Unit 3 Design

Software Design

- Activity begins with a set of requirements, and maybe an architecture
- Design done before the system is implemented
- Design focuses on module view – i.e. what modules should be in the system
- Design of a system- blue print for implementation
- Has two levels – high level (modules are defined), and detailed design (logic specified)

Design

- Goal: to produce correct designs
- Most critical activity during system development
- Design determines the major characteristics of a system
- Has great impact on testing and maintenance
- Design document forms reference for later phases
- Design methodology – systematic approach for creating a design

Design Concepts

- Design is correct, if it satisfy all requirements and is consistent with architecture
- Of the correct designs, want best design
- Focus on modularity as the main criteria (besides correctness)

Design Modularity

- Modular system – in which modules can be built separately and changes in one have minimum impact on others
- Supports independence of modules
- Enhances design clarity, eases implementation
- Reduces cost of testing, debugging and maintenance
- Cannot simply chop a program into modules to get modularly
- Need some criteria for decomposition – coupling and cohesion are such criteria

Coupling

- Independent modules: if one can function completely without the presence of other
- Independence between modules is desirable
  - Modules can be modified separately
  - Can be implemented and tested separately
  - Programming cost decreases
- All modules cannot be independent, must cooperate with each other
- More connections between modules
  - More dependent they are
  - More knowledge about one module is required to understand the other module
- Coupling captures the notion of dependence

Coupling

- Coupling between modules is the strength of interconnections between modules
- In general, the more we must know about module A in order to understand module B, the more closely connected is A to B.
- "Highly coupled" modules are joined by strong interconnection
- "Loosely coupled" modules have weak interconnections

Coupling

- Goal: modules as loosely coupled as possible
- Where possible, have independent modules
- Coupling is decided during high level design
- Cannot be reduced during implementation
- Major factors influencing coupling
  - Type of connection between modules
  - Complexity of the interface
  - Type of information flow between modules

Coupling – Type of connection

- Complexity and obscurity of interfaces increase coupling
- Minimize the number of interfaces per module
- Minimize the complexity of each interface
- Coupling is minimized if
  - Only defined entry interface of a module is used by other modules
  - E.g. Information is passed exclusively through parameters
- Coupling increases if
  - Indirect and obscure interface are used
  - Internals of a module are directly used
  - Shared variables employed for communication

Coupling – Interface complexity

- Coupling increases with complexity of interfaces eg. number and complexity of parameters
- Some level of complexity of interfaces needed to support required communication
- Often more than needed is used eg. passing entire record when only a field is needed
- Keep the interface of a module as simple as possible

Coupling – Type of Information flow

- Coupling depends on type of information flow along the interfaces
- Two kinds of information: data or control
- Transfer of control information
    - Action of module depends on the information
    - Makes modules more difficult to understand
- Transfer of data information
    - Module can be treated as input-output function
- Lowest coupling: interfaces with only data communication
- Highest: hybrid interfaces

Coupling - Summary

Coupling	Interface complexity	Type of connections	Type of communication
Low	Simple obvious	To module by name	Data
			Control
High	Complicated obscure	To internal elements	Hybrid

Coupling in Object-Oriented Systems

- In OO systems, basic modules are classes, which are richer than functions
- OO Systems have three types of coupling:
  - Interaction coupling
  - Component coupling
  - Inheritance coupling

Coupling in OO - Interaction

- Interaction coupling occurs due to methods of a class invoking methods of other classes
  - Like calling of functions
  - Worst form- methods directly access internal parts of other methods
  - Still bad if methods directly manipulate variables of other classes
  - Passing information through temporary variables is also bad
- Least interaction coupling if methods communicate directly with parameters
  - With least number of parameters
  - With least amount of information being passed
  - With only data being passed
- I.e. methods should pass the least amount of data, with least number of parameters

Coupling in OO - Component

- Component coupling – when a class A has variables of another class C
  - A has instance variables of C
  - A has a method with some parameters of type C
  - A has a method with a local variable of type C
- When A is coupled with C, it is coupled with all subclasses of C as well
- Component coupling will generally imply the presence of interaction coupling also

Coupling in OO - Inheritance

- Inheritance coupling – two classes are coupled if one is a subclass of other
- Worst form – when subclass modifies a signature of a method or deletes a method
- Coupling is bad even when same signature but a changed implementation
- Least, when subclass only adds instance variables and methods but does not modify any inherited ones

Cohesion

- Coupling reduced by minimizing relationship between elements of different modules
- Another method by maximizing relationship between elements of same module
- Cohesion considers this relationship
- Interested in determining how closely the elements of a module are related to each other

Cohesion

- Cohesion of a module represents how tightly bound the internal elements of the module are to one another
- Gives an idea about whether the different elements of a module belong together in the same module
- Cohesion and coupling are interrelated
- Greater the cohesion of each module, lower the coupling between modules

Levels of Cohesion

- There are many levels of cohesion
  - Coincidental-

    No meaningful relationship among the elements
  - Logical-

    Logical relationship between the elements, elements perform functions that are in same class (e.g. input, output module)
  - Temporal-

    As logical, except that elements are related in time and are executed together (e.g. initialization, termination module)
  - Procedural-

    Elements belong to common procedural unit (e.g. loop, decision)

Levels of Cohesion

- Communicational-

  Elements are together, as they operate on same input or output data (e.g. print record)

- Sequential-

  Elements are together, as output of one forms input of another

- Functional-

  Elements are related to perform single function (e.g. sort array)

- Coincidental is lowest, functional is highest
- Functional is considered very strong

Cohesion in OO Systems

- In OO, different types of cohesion, as classes are the modules
  - Method cohesion
  - Class cohesion
  - Inheritance cohesion

- Method cohesion –
  - Focuses on why different code elements are together in a method
  - Highest form is if each method implements a clearly defined function with all elements contributing to implementing this function

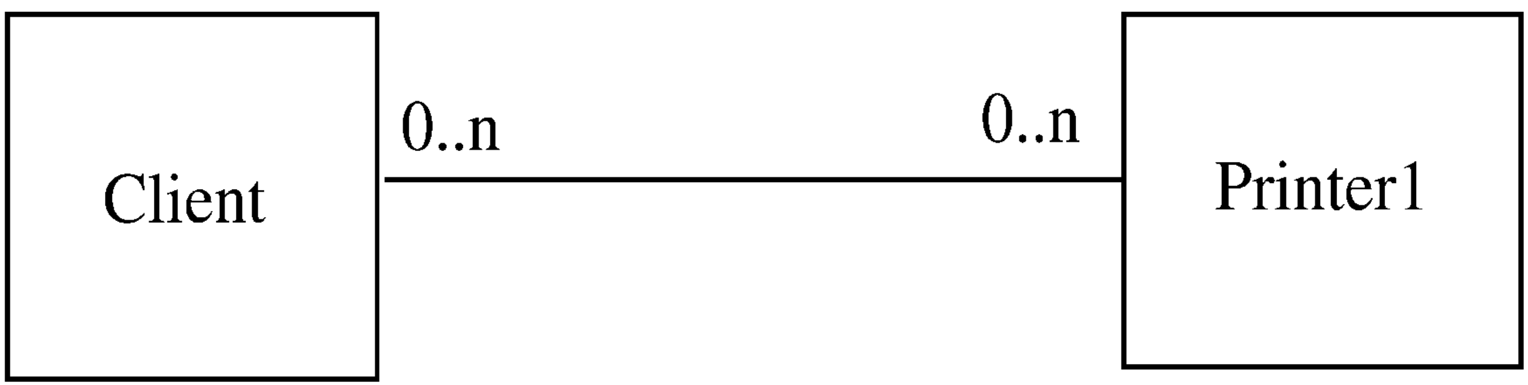
Cohesion in OO Systems

- Class cohesion –
  - Focuses on why different attributes and methods are together in a class
  - A class should implement a single concept with all elements contributing towards it
  - Whenever multiple concepts encapsulated in a class, cohesion is not as high
  - A symptom of multiple concepts – different groups of methods accessing different subset of attributes
- Inheritance cohesion –
  - Focuses on why classes are together in a hierarchy
  - Two reasons for inheritance– generalization-specialization and code reuse
  - Cohesion is higher if the hierarchy is for providing generalization-specialization

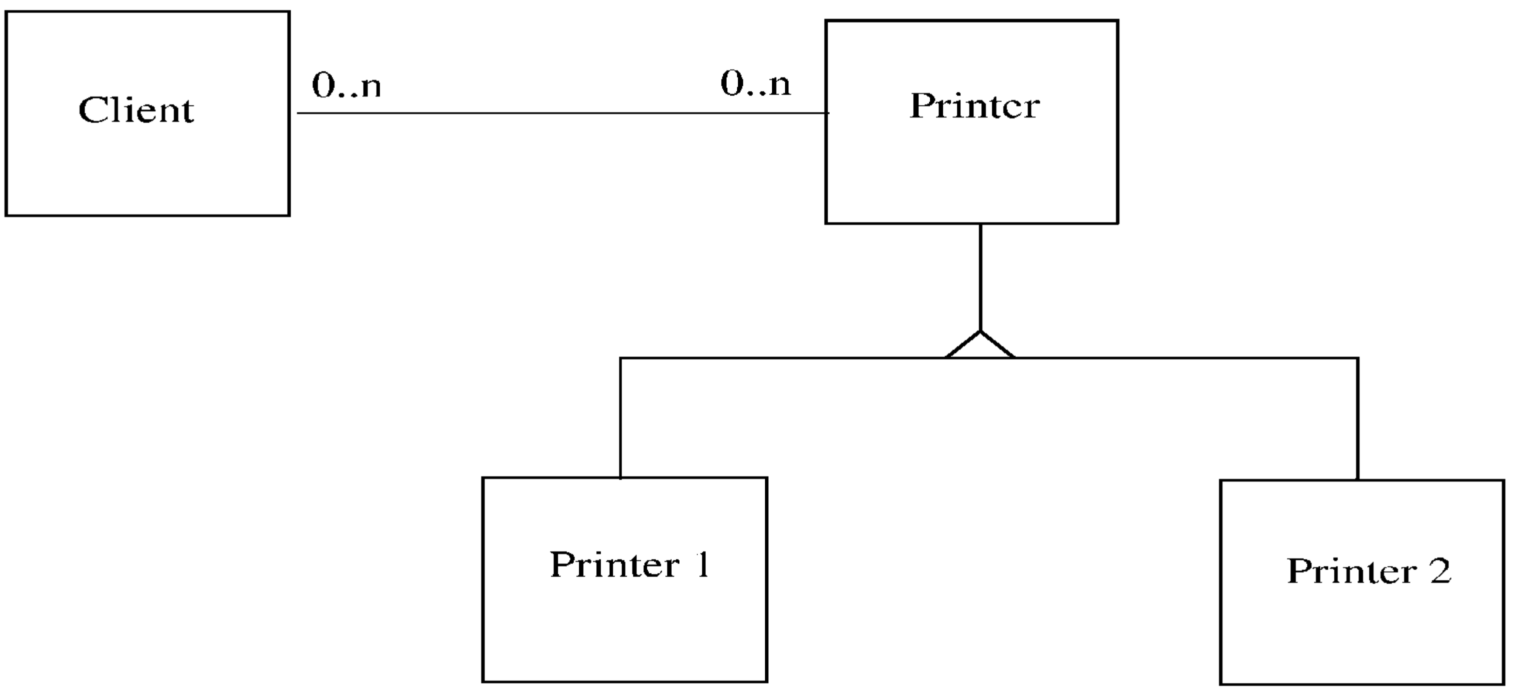
Open-closed Principle

- Besides cohesion and coupling, open closed principle also helps in achieving modularity
- Principle: A module should be open for extension but closed for modification
  - Behavior can be extended to accommodate new requirements, but existing code is not modified
  - I.e. allows addition of code, but not modification of existing code
  - Minimizes risk of having existing functionality stop working due to changes – a very important consideration while changing code
  - Good for programmers as they like writing new code

- In OO this principle is satisfied by using inheritance and polymorphism
- Inheritance allows creating a new class to extend behavior without changing the original class
- This can be used to support the open-closed principle
- Consider example of a client object which interacts with a printer object for printing



- Client directly calls methods on Printer1
- If another printer is to be allowed
  - A new class Printer2 will be created
  - But the client will have to be changed if it wants to use Printer 2
- Alternative approach
  - Have Printer1 a subclass of a general Printer
  - For modification, add another subclass Printer 2
  - Client does not need to be changed



Liskov's Substitution Principle

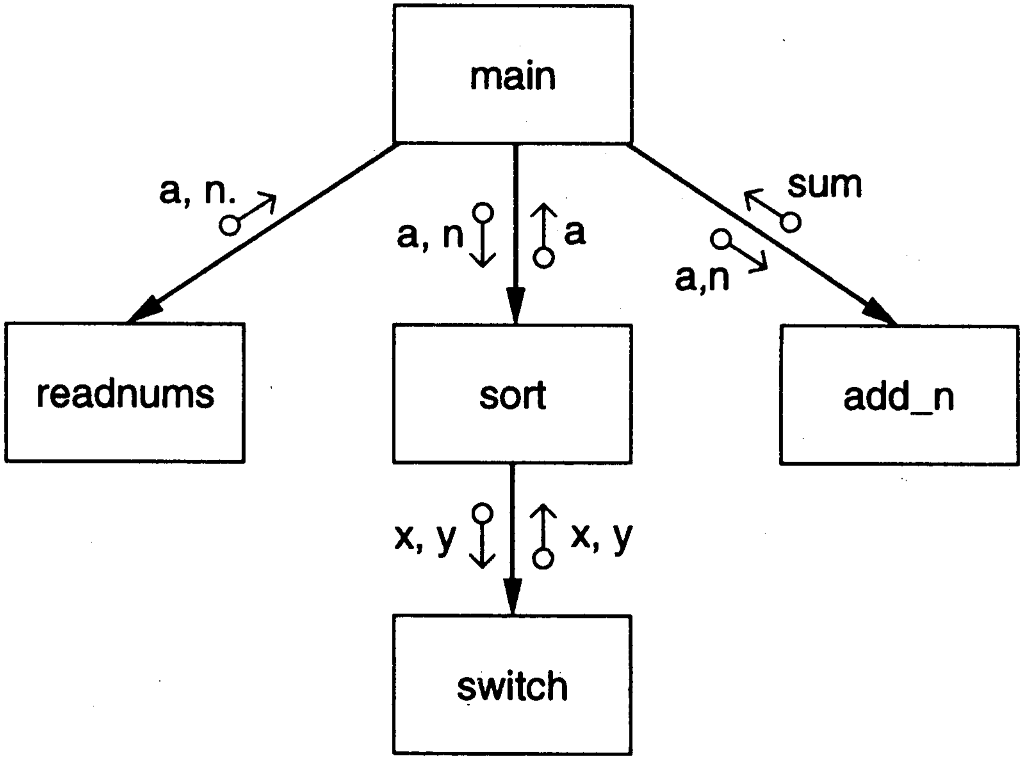
- Principle: Program using object o1 of base class C should remain unchanged if o1 is replaced by an object o2 of a subclass of C
- If hierarchies follow this principle, the open-closed principle gets supported

Function Oriented Design and Structured Design Methodology

Program Structure and Structure Charts

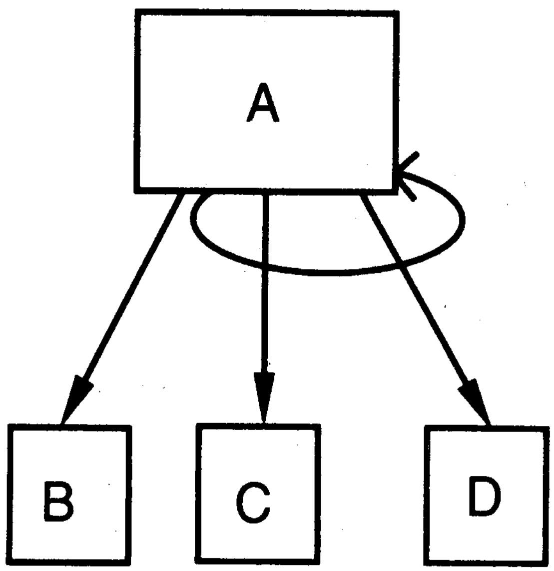
- Every program has a structure
- Structure Chart - graphic representation of structure
- SC represents modules and interconnections
- Each module is represented by a box
- If A invokes B, an arrow is drawn from A to B
- Arrows are labeled by data items
- Different types of modules in a SC
- Input, output, transform and coordinate modules
- A module may be a composite

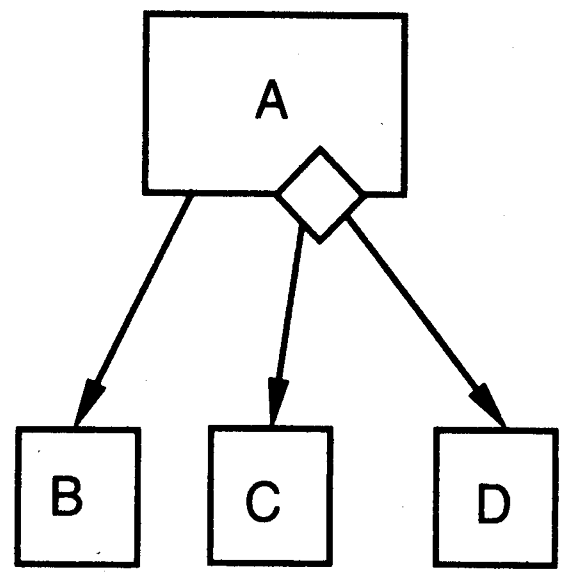
Structure Chart of a Sort Program



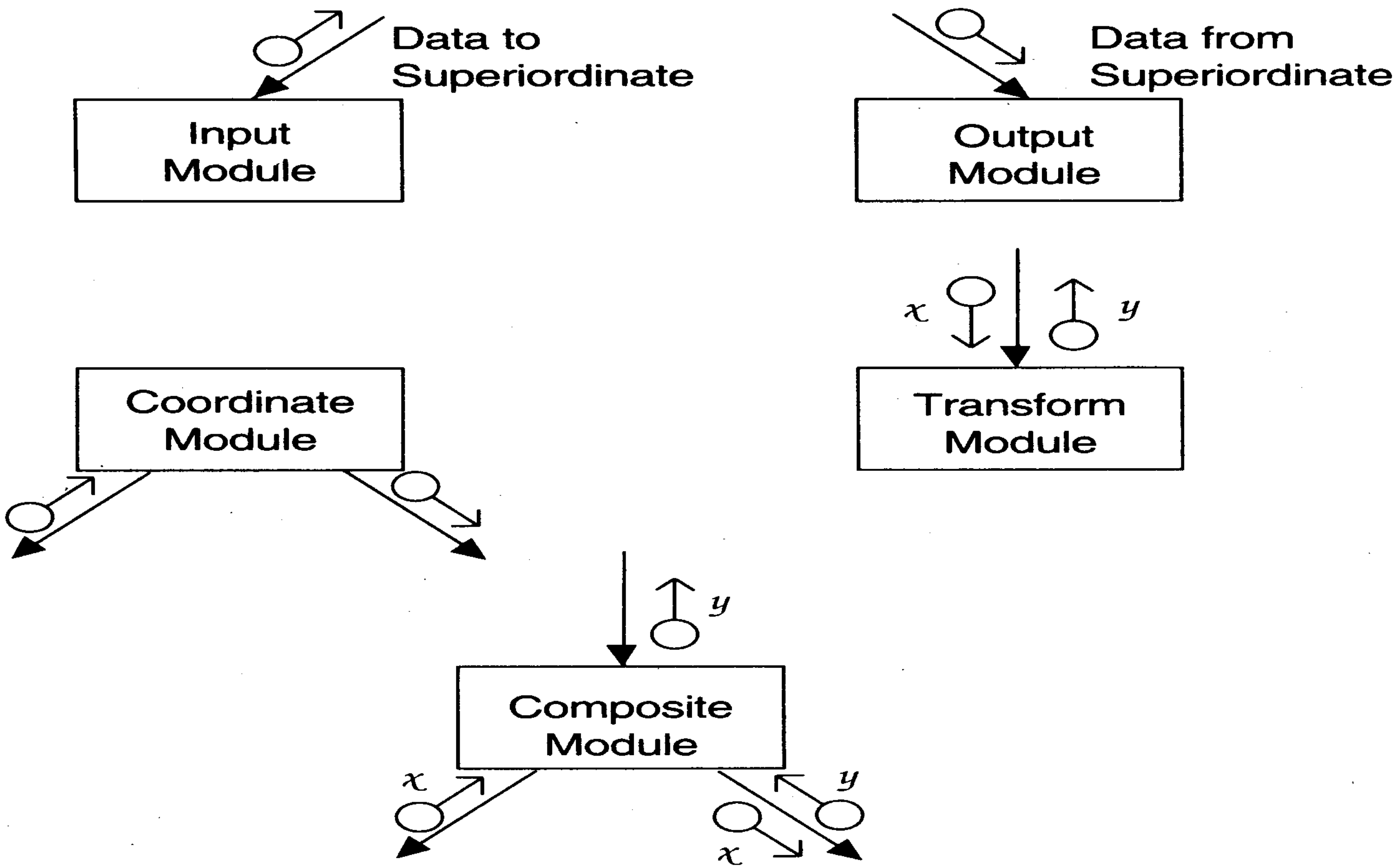
Iteration and decision representation

- Procedural information not represented, focus on hierarchy of modules representation





Different types of modules



Structured Design Methodology

- SDM views software as a transformation function that converts given inputs to desired outputs
- The focus of SD is the transformation function
- Uses functional abstraction
- Goal of SDM: Specify functional modules and connections
- Low coupling and high cohesion is the objective

Steps in Structured Design Methodology:

1. Draw a DFD of the system
2. Identify most abstract inputs and most abstract outputs
3. First level factoring
4. Factoring of input, output, transform modules
5. Improving design

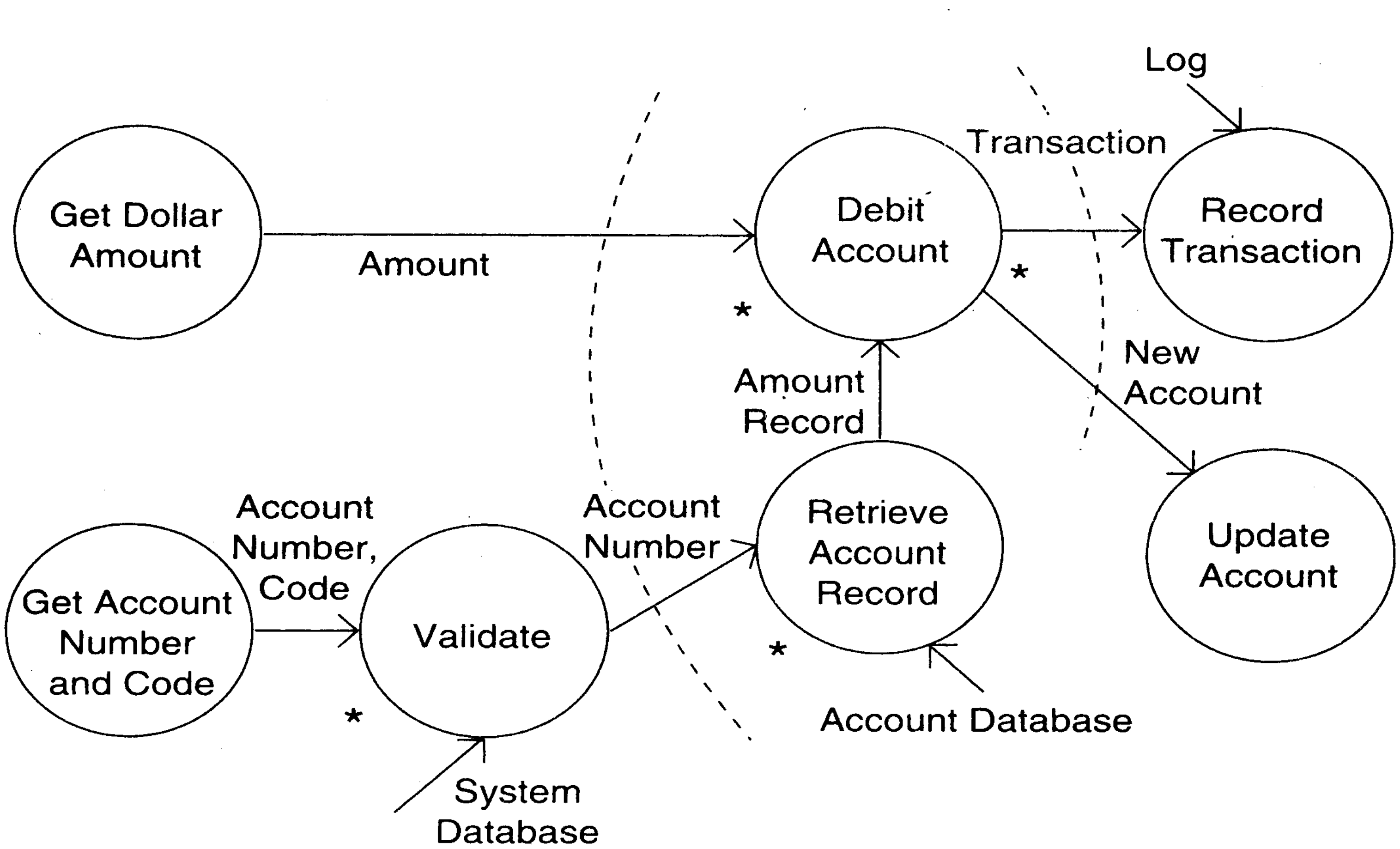
Step1. Data Flow Diagrams

- SD starts with a DFD to capture flow of data in the proposed system
- DFD is an important representation; provides a high level view of the system
- Emphasizes the flow of data through the system
- Ignores procedural aspects
- (Purpose here is different from DFDs used in requirements analysis, thought notation is the same)

Drawing a Data Flow Diagram(dfg)

- Start with identifying the inputs and outputs
- Work your way from inputs to outputs, or vice versa
  - If stuck, reverse direction
  - Ask: "What transformations will convert the inputs to outputs"
- Never try to show control logic.
  - If thinking about loops, if-then-else, start again
- Label each arrow carefully
- Make use of * and +, and show sufficient detail
- Ignore minor functions in the start
- For complex systems, make dfg hierarchical
- Never settle for the 1st dfg

Example – DFD of an ATM



Step2. Identify most abstract inputs and most abstract outputs

- Most systems perform a basic function
- Functions cannot be performed on inputs directly
- First inputs converted into a suitable form
- Similarly for outputs
- Many transforms needed for processing inputs and outputs
- Goal of step 2 is to separate such transforms from the basic transform centers

Step 2…

- Most abstract inputs: data elements in dfg that are furthest from the actual inputs, but can still be considered as incoming
- These are logical data items for the transformation
- May have little similarity with actual inputs.
- Often data items obtained after error checking, formatting, data validation, conversion etc.

Step 2…

- Abstract i/p- Travel from physical inputs towards outputs until data can no longer be considered incoming
- Go as far as possible, without loosing the incoming nature
- abstract outputs – travel in reverse manner
- Represents a value judgment, but choice is often obvious
- Bubbles between mai and mao: central transforms
- These transforms perform the basic transformation
- With mai and mao the central transforms can concentrate on the transformation

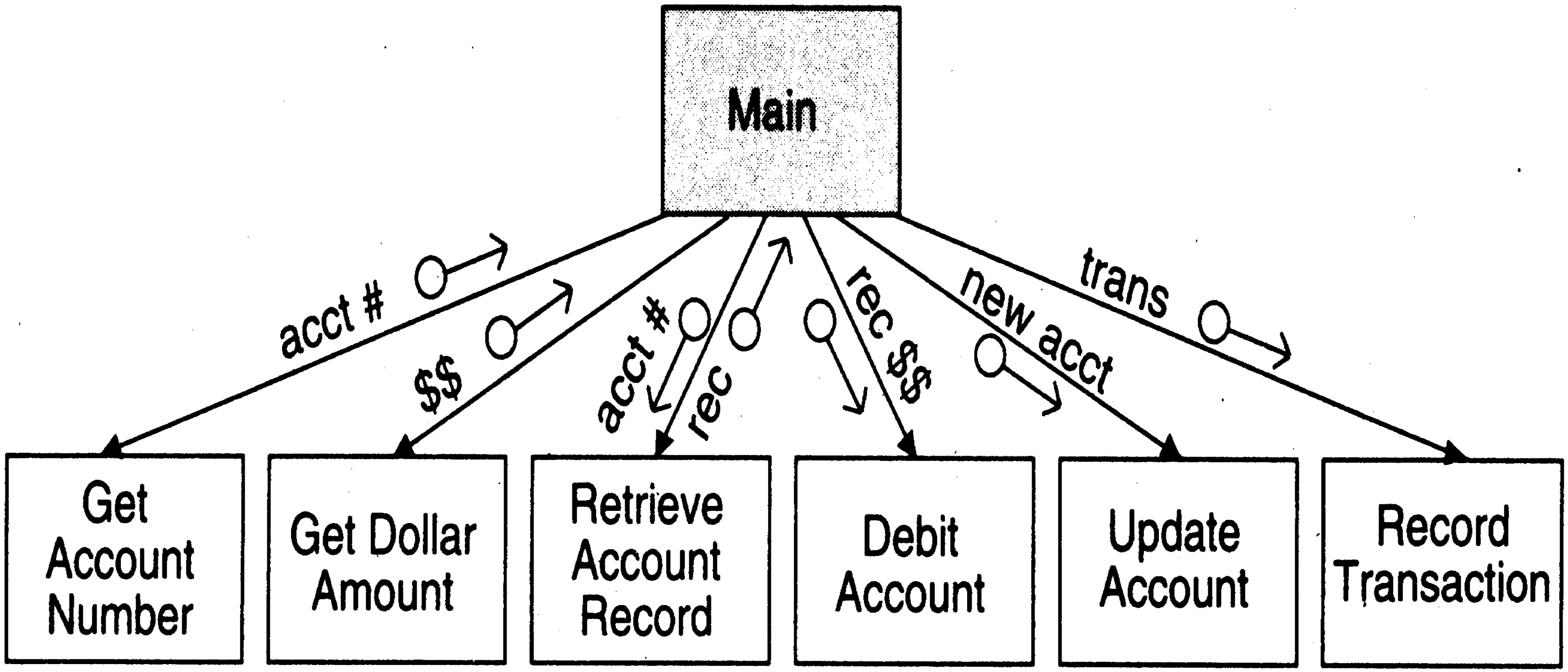
Step 2…

- Problem View: Each system does some i/o and some processing
- In many systems the i/o processing forms the large part of the code
- This approach separates the different functions
  - subsystem primarily performing input
  - subsystem primarily performing transformations
  - subsystem primarily performing output presentation

Step3. First Level Factoring

- First step towards a structure chart
- Specify a main module
- For each most abstract input data item, specify a subordinate input module
- The purpose of these input modules is to deliver to main the mai data items
- For each most abstract output data element, specify an output module
- For each central transform, specify a subordinate transform module
- Inputs and outputs of these transform modules are specified in the DFD

Example- First level factoring for ATM

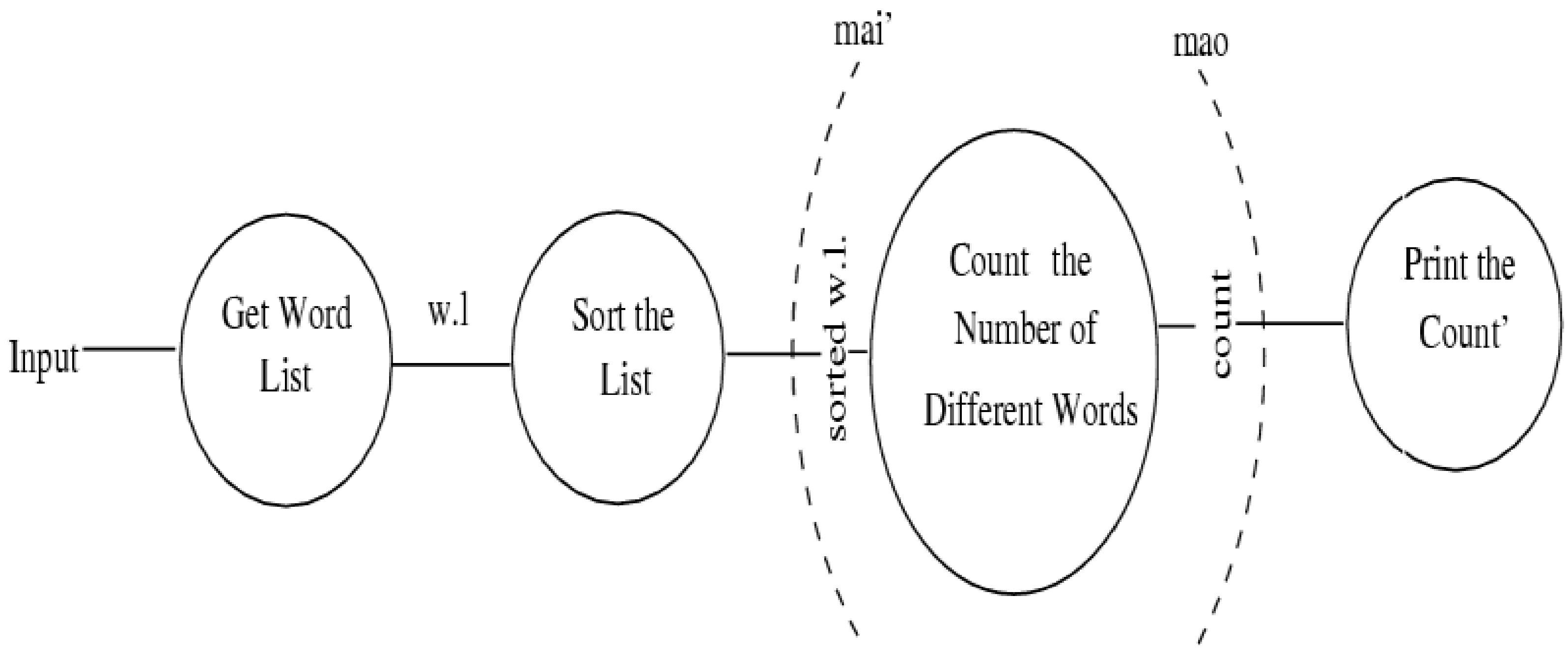


Step4. Factoring of input, output, transform modules

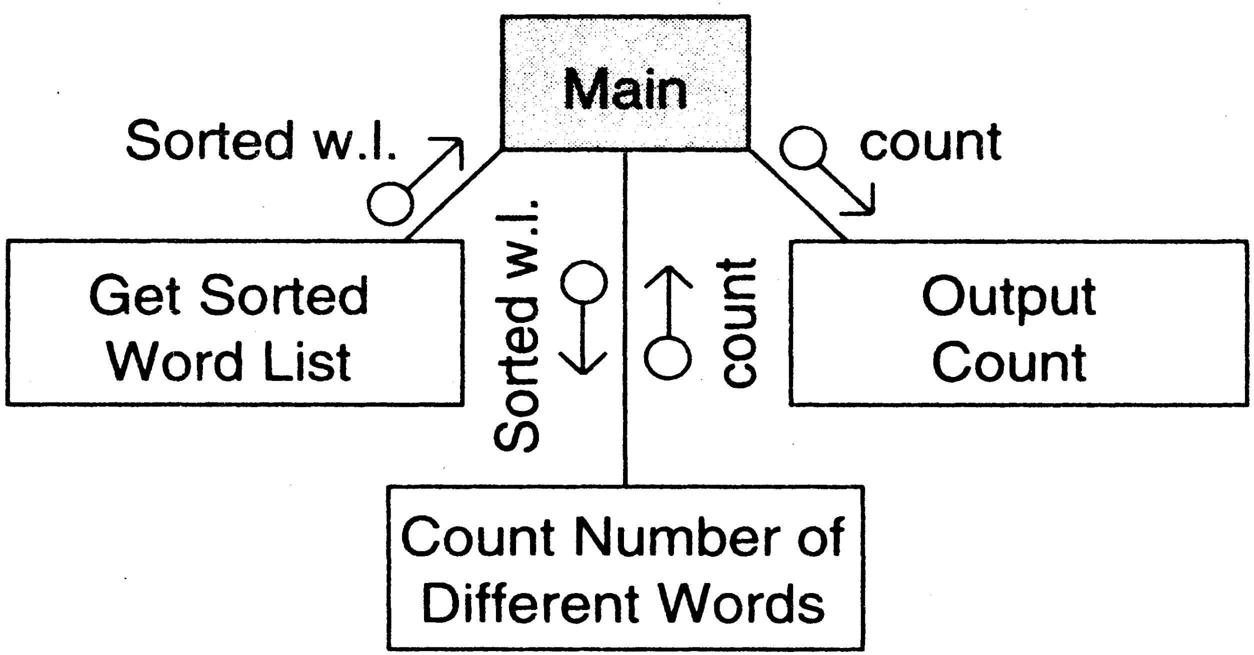
- The transform that produced the mai data is treated as the central transform
- Then repeat the process of first level factoring
- Input module being factored becomes the main module
- A subordinate input module is created for each data item coming in this new central transform
- A subordinate module is created for the new central transform
- The new input modules are factored similarly till the physical inputs are reached
- Factoring of the output modules is symmetrical
- Subordinates - a transform and output modules
- Usually no input modules

Example – DFD for word-counting problem

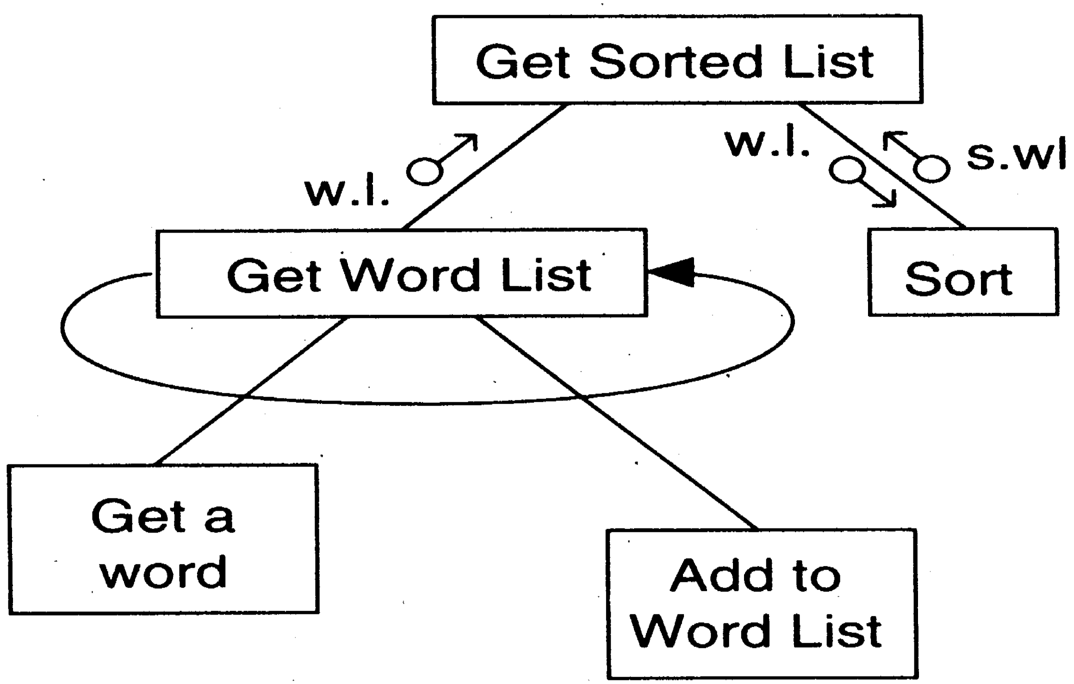
counting the no. of different words in a file



Example – Structure chart after first-level factoring of word-counting problem



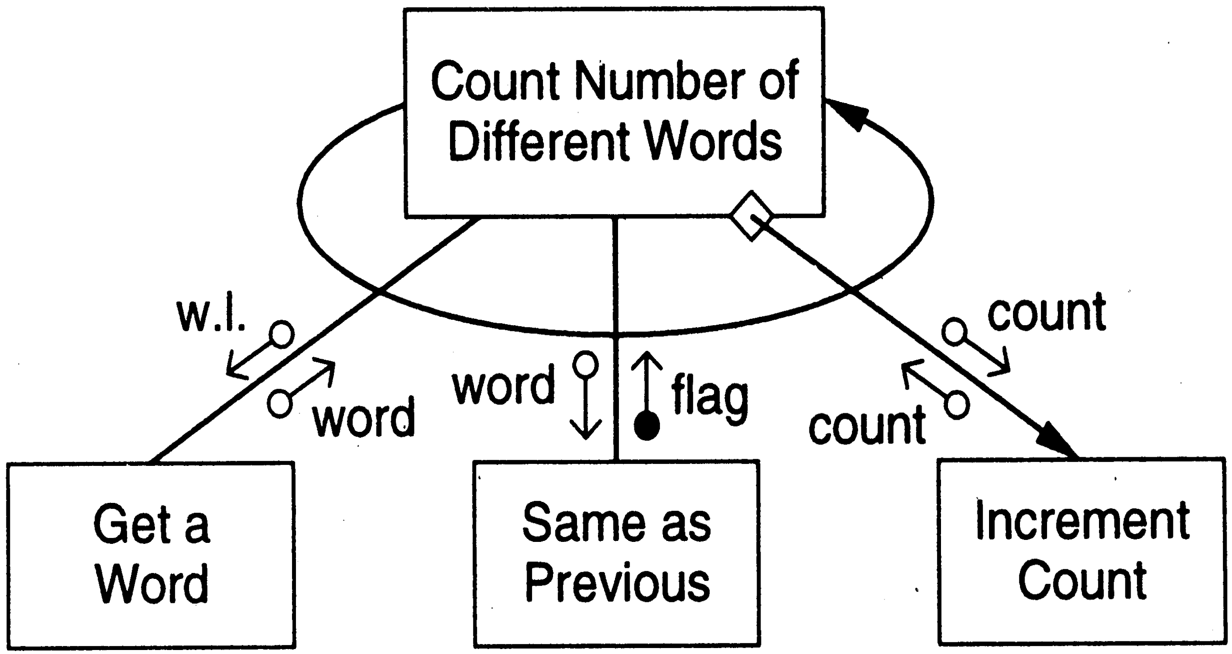
Example – Factoring of the input module get-sorted-list in the first level structure



Factoring Central Transforms

- Factoring i/o modules is straight forward if the DFD is detailed
- No rules for factoring the transform modules
- Top-down refinement process can be used
- Goal: determine sub-transforms that will together compose the transform
- Then repeat the process for newly found transforms
- Treat the transform as a problem in its own right
- Draw a data flow graph
- Then repeat the process of factoring
- Repeat this till atomic modules are reached

Example – Factoring of the central transform count-the-number-of-different-words



Step5. Improving Design through Heuristics

- The above steps should not be followed blindly
- The structure obtained should be modified if needed
- Low coupling, high cohesion being the goal
- Design heuristics used to modify the initial design
- Design heuristics - A set of thumb rules that are generally useful
- Module Size: Indication of module complexity, Carefully examine modules less than a few lines or greater than about 100 lines

Object Oriented Design and UML

OO Concepts

- Information hiding – use encapsulation to restrict external visibility
- OO encapsulates the data, provides limited access, visibility

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:

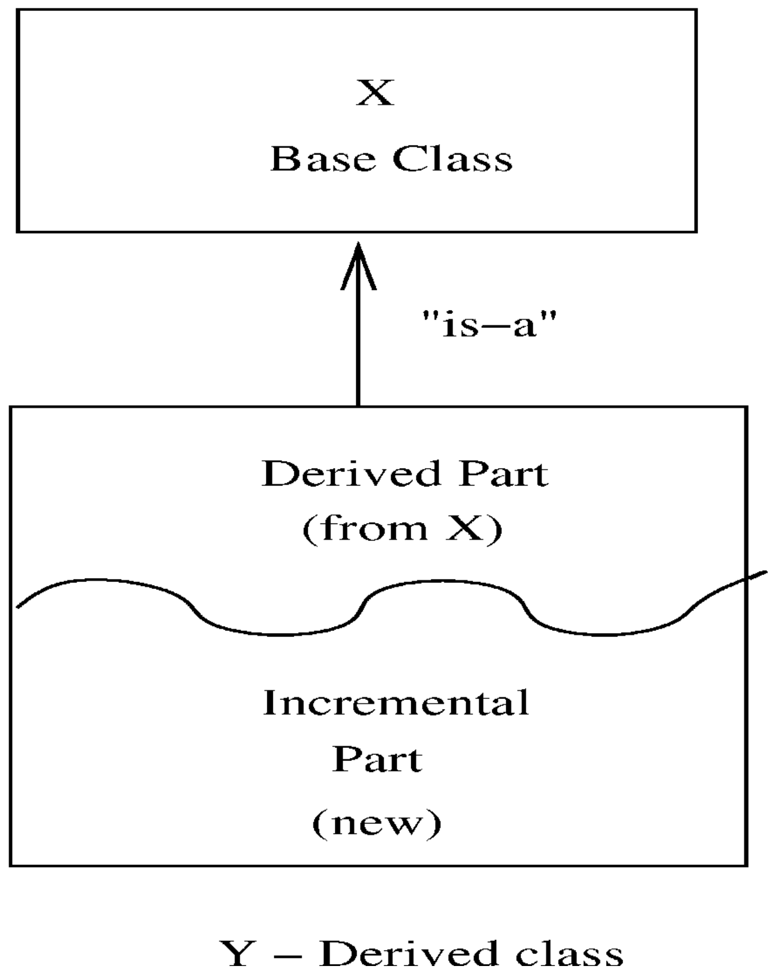
1. Supplier object is global to client
2. Supplier object is a parameter to some op of the client
3. Supplier object is part of the client object
4. Supplier object is locally declared in some operation

- Relationship can be either aggregation (whole-part relationship), or just client server relationship

Inheritance

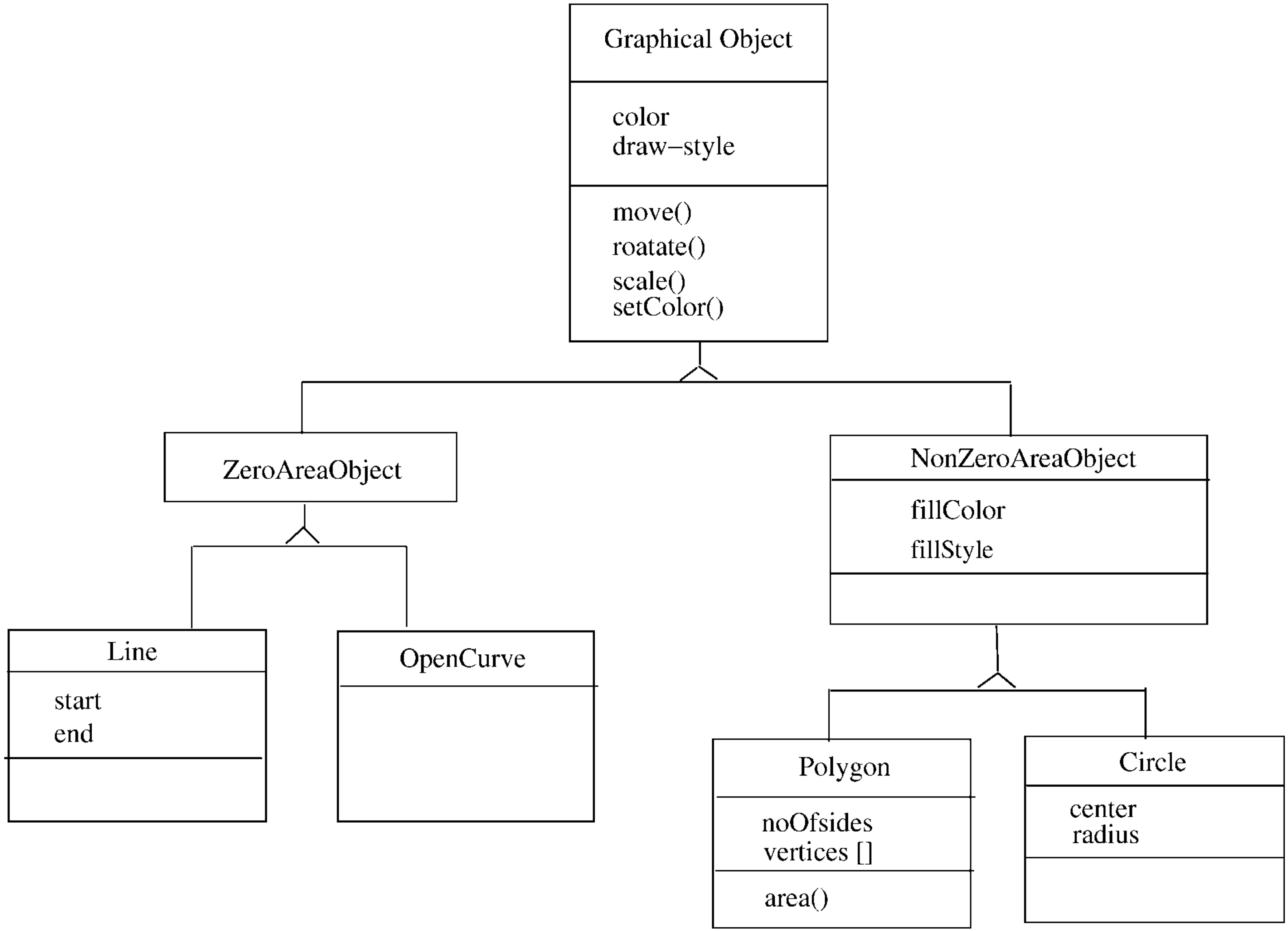
- Inheritance is unique to OO and not there in function-oriented languages/models
- Inheritance by class B from class A is the facility by which B implicitly gets the attributes and ops of A as part of itself
- Attributes and methods of A are reused by B
- When B inherits from A, B is the *subclass*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



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
- 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

Views in an UML

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

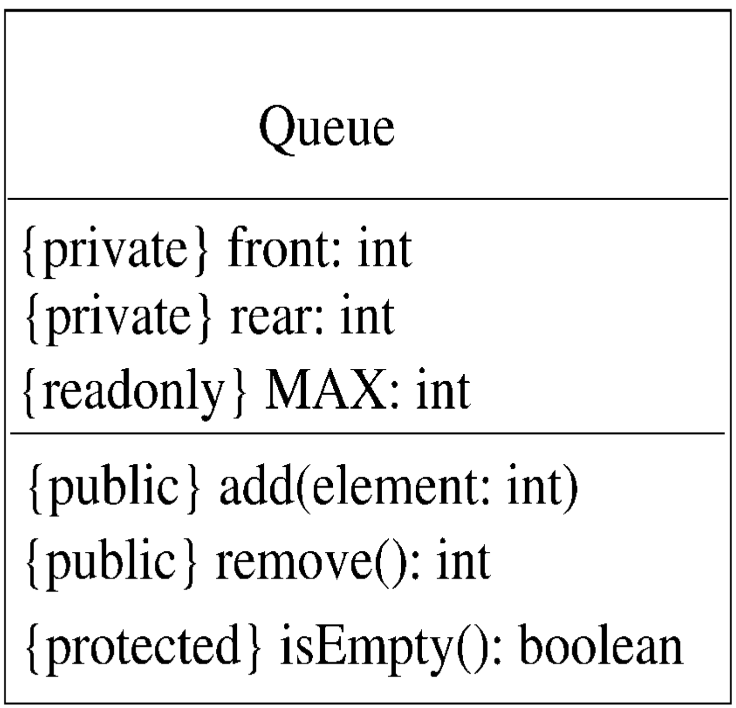
- We will focus primarily on models for design – class diagram, interaction diagram, etc.

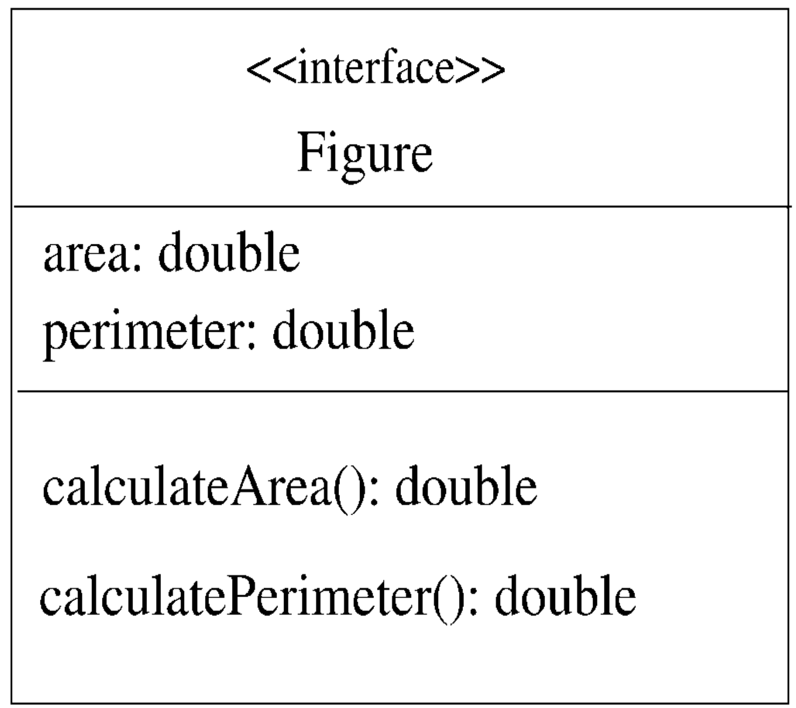
Class Diagrams

- 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…

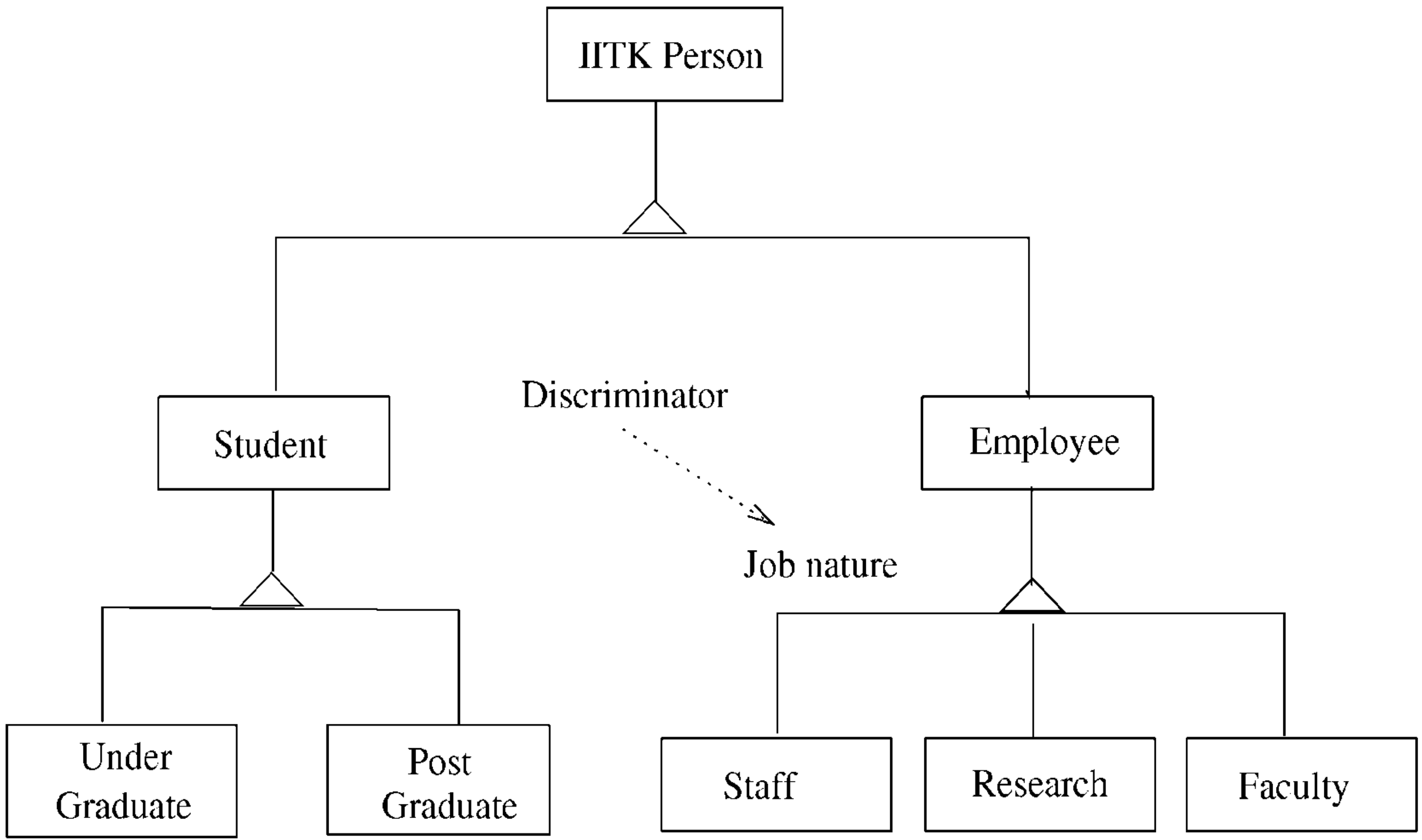
- Class itself represented as a box with name, attributes, and methods
- There are conventions for naming
- If a class is an interface, this can be specified by <<interface>> stereotype
- Properties of attributes/methods can be specified by tags between { }





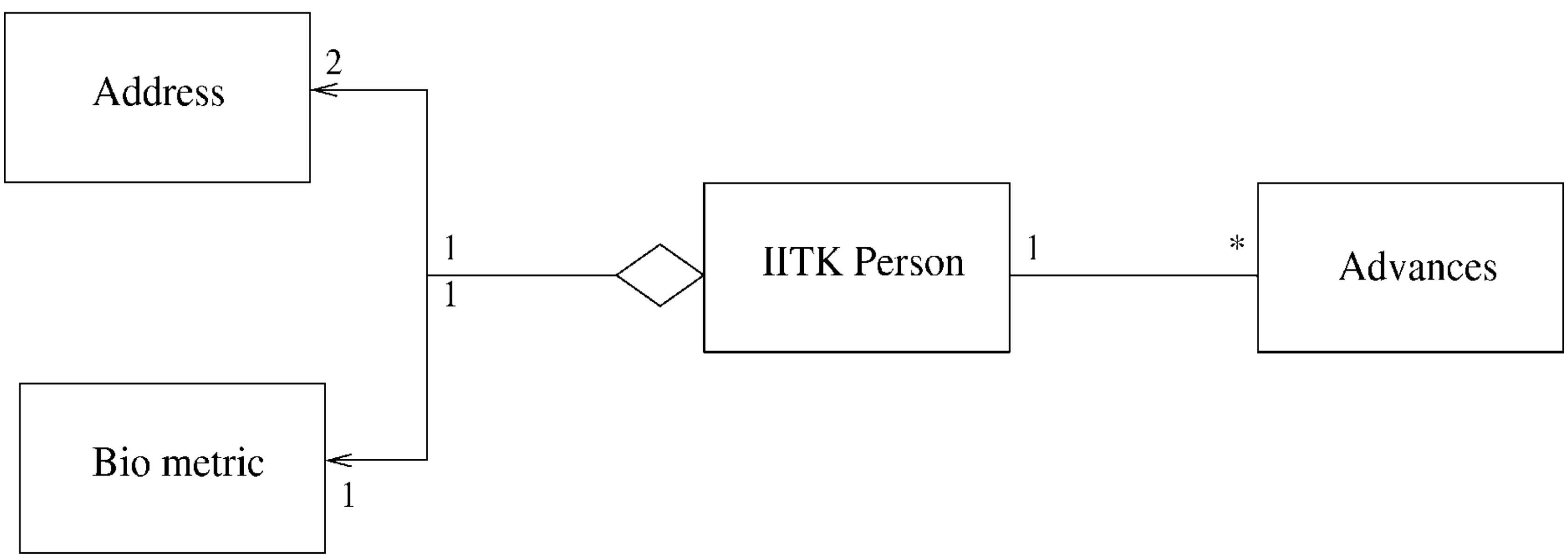
Generalization-Specialization

- 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



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

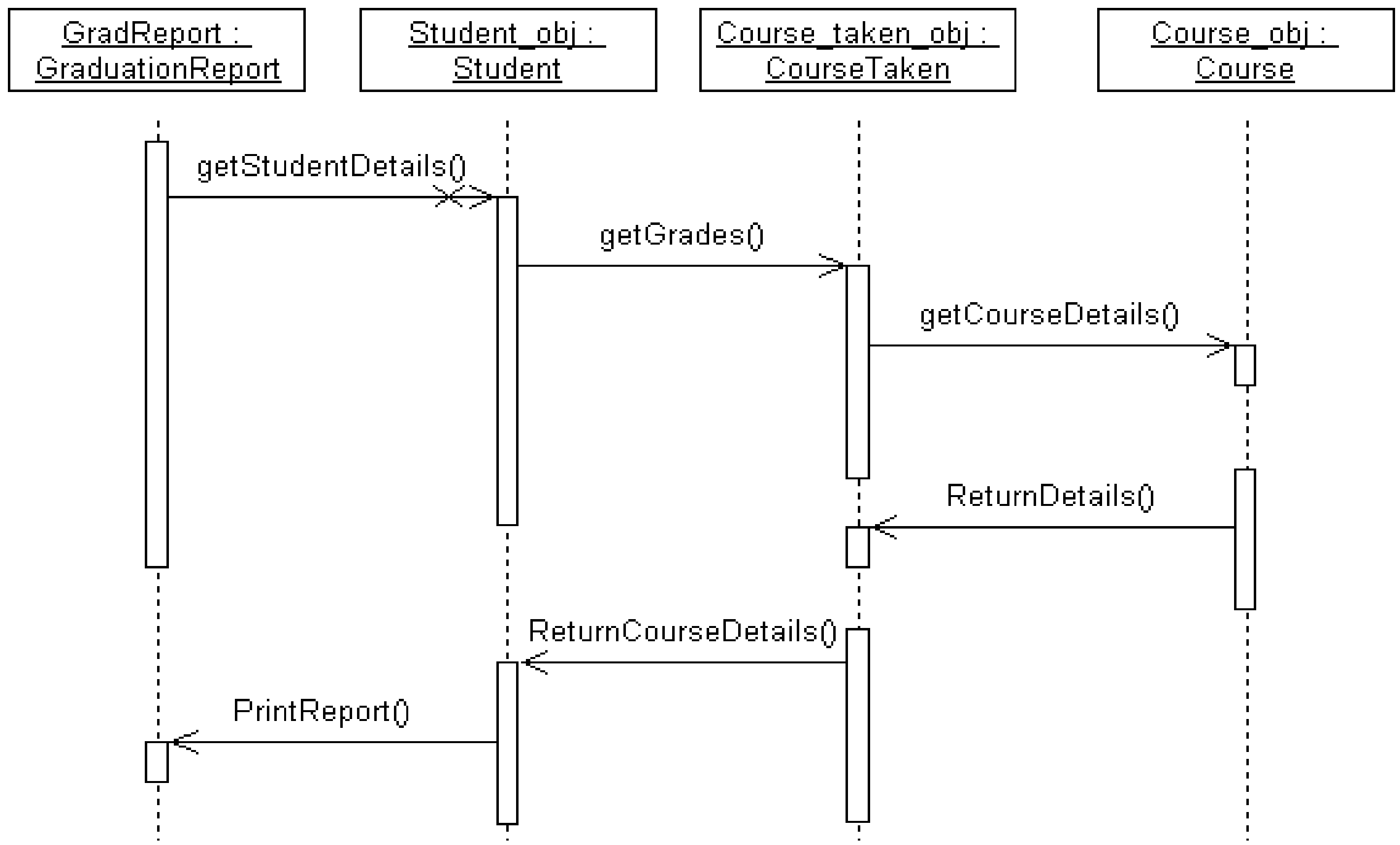


Interaction Diagrams

- Class diagram represent static structure of the system (classes and their relationships)
- 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
  - Sequence diagram
  - Collaboration diagram
- Two are equivalent in power

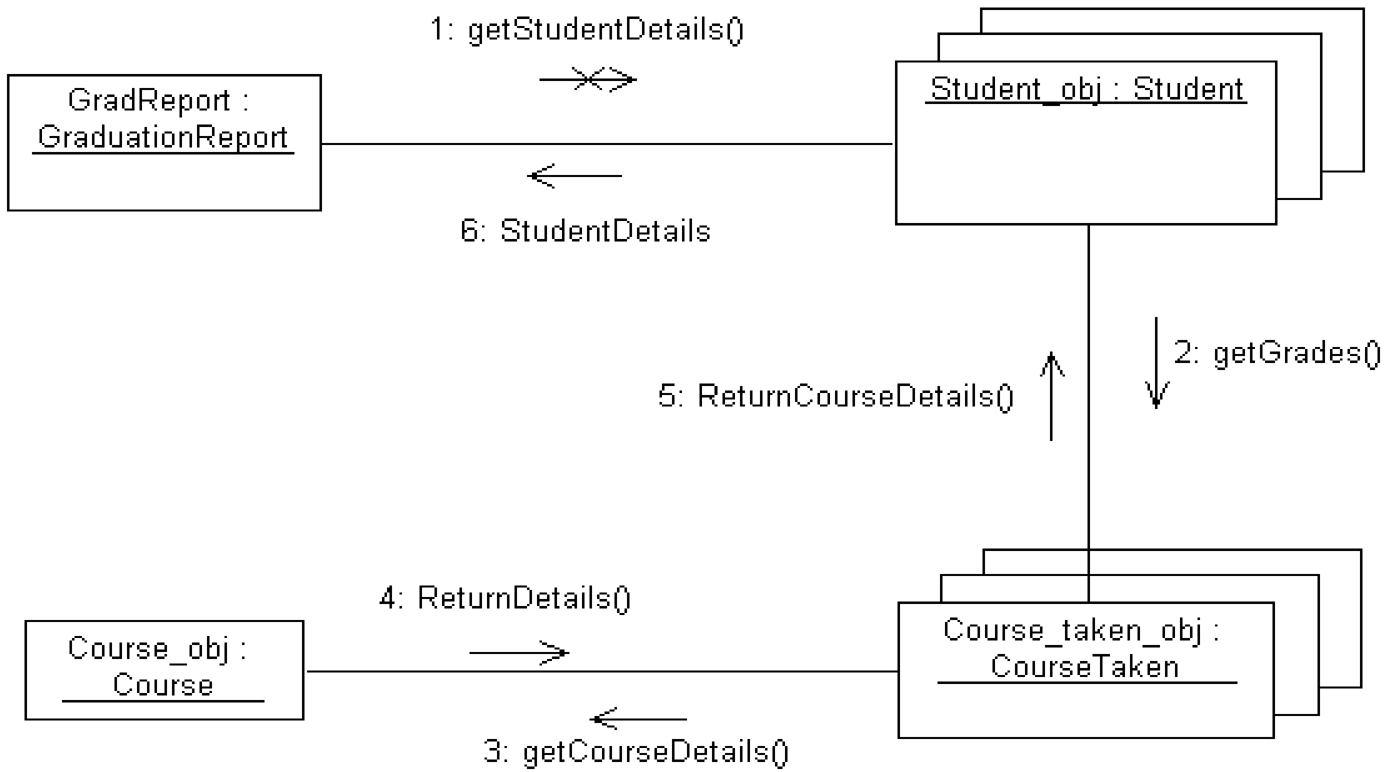
Sequence Diagram

- 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



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

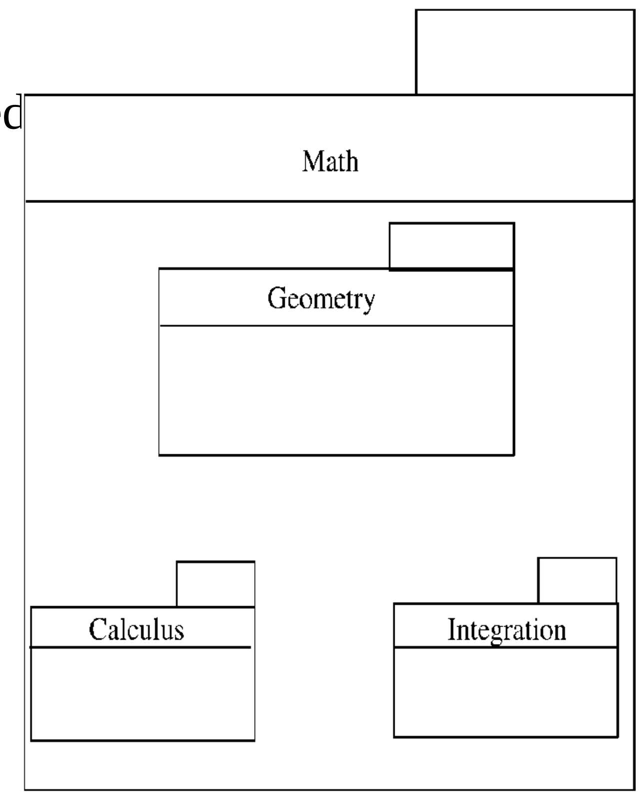


Other Diagrams

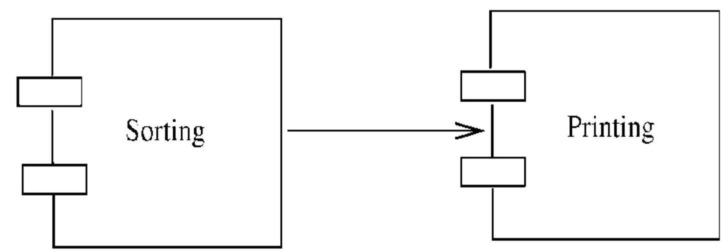
- Class diagram and interaction diagrams most commonly used during design
- There are other diagrams used to build different types of models

Other Diagrams

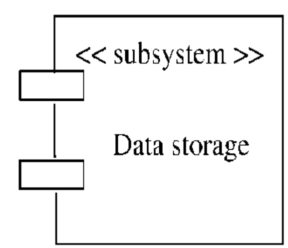
- Instead of objects/classes, can represent components, packages, subsystems
- These are useful for developing architecture structures
- UML is extensible – can model a new but similar concept by using stereotypes (by adding <<name>>), tagged values can be used to specify additional properties, e.g.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



PACKAGE



COMPONENT – CONNECTOR



SUBSYSTEM

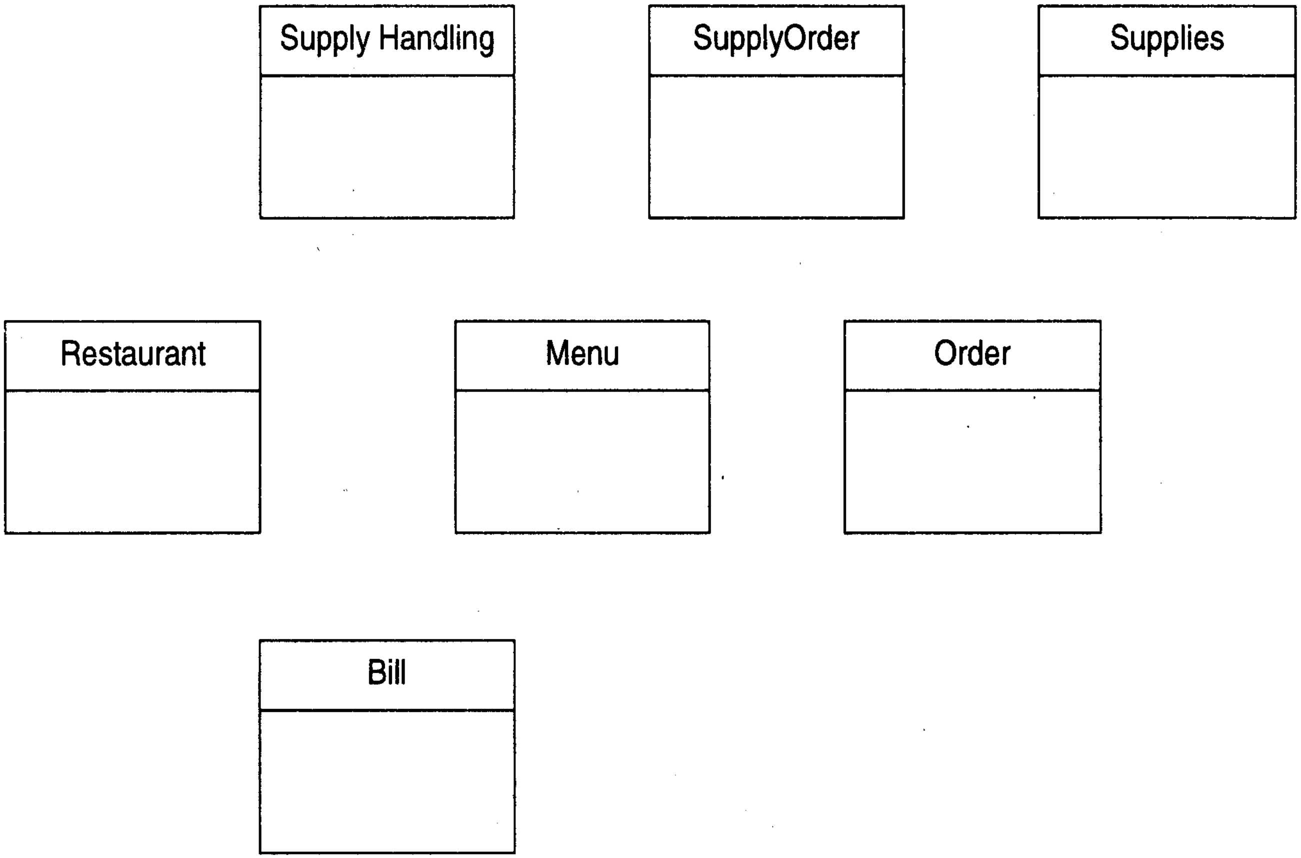
Design using UML

- 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

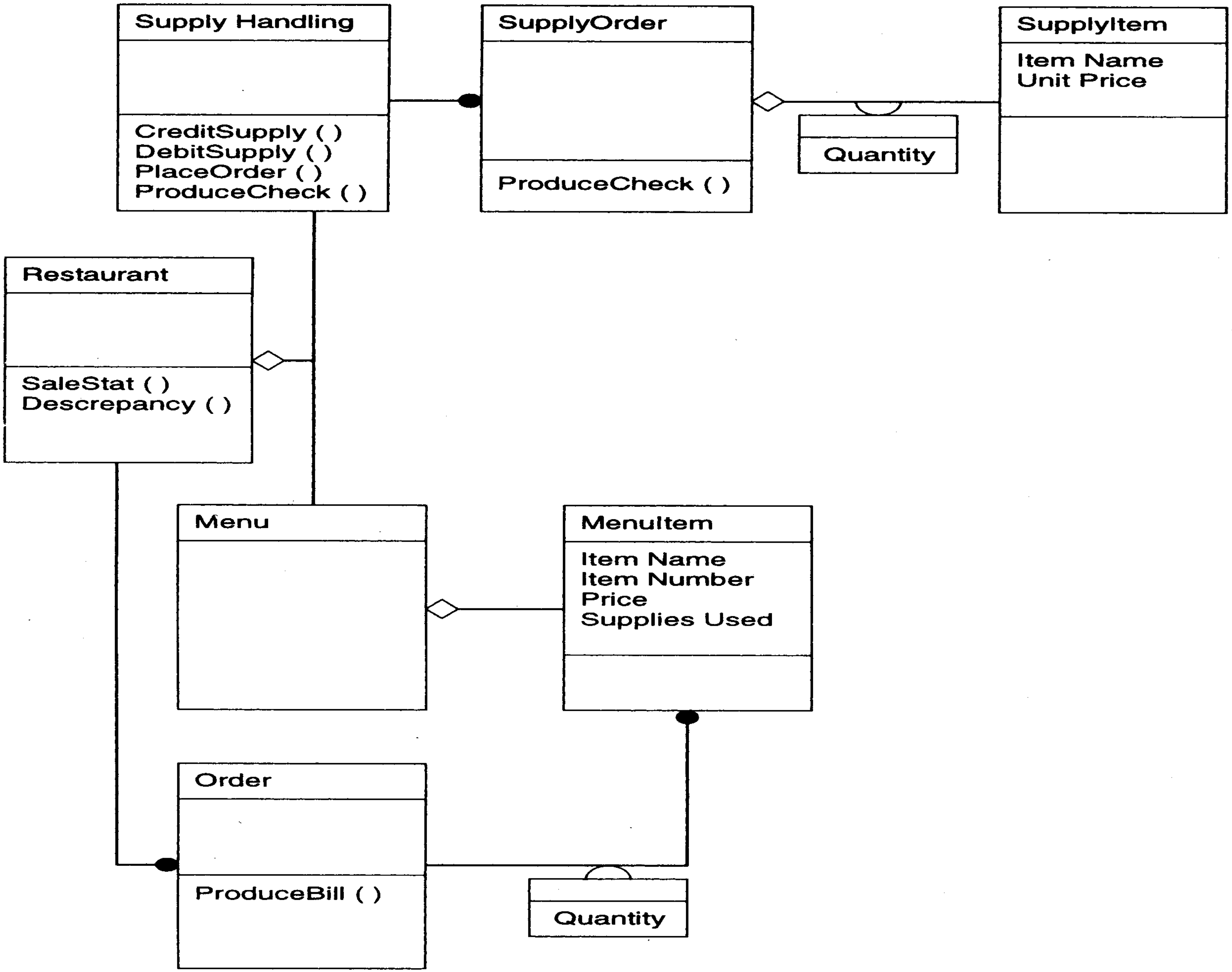
Design using UML

- 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 definition gets refined as we proceed

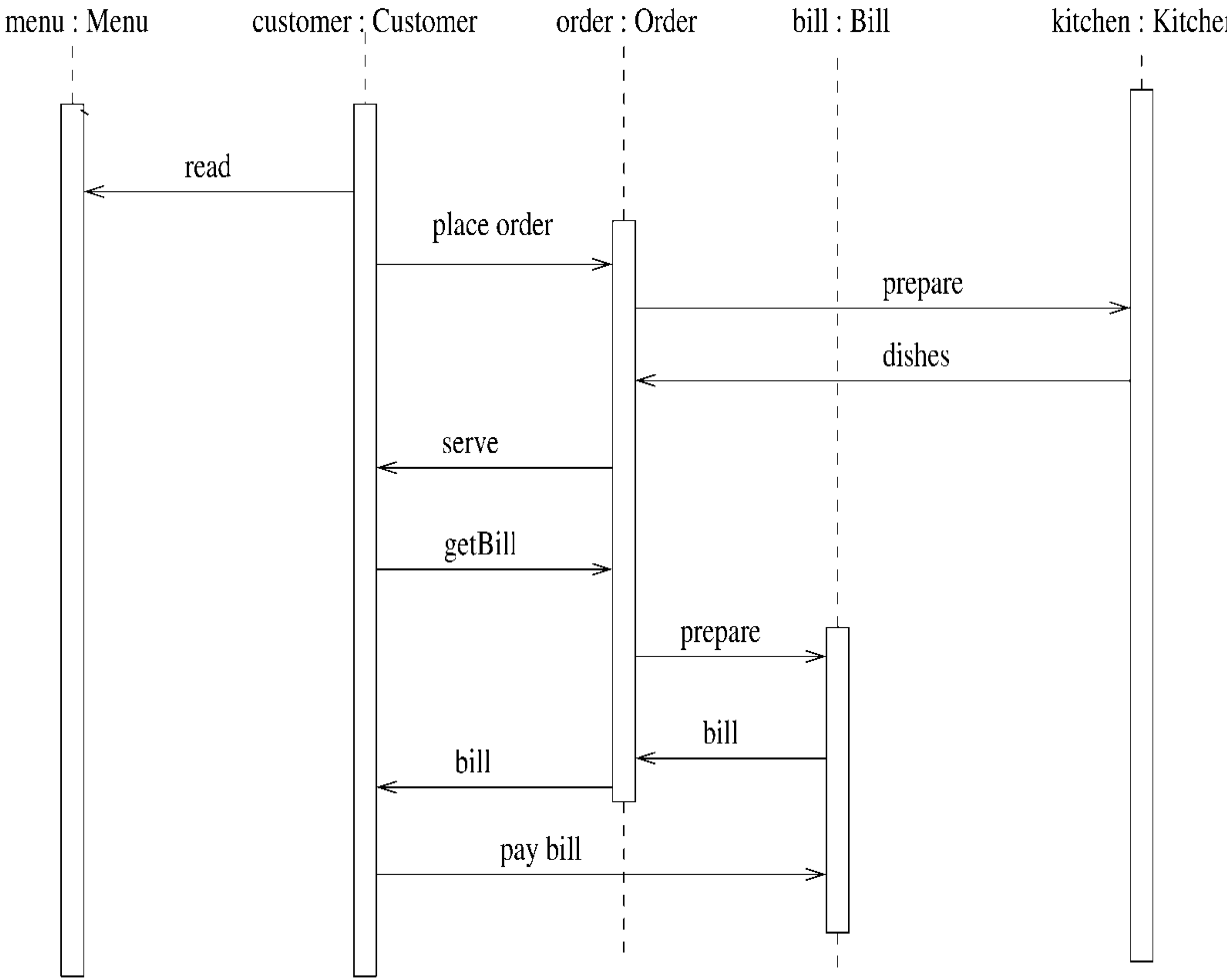
Restaurant example: Initial classes



Restaurant example: a class diagram



Restaurant example: a sequence diagram



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 algorithm to implement it is designed
- Various techniques possible for designing algorithm – in algorithms course
  - Problem statement from system design
  - Develop mathematical model for problem
  - Design algorithm – data structure and program structure

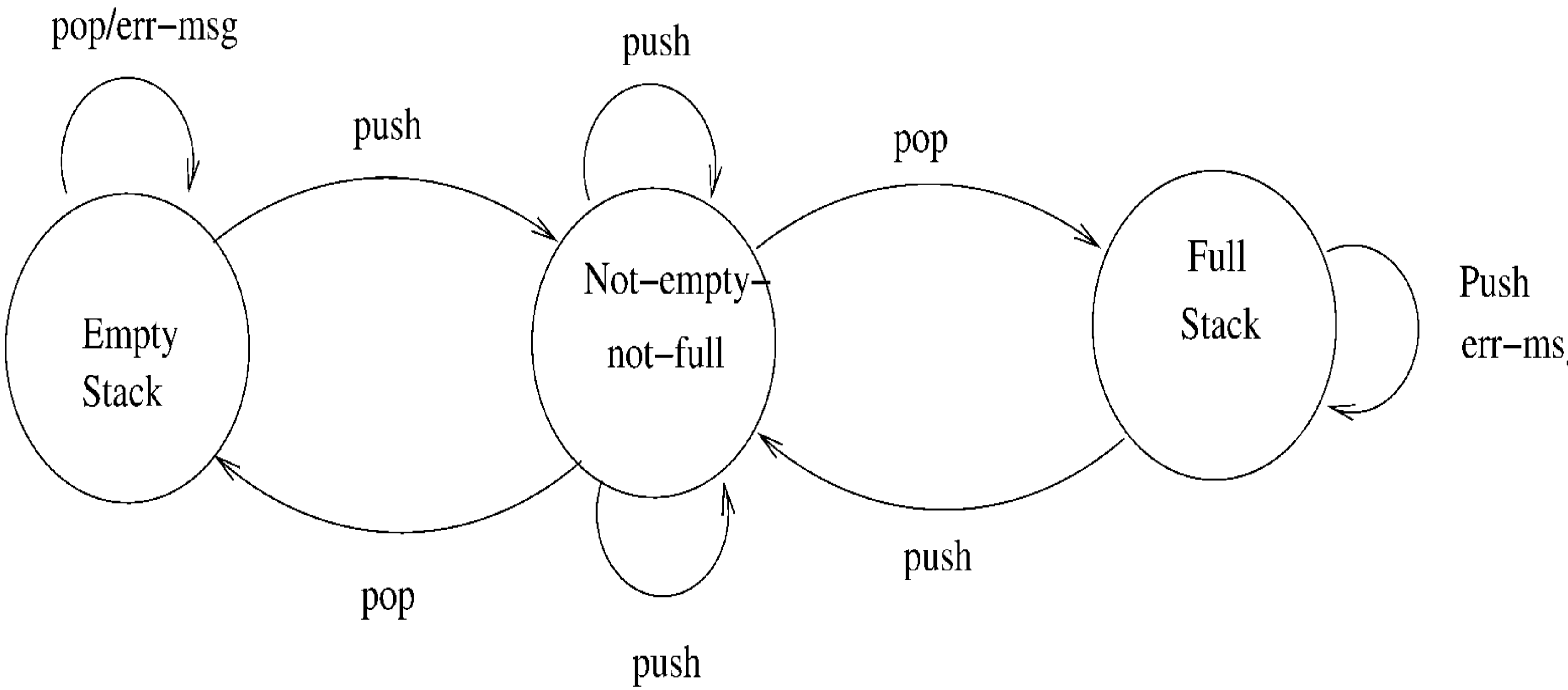
Logic/Algorithm Design

- Stepwise refinements technique is useful here
- Steps
  - Description of algorithm containing statements for each module
  - Statements are decomposed into more detailed statements
  - Refinement continues until all instructions in also can be implemented in PL statements

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
- Group of people discuss design to reveal errors and undesirable properties
- Members of system design, detailed design, requirements document author, design document author, software quality engineer

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

Complexity Metrics for Function-Oriented Design

1) Network Metrics

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

  $$\textit{Graph impurity} = n - e - 1$$

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

2) 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

3) 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 – information flowing in the module
  - Outflow – information flowing out of the module
- Dc = size * (inflow * outflow)^2
- (inflow * outflow) -represents total combination of inputs and outputs
- Its square represents interconnection between the modules

3) Information Flow Metrics

- Size represents the internal complexity of the module
- Product represents the total complexity
- D_c = fan_in * fan_out + inflow * outflow
- Size not considered
  - fan_in – no. of modules that call this module
  - fan_out – no. of modules this module calls

Identifying error-prone modules

- Uses avg_complexity of modules and std_deviation to identify error prone and complex modules:

Error prone: If D_c > avg complexity + std_deviation

Complex: If avg complexity < D_c < avg + std deviation

Normal: Otherwise

Complexity metrics for OO Design

1) Weighted Methods per Class (WMC)

- Complexity of a class depends on no. of methods in classes and their complexity
- Suppose complexity of methods is c_1, c_2..; by some functional complexity metric

  WMC = $\Sigma\, c_i$,i=1 to n
- Large WMC might mean that the class is more fault-prone

OO Metrics…

2) Depth of Inheritance Tree (DIT)

- DIT of class C is depth from the root class
- Length of the shortest path from root to node representing C
- DIT is significant in predicting fault proneness

3) Number of Children

- Immediate no. of subclasses of C
- Gives a sense of reuse

OO Metrics…

4) Coupling between classes (CBC)

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