Design

-Software Engineering

Slides compiled by Sanghamitra De

Introduction

- The design of a system is essentially a blueprint or a plan for a solution for the system.
- Design focuses on four major areas of concern: data, architecture, interfaces, and components.
- Design begins with the requirements model. Engineers work to transform this model into four levels of design detail: *the data structure, the system architecture, the interface representation, and the component level detail.*
- The final result is the Design Specification. The specification is composed of the design models that describe data, architecture, interfaces, and components.
- Design has two levels: **System level design** & **Detailed level design**.
- At the System level the focus is on deciding which modules are needed for the system, the specifications of these modules, and how the modules should be interconnected.
- In the Detailed level, the internal design of the modules, or how the specifications of the module can be satisfied, is decided.

Technical criteria for good design

- A design should exhibit an architectural structure that (a) has been created using recognizable design patterns, (b) is composed of components that exhibit good design characteristics, and (c) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
- A design should be modular; that is, the software should be logically partitioned into elements that perform specific functions and subfunctions.
- A design should contain distinct representations of data, architecture, interfaces, and components (modules).
- A design should lead to data structures that are appropriate for the objects to be implemented and are drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment.
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

Function-oriented design

Function-oriented design

- Design principles:
  - ✓Abstraction
  - ✓Problem Partitioning/ Refinement and Hierarchy
  - ✓Modularity
  - ✓Top-Down and Bottom-Up Strategies

Abstraction

- An abstraction of a component describes the external behavior of that component without bothering with the internal details that produce the behavior.
- Abstraction is essential for problem partitioning.
- Partitioning essentially is the exercise in determining the components of a system.
- These components are not isolated from each other; they interact with each other, and the designer has to specify how a component interacts with other components.
- To allow the designer to concentrate on one component at a time, abstraction of other components is used.
- Abstraction is used for existing components as well as components that are being designed.

Abstraction

- Using these abstractions, the behavior of the entire system can be understood.
- This also helps determine how modifying a component affects the system.
- The basic goal of system design is to specify the modules in a system and their abstractions.
- Once the different modules are specified, during the detailed design the designer can concentrate on one module at a time.
- The task in detailed design and implementation is essentially to implement the modules so that the abstract specifications of each module are satisfied.

Abstraction

- There are two common abstraction mechanisms for software systems: *functional abstraction* and **data abstraction**.
- In functional abstraction, a module is specified by the function it performs.
- Functional abstraction is the basis of partitioning in function-oriented approaches.
- That is, when the problem is being partitioned, the overall transformation function for the system is partitioned into smaller functions that comprise the system function.
- The decomposition of the system is in terms of functional modules

Abstraction

- Any entity in the real world provides some services to the environment to which it belongs.
- The case of data entities is similar. Certain operations are required from a data object, depending on the object and the environment in which it is used.
- Data abstraction supports this view. Data is not treated simply as objects, but is treated as objects with some predefined operations on them.
- The operations defined on a data object are the only operations that can be performed on those objects.
- From outside an object, the internals of the object are hidden; only the operations on the object are visible.
- Data abstraction forms the basis for object-oriented design.
- In using this abstraction, a system is viewed as a set of objects providing some services.
- Hence, the decomposition of the system is done with respect to the objects the system contains.

Problem Partitioning/ Refinement and Hierarchy

- For solving larger problems, the basic principle is the principle of "divide and conquer."
- For software design, therefore, the goal is to divide the problem into manageably small pieces that can be solved separately.
- The basic rationale behind this strategy is the belief that if the pieces of a problem are solvable separately, the cost of solving the entire problem is more than the sum of the cost of solving all the pieces.
- However, the different pieces cannot be entirely independent of each other, as they together form the system.
- The different pieces have to cooperate and communicate to solve the larger problem.
- This communication adds complexity, which arises due to partitioning and may not have existed in the original problem.

Problem Partitioning/ Refinement and Hierarchy

- As the number of components increases, the cost of partitioning, together with the cost of this added complexity, may become more than the savings achieved by partitioning.
- It is at this point that no further partitioning needs to be done.
- Problem partitioning also aids design verification.
- Problem partitioning, which is essential for solving a complex problem, leads to hierarchies in the design.

Modularity

Criteria that enable evaluation of a design method with respect to its ability to define an effective modular system:

- ✓Modular decomposability
- ✓Modular composability
- ✓Modular understandability
- ✓Modular continuity
- ✓Modular protection

Modularity

- A system is considered modular if it consists of discreet components so that each component can be implemented separately, and a change to one component has minimal impact on other components.
- For modularity, each module needs to support a well defined abstraction and have a clear interface through which it can interact with other modules.

Module level concepts

- Coupling between modules is the strength of interconnections between modules or a measure of interdependence among modules.
- In general, the more we must know about module A in order to understand module B, the more closely connected A is to B.
- "*Highly coupled*" modules are joined by strong interconnections, while "*loosely coupled*" modules have weak interconnections.
- Independent modules have no interconnections.
- To solve and modify a module separately, we would like the module to be loosely coupled with other modules.

Module level concepts

- Coupling increases with the **complexity and obscurity of the interface between modules**.
- To keep coupling low we would like to *minimize the number of interfaces per module* and the *complexity of each interface*.
- An interface of a module is used to pass information to and from other modules.
- Coupling is reduced if only the defined entry interface of a module is used by other modules (for example, passing information to and from a module exclusively through parameters).
- Coupling would increase if a module is used by other modules via an indirect and obscure interface, like directly using the internals of a module or using shared variables.
- Complexity of the interface is another factor affecting coupling.

Module level concepts

- The **type of information flow along the interfaces** is the third major factor affecting coupling.
- There are two kinds of information that can flow along an interface: data or control.
- Passing or receiving control information means that the action of the module will depend on this control information, so difficult to understand the module and provide its abstraction.
- Transfer of data information means that a module passes as input some data to another module and is treated as a simple input-output function.
- Coupling is considered highest if the data is hybrid, that is, some data items and some control items are passed between modules.

Module level concepts

- Coupling is reduced when elements in different modules have little or no bonds between them.
- Another way of achieving this effect is to strengthen the bond between elements of the same module by maximizing the relationship between elements of the same module.
- Cohesion of a module represents how tightly bound the internal elements of the module are to one another.
- Cohesion of a module gives the designer an idea about whether the different elements of a module belong together in the same module.
- Cohesion and coupling are clearly related.
- Usually, the greater the cohesion of each module in the system, the lower the coupling between modules is.

Module level concepts

Levels of Cohesion:

- ✓Coincidental
- ✓Logical
- ✓Temporal
- ✓Procedural
- ✓Communicational
- ✓Sequential
- ✓Functional

Top-Down and Bottom-Up Strategies

- A system consists of components, which have components of their own; indeed a system is a hierarchy of components.
- The highest-level components correspond to the total system.
- To design such hierarchies there are two possible approaches: top-down and bottom-up.
- The top-down approach starts from the highest-level component of the hierarchy and proceeds through to lower levels.

Top-Down and Bottom-Up Strategies

- A top-down design approach starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved.
- Top-down design methods often result in some form of stepwise refinement.
- Starting from an abstract design, in each step the design is refined to a more concrete level, until a level is reached where no more refinement is needed and the design can be implemented directly.
- A top-down approach is suitable only if the specifications of the system are clearly known and the system development is from scratch.

Top-Down and Bottom-Up Strategies

- A bottom-up design approach starts with designing the most basic or primitive components and proceeds to higher-level components that use these lower-level components.
- Bottom-up methods work with layers of abstraction.
- Starting from the very bottom, operations that provide a layer of abstraction are implemented.
- The operations of this layer are then used to implement more powerful operations and a still higher layer of abstraction, until the stage is reached where the operations supported by the layer are those desired by the system.
- For a bottom-up approach to be successful, we must have a good notion of the top to which the design should be heading.
- A common approach to combine the two approaches

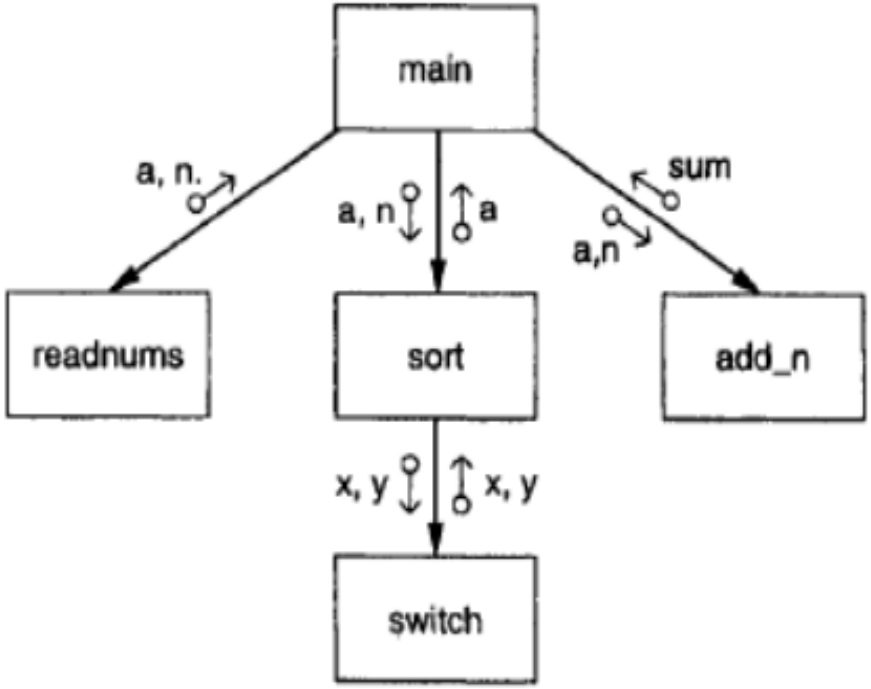
Design Notation and Specification

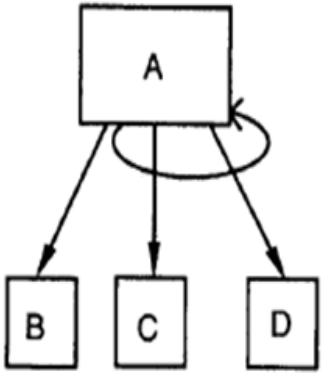
- Structure Charts
- Specification
- Structured Design Methodology

Structure Charts

- The structure chart of a program is a graphic representation of its structure.
- Procedural information is not represented in a structure chart, and the focus is on representing the hierarchy of modules.
- There are situations where the designer may wish to communicate certain procedural information explicitly, like major loops and decisions.
- Such information can also be represented in a structure chart.

Structure Charts





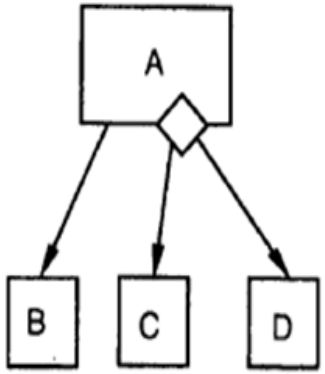
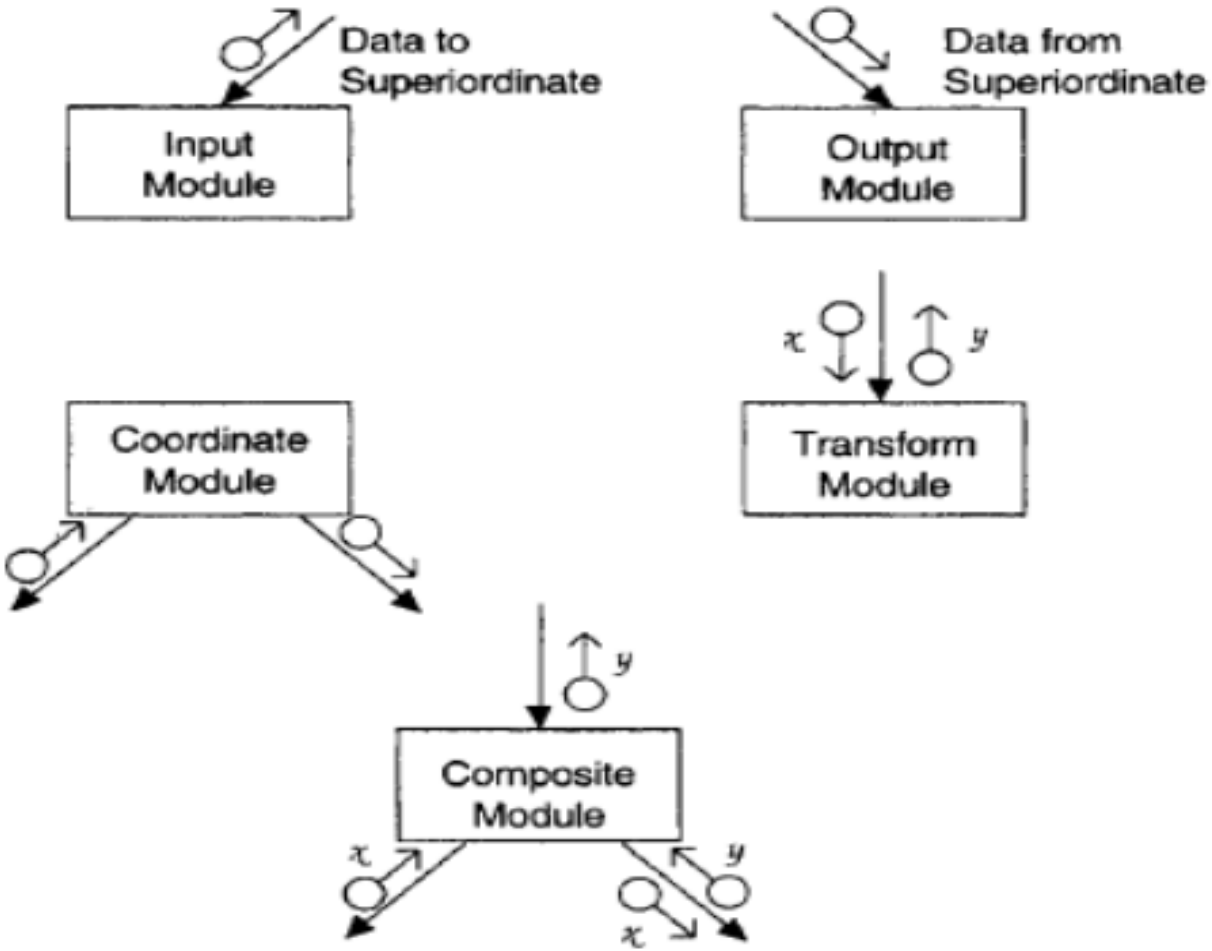


Figure : Iteration and decision representation.

Structure Charts

- In a structure chart a module is represented by a box with the module name written in the box.
- An arrow from module A to module B represents that module A invokes module B.
- B is called the subordinate of A, and A is called the superordinate of B.
- The arrow is labeled by the parameters received by B as input and the parameters returned by B as output, with the direction of flow of the input and output parameters represented by small arrows.
- The parameters can be shown to be data (unfilled circle at the tail of the label) or control (filled circle at the tail).

Structure Charts



Structure Charts

- Module Types:
  - ✓input modules
  - ✓ output modules
  - ✓transform module
  - ✓coordinate modules
  - ✓composite modules

Specification

- A design specification should define the major data structures, modules and their specifications, and design decisions.
- Module specification is the major part of system design specification.
- All modules in the system should be identified when the system design is complete, and these modules should be specified in the document.

Specification

- A design specification language reduces the effort required for translating the design to programs.
- From the design, the module headers can easily be created with some simple editing.
- All major design decisions made by the designers during the design process should be explained explicitly.

Structured Design Methodology (SDM)

- Structured design methodology (SDM) views every software system as having some inputs that are converted into the desired outputs by the software system.
- The software is viewed as a *transformation function* that transforms the given inputs into the desired outputs.
- Central problem of designing software systems is considered to be properly designing this transformation function.
- So, structured design methodology is primarily function-oriented and relies heavily on functional abstraction and functional decomposition.

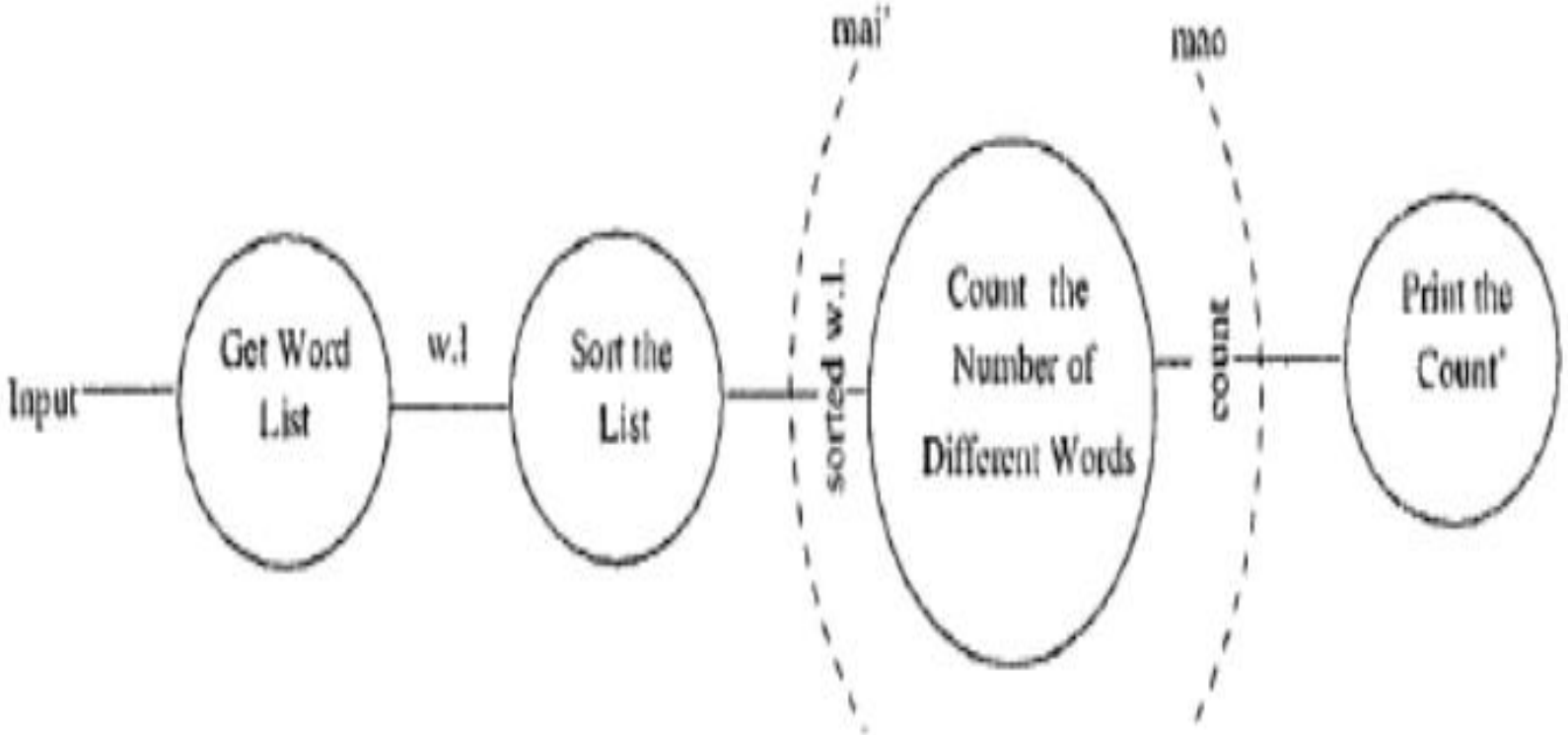
SDM

- Factoring is the process of decomposing a module so that the bulk of its work is done by its subordinates.
- A system is said to be completely factored if all the actual processing is accomplished by bottom-level atomic modules and if non-atomic modules largely perform the jobs of control and coordination.
- SDM attempts to achieve a structure that is close to being completely factored.

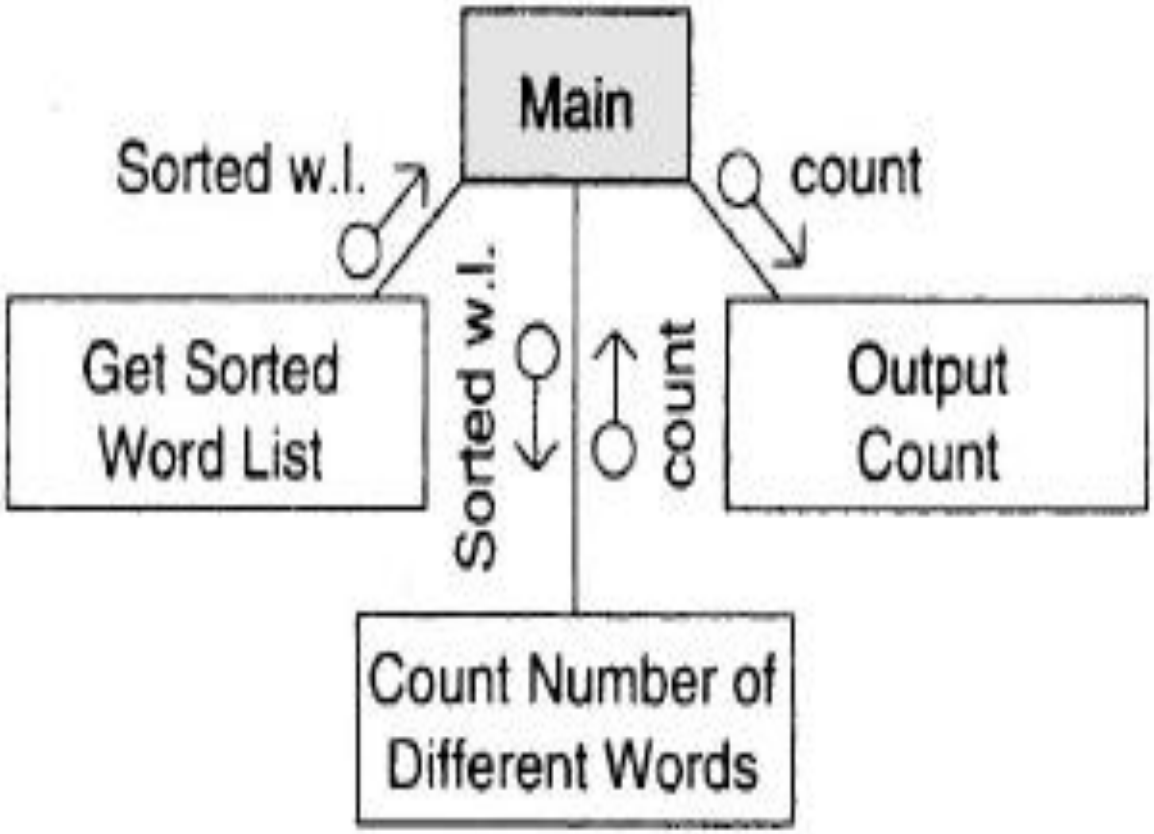
SDM

- Steps in SDM:
  - ✓ Restate the problem as a data flow diagram
  - ✓ Identify the input and output data elements
    - ❖ *Most Abstract Input (MAI)*
    - ❖ *Most Abstract Output (MAO)*
  - ✓ First-level factoring
  - ✓ Factoring of input, output, and transform branches

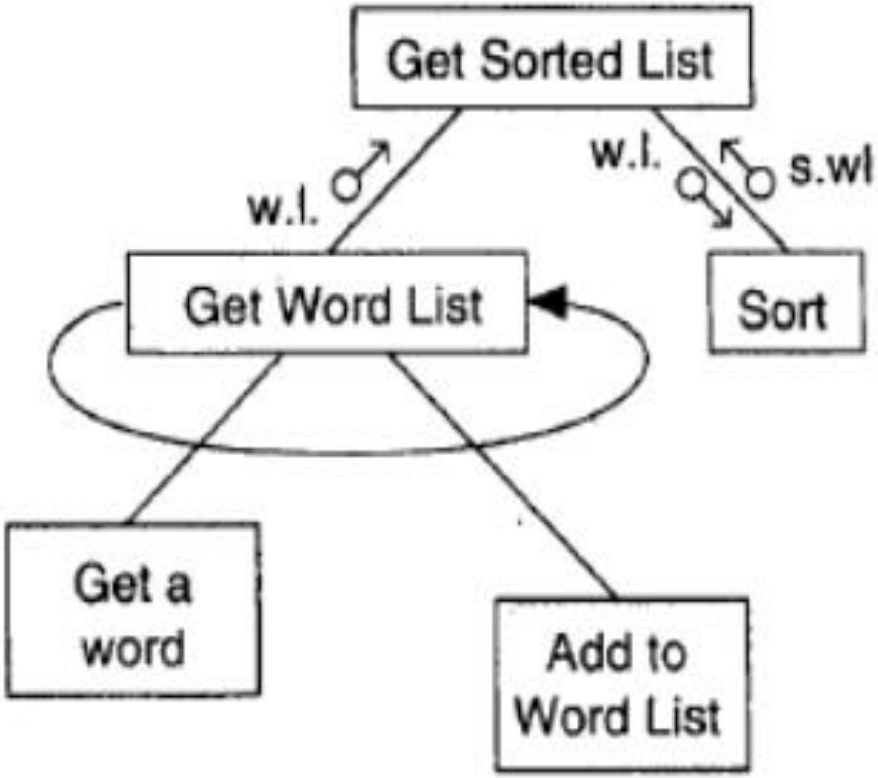
SDM


mai'
mao
Input
Get Word List
w.l
Sort the List
sorted w.l.
Count the Number of Different Words
count
Print the Count'

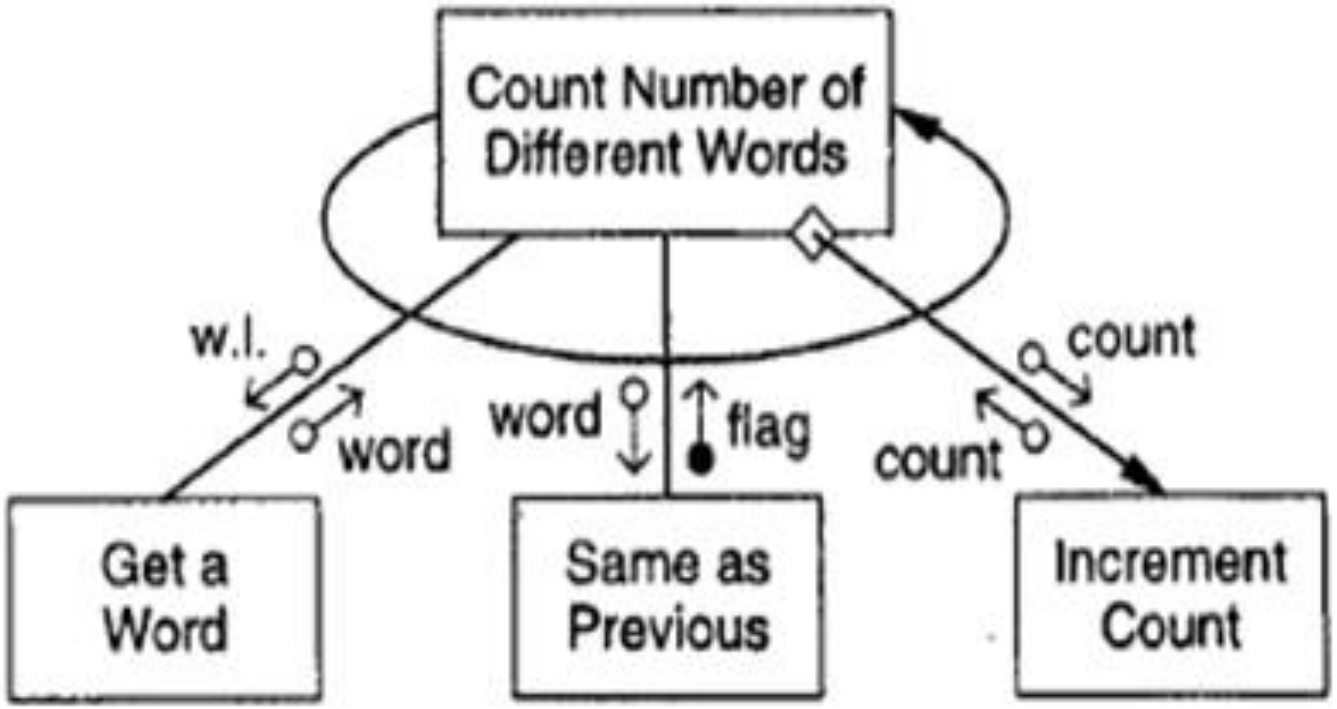
SDM


Main
Sorted w.l.
count
Get Sorted
Word List
Sorted w.l.
count
Output
Count
Count Number of
Different Words

SDM


Get Sorted List
w.l.
w.l.
s.wl
Get Word List
Sort
Get a word
Add to Word List

SDM


Count Number of Different Words
w.l.
word
word
flag
count
count
Get a Word
Same as Previous
Increment Count

Design Heuristics

- Module size
- Fan-in & fan-out of modules
- Correlation of the scope of effect & scope of control

Design Heuristics

- Modules that are very large may not be implementing a single function and can therefore be broken into many modules, each implementing a different function.
- Modules that are too small may not require any additional identity and can be combined with other modules.
- A module should be split into separate modules only if the cohesion of the original module was low, the resulting modules have a higher degree of cohesion, and the coupling between modules does not increase.
- Two or more modules should be combined only if the resulting module has a high degree of cohesion and the coupling of the resulting module is not greater than the coupling of the sub modules.

Design Heuristics

- Fan-in of a module is the number of arrows coming in the module, indicating the number of superordinates of a module.
- Fan-out of a module is the number of arrows going out of that module, indicating the number of subordinates of the module.
- Very high fan-out is not very desirable, as it means that the module has to control and coordinate too many modules and may therefore be too complex.
- Fan-out can be reduced by creating a subordinate and making many of the current subordinates subordinate to the newly created module.
- Fan-out should not be increased above five or six.
- Whenever possible, the fan-in should be maximized but not at the cost of increasing the coupling or decreasing the cohesion of modules.

Design Heuristics

- *Scope of effect* of a decision (in a module) is the collection of all the modules that contain any processing that is conditional on that decision or whose invocation is dependent on the outcome of the decision.
- *Scope of control* of a module is the module itself and all its subordinates (not just the immediate subordinates).
- System is usually simpler when the scope of effect of a decision is a subset of the scope of control of the module in which the decision is located.

Verification

- Design using formal notation can be verified by analysis tools.
- Most popular method of system design verification is **Design Review** or **Inspections**
- Most significant design error is *omission or misinterpretation of specified requirements*.
- Other *quality factors (e.g.* weak modularity due to weak cohesion and/or strong coupling)
- Checklists must be used for design review

Verification

Typical checklist questions for a design review:

- ✓ Is each of the functional requirements taken into account?
- ✓ Are there analyses to demonstrate that performance requirements can be met?
- ✓ Are all assumptions explicitly stated, and are they acceptable?
- ✓ Are there any limitations or constraints on the design beyond those in the requirements?
- ✓ Are external specifications of each module completely specified?
- ✓ Have exceptional conditions been handled?
- ✓ Are all the data formats consistent with the requirements?
- ✓ Are the operators and user interfaces properly addressed?
- ✓ Is the design modular, and does it conform to local standards?
- ✓ Are the sizes of data structures estimated? Are provisions made to guard against overflow?

Software architecture as part of design

- Certain properties should be specified as part of an architectural design, these are:
  - ✓ Structural properties
  - ✓ Extra-functional properties
  - ✓ Families of related systems
- Architectural design can be represented using one or more of a number of different models, e.g. Structural models, Framework models, Dynamic models, Process models, Functional models.
- Architectural Description Languages (ADLs) have been developed to represent these models

Software architecture as part of design

- *Structural models* represent architecture as an organized collection of program components.
- *Framework models* increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications.
- *Dynamic models* address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.
- *Process models* focus on the design of the business or technical process that the system must accommodate.
- *Functional models* can be used to represent the functional hierarchy of a system.

Object-oriented design

OOAD

- Object-oriented analysis (OOA) *models the problem domain*, leading to an understanding and specification of the problem
- Object-oriented design (OOD) *models the solution to the problem*
- In OOAD it is believed that the solution domain representation, created by OOD, generally contains much of the representation created by OOA
- Frequently the OOA and OOD processes (i.e., the methodologies) and the representations look quite similar

OOAD

- The main difference between OOA and OOD, due to the different domains of modeling, is in the type of objects that come out of the analysis and design processes.
- Objects during OOA focus on the problem domain and generally represent some things or concepts in the problem.
- These objects are called semantic objects as they have a meaning in the problem domain.
- The solution domain, on the other hand, consists of semantic objects, interface, application & utility objects.

OOAD

- The interface objects deal with the user interface, which is not directly a part of the problem domain but represents some aspect of the solution desired by the user.
- The application objects specify the control mechanisms for the proposed solution. They are driver objects that are specific to the application needs.
- Utility objects are those needed to support the services of the semantic objects or to implement them efficiently (e.g. queues, trees, and tables). These objects are frequently general-purpose objects and are not application-dependent.

Object oriented concepts: Classes & Objects

- Classes and objects are the basic building blocks of an OOD.
- During analysis, we viewed an object as an entity in the problem domain that had clearly defined boundaries and behavior.
- During design, this has to be extended to accommodate software objects.
- Objects represent the basic run-time entities in an OO system, they occupy space in memory that keeps its state and is operated on by the defined operations on the object.

Classes & Objects

- Class, defines a possible set of objects.
- Objects have some attributes, whose values constitute much of the state of an object.
- What attributes an object has are defined by the class of the object.
- The operations allowed on an object or the services it provides, are defined by the class of the object.

Classes & Objects

- Classes have:
  - ✓ An interface that defines which parts of an object of a class can be accessed from outside and how
  - ✓ A class body that implements the operations in the interface
  - ✓ Instance variables that contain the state of an object of that class

Classes & Objects

- Data and operations of a class can be declared as one of three types:
  - ✓ Public: These are (data or operation) declarations that are accessible from outside the class to anyone who can access an object of this class.
  - ✓ Protected: These are declarations that are accessible from within the class itself and from within subclasses (actually also to those classes that are declared as friends).
  - ✓ Private: These are declarations that are accessible only from within the class itself (and to those classes that are declared as friends).

Classes & Objects

- The operation creating and initializing objects is called **constructor**, and the operation destroying objects is called **destructor**.
- Remaining operations can be broadly divided into two categories: **modifiers** and **value-ops**.
- Modifiers are operations that modify the state of the object, while value-ops are operations that access the object state but do not alter it.

State, Behavior, and Identity

- An object has state, behavior, and identity. The encapsulated data for an object defines the state of the object.
- An important property of objects is that this state persists, in contrast to the data defined in a function or procedure.
- The various components of the information an object encapsulates can be viewed as "attributes" of the object.
- That is, an object can be viewed as having various attributes, whose values form the state of the object.

State, Behavior, and Identity

- The state and services of an object together define its behavior.
- The behavior of an object is how an object reacts in terms of state changes when it is acted on, and how it acts upon other objects by requesting services and operations.
- A side effect of performing an operation may be that the state of the object is modified.

State, Behavior, and Identity

- Identity is the property of an object that distinguishes it from all other objects. In most programming languages, variable names are used to distinguish objects from each other.
- E.g. one can declare objects s1, s2, ... of class type Stack. Each of these variables s1, s2, ... will refer to a unique stack having a state of its own.

Encapsulation

- Basic property of an object is encapsulation: it encapsulates the data and information it contains, and supports a well-defined abstraction.
- For this, an object provides some well-defined services its clients can use, with the additional constraint that a client can access the object only through these services.
- The set of services that can be requested from outside the object forms the interface of the object.

Encapsulation

- A major advantage of encapsulation is that access to the encapsulated data is limited to the operations defined on the data.
- Hence, it becomes much easier to ensure that the integrity of data is preserved, something very hard to do if any program from outside can directly manipulate the data structures of an object.
- As long as the interface is preserved, implementation of an object can be changed without affecting any user of the object.

Relationships among Objects

- Any complex system will be composed of many objects of different classes, and these objects will interact with each other so that the overall system objectives are met.
- In object-oriented systems, an object interacts with another by sending a message to the object to perform some service it provides.
- On receiving the message, the object invokes the requested service or the method and sends the result, if needed.
- Frequently, the object providing the service is called the **server** and the object requesting the service is called the **client**.

Relationships among Objects

- If an object invokes some services in other objects, we can say that the two objects are related in some way to each other.
- All objects in a system are not related to all other objects.
- During design, which objects are related has to be clearly defined so that the system can be properly implemented.

Relationships among Objects

- If an object uses some services of another object, there is an association between the two objects.
- This **association** is also called a link—a link exists from one object to another if the object uses some services of the other object.
- If there is a link from object A to object B, for A to be able to send a message to B, B must be visible to A in the final program.

Relationships among Objects

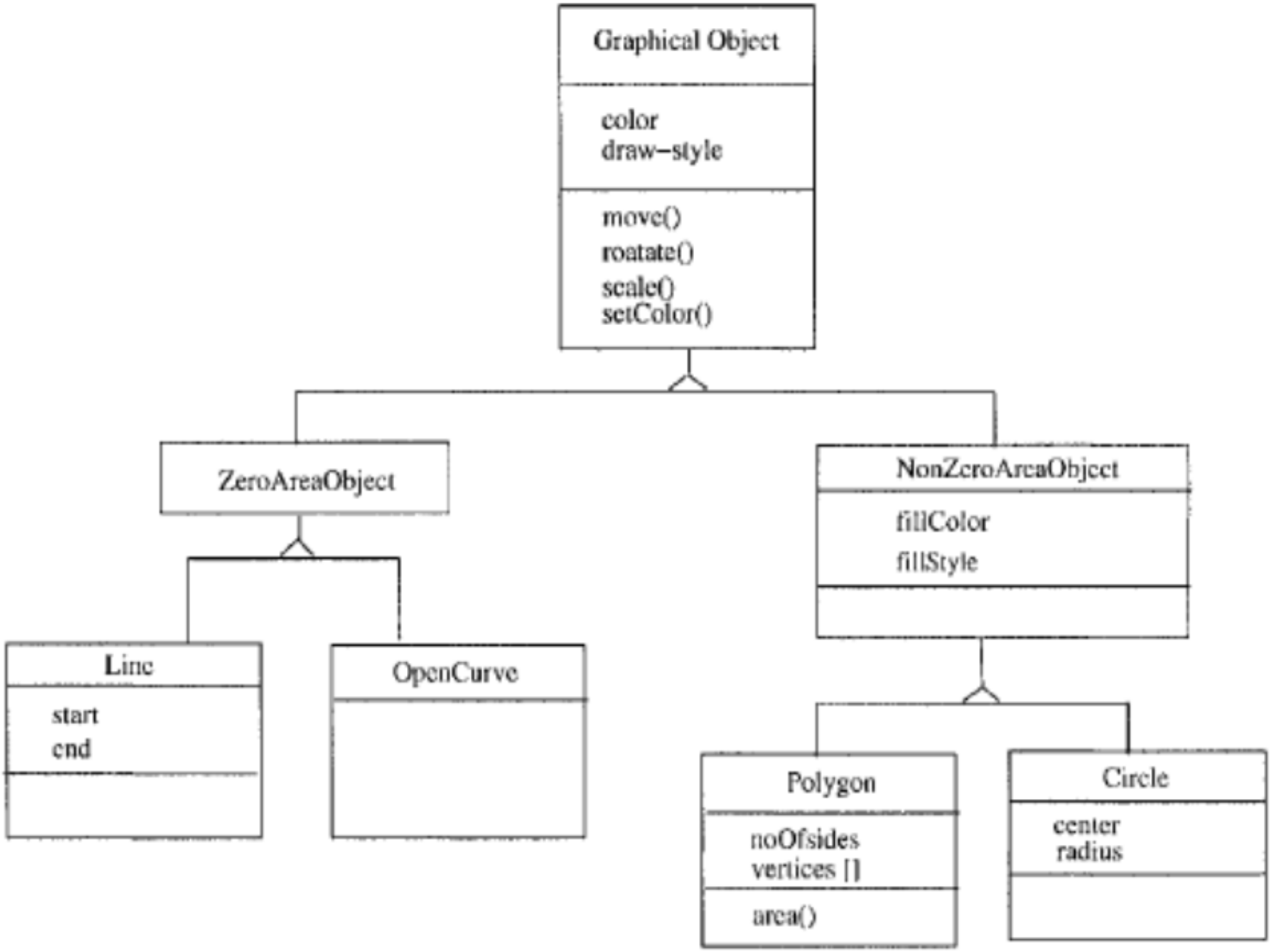
- Another type of relationship between objects is **aggregation**, which reflects the whole/part-of relationship.
- Though not necessary, aggregation generally implies containment.
- That is, if an object A is an aggregation of objects B and C, then objects B and C will generally be within object A.

Inheritance and Polymorphism

- Inheritance is a concept unique to object orientation.
- Inheritance is a relation between classes that allows for definition and implementation of one class based on the definition of existing classes.
- When a class B inherits from another class A, B is referred to as the *subclass* or the *derived class* and A is referred to as the *superclass* or the *base class*.
- In general, a subclass B will have two parts: a derived part and an incremental part.
- The derived part is the part inherited from A and the incremental part is the new code and definitions that have been specifically added for B.

Inheritance and Polymorphism

- The inheritance relation between classes forms a hierarchy.
- As inheritance represents an "*is-a*" relation, it is important that the hierarchy represent a structure present in the application domain and is not created simply to reuse some parts of an existing class.
- A feature is placed in the higher level of abstractions.
- Once this is done, such features can be inherited from the parent class and used in the subclass directly.
- This implies that if there are many abstract class definitions available, when a new class is needed, it is possible that the new class is a specialization of one or more of the existing classes.
- Inheritance promotes reuse by defining the common operations of the subclasses in a superclass.

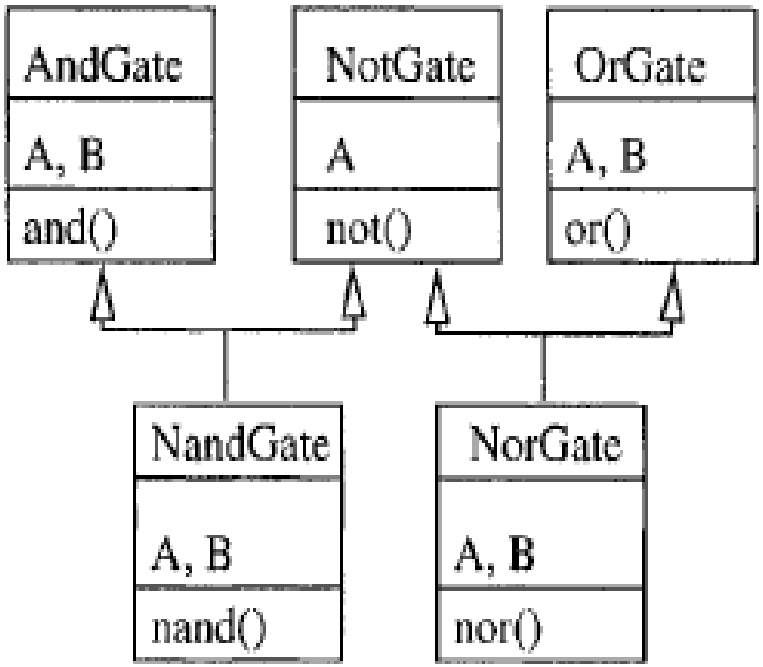

Graphical Object
color
draw-style
move()
roatate()
scale()
setColor()
ZeroAreaObject
NonZeroAreaObject
fillColor
fillStyle
Line
start
end
OpenCurve
Polygon
noOfsides
vertices []
area()
Circle
center
radius

Inheritance and Polymorphism

- In *strict inheritance* a subclass takes all the features from the parent class and adds additional features to specialize it.
- This form supports the "is-a" relation and is the easiest form of inheritance.
- *Nonstrict inheritance* occurs when the subclass does not have all the features of the parent class or some features have been redefined.
- This form of inheritance has consequences in the dynamic behavior and complicates testing

Inheritance and Polymorphism

- When a subclass inherits from many superclasses, is called **multiple inheritance**.



Inheritance and Polymorphism

- Inheritance brings in *polymorphism*, a general concept widely used in type theory that deals with the ability of an object to be of different types.
- With polymorphism, an entity has a *static type* and a *dynamic type*.
- The static type of an object is the type of which the object is declared in the program text, and it remains unchanged.
- The dynamic type of an entity, on the other hand, can change from time to time and is known only at reference time.
- Once an entity is declared, at compile time the set of types that this entity belongs to can be determined from the inheritance hierarchy that has been defined.
- The dynamic type of the object will be one of this set, but the actual dynamic type will be defined at the time of reference of the object.
- This type of polymorphism is called **object polymorphism**, in which wherever an object of a superclass can be used, objects of subclasses can be used.

Inheritance and Polymorphism

- There is another type of polymorphism which requires *dynamic binding* of operations, which brings in feature polymorphism.
- Dynamic binding means that the code associated with a given procedure call is not known until the moment of the call.
- ***Feature polymorphism***, is essentially overloading of the feature (i.e., a feature can mean different things in different contexts and its exact meaning is determined only at run time) causes no problem in strict inheritance because all features of a superclass are available in the subclasses.
- But in nonstrict inheritance, it can cause problems, because a child may lose a feature.
- Since the binding of the feature is determined at run time, this can cause a run-time error as a situation may arise where the object is bound to the superclass in which the feature is not present.

Design Concepts

- Cohesion
- Coupling
- Open-closed principle

Cohesion

➢ Method cohesion

Method cohesion is same as cohesion in functional modules. It focuses on why the different code elements of a method are together within the method.

➢ Class cohesion

Class cohesion focuses on why different attributes and methods are together in this class. The goal is to have a class that implements a single concept or abstraction with all elements contributing towards supporting this concept.

➢ Inheritance cohesion

Inheritance cohesion focuses on why classes are together in a hierarchy. The two main reasons for inheritance are to model generalization-specialization relationship, and for code reuse.

Coupling

➢ Interaction coupling

Interaction coupling occurs due to methods of a class invoking methods of other classes. In many ways, this situation is similar to a function calling another function and hence this coupling is similar to coupling between functional modules.

➢ Component coupling

Component coupling refers to the interaction between two classes where a class has variables of the other class.

➢ Inheritance coupling

Inheritance coupling is due to the inheritance relationship between classes. Two classes are considered inheritance coupled if one class is a direct or indirect subclass of the other. If inheritance adds coupling, why not do away with inheritance altogether. The reason is that inheritance may reduce the overall coupling in the system.

Open-Closed Principle

- Software entities should be open for extension, but closed for modification.
- A module being "*open for extension*" means that its behaviour can be extended to accommodate new demands placed on this module due to changes in requirements and system functionality.
- The modules being "*closed for modification*" means that the existing source code of the module is not changed when making enhancements.

Open-Closed Principle

- If this principle is satisfied, then we can expand existing systems by mostly adding new code to old systems, and minimizing the need for changing code.
- This principle can be satisfied in OO designs by properly using inheritance and polymorphism.
- Inheritance allows creating new classes that will extend the behaviour of existing classes without changing the original class. And it is this property that can be used to support this principle.

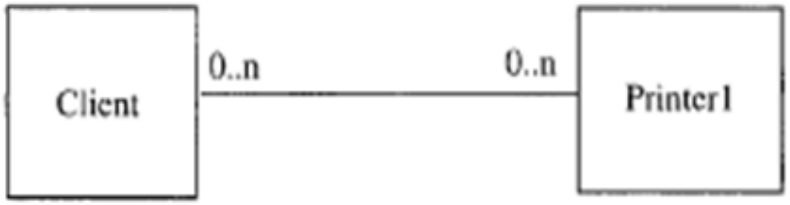


Figure : Example without using subtyping.

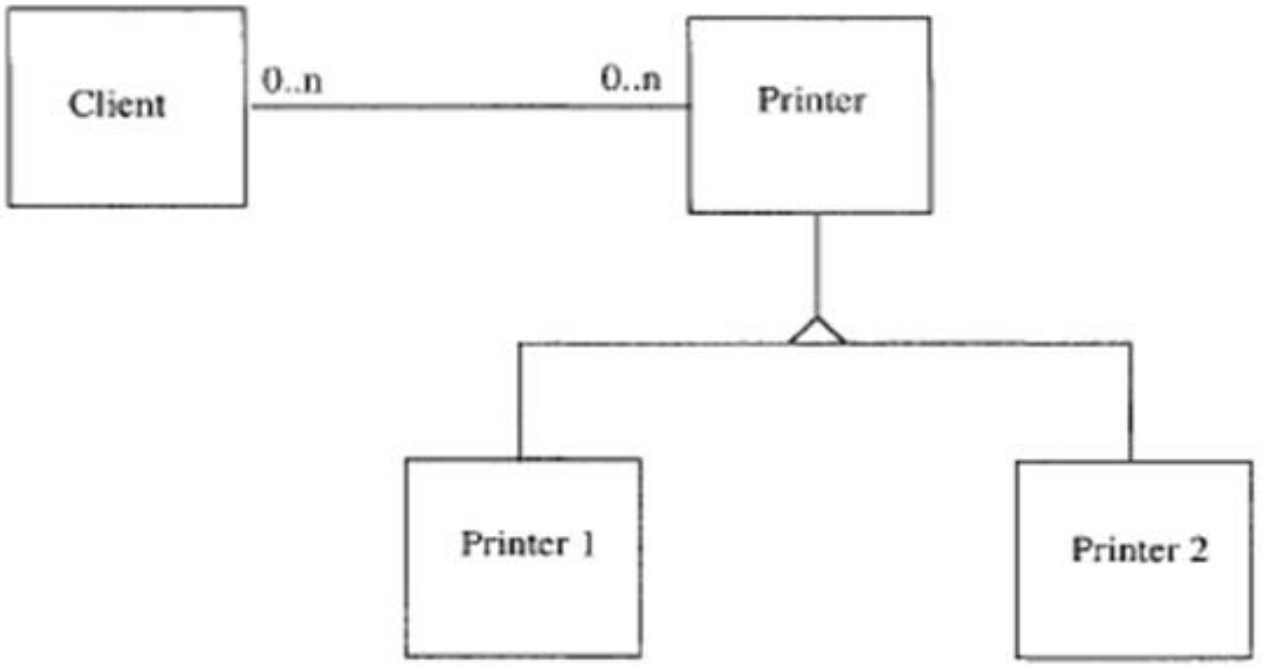


Figure : Example using subtyping.

UML diagrams

- See notes on UML Diagrams

Detailed Design

PDL/ Structured English

- PDL has an overall outer syntax of a structured programming language and has a vocabulary of a natural language (English in our case).
- It can be thought of as "*structured English*.“
- As the structure of a design expressed in PDL is formal, using the formal language constructs, some amount of automated processing can be done on such designs.

PDL/ Structured English

- In the PDL program we have the entire logic of the procedure, but little about the details of implementation in a particular language.
- To implement this in a language, each of the PDL statements will have to be converted into programming language statements.
- One way to use PDL is to first generate a rough outline of the entire solution at a given level of detail. When the design is agreed on at this level, more detail can be added.

PDL/ Structured English

- This allows a successive refinement approach, and can save considerable cost by detecting the design errors early during the design phase.
- It also aids design verification by phases, which helps in developing error-free designs.
- The structured outer syntax of PDL also encourages the use of structured language constructs while implementing the design.

PDL/ Structured English

- IF construct. It is similar to the if-then-else construct of Pascal.
- For a general selection, there is a CASE statement. Some examples of CASE statements are:

  CASE OF transaction type

  CASE OF operator type
- The DO construct is used to indicate repetition. The construct is indicated by:

  DO iteration criteria

  one or more statements

  ENDDO
- The iteration criteria can be chosen to suit the problem, and unlike a formal programming language, they need not be formally stated. Examples of valid uses are:

  DO WHILE there are characters in input file

  DO UNTIL the end of file is reached

  DO FOR each item in the list EXCEPT when item is zero
- A variety of data structures can be defined and used in PDL such as lists, tables, scalar, and integers.

Logic/Algorithm Design

- The basic goal in detailed design is to specify the logic for the different modules that have been specified during system design.
- Specifying the logic will require developing an algorithm that will implement the given specifications.
- The starting step in the design of algorithms is *statement of the problem*.
- The next step is the design of the algorithm. During this step the data structure and program structure are decided.
- The *stepwise refinement* technique breaks the logic design problem into a series of steps, so that the development can be done gradually.
- During refinement, both data and instructions have to be refined.
- The stepwise refinement technique is a top-down method for developing detailed design.

State Modeling of Classes

- A class is not a functional abstraction and cannot be viewed as an algorithm.
- An object of a class has some state and many operations on it.
- To better understand a class, the relationship between the state and various operations and the effect of interaction of various operations have to be understood.
- A method to understand the behavior of a class is to view it as a finite state automata (FSA).
- An FSA consists of states and transitions between states, which take place when some events occur.

State Modeling of Classes

- When modeling an object, the state is the value of its attributes, and an event is the performing of an operation on the object.
- A state diagram relates events and states by showing how the state changes when an event is performed.
- A state diagram for an object will generally have an initial state, from which all states in the FSA are reachable.

State Modeling of Classes

- A state diagram attempts to represent only the logical states of the object.
- A logical state of an object is a combination of all those states from which the behavior of the object is similar for all possible events.
- The finite state modeling of objects is an aid to understand the effect of various operations defined on the class on the state of the object.

Decision Trees & Decision Tables

- Decision tables provide a useful way of viewing and managing large sets of similar business rules.
- Decision tables are composed of rows and columns.
- Each of the columns defines the conditions and actions of the rules.
- Decision tables is a list of causes (conditions) and effects (actions) in a matrix, where each column represents a unique rule.
- The purpose is to structure the logic for a specific specification feature.

Decision Trees & Decision Tables

- A Decision Table can be constructed from the following steps:
  a) Identify the Conditionals (Purchase Amount, Number of Items, etc.) and put each of them in a separate row in the leftmost column.
  b) Identify the Actions (Shipping Charge) and put each of them in the leftmost column beneath the Conditionals.
  c) Identify the Rules and match them to the Conditionals. Each rule is given a separate column.
  d) If necessary, reorder the table so that the Conditionals with the fewest rules are above the Conditionals with more rules.

Example

- If the number of items is 3 or less:

Delivery Day	Shipping Charge
Next Day	$35.00
2nd day	$15.00
Standard	$10.00

- If the number of items is 4 or more:

Delivery Day	Shipping Charge, N = number of items
Next Day	N * $7.50
2nd day	N * $3.50
Standard	N * $2.50

Example

- For orders over $250.00, the shipping charge will be calculated as follows:
- If the number of items is 3 or less:

Delivery Day	Shipping Charge
Next Day	$25.00
2nd day	$10.00
Standard	N * $1.50

- If the number of items is 4 or more:

Delivery Day	Shipping Charge, N = number of items
Next Day	N * $6.00
2nd day	N *$2.50
Standard	FREE

Decision Table Equivalent

Calculating Shipping Charges

Purchase Amount	Over $250.00						Less Than $250.00					
Number of Items	3 or less			4 or more			3 or less			4 or more		
Delivery Day	Next	2nd day	Std.	Next	2nd day	Std.	Next	2nd day	Std.	Next	2nd day	Std.
Shipping Charge ($)	25	10	N * $1.50	N * $6.00	N * $2.50	FREE	35	15	10	N * $7.50	N * $3.50	N * $2.50

Verification

- Design Walkthroughs
- Critical Design Review
- Consistency Checkers
- Cyclomatic Complexity

Design Walkthroughs

- It is a manual method of verification.
- A design walkthrough is done in an informal meeting called by the designer or the leader of the designer's group.
- The walkthrough group is usually small and contains, along with the designer, the group leader and/or another designer of the group.
- The designer might just get together with a colleague for the walkthrough or the group leader might require the designer to have the walkthrough with him.
- In a walkthrough the designer explains the logic step by step, and the members of the group ask questions, point out possible errors or seek clarification.
- A benefit of walkthroughs is that in the process of articulating and explaining the design in detail, the designer himself can uncover some of the errors.
- Walkthroughs are essentially a form of peer review. Due to its informal nature, they are usually not as effective as the design review.

Critical Design Review

- The purpose of critical design review is to ensure that the detailed design satisfies the specifications laid down during system design.
- The critical design review process is same as the inspections process in which a group of people get together to discuss the design with the aim of revealing design errors or undesirable properties.
- The review group includes, besides the author of the detailed design, a member of the system design team, the programmer responsible for ultimately coding the module(s) under review, and an independent software quality engineer.
- While doing design review it should be kept in mind that the aim is to uncover design errors, not try to fix them. Fixing is done later.
- The use of checklists, is considered important for the success of the review.
- Checklists can be used by each member during private study of the design and during the review meeting.

Critical Design Review

- Typical checklist questions:
  - Does each of the modules in the system design exist in detailed design?
  - Are there analyses to demonstrate that the performance requirements can be met?
  - Are all the assumptions explicitly stated, and are they acceptable?
  - Are all relevant aspects of system design reflected in detailed design?
  - Have the exceptional conditions been handled?
  - Are all the data formats consistent with the system design?
  - Is the design structured, and does it conform to local standards?
  - Are the sizes of data structures estimated? Are provisions made to guard against overflow?
  - Is each statement specified in natural language easily code-able?
  - Are the loop termination conditions properly specified?
  - Are the conditions in the loops OK?
  - Are the conditions in the if statements correct?
  - Is the nesting proper?
  - Is the module logic too complex?
  - Are the modules highly cohesive?

Consistency Checkers

- Consistency checkers are essentially compilers that take as input the design specified in a design language (PDL in our case).
- Consistency checkers cannot produce executable code.
- A consistency checker can ensure that any modules invoked or used by a given module actually exist in the design and that the interface used by the caller is consistent with the interface definition of the called module.
- It can also check if the used global data items are indeed defined globally in the design.
- These tools can be used to compute the complexity of modules and other metrics, because these metrics are based on alternate and loop constructs, which have a formal syntax in PDL.

Consistency Checkers

- Trade-off : checking done during design vs. flexibility of design language
- Many of the metrics that are traditionally associated with code can be used effectively after detailed design.
- When logic of modules is available after detailed design, one can talk about the complexity of a module.
- Complexity metrics are applied to code, but they can easily be applied to detailed design as well.

Cyclomatic Complexity

- It is generally recognized that conditions and control statements add complexity to a program.
- Given two programs with the same size, the program with the larger number of decision statements is likely to be more complex.
- The simplest measure of complexity, then, is the number of constructs that represent branches in the control flow of the program, like if then else, while do, repeat until, and goto statements.
- A more refined measure is the cyclomatic complexity measure proposed by **McCabe**, which is a graph-theoretic-based concept.
- For a graph G with *n nodes*, *e edges*, *and p connected components*, the cyclomatic number V (G) is defined as:

$$\mathbf{V(G) = e - n + p}$$

Cyclomatic Complexity

- McCabe proposed that the cyclomatic complexity of modules should, in general, be kept below 10.
- The cyclomatic number can also be used as a number of paths that should be tested during testing.

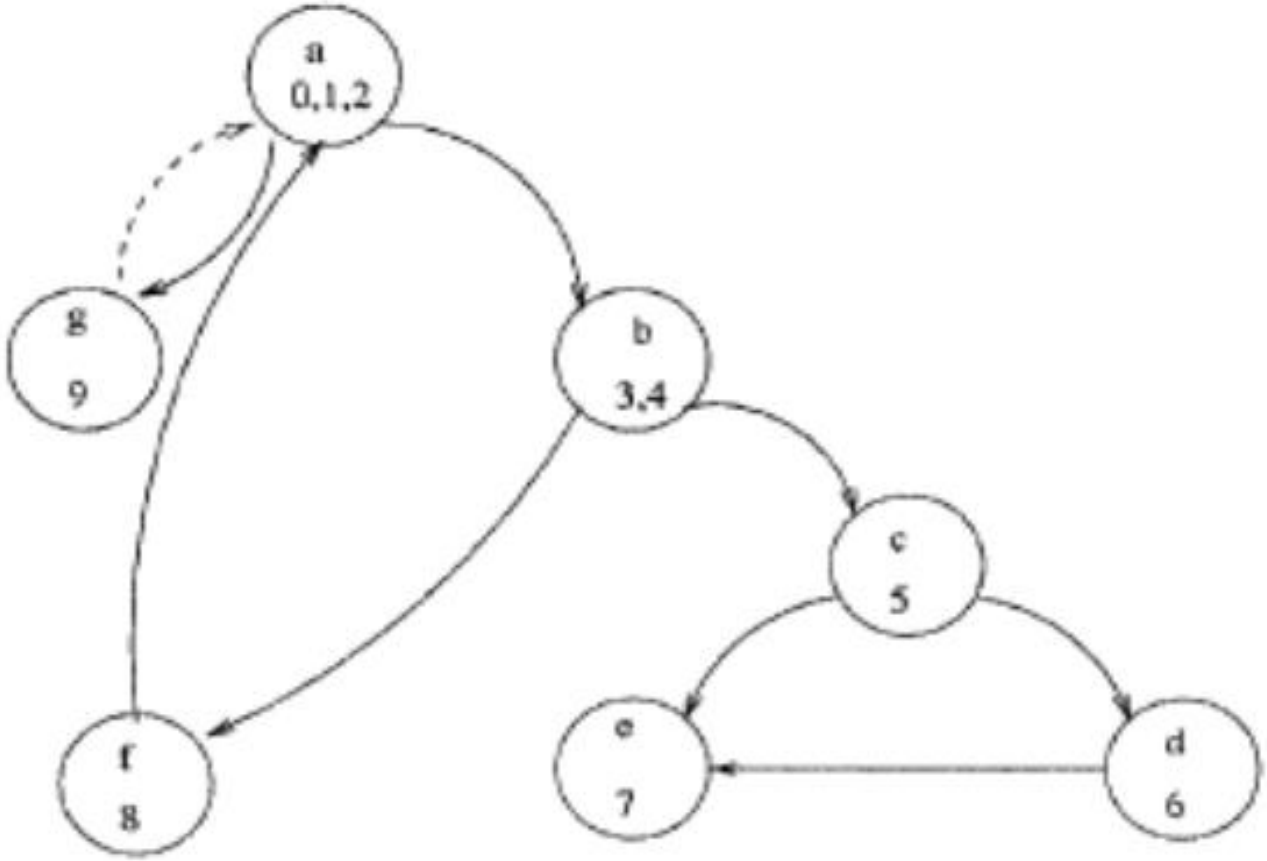
Cyclomatic Complexity

- To define the cyclomatic complexity of a module, the *control flow graph* G of the module is first drawn.
- To construct a control flow graph of a program module, break the module into blocks delimited by statements that affect the control flow, like **if, while, repeat**, and **goto**.
- These blocks form the nodes of the graph.
- If the control from a block **i** can branch to a block **j**, then draw an arc from node **i** to node **j** in the graph.

Example

```
0. {
1.     i = 1;
2.     while (i <= n) {
3.         j = i;
4.         while (j <= i) {
5.             if (A[i] < A[j])
6.                 swap(A[i], A[j]);
7.             j = j + 1; }
8.     i = i + 1; }
9. }
```

Example



Example

- V (G) = 1 0 - 7 + 1 = 4.
- The independent circuits are:
  - ckt 1: b c e b
  - ckt 2: b c d e b
  - ckt 3: a b f a
  - ckt 4: a g a

Example

- It can also be shown that the cyclomatic complexity of a module is the number of decisions in the module plus one, where a decision is effectively any conditional statement in the module.
- Hence, we can also compute the cyclomatic complexity simply by counting the number of decisions in the module.
- For this example, as we can see, we get the same cyclomatic complexity for the module if we add 1 to the number of decisions in the module.(The module has three decisions: two in the two while statements and one in the if statement.)

End…