# Object Oriented Design and UML

# OO Concepts

- Information hiding – use encapsulation to restrict external visibility
- OO encapsulates the data, provides limited access, visibility
- Info hiding can be provided without OO – is an old concept

# OO Concepts...

- State retention – fns, procedures do not retain state; an object is aware of its past and maintains state
- Identity – each object can be identified and treated as a distinct entity
- Behavior – state and services together define the behavior of an object, or how an object responds

# OO Concepts..

- Messages – through which a sender obj conveys to a target obj a request
- For requesting O1 must have – a handle for O2, name of the op, info on ops that O2 requires
- General format O2.method(args)

# OO Concepts..

- Classes – a class is a stencil from which objects are created; defines the structure and services. A class has
  - An interface which defines which parts of an object can be accessed from outside
  - Body that implements the operations
  - Instance variables to hold object state
- Objects and classes are different; class is a type, object is an instance
- State and identity is of objects

# Relationship among objects

- An object has some capability – for other services it interacts with other objects
- Some different ways for interaction:
  1. Supplier object is global to client
  2. Supplier obj is a parm to some op of the client
  3. Supplier obj is part of the client obj
  4. Supplier obj is locally declared in some op
- Relationship can be either aggregation (whole-part relationship), or just client server relationship

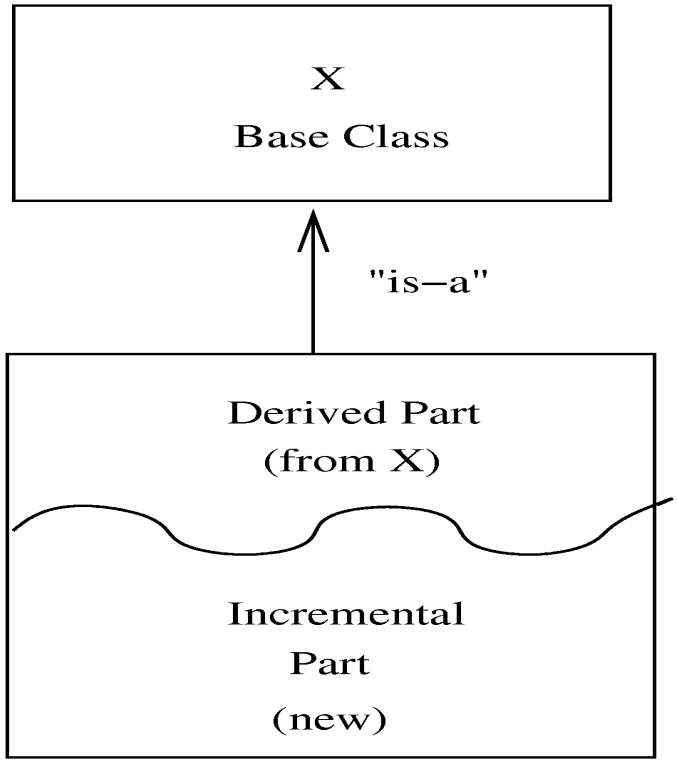
# Inheritance

- Inheritance is unique to OO and not there in function-oriented languages/models
- Inheritance by class B from class A is the facility by which B implicitly gets the attributes and ops of A as part of itself
- Attributes and methods of A are reused by B
- When B inherits from A, B is the *subclass* or *derived* class and A is the *base* class or *superclass*

# Inheritance..

- A subclass B generally has a derived part (inherited from A) and an incremental part (is new)
- Hence, B needs to define only the incremental part
- Creates an “is-a” relationship – objects of type B are also objects of type A

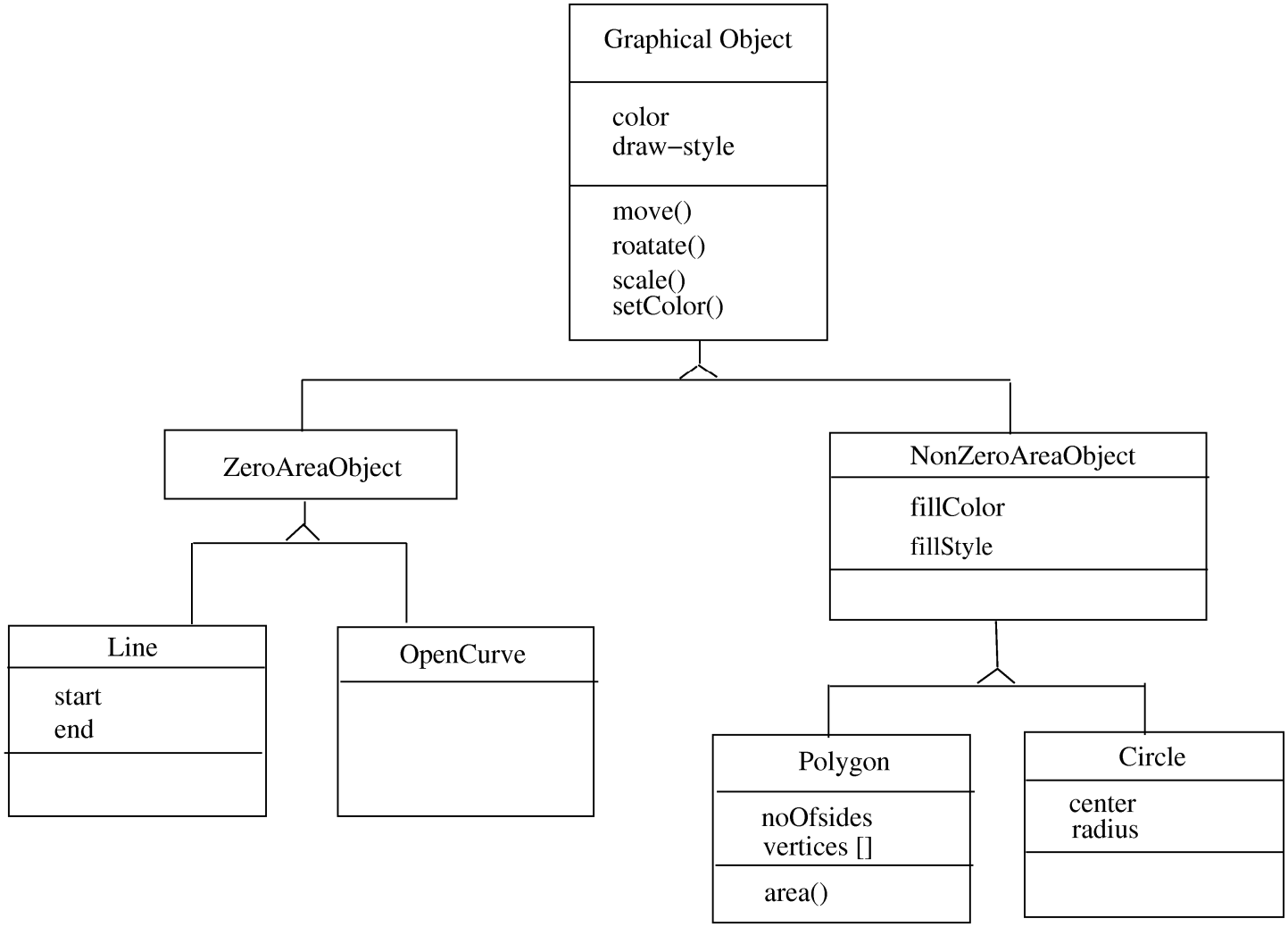
# Inheritance…




Y – Derived class

# Inheritance…

- The inheritance relationship between classes forms a class hierarchy
- In models, hierarchy should represent the natural relationships present in the problem domain
- In a hierarchy, all the common features can be accumulated in a superclass
- An existing class can be a specialization of an existing general class – is also called generalization-specialization relationships


Graphical Object
color
draw−style
move()
roatate()
scale()
setColor()
ZeroAreaObject
NonZeroAreaObject
fillColor
fillStyle
Line
start
end
OpenCurve
Polygon
noOfsides
vertices []
area()
Circle
center
radius

# Inheritance…

- Strict inheritance – a subclass takes all features of parent class
- Only adds features to specialize it
- Non-strict: when some of the features have been redefined
- Strict inheritance supports “is-a” cleanly and has fewer side effects

# Inheritance…

- Single inheritance – a subclass inherits from only one superclass
  - Class hierarchy is a tree
- Multiple inheritance – a class inherits from more than one class
  - Can cause runtime conflicts
  - Repeated inheritance - a class inherits from a class but from two separate paths

# Inheritance and Polymorphism

- Inheritance brings polymorphism, i.e. an object can be of different types
- An object of type B is also an object of type A
- Hence an object has a static type and a dynamic type
  - Implications on type checking
  - Also brings dynamic binding of operations which allows writing of general code where operations do different things depending on the type

# Unified Modeling Language (UML) and Modeling

- UML is a graphical notation useful for OO analysis and design
- Allows representing various aspects of the system
- Various notations are used to build different models for the system
- OOAD methodologies use UML to represent the models they create

# Modeling

- Modeling is used in many disciplines – architecture, aircraft building, …
- A model is a simplification of reality
- “All models are wrong, some are useful”
- A good model includes those elts that have broad effect and omits minor elts
- A model of a system is not the system!

# Why build models?

- Models help us visualize a system
- Help specify the system structure
- Gives us a template that can guide the construction
- Document the decisions taken and their rationale

# Modeling

- Every complex system requires multiple models, representing diff aspects
- These models are related but can be studied in isolation
- Eg. Arch view, electrical view, plumbing view of a building
- Model can be structural, or behavioral

# Views in an UML

- A use case view
- A design view
- A process view
- Implementation view
- Deployment view

- We will focus primarily on models for design – class diagram, interaction diagram, etc.

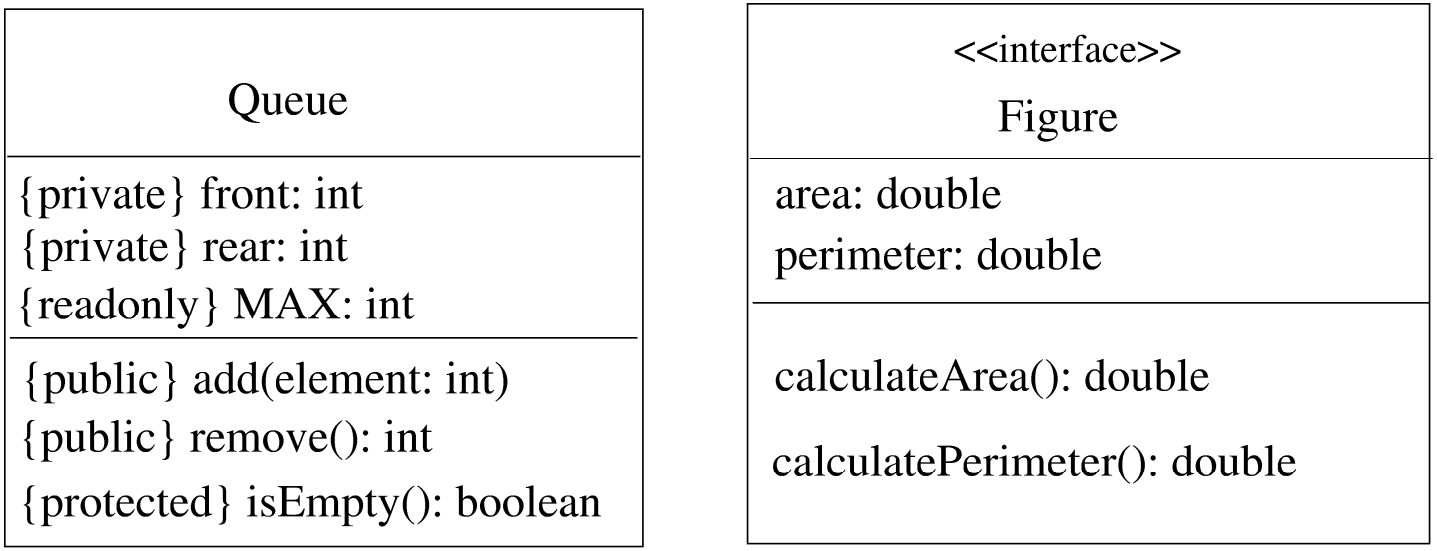
# Class Diagrams

- Classes are the basic building blocks of an OO system as classes are the implementation units also
- Class diagram is the central piece in an OO design. It specifies
  - Classes in the system
  - Association between classes
  - Subtype, supertype relationship

# Class Diagram…

- Class itself represented as a box with name, attributes, and methods
- There are conventions for naming
- If a class is an interface, this can be specified by <<interface>> stereotype
- Properties of attr/methods can be specified by tags between { }

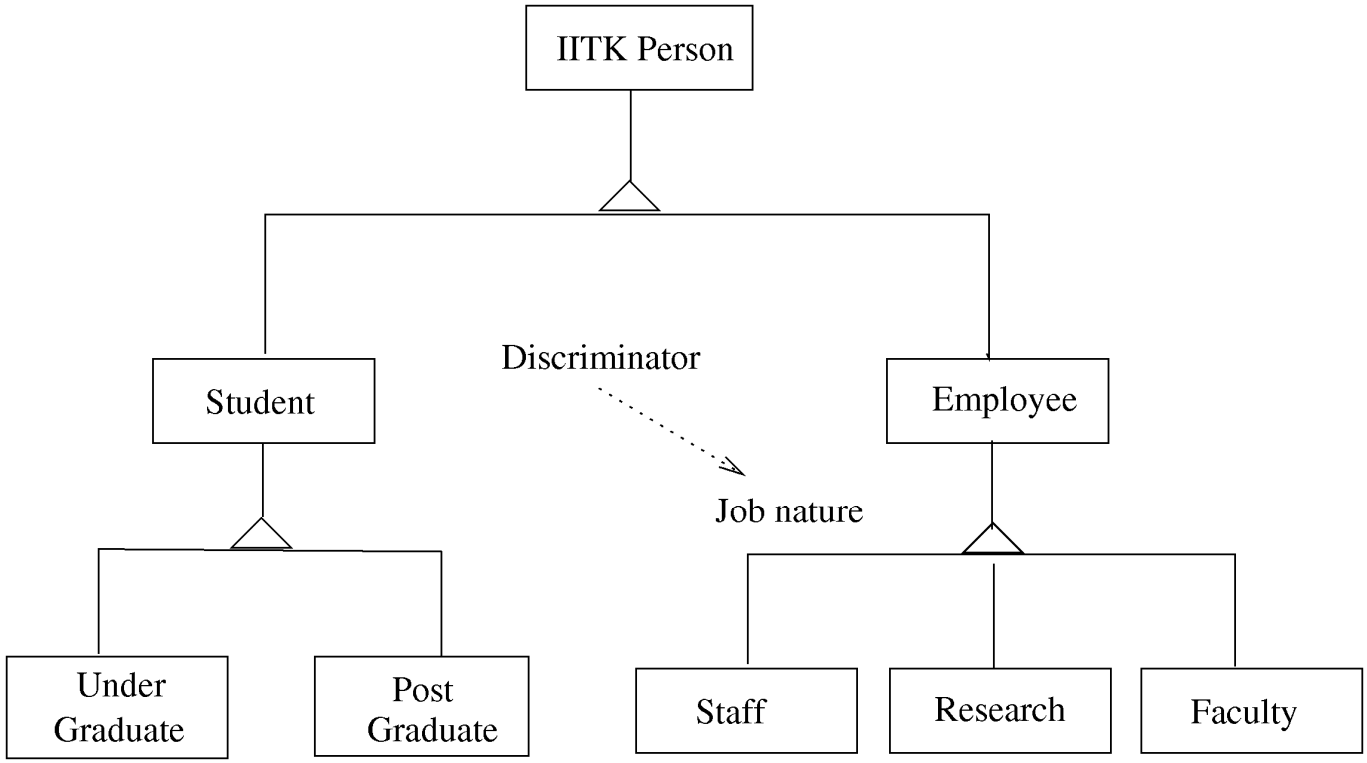
# Class – example



# Generalization-Specialization

- This relationship leads to class hierarchy
- Can be captured in a class diagram
  - Arrows coming from the subclass to the superclass with head touching super
  - Allows multiple subclasses
  - If specialization is done on the basis of some discriminator, arrow can be labeled

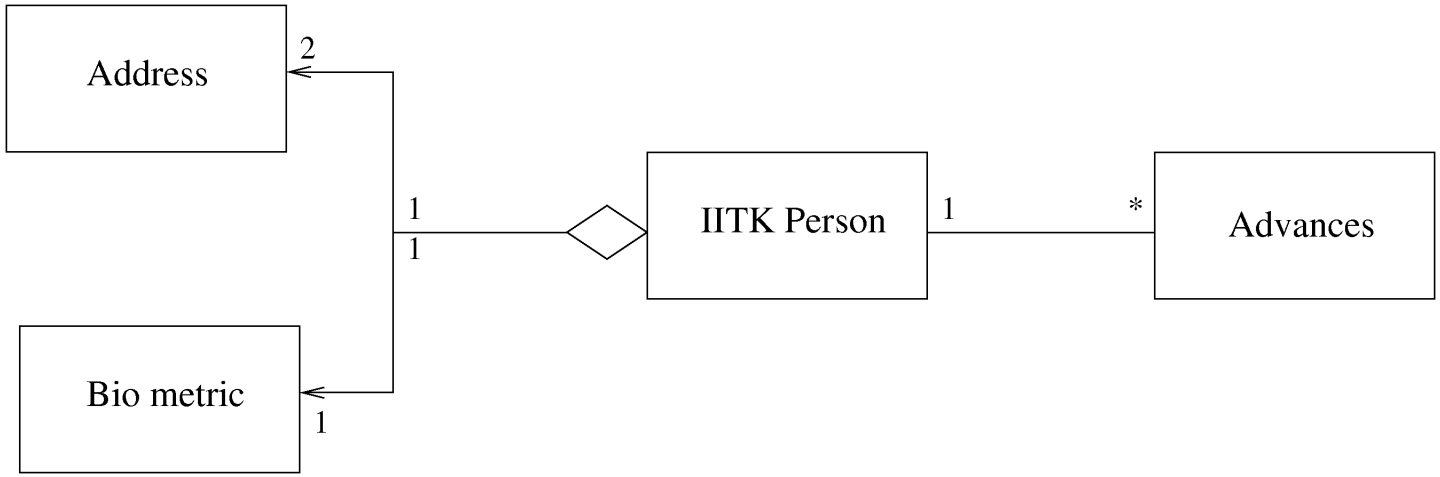
# Example – class hierarchy



# Association/aggregation

- Classes have other relationships
- Association: when objects of a class need services from other objects
  - Shown by a line joining classes
  - Multiplicity can be represented
- Aggregation: when an object is composed of other objects
  - Captures part-whole relationship
  - Shown with a diamond connecting classes

# Example – association/aggregation



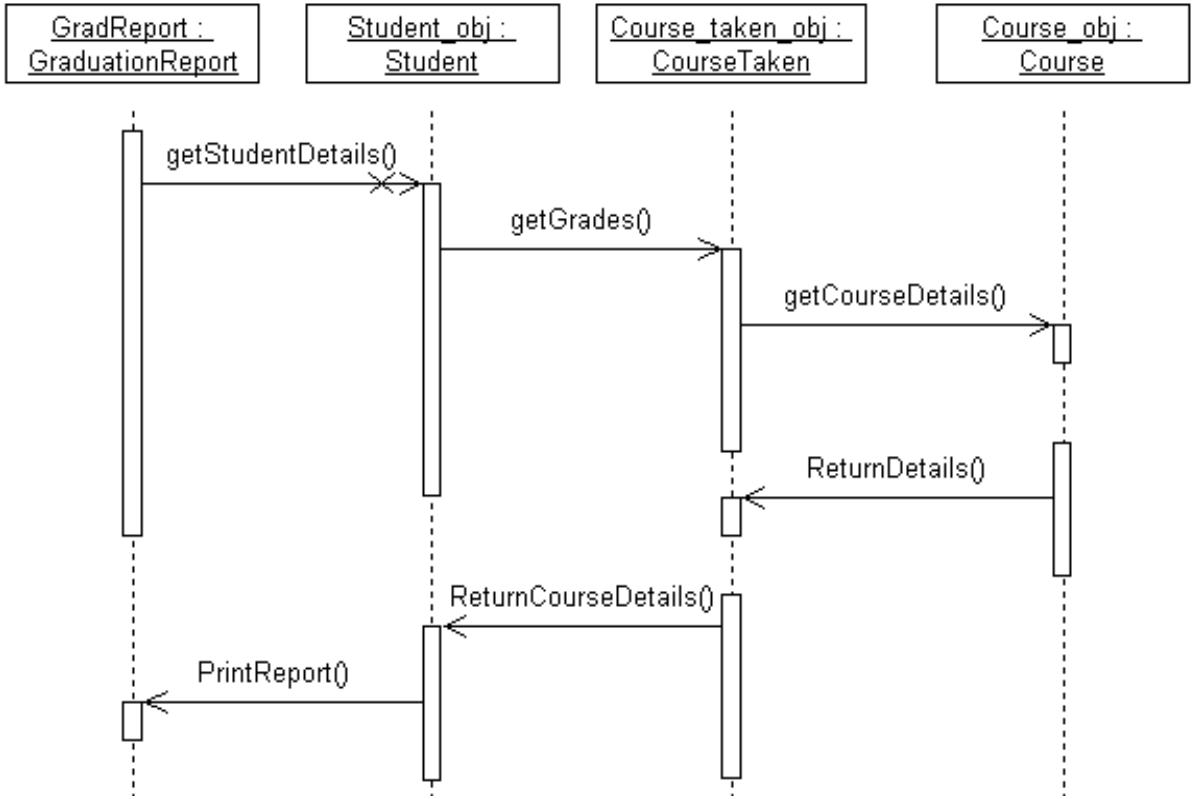
# Interaction Diagrams

- Class diagram represent static structure of the system (classes and their rel)
- Do not model the behavior of system
- Behavioral view – shows how objects interact for performing actions (typically a use case)
- Interaction is between objects, not classes
- Interaction diagram in two styles
  - Collaboration diagram
  - Sequence diagram
- Two are equivalent in power

# Sequence Diagram

- Objects participating in an interaction are shown at the top
- For each object a vertical bar represents its lifeline
- Message from an object to another, represented as a labeled arrow
- If message sent under some condition, it can be specified in bracket
- Time increases downwards, ordering of events is captured

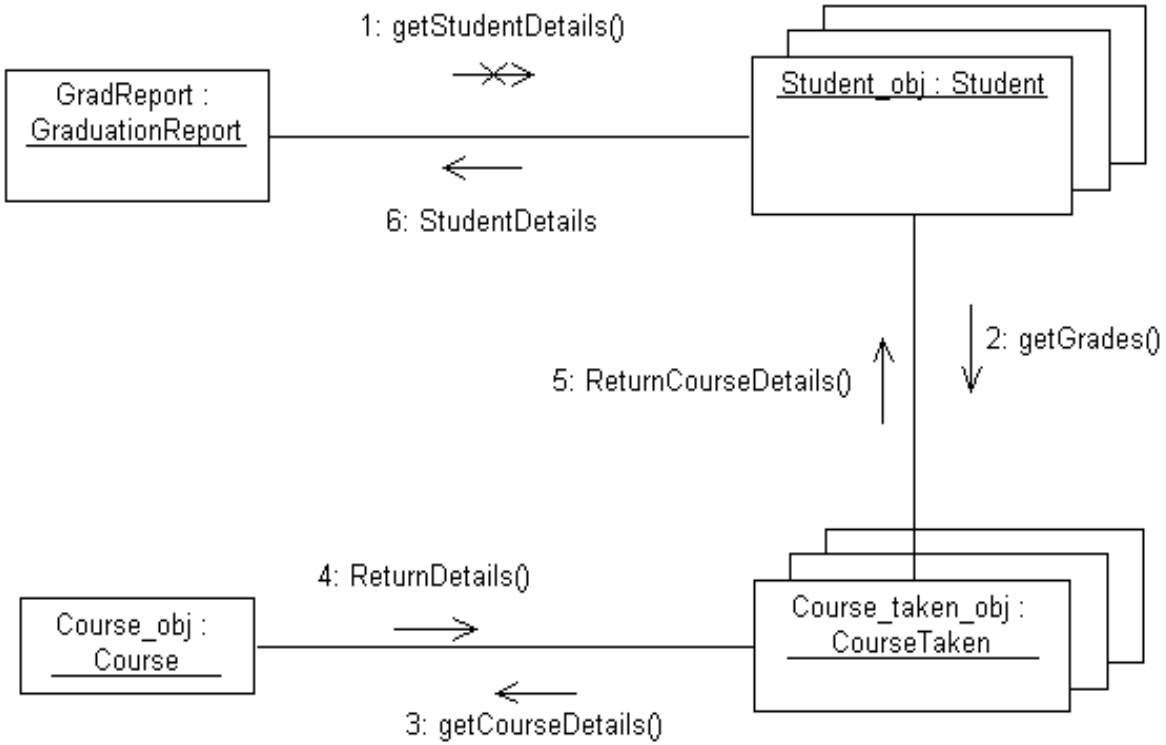
# Example – sequence diag.



# Collaboration diagram

- Also shows how objects interact
- Instead of timeline, this diagram looks more like a state diagram
- Ordering of messages captured by numbering them
- Is equivalent to sequence diagram in modeling power

# Example – collaboration diag



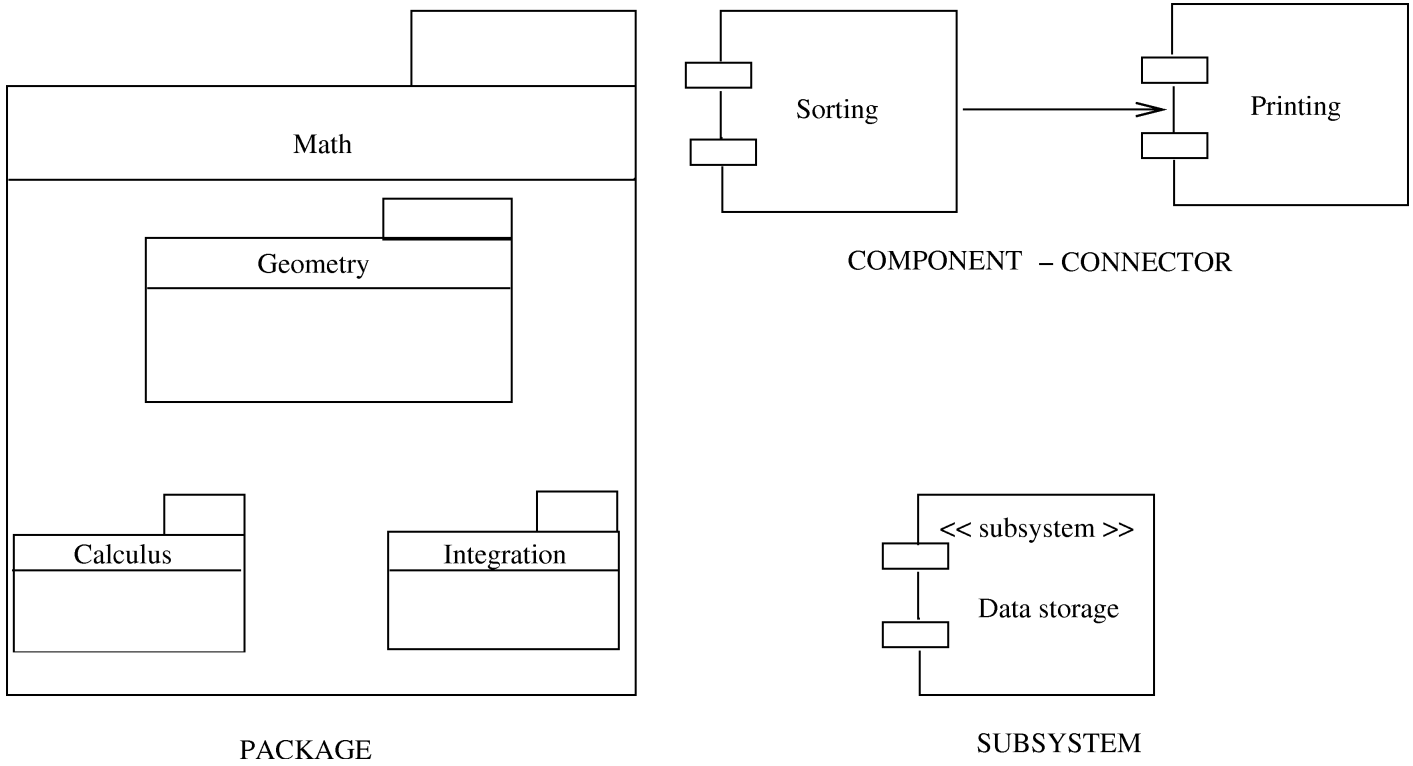
# Other Diagrams

- Class diagram and interaction diagrams most commonly used during design
- There are other diagrams used to build different types of models

# Other Diagrams

- Instead of objects/classes, can represent components, packages, subsystems
- These are useful for developing architecture structures
- UML is extensible – can model a new but similar concept by using stereotypes (by adding <<name>>)
- Tagged values can be used to specify additional properties, e.g. private, readonly..
- Notes can be added

# Other symbols



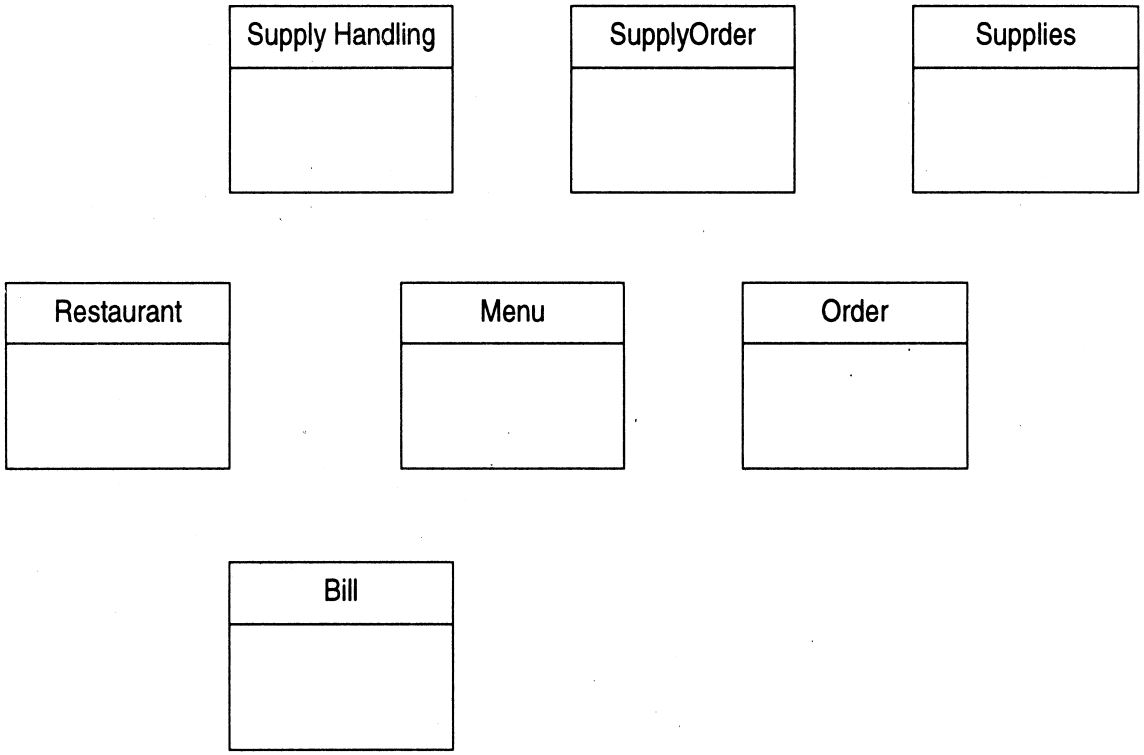
# Design using UML

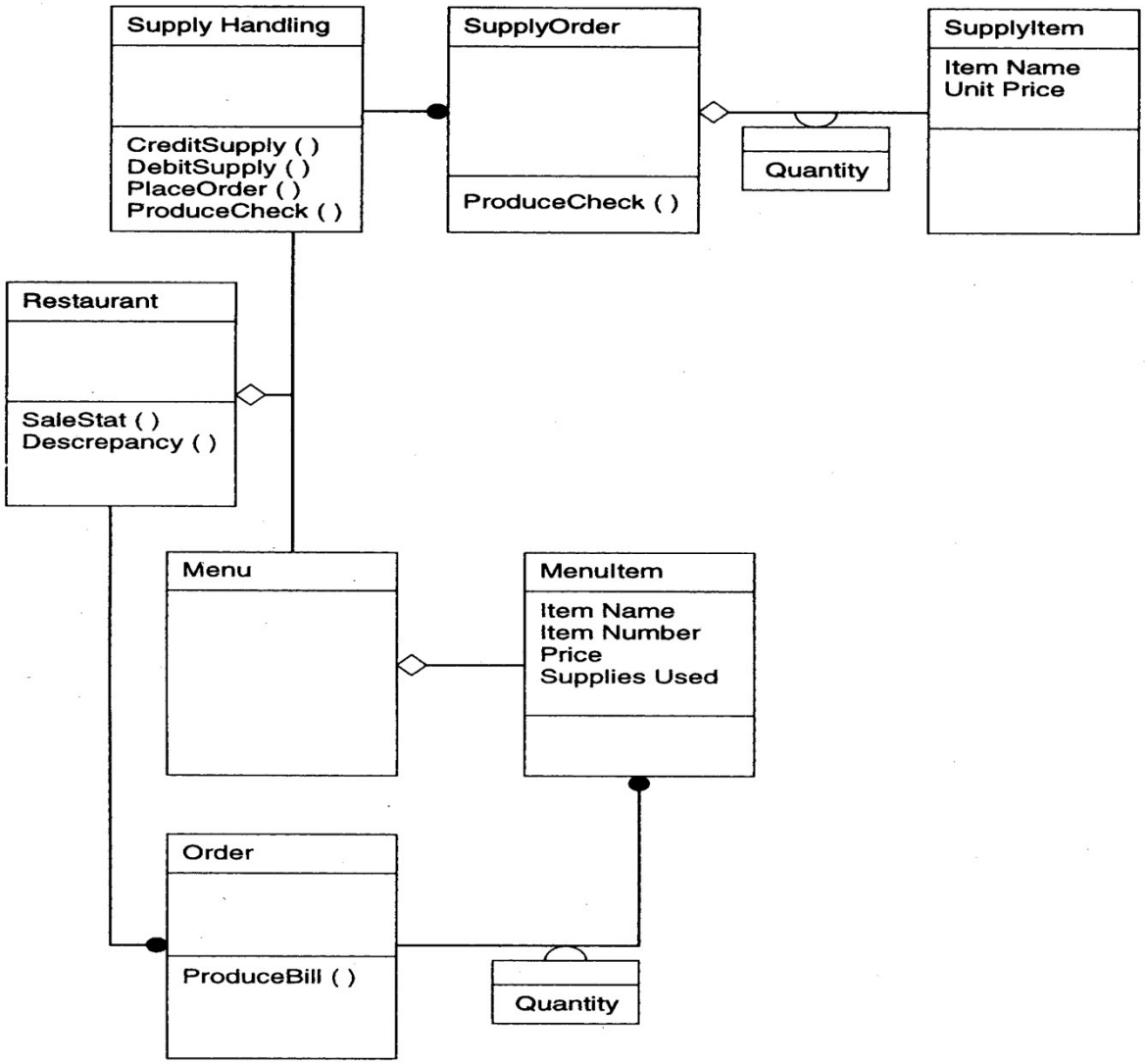
- Many OOAD methodologies have been proposed
- They provide some guidelines on the steps to be performed
- Basic goal is to identify classes, understand their behavior, and relationships
- Different UML models are used for this
- Often UML is used, methodologies are not followed strictly

# Design using UML

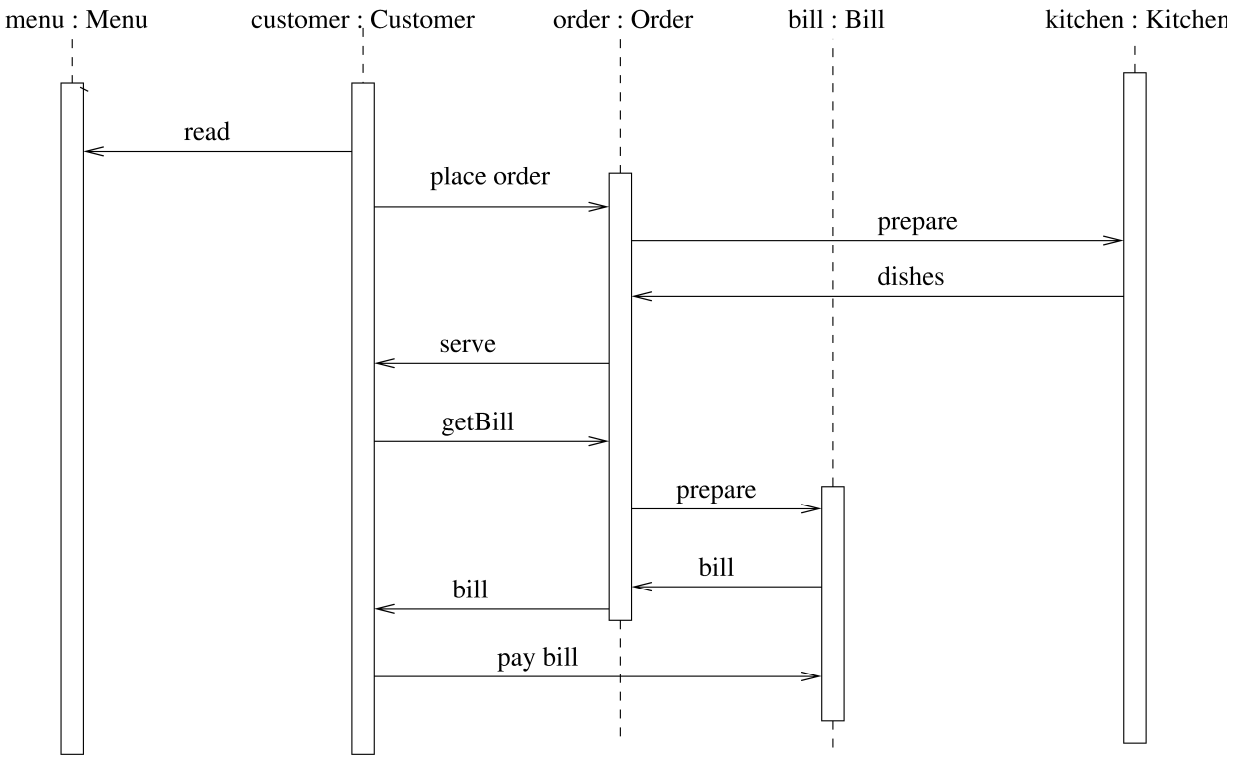
- Basic steps
  - Identify classes, attributes, and operations from use cases
  - Define relationships between classes
  - Make dynamic models for key use cases and use them to refine class diagrams
  - Make a functional model and use it to refine the classes
  - Optimize and package
- Class diagrams play the central role; class defn gets refined as we proceed

# Restaurant example: Initial classes




Supply Handling
CreditSupply ( )
DebitSupply ( )
PlaceOrder ( )
ProduceCheck ( )
SupplyOrder
ProduceCheck ( )
Quantity
SupplyItem
Item Name
Unit Price
Restaurant
SaleStat ( )
Descrepancy ( )
Menu
MenuItem
Item Name
Item Number
Price
Supplies Used
Order
ProduceBill ( )
Quantity

# Restaurant example: a seq diag



# Detailed Design

- HLD does not specify module logic; this is done during detailed design
- One way to communicate the logic design: use natural language
- Is imprecise and can lead to misunderstanding
- Other extreme is to use a formal language
- Generally a semi-formal language is used – has formal outer structures but informal inside

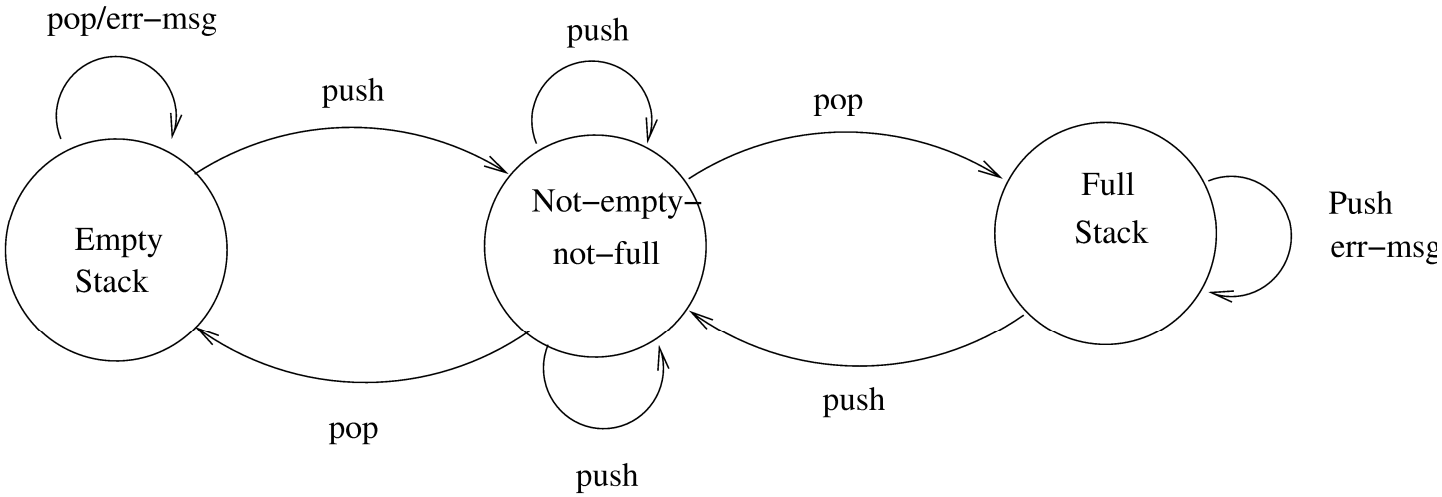
# Logic/Algorithm Design

- Once the functional module (function or methods in a class) are specified, the algo to implement it is designed
- Various techniques possible for designing algorithm – in algos course
- Stepwise refinements technique is useful here

# State Modeling of Classes

- Dynamic model to represent behavior of an individual object or a system
- Shows the states of an object and transitions between them
- Helps understand the object – focus only on the important logical states
- State diagrams can be very useful for automated and systematic testing

# State diagram of a stack



# Design Verification

- Main objective: does the design implement the requirements
- Analysis for performance, efficiency, etc may also be done
- If formal languages used for design representation, tools can help
- Design reviews remain the most common approach for verification

# Metrics

# Background

- Basic purpose to provide a quantitative evaluation of the design (so the final product can be better)
- Size is always a metric – after design it can be more accurately estimated
  - Number of modules and estimated size of each is one approach
- Complexity is another metric of interest – will discuss a few metrics

# Network Metrics

- Focus on structure chart; a good SC is considered as one with each module having one caller (reduces coupling)
- The more the SC deviates from a tree, the more impure it is

  *Graph impurity = n – e – 1*

  *n – nodes, e- edges* in the graph
- Impurity of 0 means tree; as this no increases, the impurity increases

# Stability Metrics

- Stability tries to capture the impact of a change on the design
- Higher the stability, the better it is
- Stability of a module – the number of assumptions made by other modules about this module
  - Depends on module interface, global data the module uses
  - Are known after design

# Information Flow Metrics

- Complexity of a module is viewed as depending on intra-module complexity
- Intramodule estimated by module size and the information flowing
  - Size in LOC
  - Inflow – info flowing in the module
  - Outflow – info flowing out of the module
- $Dc = size * (inflow * outflow)^2$

# Information flow metrics…

- (inflow * outflow) represents total combination of inputs and outputs
- Its square reps interconnection between the modules
- Size represents the internal complexity of the module
- Product represents the total complexity

# Identifying error-prone modules

- Uses avg complexity of modules and std dev to identify error prone and complex modules:

  Error prone: If Dc > avg complexity + std_dev

  Complex: If avg complexity < Dc < avg + std dev

  Normal: Otherwise

# Complexity metrics for OO

- Weighted methods per class
  - Complexity of a class depends on no of classes and their complexity
  - Suppose complexity of methods is c1, c2..; by some functional complexity metric

    $WMC = \Sigma ci$
  - Large WMC might mean that the class is more fault-prone

# OO Metrics…

- Depth of Inheritance Tree
  - DIT of C is depth from the root class
  - Length of the path from root to C
  - DIT is significant in predicting fault proneness
- Number of Children
  - Immediate no of subclasses of C
  - Gives a sense of reuse

# OO Metrics…

- Coupling between classes
  - No of classes to which this class is coupled
  - Two classes are coupled if methods of one use methods or attr of other
  - Can be determined from code
  - (There are indirect forms of coupling that cannot be statically determined)

# Metrics…

- Response for a class
  - The total no of methods that can be invoked from this class
  - Captures the strength of connections
- Lack of cohesion in methods
  - Two methods form a cohesive pair if they access some common vars (form a non-cohesive pair if no common var)
  - LCOM is the number of method pairs that are non-cohesive – the no of cohesive pairs

# Metrics with detailed design

- When logic is known, internal complexity metrics can be determined
- We will cover all detailed design based metrics along with code metrics

# Summary

- Design for a system is a plan for a solution – want correct and modular
- Cohesion and coupling key concepts to ensure modularity
- Structure charts and structured design methodology can be used for function-oriented design
- UML can be used for OO design
- Various complexity metrics exist to evaluate a design complexity