THE OBJECT-ORIENTED DESIGN PROCESS AND DESIGN AXIOMS (CH -9)

By:
Mr.Prachet Bhuyan
Assistant Professor,
School of Computer Engineering,
KIIT University

Topics to be Discussed

- 9.1 INTRODUCTION
- 9.2 THE O-O DESIGN PROCESS
- 9.3 O-O DESIGN AXIOMS
- 9.4 COROLLARIES
  - 9.4.1 Corollary 1: Uncoupled Design with Less Information Content
  - 9.4.2 Corollary 2: Single Purpose
  - 9.4.3 Corollary 3: Large Number of Simpler Classes, Reusability

Topics to be Discussed contd..

9.4.4 Corollary 4: Strong Mapping

9.4.5 Corollary 5: Standardization

9.4.6 Corollary 6: Designing with Inheritance

9.5 DESIGN PATTERNS

9.1 INTRODUCTION

- OOA phase of software development was on "what needs to be done".
- The objects discovered during analysis can serve as the framework for design.
- The class's attributes, methods and associations identified during analysis must be designed for as a data type expressed in the implementation language.
- New classes can be introduced to store intermediate result during program execution.

9.1 INTRODUCTION contd..

- During design phase we elevate the various object models (individuals, organizations, machines, etc) into logical entities, some of which might relate more to the computer domain (such as UIs or access layer).
- Good design simplifies the implementation and maintenance of a project.
- The design model does not look terribly different from the analysis model.

9.1 INTRODUCTION contd..

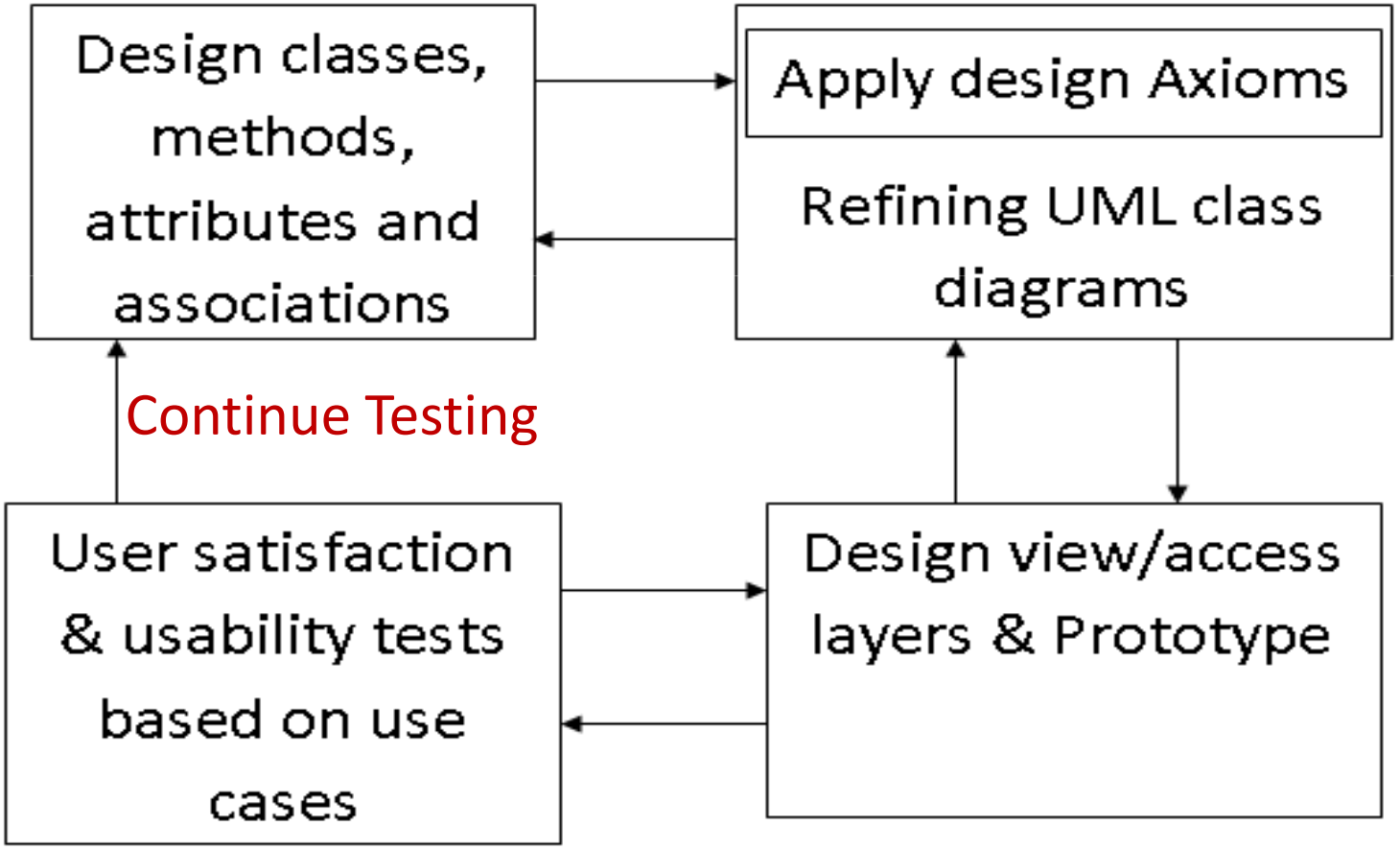
- The difference between OOA & OOD is that, at OOD level, we focus on the view and access classes, such as:
  - How to maintain information or
  - The best way to interact with the a user or
  - Present information
- However the time spent on design has a great impact on the over all success of the software development project.

9.1 INTRODUCTION contd..

- Here we look at the O-O design process and axioms.
- The basic goal of axiomatic process is :
  - To formalize the design process
  - Assist in establishing a scientific foundation for the O-O design process
  - To provide a fundamental basis for creation of systems.

9.2 THE O-O DESIGN PROCESS

OOD Design Process in UA



9.2 THE O-O DESIGN PROCESS

The O-O design process consists of the following activities:

1. Apply design axioms to design classes, their attributes, methods, associations, structures & protocols.
   i. Refine and complete the static UML class diagram by adding details to the UML class diagram. This steps consists of:
      a) Refine attributes

9.2 THE O-O DESIGN PROCESS contd..

b) Design methods & Protocols by utilizing a UML activity diagram to represent the method's algorithm.

c) Refine associations between classes (if required)

d) Refine class hierarchy & design with inheritance (if required)

ii. Iterate and refine again.

2. Design the access layer (CH-11)

i. Create mirror classes. For every business class identified and created, create one access class.

ii. Identify access layer class relationships.

9.2 THE O-O DESIGN PROCESS contd..

iii. Simplify classes and their relationships. Eliminate redundant classes & structures.

a) Redundant classes: Do not keep two classes that perform similar translate request and translate results activities. Simply select one and eliminate the other.

b) Method classes: Revisit the classes consisting of only one or two methods to see if they can be eliminated or combined with existing clases.

iv. Iterate and refine again.

3. Design the view layer classes (CH-12)

9.2 THE O-O DESIGN PROCESS contd..

i. Design the macro level user interface, identifying view layer objects.

ii. Design the micro level user interface, consists of:

   a) Design the view layer objects by applying the design axioms and corollaries.

   b) Build a prototype of the view layer interface.

iii. Test usability & user satisfaction

iv. Iterate & refine.

4. Iterate & refine the whole design.

9.3 O-O DESIGN AXIOMS

- 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. (hence equivalent to a law or principle)
- A corollary is a proposition that follows from an axiom or another proposition that has been proven.

9.3 O-O DESIGN AXIOMS contd..

Suh's design axioms applied to O-O design:

- AXIOM 1: *The independence axiom*. Maintain the independence of components.
  - Axiom 1 deals with relationships between system components (such as classes, requirements and software components)
- AXIOM 2: *The information axiom*. Minimize the information content of the design.
  - Axiom 2 deals with the complexity of design.

9.3 O-O DESIGN AXIOMS contd..

- Axiom 2 is concerned with simplicity. Occam's razor rule of simplicity in terms of O-O:
  - The best designs usually involves the least complex code but not necessarily the fewest number of classes or methods.
  - Minimizing complexity should be the goal, because that produces the most easily maintained and enhanced application.
  - In an o-o system, the best way to minimize complexity is to use inheritance and the system's built-in classes and to add as little as possible to what already is there.

9.3 O-O DESIGN AXIOMS contd..

- OCCAM'S RAZOR Says..

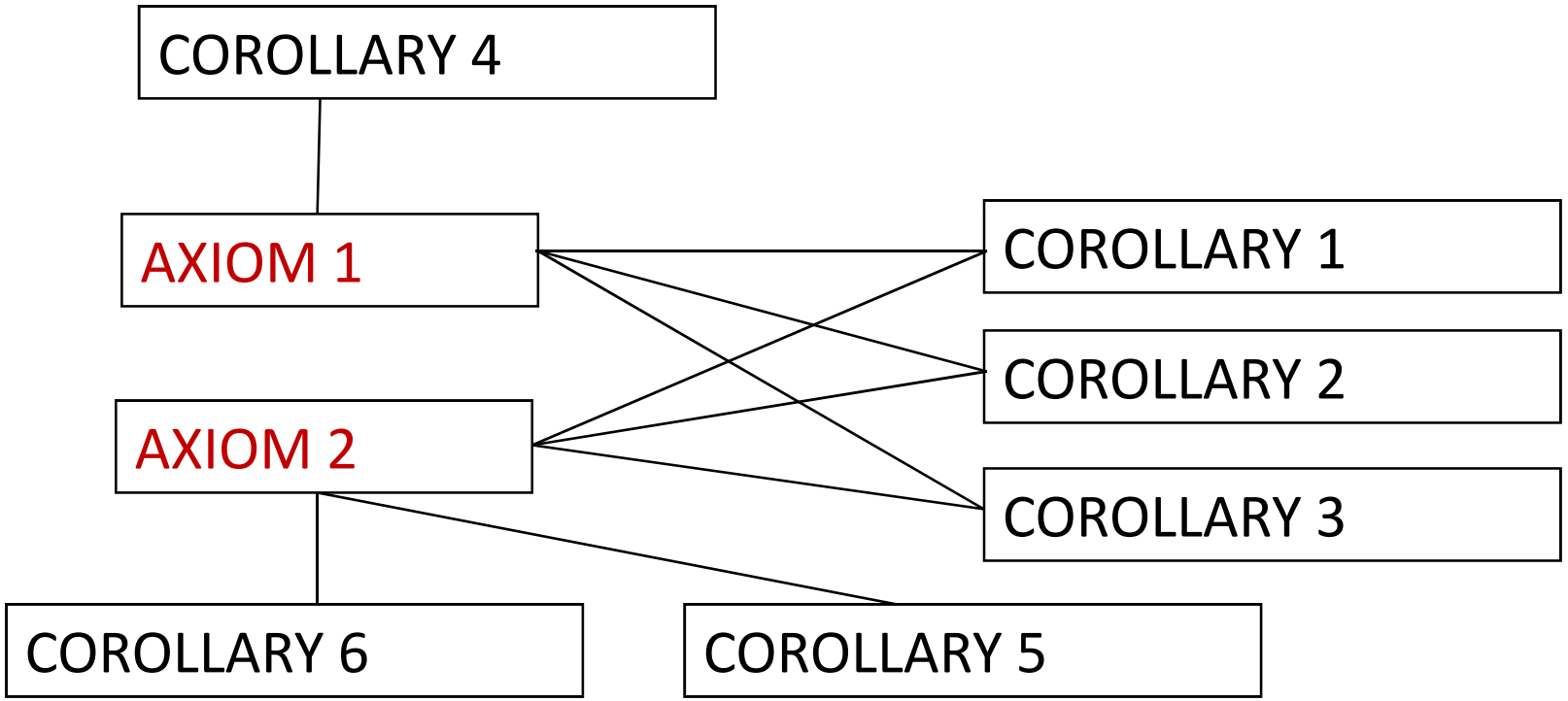
- "The best theory explains the known facts with a minimum amount of complexity and maximum simplicity and straightforwardness."

9.4 COROLLARIES

- From the two design axioms, many corollaries may be derived.
- These corollaries may be more useful in making specific design decisions, than the original axioms in actual situations.
- They may even be called *design rules* derived from two basic axioms.

9.4 COROLLARIES contd..

- The Origin of Corollaries:



9.4 COROLLARIES contd..

- Corollary 1: ***Uncoupled Design with Less Information Content.***
  - Highly cohesive objects can improve coupling because only a minimal amount of essential information need be passed between objects.
- Corollary 2: ***Single Purpose.***
  - Each class must have a single, clearly defined purpose which can be describe in few sentences.

9.4 COROLLARIES contd..

- Corollary 3: **Large Number of Simple Classes.**
  - Keeping the classes simple allows reusability.

- Corollary 4: ***Strong Mapping*.**
  - There must be strong association between the physical system (analysis's object) & logical design (design's object)

9.4 COROLLARIES contd..

- Corollary 5: ***Standardization***. (Promote it.)
  - By designing interchangeable components &
  - By reusing existing classes & components.
- Corollary 6: ***Design with Inheritance***.
  - Common behavior (methods) must be moved to superclasses.
  - The superclass-sub class structure must make logical sense.

9.4.1 Corollary 1: Uncoupled Design with Less Information Content

- The main goal here is:
  - To maximize objects cohesiveness among objects & software components to improve coupling.
- 9.4.1.1 COUPLING: Coupling is a measure of the strength of association established by a connection from one object or software component to another.
  - Coupling is a binary relationship: A is coupled with B
  - Strong coupling among objects complicates a system.

9.4.1 Corollary 1: Uncoupled Design with Less Information Content contd..

- The degree of coupling is a function of:
  1. How complicated the connection is.
  2. Whether the connection refers to the object itself or something inside it.
  3. What is being sent or received.
- The degree or strength of coupling between two components is measured by the amount & complexity of information transmitted between them.

9.4.1 Corollary 1: Uncoupled Design with Less Information Content contd..

- Strong Coupling: Coupling increases (becomes stronger) with increasing complexity or obscurity of the interface.
- Low or Weak Coupling: Coupling decreases (becomes lower) when the connection is to the component interface rather than to an internal component.
  - Coupling also is lower for data connections than for control connections.

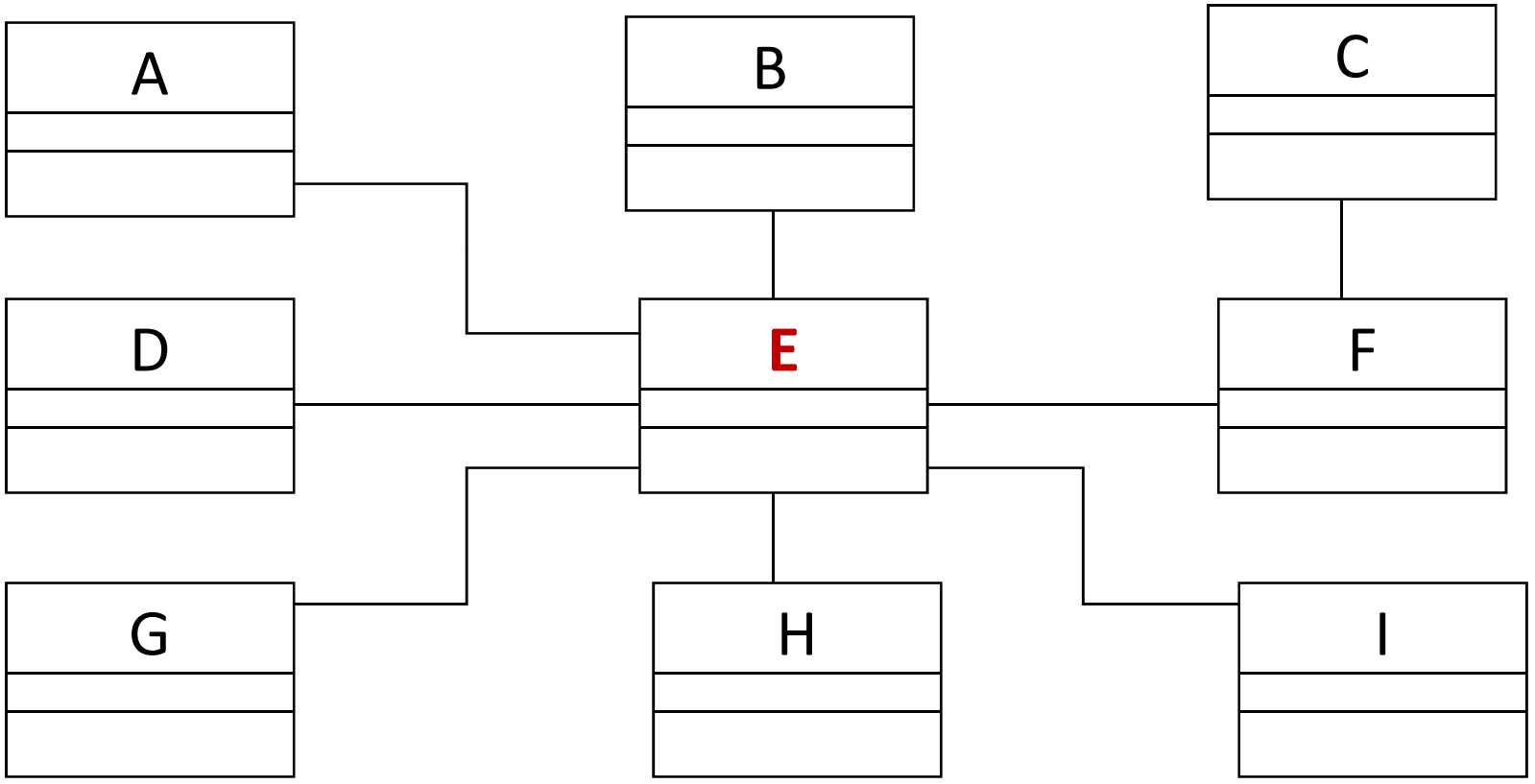
9.4.1 Corollary 1: Uncoupled Design with Less Information Content contd..

✓ O-O Design has TWO types of Coupling:

1. Interaction Coupling : Involves the amount and complexity of messages between components.
   - General Guideline: is to keep the messages as simple & infrequent as possible.
   - Ex: If a message connection involves more than three parameters (eg: in Method(X,Y,Z) where X,Y,Z are parameters and any change in one will have ripple effect of changes in other. Hence TIGHTLY COUPLED.

9.4.1 Corollary 1: Uncoupled Design with Less Information Content contd..

- E is a Tightly Coupled Object:



9.4.1 Corollary 1: Uncoupled Design with Less Information Content contd..

- Types of Interaction Coupling are:

DEGREE OF COUPLING	NAME OF COUPLING	DESCRIPTION
VERY HIGH	CONTENT COUPLING	The connection involves direct reference to attributes or methods of another object.
HIGH	COMMON COUPLING	The connection involves two objects accessing a global data space for both to read & write.
MEDIUM	CONTROL COUPLING	The connection involves explicit control of the processing logic of one object by another.
LOW	STAMP COUPLING	Involves passing an aggregate data structure to another object, using partial of it.
VERY LOW	DATA COUPLING (Should be the goal of Architectural Design)	Involves either simple data items or aggregate structures all of whose elements are used by the receiving object.

9.4.1 Corollary 1: Uncoupled Design with Less Information Content contd..

2. Inheritance Coupling: It is a form of coupling between super and sub classes.
   - A subclass is coupled to its super class in terms of attributes and methods.
   - Unlike interaction coupling, high inheritance coupling is desirable.

9.4.1 Corollary 1: Uncoupled Design with Less Information Content contd..

- 9.4.1.2 COHESION : There is a need to consider interactions within a single object or software component, called ***cohesion***.
  - It reflects the "single-purposeness" of an object.
  - Highly cohesive components can lower coupling as a minimum information of essential information need be passes between components.
  - It helps to design classes which have very specific purpose.

9.4.1 Corollary 1: Uncoupled Design with Less Information Content contd..

- Types of Cohesion are:
  - Method Cohesion: Like function cohesion, means that method should carry only one function.
  - Class Cohesion: It means that all the class's methods and attributes must be highly cohesive, (used by internal methods or derived classes' methods).
  - Inheritance Cohesion: - Concerned with:
    - How interrelated are the classes ?
    - Does specialization really portray specialization or it is just something arbitrary ?

9.4.2 Corollary 2: Single Purpose

- Every class should be clearly defined.
- It is necessary to achieve the system's goal.
- During documentation we should be able to explain the purpose of a class in one or two sentences.
- If not, then rethink the class and try to subdivide the class into more independent pieces.
- Keep it simple. (Each method to provide one service.)

9.4.3 Corollary 3: Large Number of Simpler Classes, Reusability

- Simpler classes is beneficial.
- It is difficult to foresee all future scenarios of the class to be reused.
- The less specialized the classes are, more problems can be solved by combining them with less subclasses. GUIDELINE:
- "The smaller are your classes, the better are your chances of reusing them in other projects. Large and complex classes are too specialized to be reused"

9.4.3 Corollary 3: Large Number of Simpler Classes, Reusability contd..

- Coad & Yourdon describes FOUR reasons why people are not utilizing reusability concept more:
  1. Software engineering textbooks teach new practitioners to build systems from "first principle"; reusability is not promoted or even discussed.

9.4.3 Corollary 3: Large Number of Simpler Classes, Reusability contd..

2. The "not invented here" syndrome and the intellectual challenge of solving problem an interesting problem in one's own unique way mitigates against reusing someone else's software component.
3. Unsuccessful experiences with software reusability in the past have convinced many practitioners and development managers that the concept is not practical.

9.4.3 Corollary 3: Large Number of Simpler Classes, Reusability contd..

4. Most organization provide no reward for reusability; sometimes productivity is measured in terms of new lines of codes written plus a discounted credit (e.g., 50% less credit) for reused lines of code.

- Benefits of Software Reusability:
  - Higher productivity
  - The software development team that achieves 80% reusability is four times as productive as the team that achieves only 20% reusability.

9.4.3 Corollary 3: Large Number of Simpler Classes, Reusability contd..

- Using successful design patterns we can recreate.
- O-O design encourages reusable libraries supported by OOP languages.
- O-O design emphasizes on reusing concepts like:
  i. Encapsulation
  ii. Modularization (e.g., class structure)
  iii. Polymorphism

9.4.4 Corollary 4: Strong Mapping

- OOA & OOD are based on same UML model.
- Eg; During analysis we might identify a class EMPLOYEE. During the design phase, we need to design this class:
  - Design its methods
  - Design its association with other objects
  - Design its view & access classes

9.4.4 Corollary 4: Strong Mapping contd..

- A strong mapping links classes identified during analysis and classes designed during the design phase.
- According to Martin & Odell:
  - With O-O technique the same paradigm is used for analysis, design & implementation.
  - The analyst identifies objects' types & inheritance & think about the events that change the state of object.

9.4.4 Corollary 4: Strong Mapping contd..

- The designer adds details to this model perhaps designing screens, user interaction, and client-server interaction.
- The thought process flows so naturally from analyst to design that it may be difficult to tell where analysis ends and design begins.

9.4.5 Corollary 5: Standardization

- To reuse one should have good understanding of classes in OOP environment.
- Most OOP such as Smalltalk, Java, C++, or PowerBuilder come with built-in class libraries.
- O-O systems grow as you create new applications.
- The knowledge of existing class will help in reuse by inheriting from the existing class libraries.

9.4.5 Corollary 5: Standardization contd..

Some shortfall:

- However class libraries are not always well documented and updated.
- Class libraries should be easily searched based on user's criteria.

Solution:

- Making a repository of Design Patterns can give some solutions to all these problems.

9.4.6 Corollary 6: Designing with Inheritance

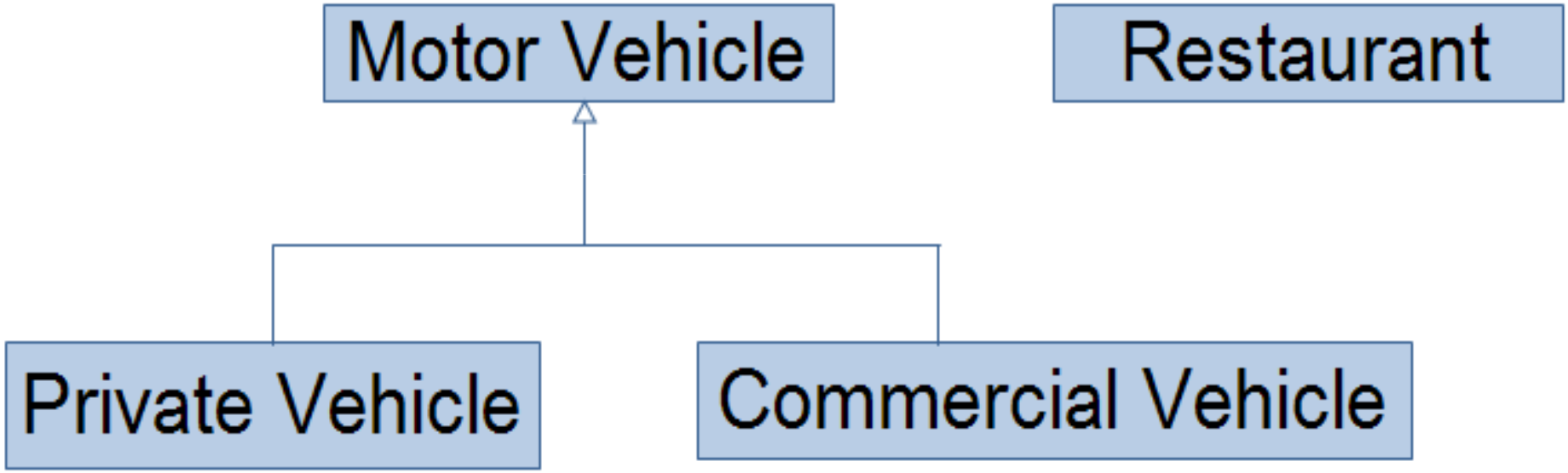
- When you implement a class you have to determine its ancestor, along with:
  - What attributes it will have &
  - What messages it will understand.
- Then its methods & protocols are constructed.
- Ideally you will choose inheritance to minimize the amount of program instructions.

9.4.6 Corollary 6: Designing with Inheritance contd..

- Issue here will be:
  - Achieving Multiple Inheritance in a Single Inheritance System.
  - Avoiding Inheriting Inappropriate Behaviors.
- Eg: Developing an application for the government that manages the licensing procedure for a variety of regulated entities:
- Like: License, Motor Vehicle, Private Vehicle, Commercial Vehicle, Restaurant, FoodTruck.

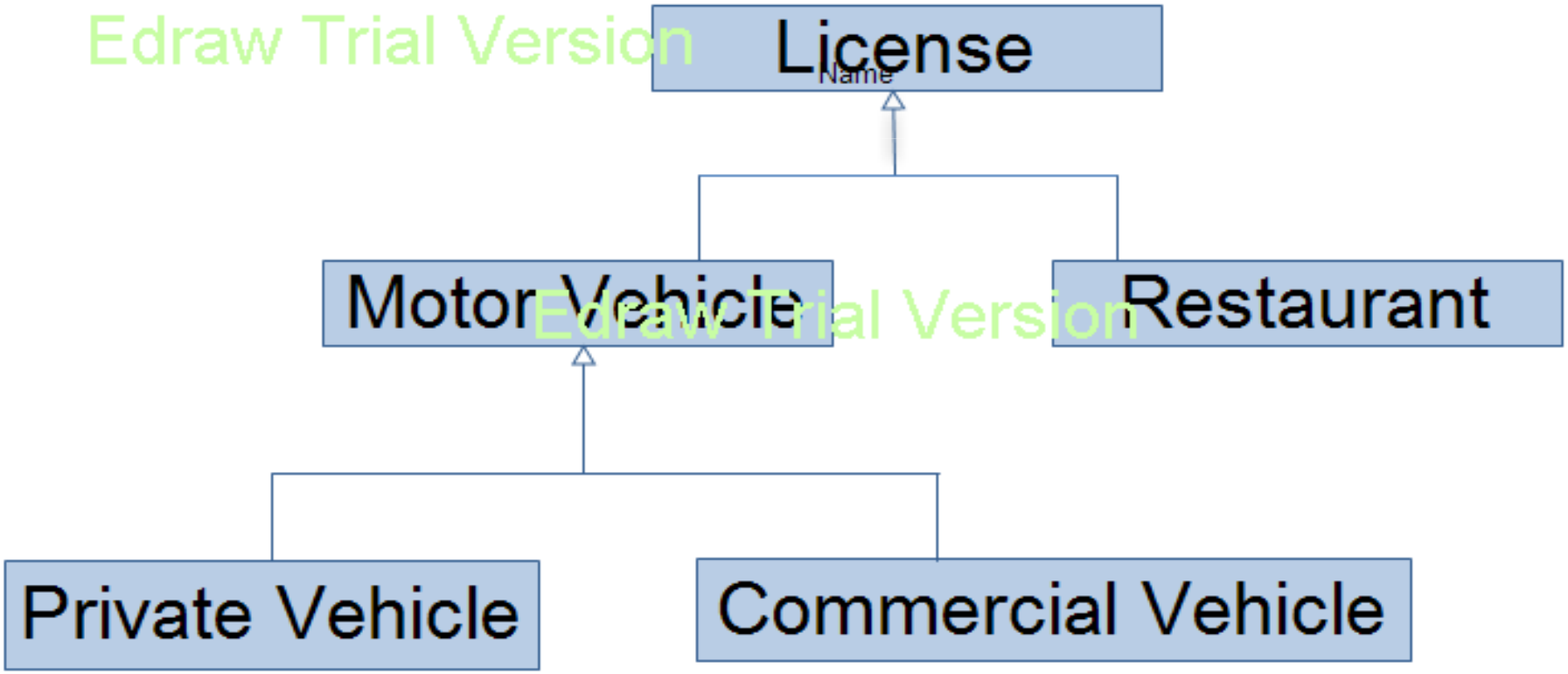
9.4.6 Corollary 6: Designing with Inheritance contd..

- The Initial Inheritance Design



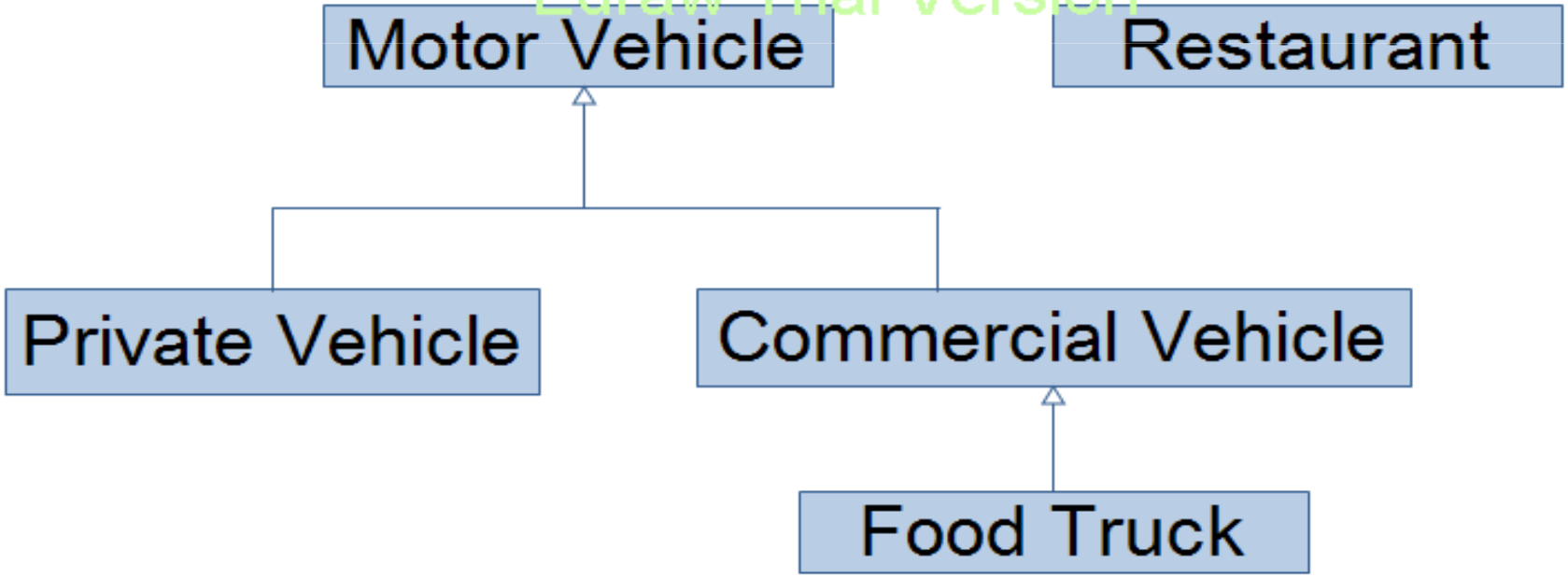
9.4.6 Corollary 6: Designing with Inheritance contd..

- The Single Inheritance Design Modified to Allow Licensing Food Trucks.



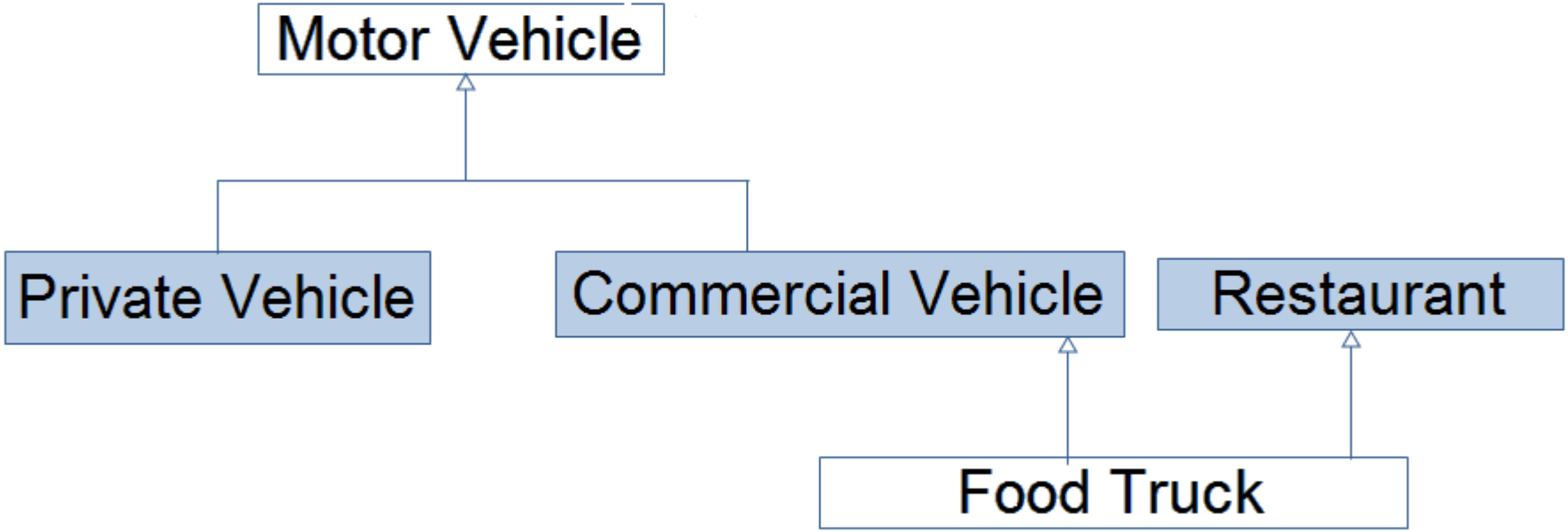
9.4.6 Corollary 6: Designing with Inheritance contd..

- Alternately you can Modify the Single Inheritance Design to Allow Licensing Food Truck.



9.4.6 Corollary 6: Designing with Inheritance contd..

- Multiple Inheritance Design of the System Structure



9.5 DESIGN PATTERNS (Eg:)

- Design Patterns are devices:
  - that allows systems to share knowledge about their design,
  - by describing commonly recurring structures of communicating components
  - that solve a general design problem within a particular context.
- Documenting patterns is one way that allows reuse & sharing information of best practices.

9.5 DESIGN PATTERNS (Eg:) contd..

Design Pattern example created by Kutotsuchi

- Pattern Name: **Facade**
- Rational & Motivation: This pattern can make the task of accessing a large number of modules much simpler by providing an additional interface layer.
  - This is done by creating a small collection of classes that have a single class that is used to access them, the **FACADE**.

9.5 DESIGN PATTERNS (Eg:) contd..

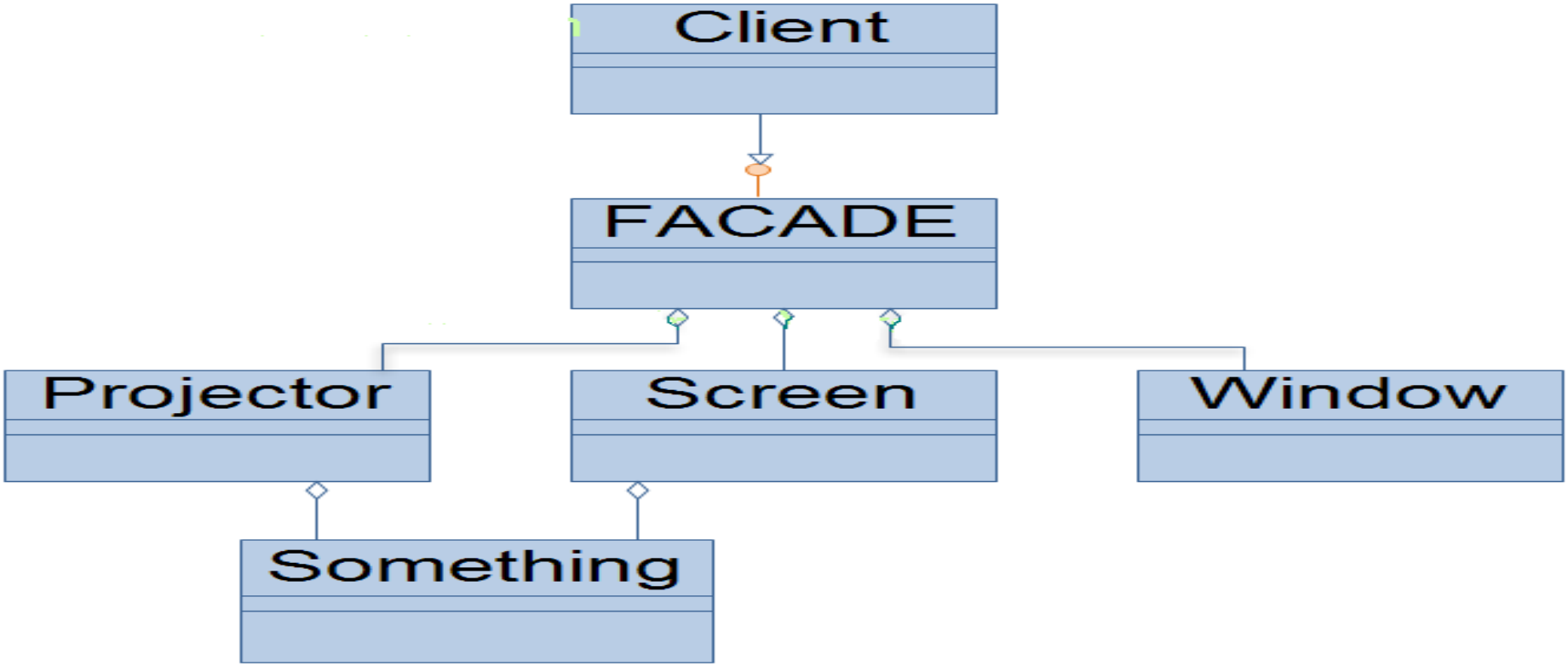
- Classes: There can be any number of classes involved in this “facade” system, but
  - At least four or more classes are required: *One client, the facade, & the classes underneath the facade.*
  - *A facade would be having limited coding most of the time making calls to lower layers.*
- Advantages: Using the facade to make the interfacing between many modules or classes.

9.5 DESIGN PATTERNS (Eg:) contd..

- Disadvantages: We may loose some functionality contained in the lower level of classes, but this depends on how the facade was designed.
- Example: Imagine there is a need to write a program that needs to represent a building as rooms that can be manipulated to interact with objects(windows, screens, projector, etc) in the room to change their state.

9.5 DESIGN PATTERNS (Eg:) contd..

- Using a Design Pattern FAÇADE Eliminates the need for the client class to deal with a large number of classes.



- END of CHAPTER 9

- USE DESIGN PATTERNS IN YOUR O-O SOFTWARE DESIGN