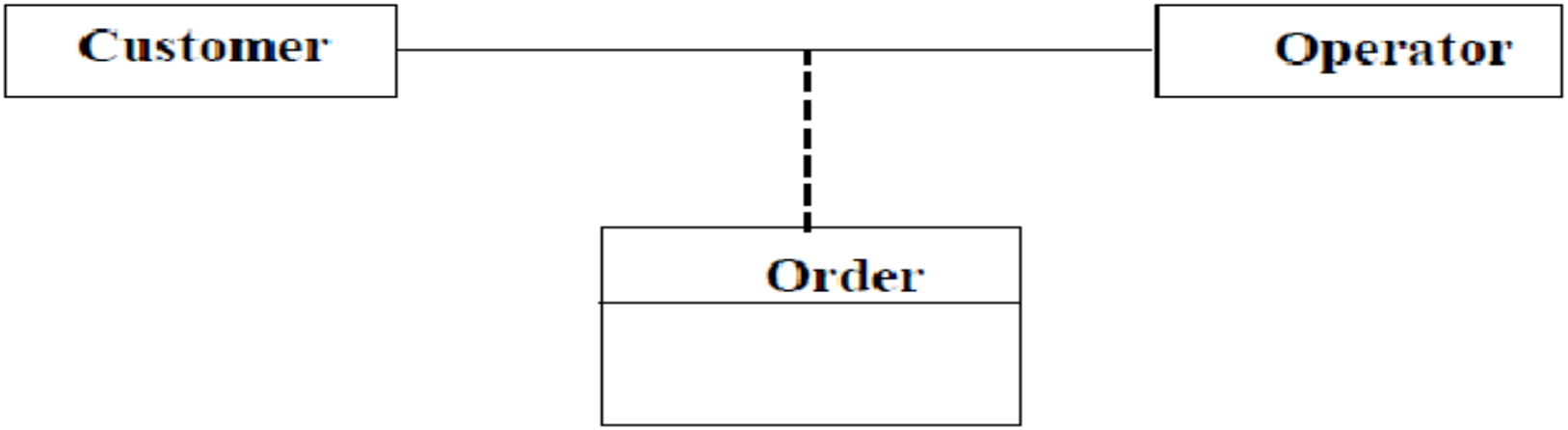
Common Association Patterns

- **Common association patterns include:**
- **Location Association:** ***next To, part of, contained in, ingredient of etc. :***
- **For example cheddar cheese is an *ingredient of* the French soup.**

Common Association Patterns (Con't)

- Communication association — *talk to*, *order to*.

- For example, a customer places an order with an operator person.



Eliminate Unnecessary Associations

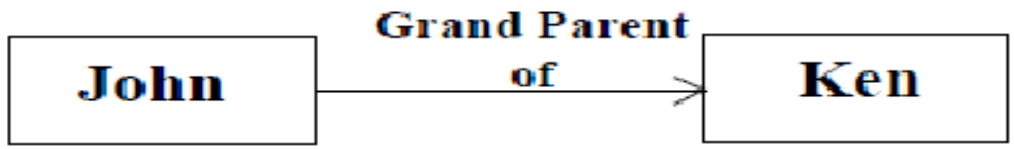
- ***Implementation association***. **Defer implementation-specific associations to the design phase.**
- ***Ternary associations***. ***Ternary* or n-ary *association* is an association among more than two classes**

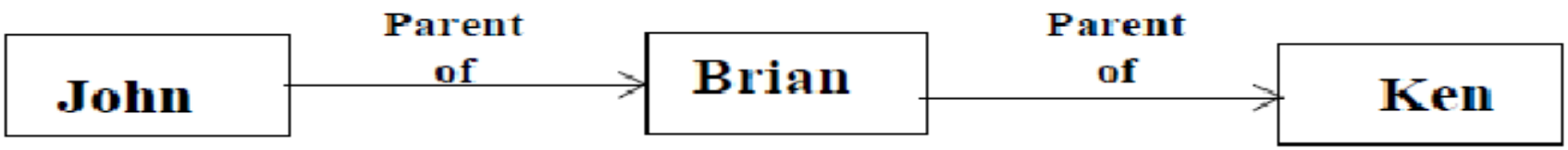
Eliminate Unnecessary Associations (Con't)

- ***Directed actions*** **(derived)** ***associations*** **can be defined in terms of other associations.**
- **Since they are redundant you should avoid these types of association.**

Eliminate Unnecessary Associations (Con't)

- **Grandparent of Ken can be defined in terms of the parent association.**





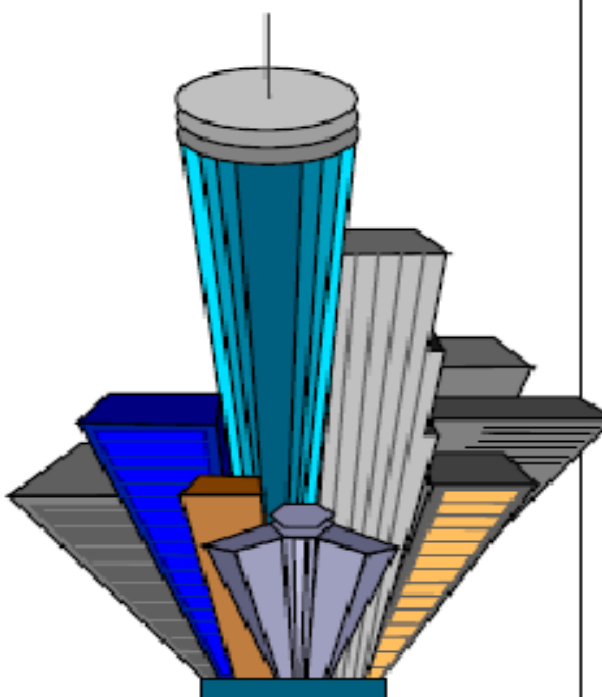
Super-Class and Sub-Class

Superclass-Subclass Relationships

- **Recall that at the top of the class hierarchy is the most general class, and from it descend all other, more specialized classes.**
- **Sub-classes are more specialized versions of their super-classes.**

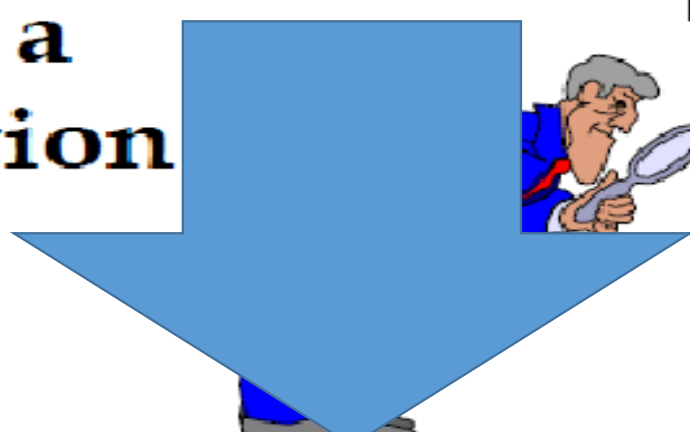
Guidelines For Identifying Super-sub Relationships: Top-down

- Look for noun phrases composed of various adjectives on class name.
- Example, Military Aircraft and Civilian Aircraft.
- Only specialize when the sub classes have significant behavior.



Guidelines For Identifying Super-sub Relationships: Bottom-up

- Look for classes with similar attributes or methods.
- Group them by moving the common attributes and methods to super class.
- Do not force classes to fit a preconceived generalization structure.

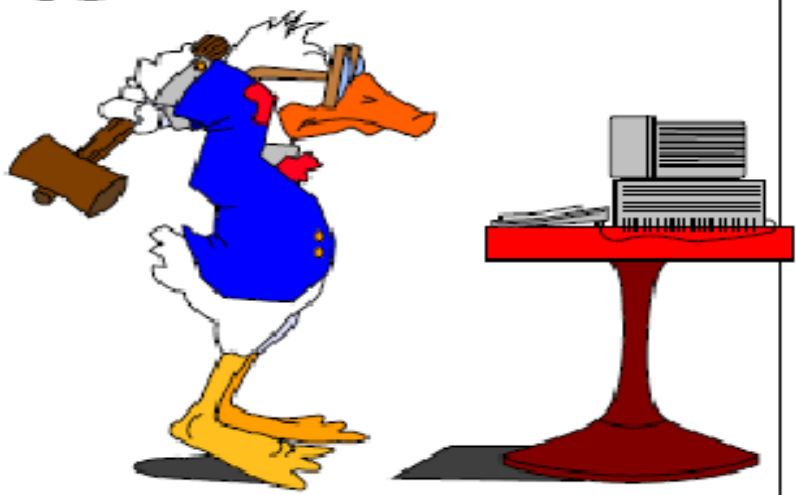


Guidelines For Identifying Super-sub Relationships: Reusability

- Move attributes and methods as high as possible in the hierarchy.
- At the same time do not create very specialized classes at the top of hierarchy.
- This balancing act can be achieved through several iterations.

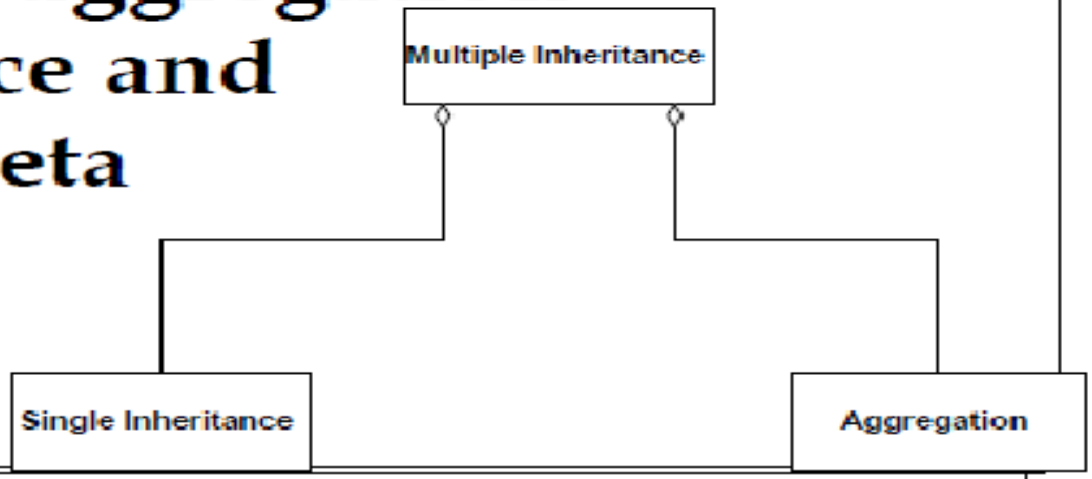
Guidelines For Identifying Super-sub Relationships: Multiple inheritance

- **Avoid excessive use of multiple inheritance.**
- **It is also more difficult to understand programs written in multiple inheritance system.**



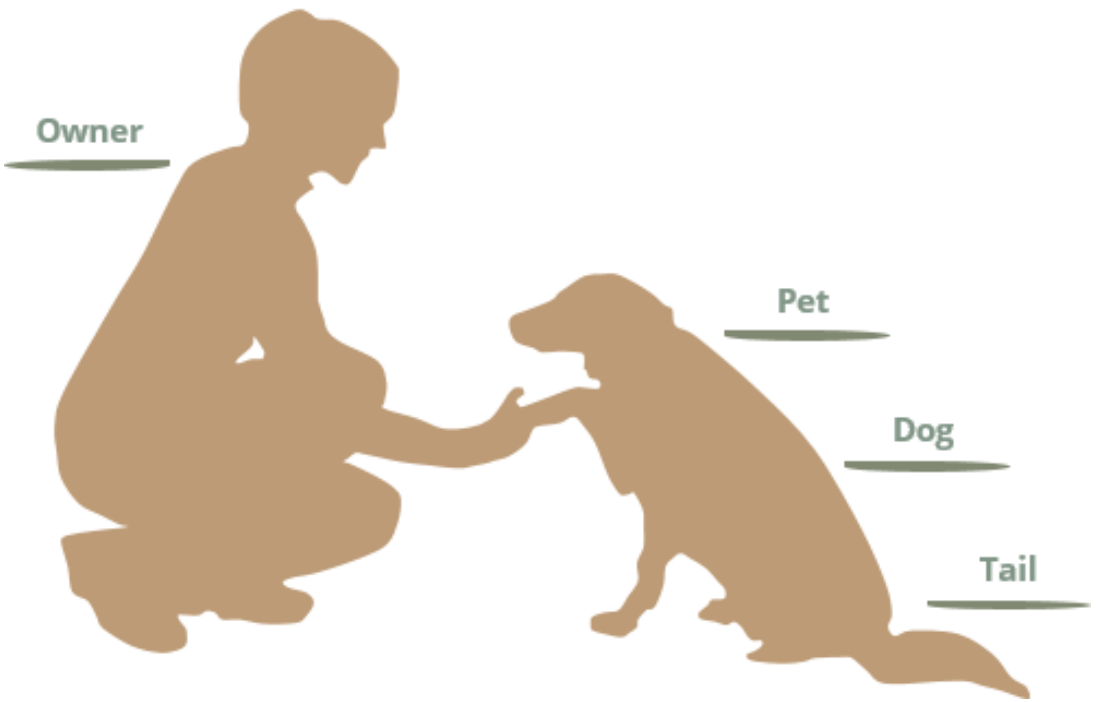
Multiple inheritance (Con't)

- One way to achieve the benefits of multiple inheritance is to inherit from the most appropriate class and add an object of other class as an attribute.
- In essence, a multiple inheritance can be represented as an aggregation of a single inheritance and aggregation. This meta model reflects this situation.



a part of relationship

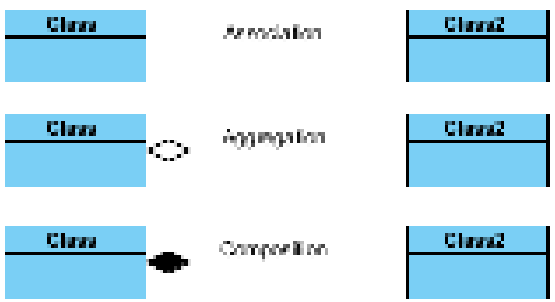
Association • Aggregation • Composition



We see the following relationships:

- owners feed pets, pets please owners (**association**)
- a tail is a part of both dogs and cats (**aggregation** / **composition**)
- a cat is a kind of pet (**inheritance** / **generalization**)

The figure below shows the three types of association connectors: association, aggregation and composition.



The figure below shows a generalization.

A-Part-of Relationship - Aggregation

- ***A-part-of relationship*, also called *aggregation*, represents the situation where a class consists of several component classes.**

A-Part-of Relationship - Aggregation (Con't)

- This does not mean that the class behaves like its parts.
- For example, a car consists of many other classes, one of them is a radio, but a car does not behave like a radio.

A-Part-of Relationship - Aggregation (Con't)

- Two major properties of a-part-of relationship are:
  - transitivity
  - antisymmetry

Transitivity

- If *A* is part of *B* and B is part of *C*, then *A* is part of *C*.
- For example, a carburetor is part of an engine and an engine is part of a car; therefore, a carburetor is part of a car.

Antisymmetry

- If A is part of B, then B is not part of A.
- For example, an engine is part of a car, but a car is not part of an engine.

A-Part-of Relationship Patterns Assembly

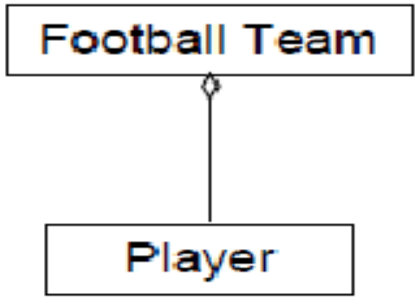
- An assembly-Part situation physically exists.
- For example, a French soup consists of onion, butter, flour, wine, French bread, cheddar cheese, etc.

A-Part-of Relationship Patterns Container

- A case such as course-teacher situation, where a course is considered as a container. Teachers are assigned to specific courses.

A-Part-of Relationship Patterns
Collection-Member

- **A soccer team is a collection of players.**



Class responsibility

Class Responsibility: Identifying Attributes and Methods

- Identifying attributes and methods, like finding classes, is a difficult activity.
- The use cases and other UML diagrams will be our guide for identifying attributes, methods, and relationships among classes.

Identifying Class Responsibility by Analyzing Use Cases and Other UML Diagrams

- **Attributes can be identified by analyzing the use cases, sequence/collaboration, activity, and state diagrams.**

Responsibility

- How am I going to be used?
- How am I going to collaborate with other classes?
- How am I described in the context of this system's responsibility?
- What do I need to know?
- What state information do I need to remember over time?
- What states can I be in?

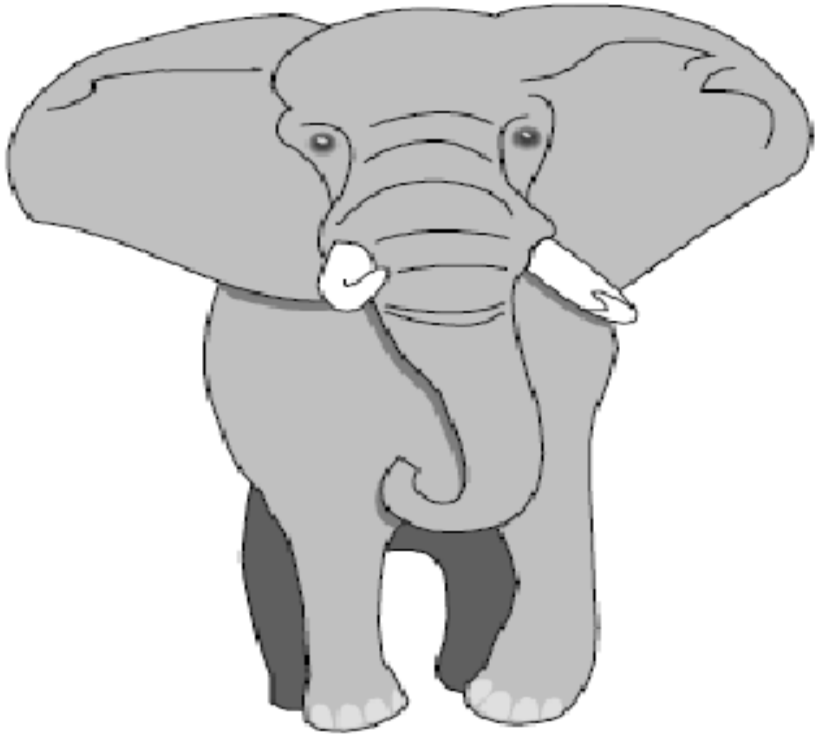
Assign Each Responsibility To A Class

- Assign each responsibility to the class that it logically belongs to.
- This also aids us in determining the purpose and the role that each class plays in the application.

Identifying attributes and methods by analyzing use cases and other UML diagrams

Object Responsibility: Attributes

- **Information that the system needs to remember.**

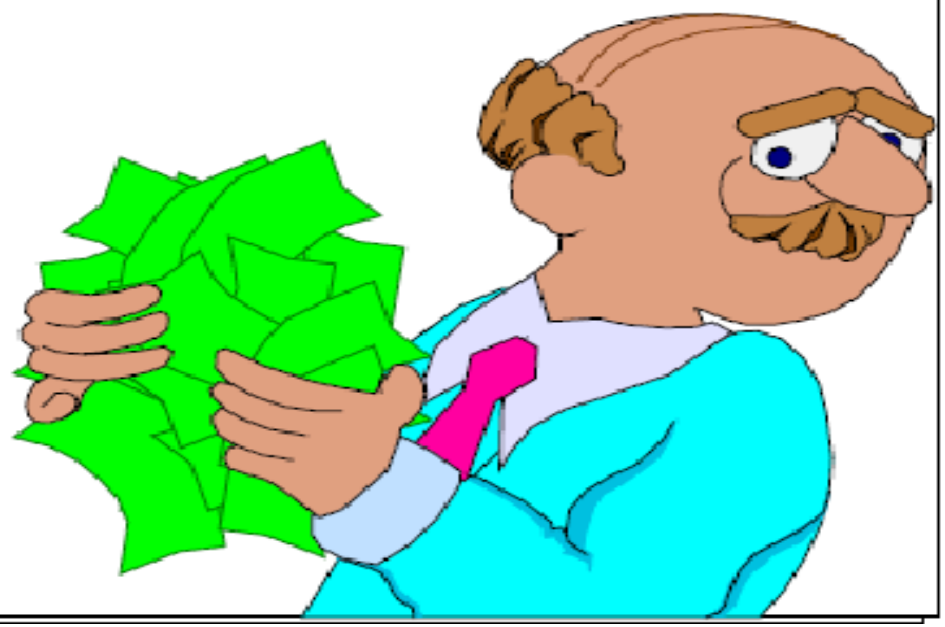


Guidelines For Identifying Attributes Of Classes

- **Attributes usually correspond to nouns followed by possessive phrases such as *cost of* the soup.**

Guidelines For Identifying Attributes Of Classes (Con't)

- **Keep the class simple; only state enough attributes to define the object state.**



Guidelines For Identifying Attributes Of Classes (Con't)

- Attributes are less likely to be fully described in the problem statement.
- You must draw on your knowledge of the application domain and the real world to find them.

Guidelines For Identifying Attributes Of Classes (Con't)

- Omit derived attributes.
- For example, don't use age as an attribute since it can be derived from date of birth.
- Drive attributes should be expressed as a method.

Guidelines For Identifying Attributes Of Classes (Con't)

- Do not carry discovery of attributes to excess.
- You can always add more attributes in the subsequent iterations.

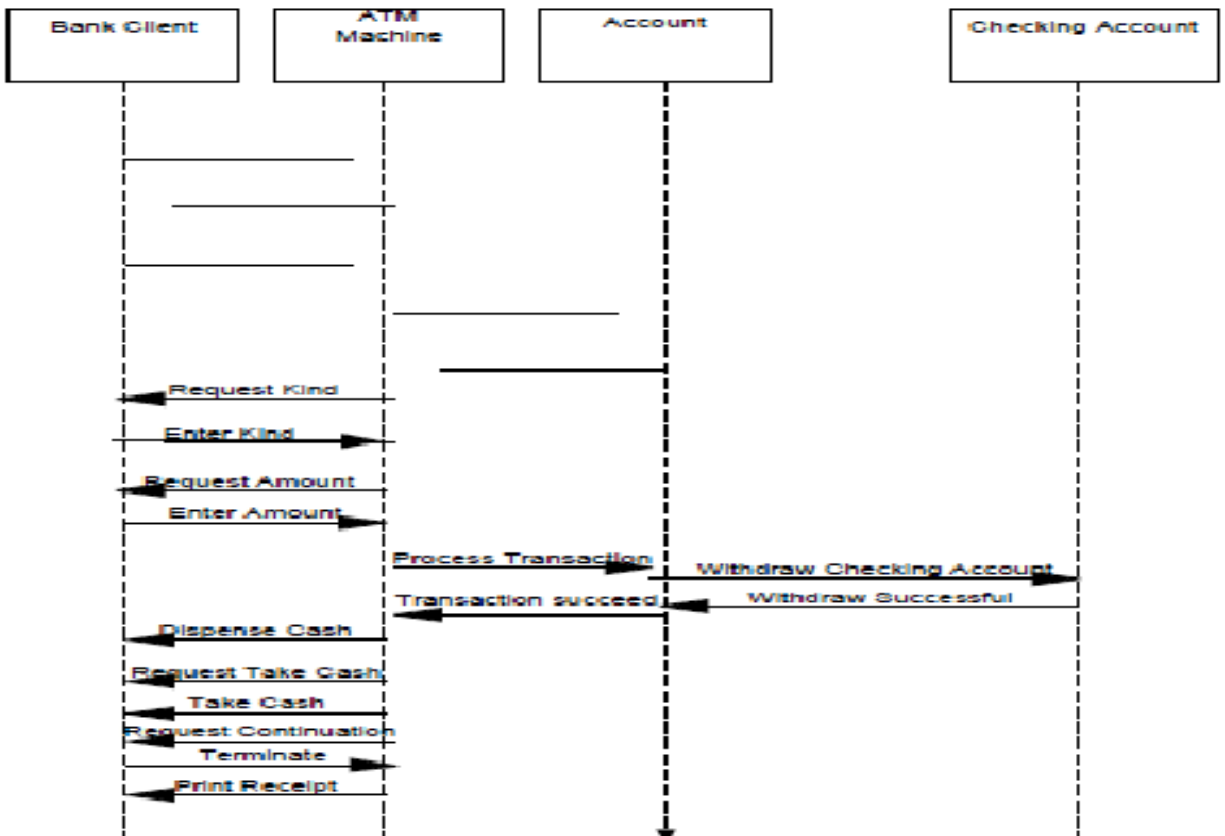
Object Responsibility: Methods & Messages

- Methods and messages are the work horses of object-oriented systems.
- In O-O environment, every piece of data, or object, is surrounded by a rich set of routines called methods.

Identifying Methods by Analyzing UML Diagrams and Use Cases

- **Sequence diagrams can assist us in defining the services the objects must provide.**

Identifying Methods (Con't)



Identifying Methods (Con't)

- Methods usually correspond to queries about attributes (and sometimes association) of the objects.
- Methods are responsible for managing the value of attributes such as query, updating, reading and writing.

Identifying Methods (Con't)

- For example, we need to ask the following questions about soup class:
- What services must a soup class provide? And
- What information (from domain knowledge) is soup class responsible for storing?

Identifying Methods (Con't)

- Let's first take a look at its attributes which are:
- name
- preparation,
- price,
- preparation time and
- oven temperature.

Identifying Methods (Con't)

- Now we need to add methods that can maintain these attributes.
- For example, we need a method to change a price of a soup and another operation to query about the price.

Identifying Methods (Con't)

- setName
- getName
- setPreparation
- get Preparation
- setCost
- getCost
- setOvenTemperature
- getOvenTemperature
- setPreparationTime
- getPreparationTime

Summary

Summary

- We learned how to identify three types of object relationships:
- Association
- Super-sub Structure (Generalization Hierarchy)
- A-part-of Structure

Summary (Con't)

- The hierarchical relation allows the sharing of properties or inheritance.
- A reference from one class to another is an association.
- The A-Part-of Structure is a special form of association.

Summary (Con't)

- Every class is responsible for storing certain information from domain knowledge .
- Every class is responsible for performing operations necessary upon that information.

- MODULE-3

• OBJECT ORIENTED DESIGN

- MODULE-3

- COURSE OUTCOMES:

Identify suitable object oriented design methodologies

OBJCT ORIENTED DESIGN

Object Oriented Design Axioms-Designing Classes -Class visibility -Redefining attributes -Designing methods and protocols -Packages and managing classes -Access Layer- Object Storage Persistence - Object oriented Database System-Designing view layer classes -Macro level process -Micro level process.

OO Design Process and Design Axioms

Goals

- The object-oriented design process.
- Object-oriented design axioms and corollaries.
- Design patterns.

Object-Oriented Design Process in the Unified Approach

- **Objects discovered during the analysis can serve as the framework for design.**
- **During the design phase the classes identified in object-oriented analysis must be revisited with a shift in focus to their implementation.**
- **New classes or attributes and methods must be added for implementation purposes and user interfaces.**
- The object-oriented design process consists of the following activities

Object-Oriented Design Process in the Unified Approach

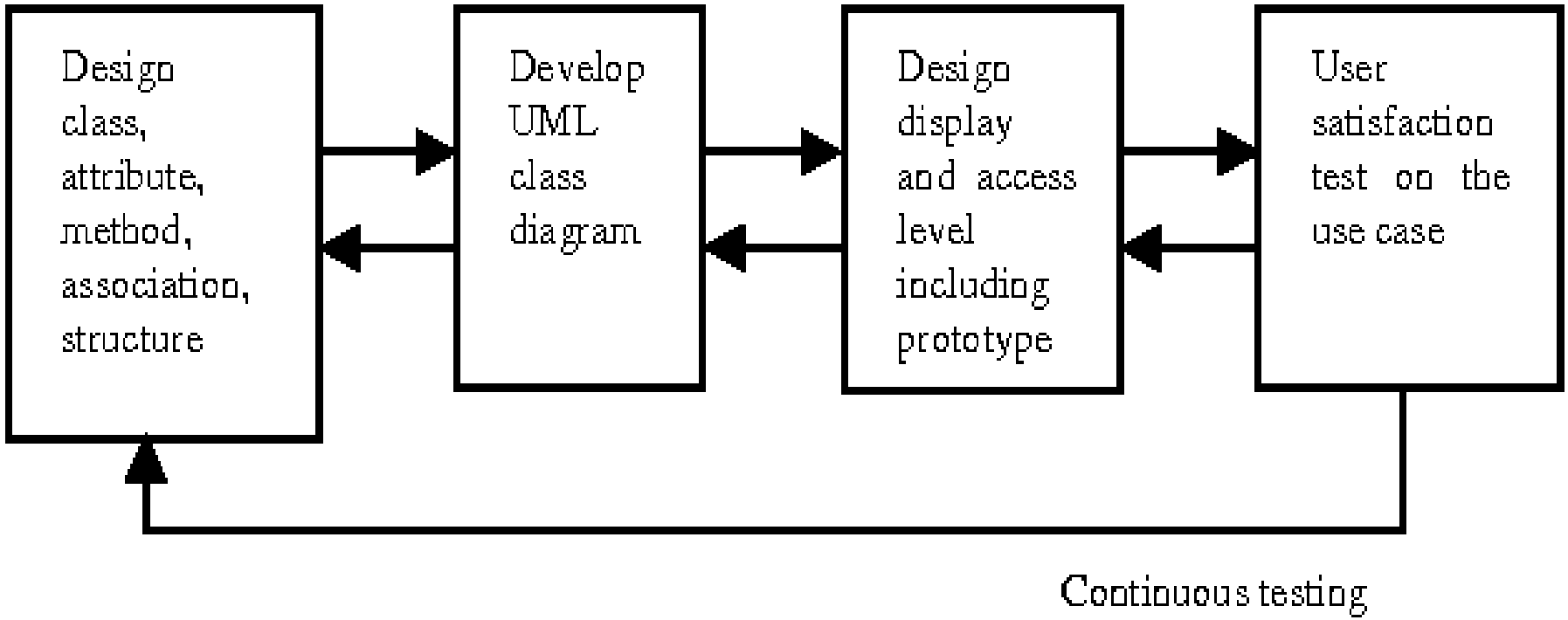


Fig. 2. Object-oriented Design (OOD) Process

OO Design Process

1. **Apply design axioms**
2. **Design the access layer**
3. **Design the view layer classes.**
4. **Iterate and refine the preceding steps.**

OO Design Process

- 1. **Apply design axioms to design classes, their attributes, methods, associations, structures, and protocols.**

OO Design Process

1.Apply Design Axioms

1.1. Refine and complete the static UML class diagram (object model) by adding details to the UML class diagram.

OO Design Process

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.**

OO Design Process (Con't)

- **2. Design the access layer**
- **2.1.** Create mirror classes. For every business class identified and created, create one access class.
- For example if there are 3 business class(class1, class2, class3) create 3 access class layers.(class 1DB,class2DB and class3DB)
  - **2.2.** define relationships among access layer classes.

OO Design Process (Con't)

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.

OO Design Process (Con't)

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.

OO Design Process (Con’t)

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

Object-Oriented Design

Axioms, Theorems and Corollaries

- **Axiom:**
  An axiom is a fundamental truth that always is observed to be valid and for which there is no counterexample or exception.
- The axioms cannot be proven or derived but they cannot be invalidated by counterexamples or exceptions.

Object-Oriented Design

Axioms, Theorems and Corollaries

- **Axiom:**
  There are two design axioms applied to object-oriented design.
- Axiom 1 deals with relationships between system components and
- Axiom 2 deals with the complexity of design.

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.**

Axioms, Theorems and *Corollaries*

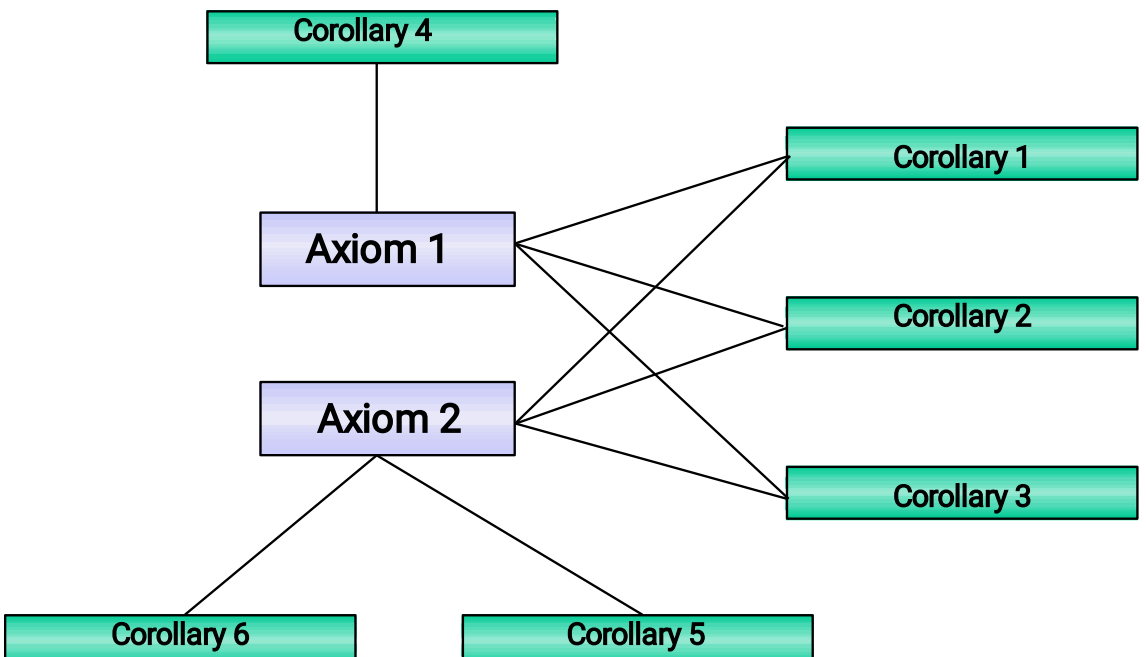
- A *Corollary* is a proposition that follows from an axiom or another proposition that has been proven.

(Cont'd)

Occam's Razor

- **The best theory explains the known facts with a minimum amount of complexity and maximum simplicity and straight forwardness.**

The Origin of Corollaries



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.

Corollaries (Con't)

- 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.

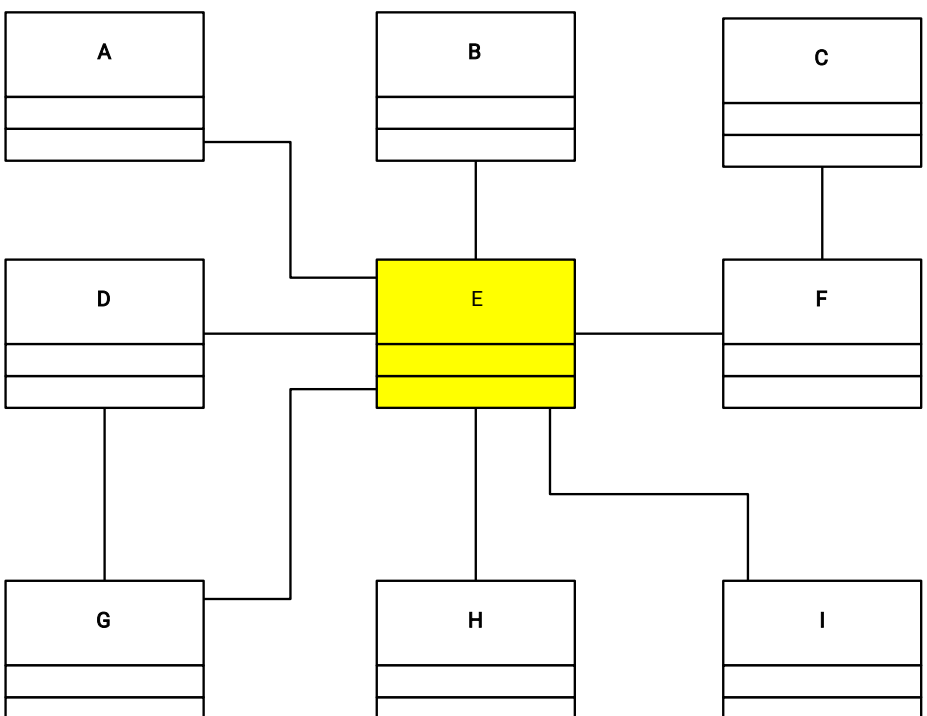
Corollaries (Con't)

- **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 in a previous topic.**
- **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.**

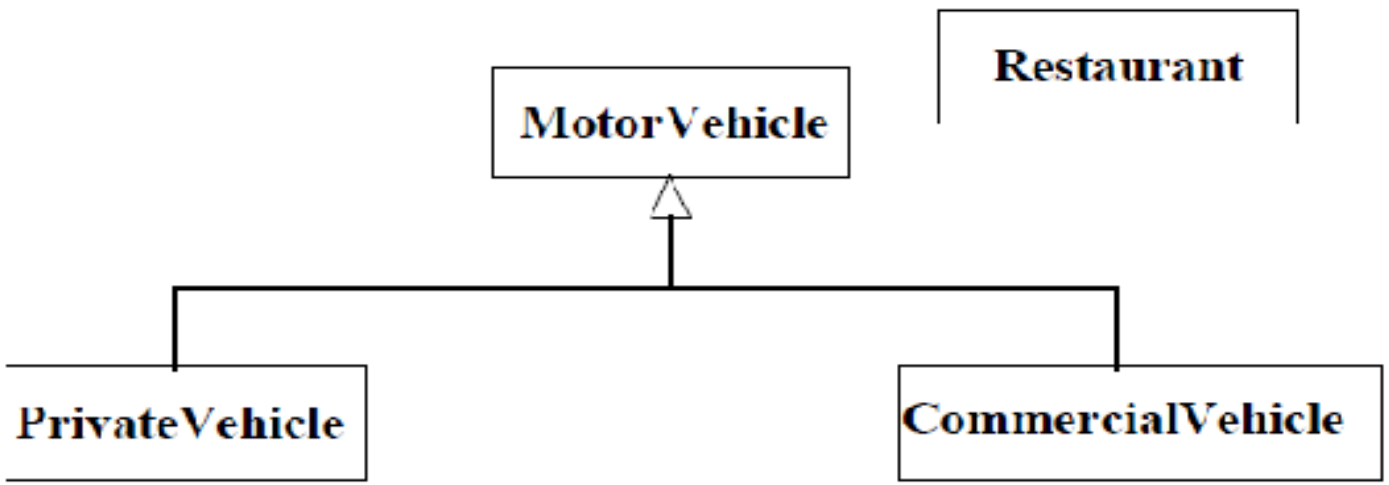
Corollary 5. Standardization

- 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.

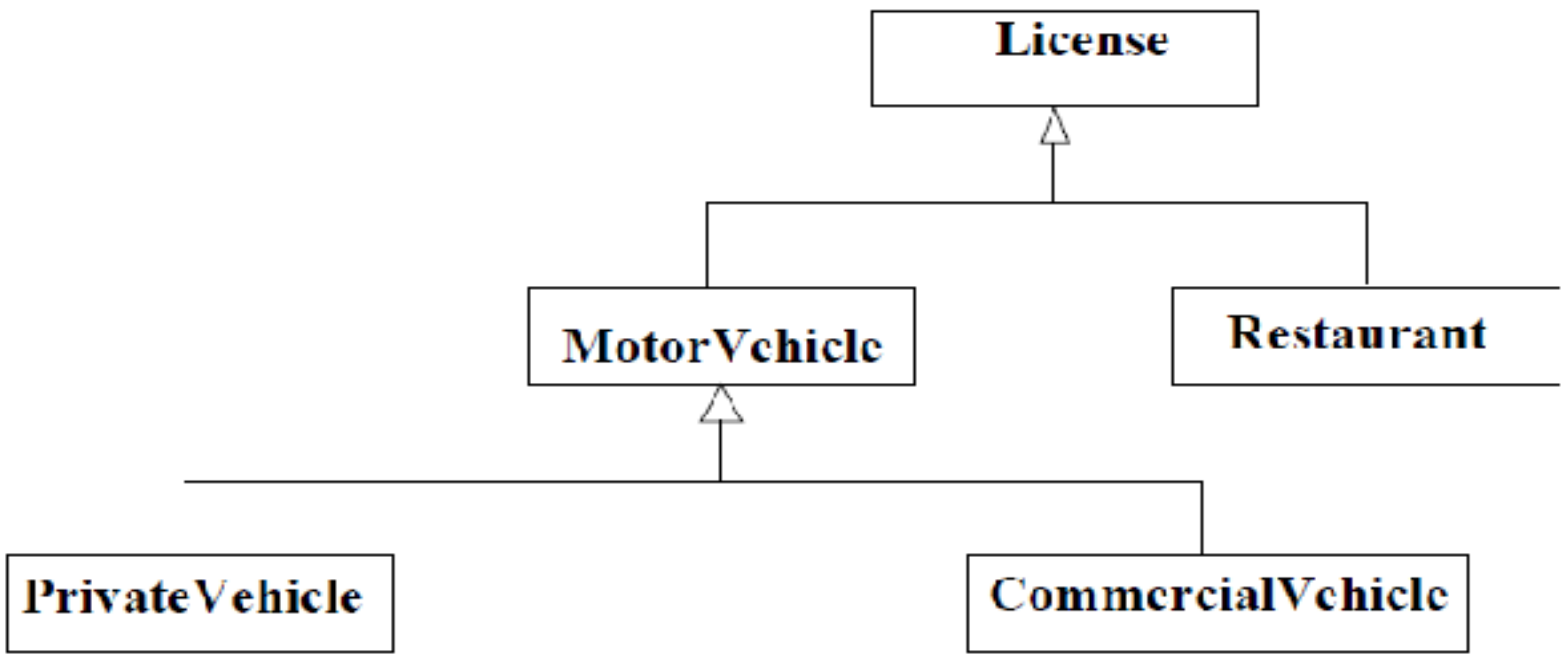
Corollary 6. Designing with *Inheritance*

- **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 (Con't)



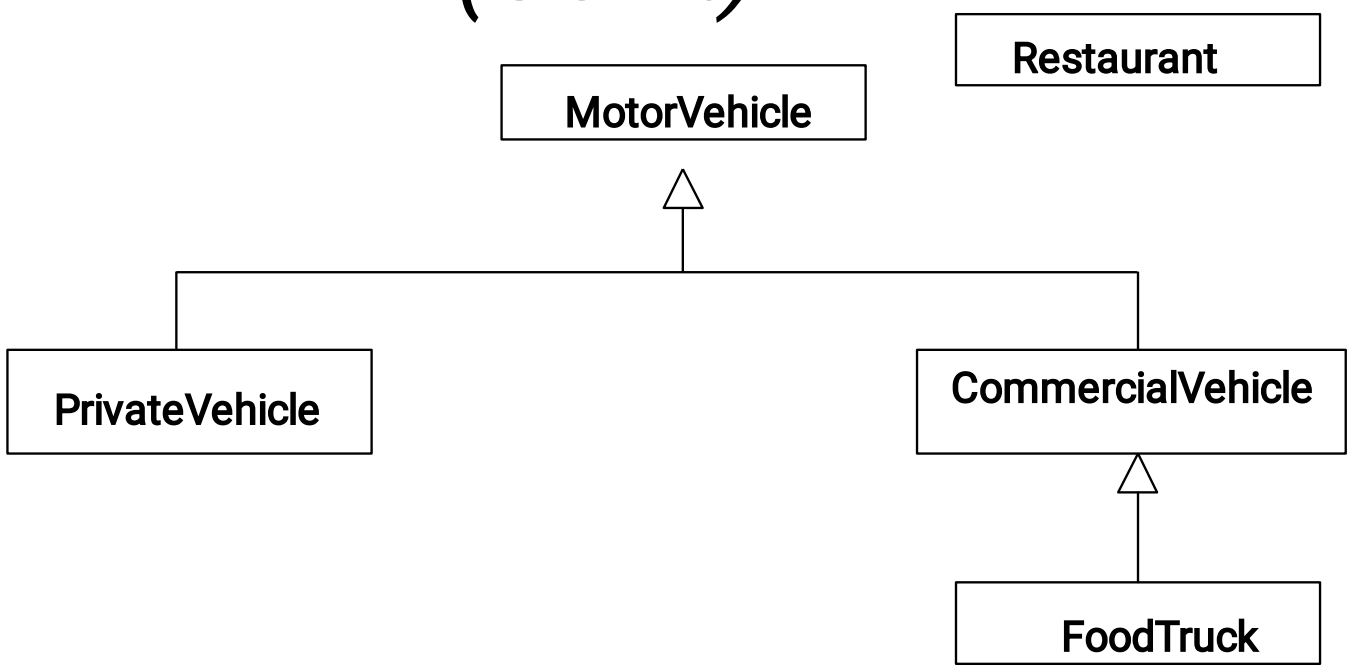
Designing With Inheritance (Con't)



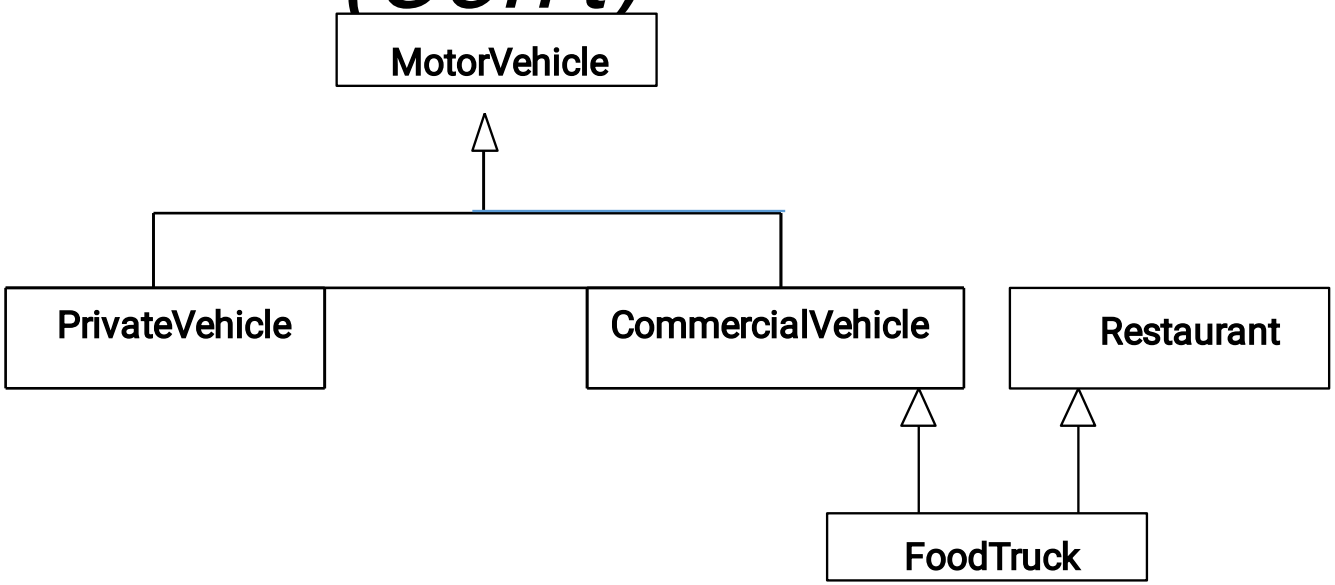
Designing With Inheritance *Weak Formal Class*

- **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?**

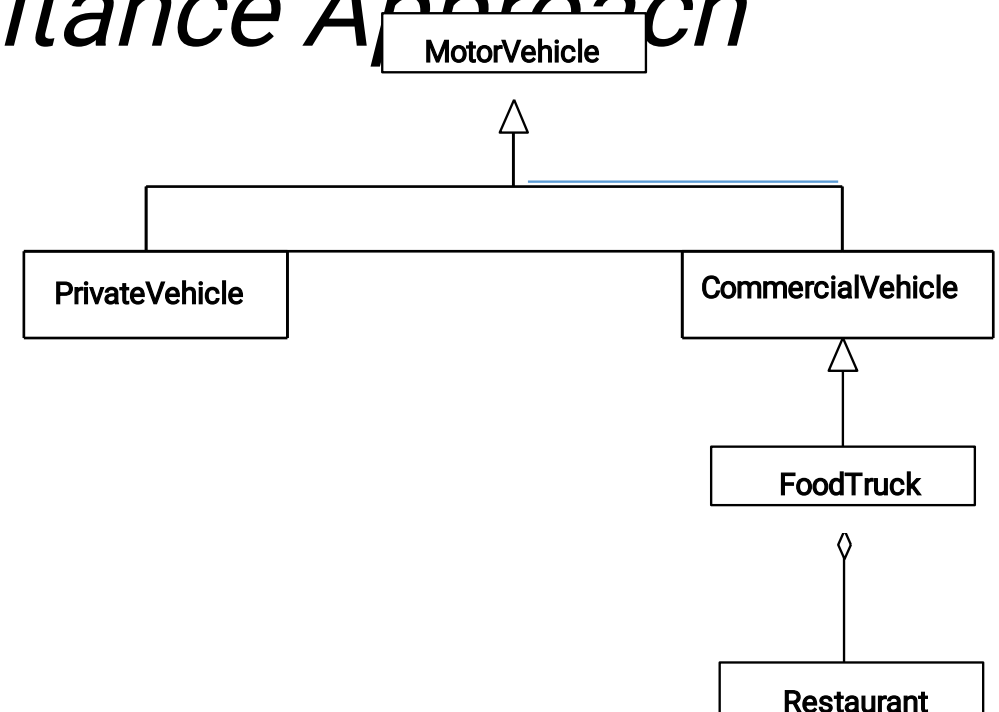
Designing With Inheritance *(Con't)*



Designing With Inheritance *(Con't)*

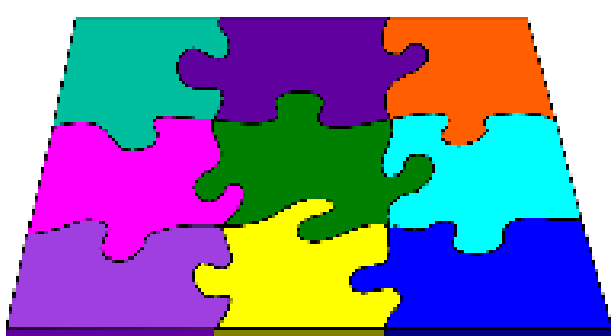


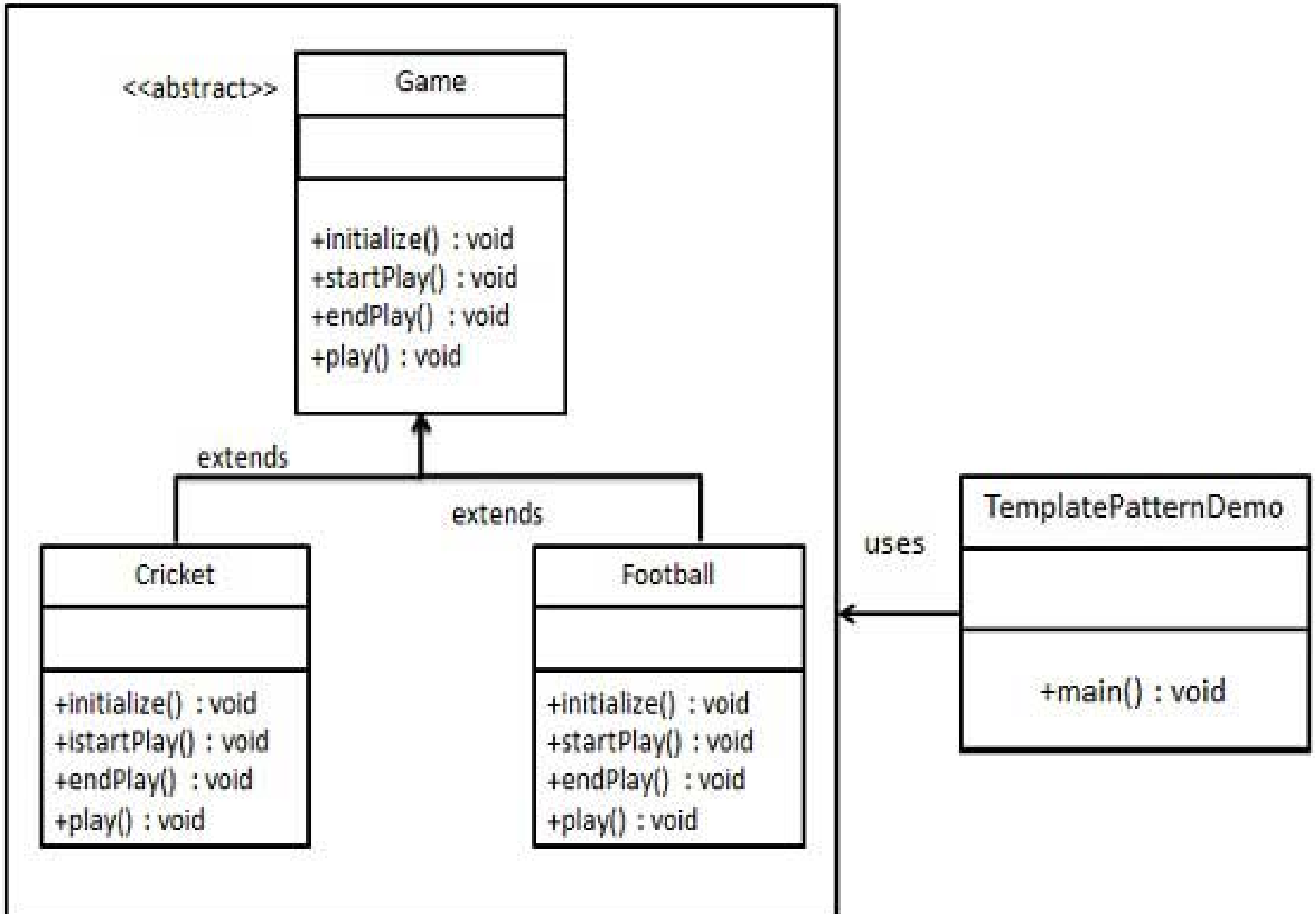
Achieving Multiple Inheritance*using Single Inheritance Approach*

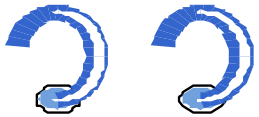


Design Patterns

- **Patterns provide a mechanism for capturing and describing commonly recurring design ideas that solve a general design problem.**

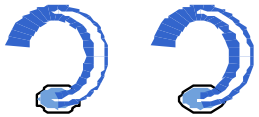



<<abstract>>
Game
+initialize() : void
+startPlay() : void
+endPlay() : void
+play() : void
extends
extends
Cricket
+initialize() : void
+istartPlay() : void
+endPlay() : void
+play() : void
Football
+initialize() : void
+startPlay() : void
+endPlay() : void
+play() : void
uses
TemplatePatternDemo
+main() : void



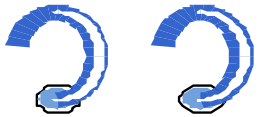
Summary

- We studied the object-oriented design process and axioms.
- The two design axioms are
  - Axiom 1. The independence axiom. Maintain the independence of components.
  - Axiom 2. *The information axiom. Minimize the information content of the design .*



Summary (Con't)

- The six design corollaries are
  - **Corollary 1. Uncoupled design with less information content.**
  - **Corollary 2. Single purpose.**
  - **Corollary 3. Large number of simple classes.**
  - **Corollary 4. Strong mapping.**
  - **Corollary 5. Standardization.**
  - **Corollary 6. Design with inheritance.**



Summary (Con’t)

- **We also studied the concept of design patterns, which allow systems to share knowledge about their design.**

Designing Classes

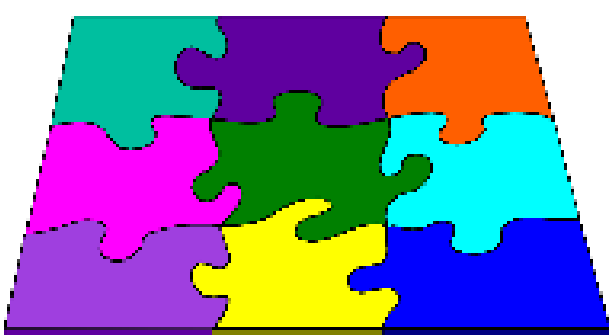
Object-Oriented Systems
Development

Goals

- Designing classes.
- Designing protocols and class visibility.
- Defining attributes.
- Designing methods.
- Designing access layer objects.

Object-Oriented 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.

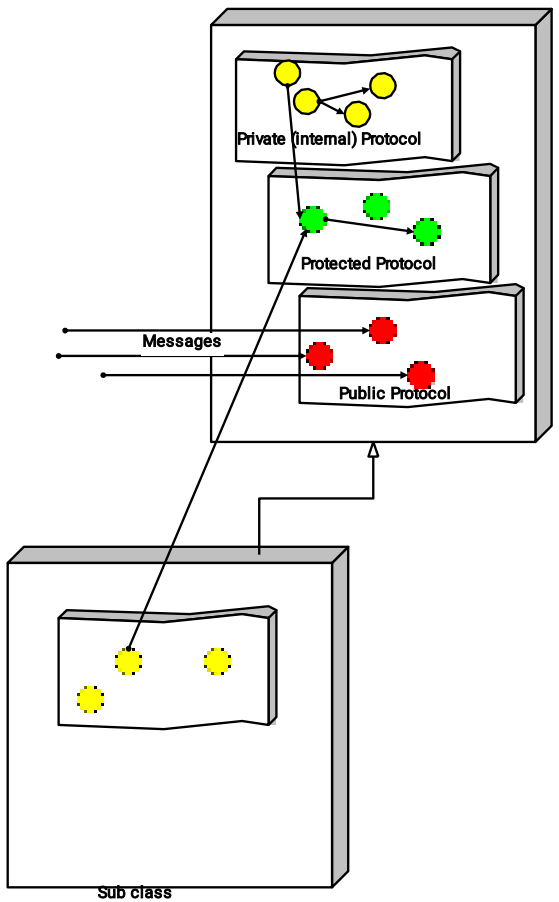


Class Visibility

- 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.

Class Visibility (Con't)

- **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

Protected Protocol (Visibility)

- 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.**

Guidelines for Designing *Protocols (Con't)*

- **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?**

Questions (Con't)

- **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.**

Designing Methods and *Protocols (Con't)*

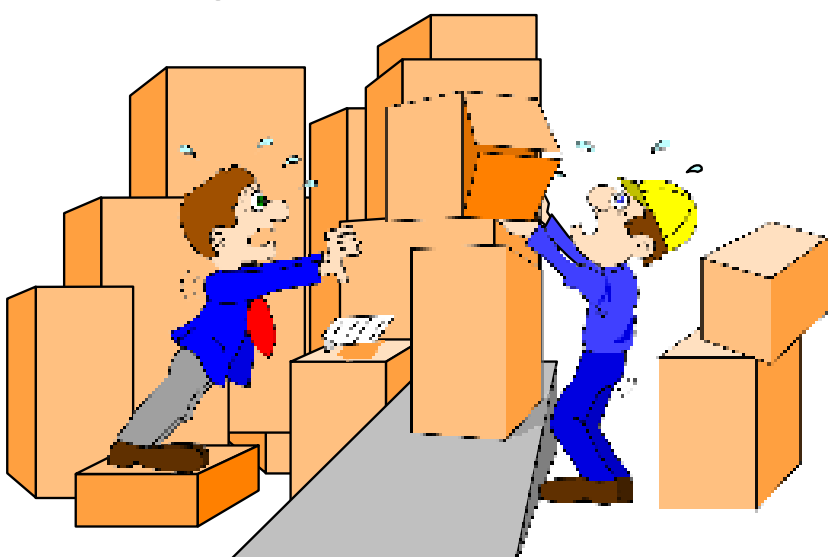
- ***Copy method*** **. The method that copies the contents of one instance to another instance.**
- ***Attribute set*** **. The method that sets the values of one or more attributes.**
- ***Attribute get*** **. The method that returns the values of one or more attributes.**

Designing Methods and *Protocols (Con't)*

- ***I/O methods*** **. The methods that provide or receive data to or from a device.**
- ***Domain specific*** **. The method specific to the application.**

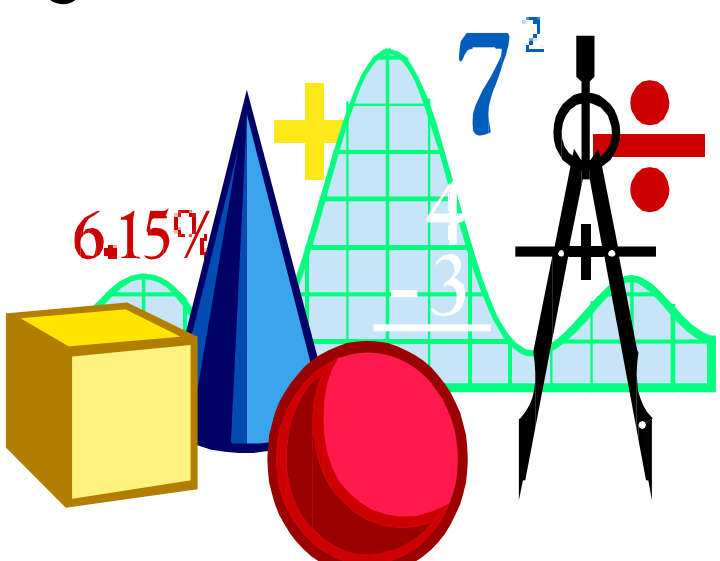
Five Rules For Identifying Bad *Design*

- **I. If it looks messythen it's probably a bad design.**



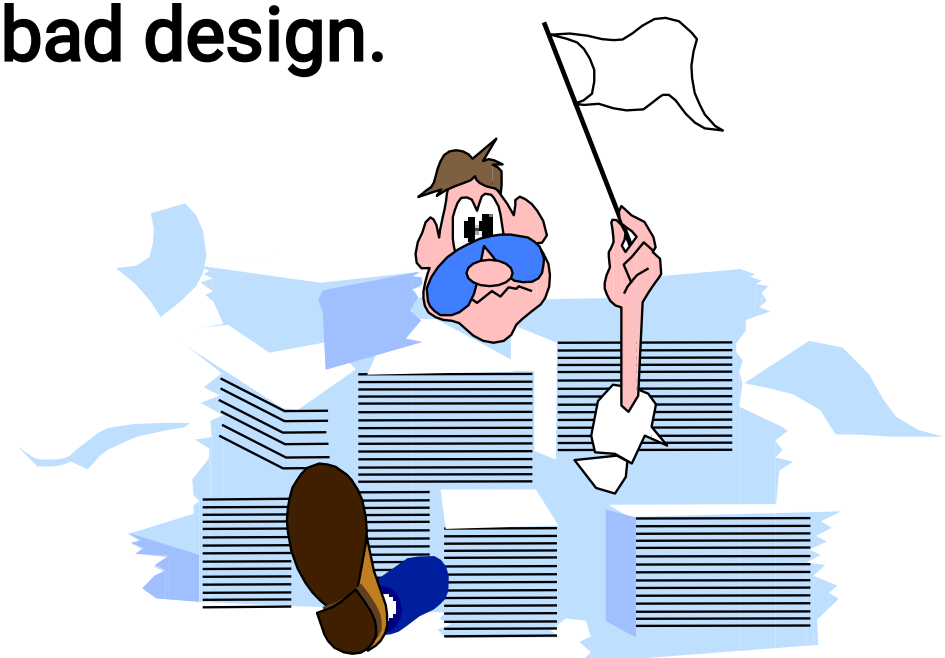
Five Rules For Identifying Bad *Design (Con't)*

- **II. If it is too complex then it's probably a bad design.**



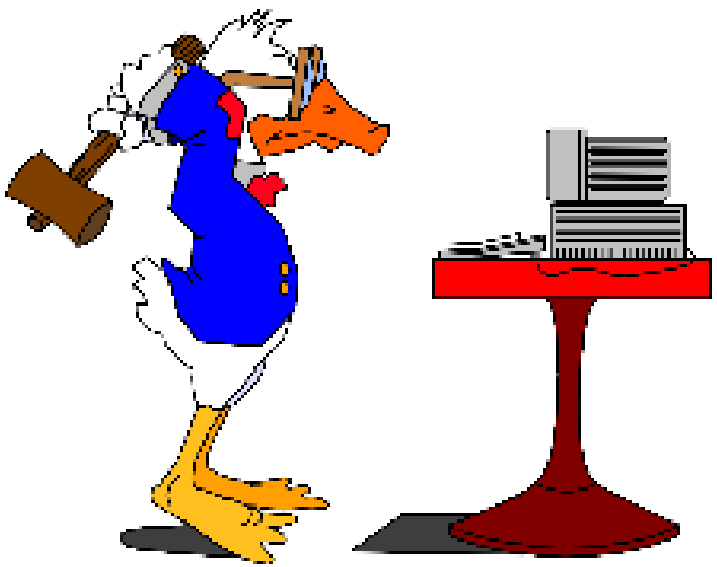
Five Rules For Identifying Bad *Design (Con't)*

- **III. If it is too big then it's probably a bad design.**



Five Rules For Identifying Bad*Design* *(Con't)*

- **IV. If people don't like it then it's probably a bad design.**



Five Rules For Identifying Bad *Design (Con't)*

- **V. 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)

Avoiding Design Pitfalls (Con't)

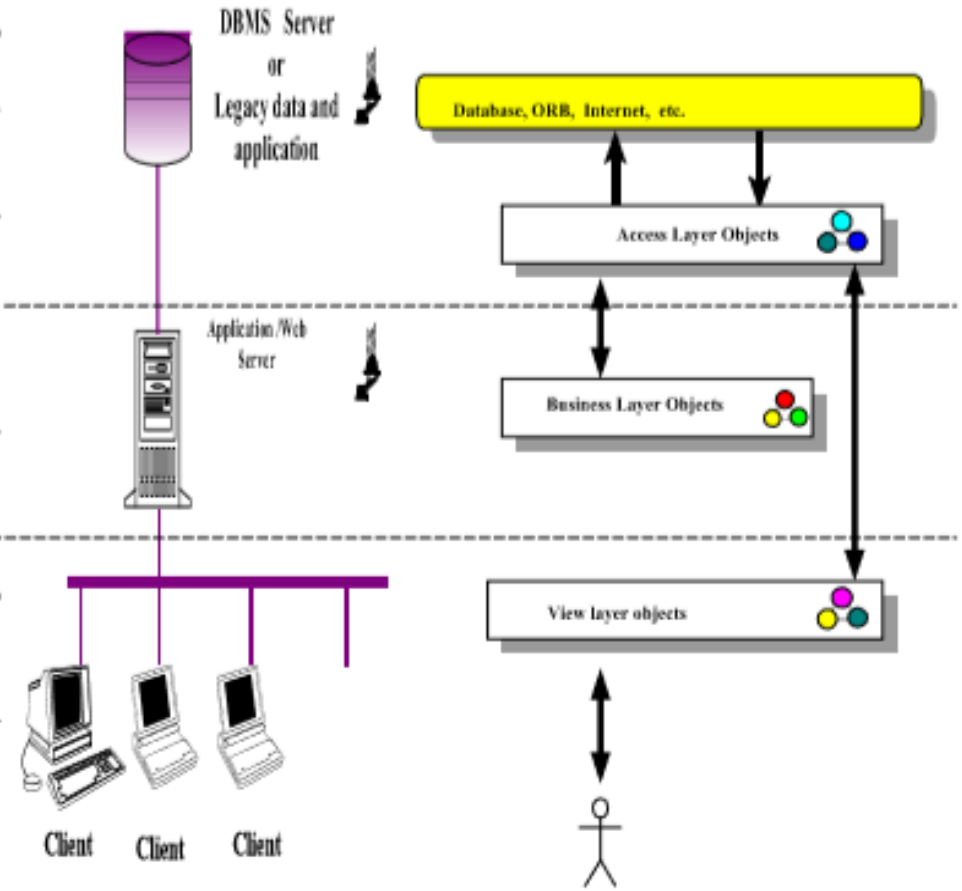
- Move some functions into new classes that the object would use.
- Apply Corollary 1 (uncoupled design with less information content).
- Break up the class into two or more classes.
- Apply Corollary 3 (large number of simple classes).

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.**

Access Layer Classes (Con't)

- 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.



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.**

Proce ss

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

1. If methods will be stored in a program then

For every business class identified, *determine if the class has persistent data.*

else

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.

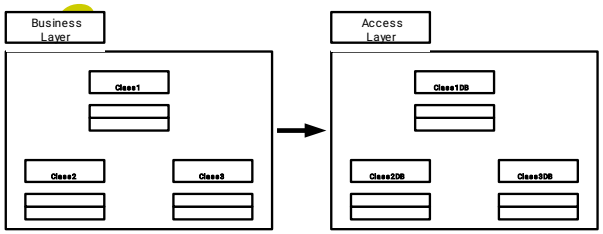
Process (Con't)

3. *Simplify classes and relationships* . The main goal here is to eliminate redundant or unnecessary classes or structures.
    1. *Redundant classes* . If you have more than one class that provides similar services, simply select one and eliminate the other(s).
    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.

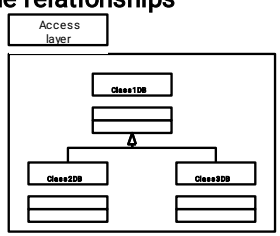
Process of Creating Access Layer

classe

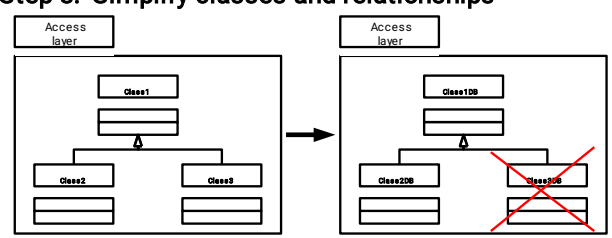
Step 1: Mirror business class package



Step 2: Define relationships

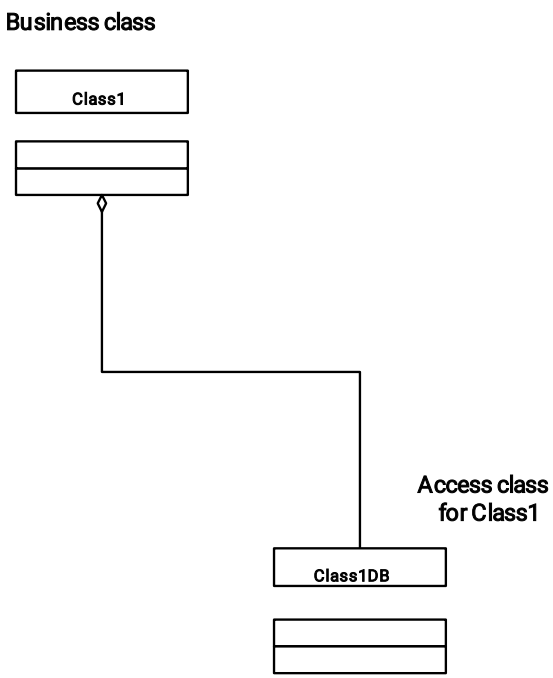


Step 3: Simplify classes and relationships



Access Layer Classes (Con't)

- **The relation between a business class and its associated access class.**



Summary

- This chapter concentrated on the first step of the object-oriented design process, which consists of applying the design axioms and corollaries to design classes, their attributes, methods, associations, structures, and protocols; then, iterating and refining.

Summary (Cont'd)

- Object-oriented design is an iterative process.
- Designing is as much about discovery as construction.
- Do not be afraid to change a class design, based on experience gained, and do not be afraid to change it a second, third, or fourth time.

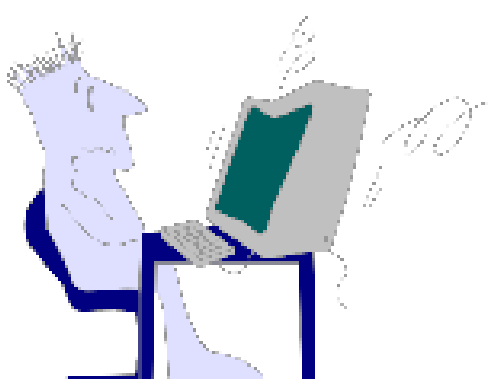
View Layer: Designing Interface Objects

Goals

- **Identifying View Classes**
- **Designing Interface Objects**
- **Guidelines to Graphical User Interface (GUI)**

. . . The design of your software's *interface, more than anything else, affects how a user interacts and therefore experiences your application.*

Tandy Trower



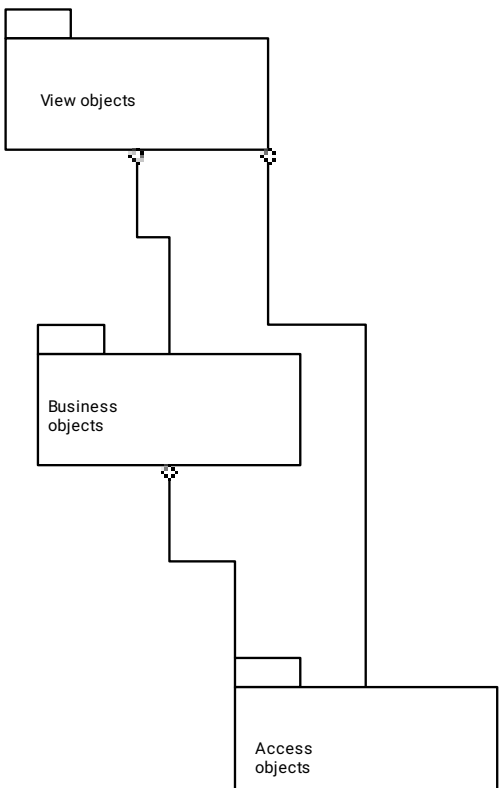
Designing View Layer Classes

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

- **Input-Responding to user interaction**
- **Output-Displaying business objects**

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.**



Designing View Layer Classes (Cont'd)

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

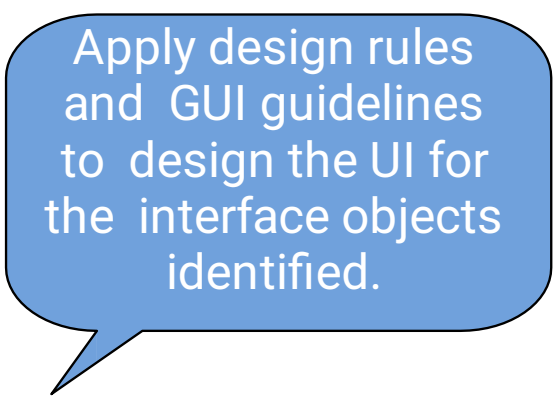
View Layer Macro Level

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

View Layer Micro Level

- 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.**
- **KISS, Keep It Simple, Stupid.**
- **Simplicity is different from being simplistic.**
- **Making something simple requires a good deal of work and code.**

Making The Interface Simple(Con’t)

- **Every additional feature potentially affects performance, complexity, stability, maintenance, and support costs of an application.**

UI Design Rule 1 (Con't)

- **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

- **Metaphors can assist the users to transfer their previous knowledge from their work environment to your application interface.**
- **For example, forms that users are accustomed to seeing.**

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.**
  - **Avoiding Modes.**
  - **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.**

Avoiding Modes

- Users are in a mode whenever they must cancel what they are doing before they can do something else.
- Modes force users to focus on the way an application works, instead of on the task they want to complete.

Can Modes be useful?

Yes, however:

- **You should make modes an exception and limit their use.**
- **Whenever users are in a mode, you should make it obvious by providing good visual cues.**
- **The method for ending the mode should be easy to learn and remember.**

Modes can be useful (Con’t)

These are some of the modes that can be used in the user interface.

- **Modal Dialog**
- **Spring-Loaded Modes**
- **Tool-Driven Modes**

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 User 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.**

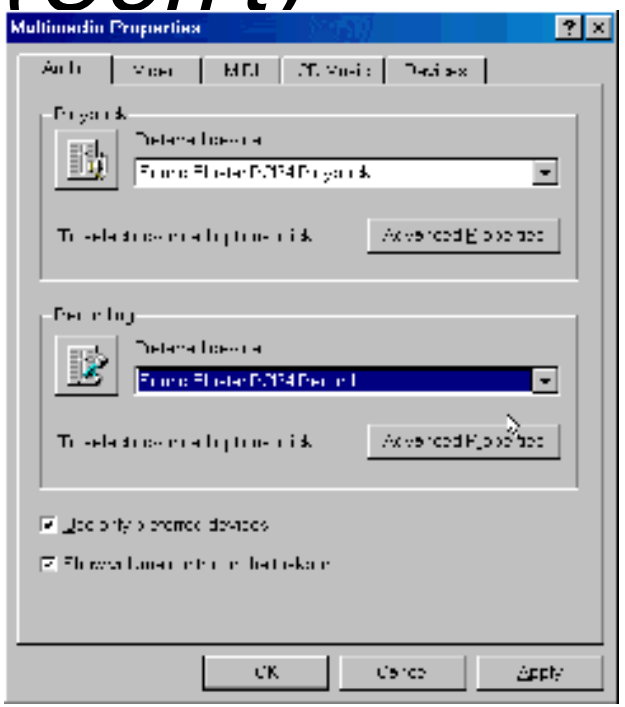
Guidelines For Designing Data *Entry Windows (Con't)*

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.**

Guidelines For Designing Data *Entry Windows (Con't)*

- **An example of a dialog box with multiple pages in the Microsoft multimedia setup.**

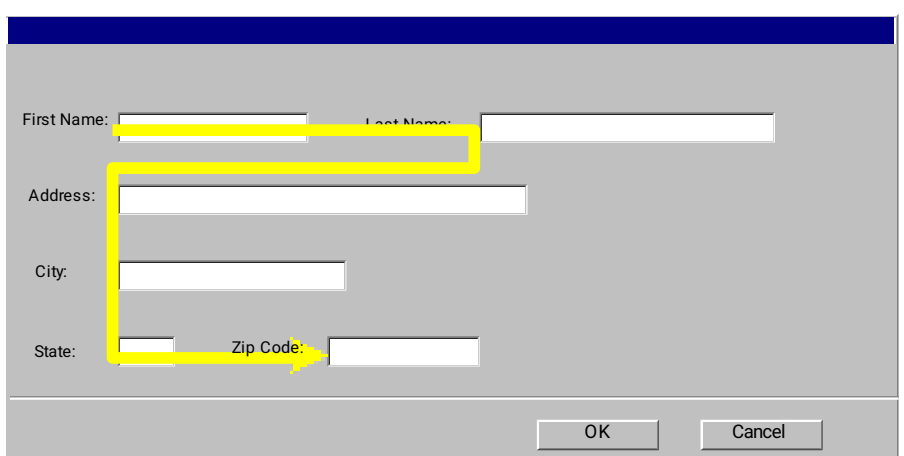


Designing Data *Entry Windows (Con't)*

Users scan a screen in the same way they read a page of a book, from left to right, and top to bottom.

Guidelines For Designing Data *Entry Windows (Con't)*

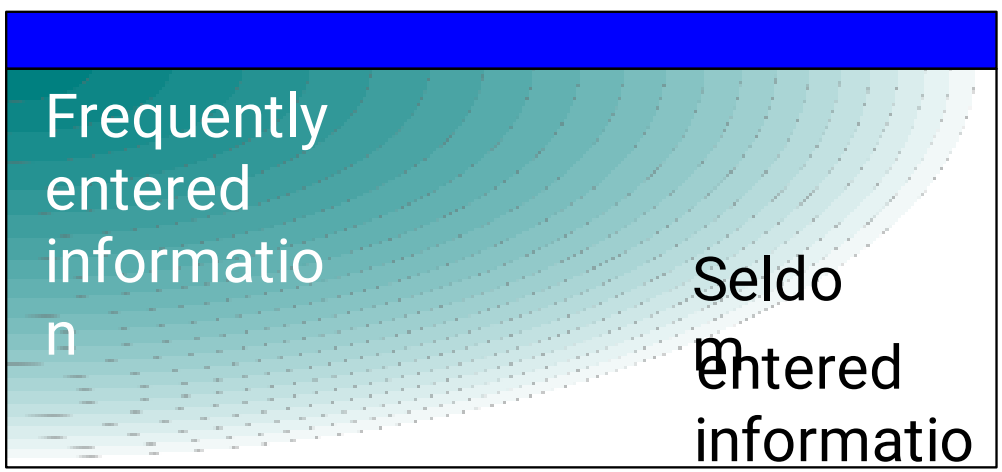
- Orient the controls in the dialog box in the direction people read.
- In the Western world this usually means left
- to right, top to bottom.



Guidelines For Designing Data

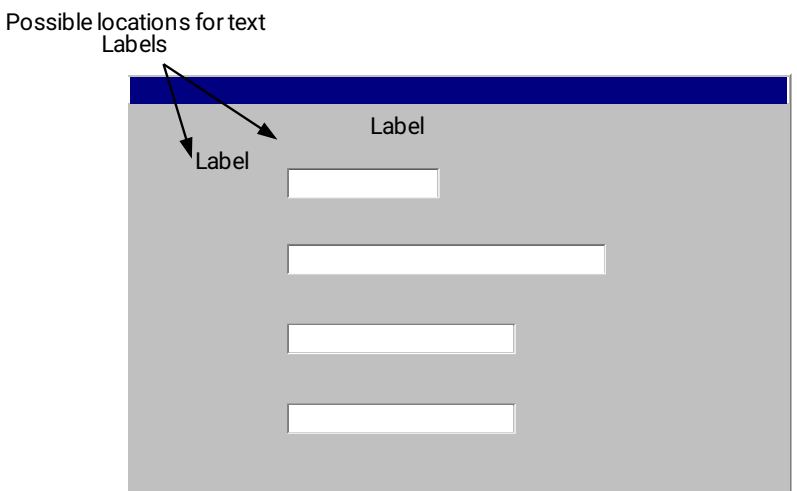
Entry Windows (Con't)

- **Required information should be put toward the top and left side of the form, entering optional or seldom entered information toward the bottom.**



Guidelines For Designing Data *Entry Windows (Con't)*

- **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."

Guidelines For Designing Dialog *Boxes(Con't)*

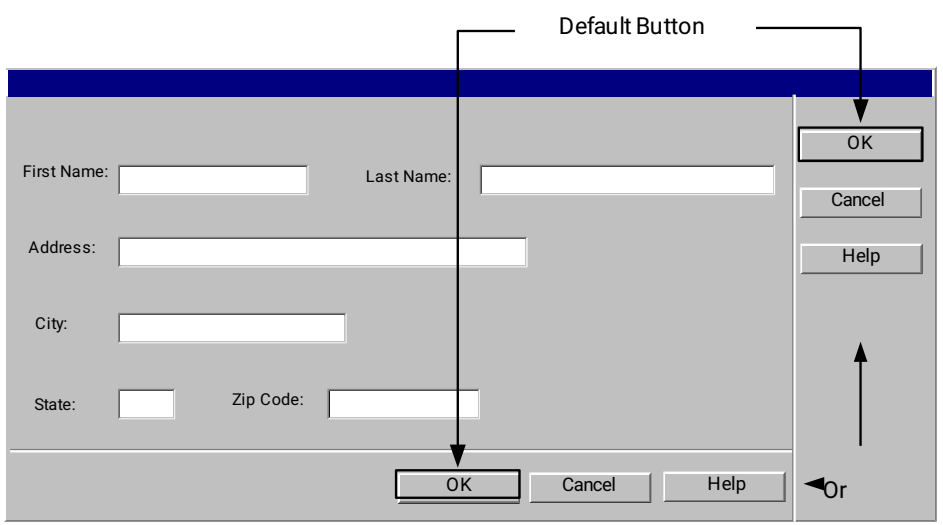
- Your error message should be constructive, brief and meaningful.
- For example, avoid messages such as

  "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.**



Buttons Layout (Con't)

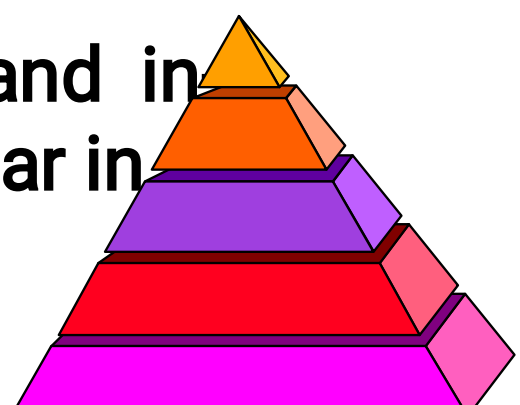
- **Positioning buttons on the left or center is popular in Web interfaces.**

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.**

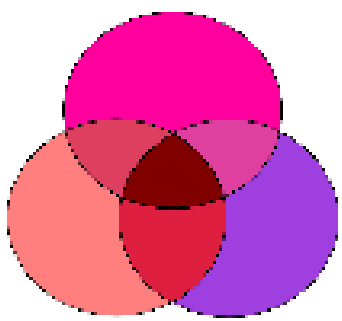
Guidelines For Using Colors

- 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.



Guidelines For Using Colors *(Con't)*

- **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.**



Guidelines For Using Colors *(Con't)*

- **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.**

Guidelines For Using Colors*(Con't)*

- **Use colors consistently within each window and among all Windows in your application.**
- **For example the colors for Pushbuttons should be the same throughout.**

Guidelines For Using Colors *(Con't)*

- **Using too many colors can be visually distracting, and will make your application less interesting.**

Guidelines For Using Colors*(Con't)*

- **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.**

Guidelines For Using Fonts (Con't)

- 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.

Guidelines For Using Fonts (Con't)

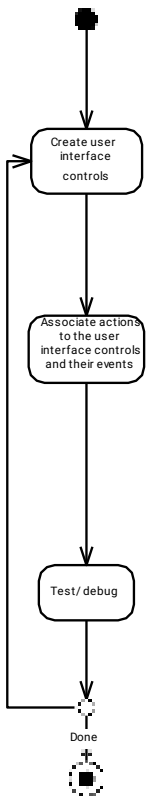
- **Using too many font styles, sizes and colors can be visually distracting and should be avoided.**

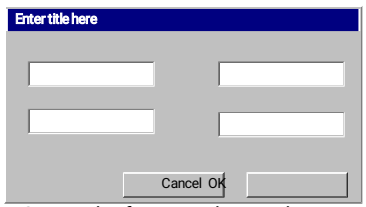
Prototyping the User Interface

- **Rapid prototyping encourages the incremental development approach, "grow, don't build."**

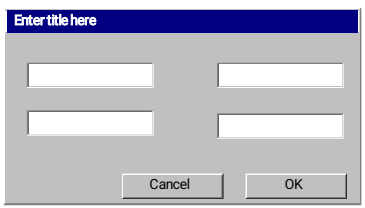
Three General Steps

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

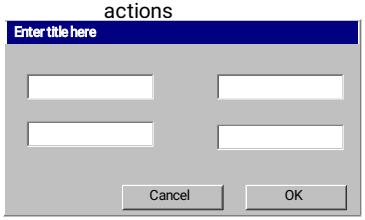




Create the forms and controls



Add actions



Test the UI

Make Users Feel in Charge

- Instead of using leading phrases like,

  "we could do this ..." or "It would be easier if we ..."

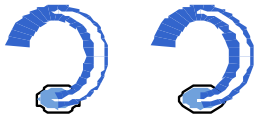
- Choose phrases that give the user the feeling that he/she is in charge:

  "Do you think that if we did ... it would make it easier for the users?"

  "Do users ever complain about ...? We could add .. to make it easier."

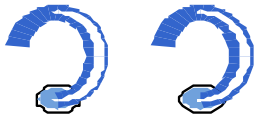
Summary

- The main goal of UI is to display and obtain information you need in an accessible, efficient manner.
- The design of your software's interface, more than anything else, affects how a user interacts and therefore experiences your application.



Summary (Con't)

- UI must provide users with the information they need and clearly tell them what they need to successfully complete a task.
- A well-designed UI has visual appeal that motivates users to use your application.
- UI should use limited screen space efficiently.



Summary (Con't)

- **Designing View layer classes consists of the following steps:**

I. **Macro Level UI Design Process- Identifying View Layer Objects**

II. **Micro Level UI Design Activities**

II.1 Designing the View Layer Objects by applying Design Axioms and corollaries .

II. **2 Prototyping the View Layer Interface.**

III. **Usability and User Satisfaction Testing**

IV. **Refine and Iterate**

Summary (Con't)

- Guidelines are not a standalone tool, and they cannot substitute for effective evaluation and iterative refinement within a design.
- However, they can provide helpful advice during the design process.