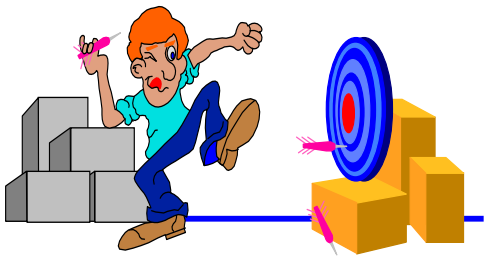
Object Oriented Analysis and Design: The Unified Modeling Language (UML)



Programming in the Small

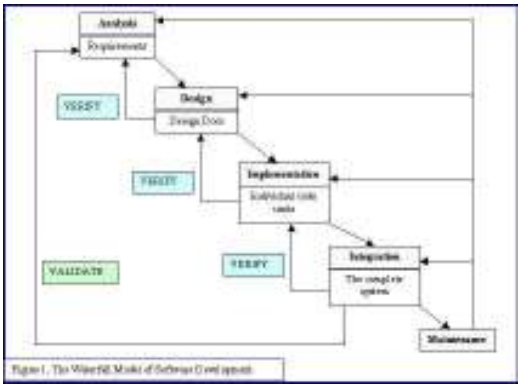
- What you have done so far
- Code is developed by one programmer
  - Maybe a small close knit group
- One person can understand the entire system
  - To them the system is self-documenting
  - The creator is the maintainer
- Designed to solve a particular problem
- System does not have a long life cycle
- The biggest problem is getting it done on time

Programming in the Large

- What you will be doing in the real world
- Developed by a large team of programmers with a lot of "help" from management
- Nobody really knows what is really going on inside all parts of the system
  - Everybody has their little piece
- The system is designed to solve a "systems" level problem
- Ideally, the system will be around for a long time
- Communication amongst developers is the biggest problem

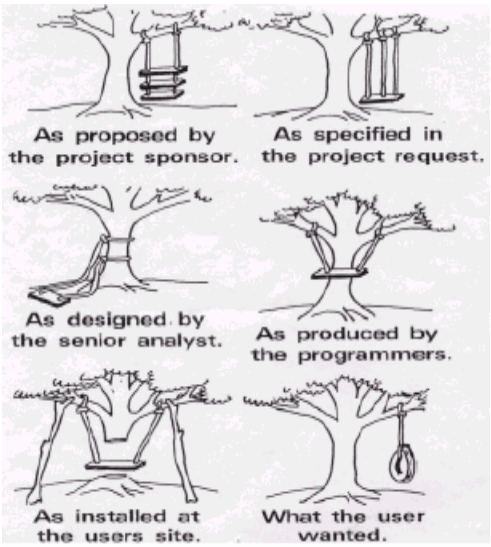
Software Development

- The creation of software involves four basic activities:
  - establishing the requirements
  - creating a design
  - implementing the code
  - testing the implementation
- The development process is much more involved than this, but these basic steps are a good starting point
  - There are always multiple ways to design and implement a program
  - Any design has advantages and disadvantages; there are always trade-offs



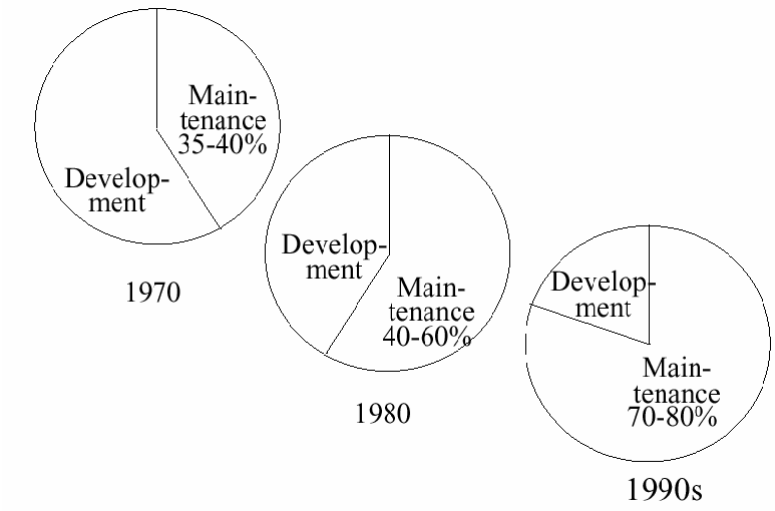
The Software Crisis

- Technological advances outpace ability to build software
- Demand is more than ability to build new programs
- Society increasingly depends on and expects high quality software
- Ability to maintain programs is threatened by poor design and few resources
- and more …..



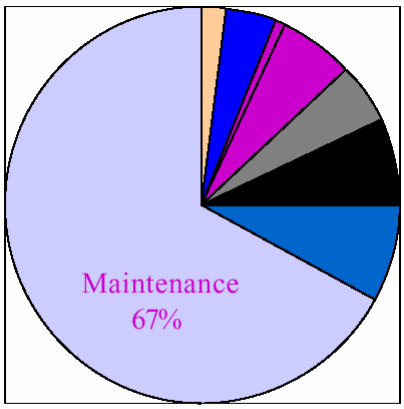
5

Software Maintenance Costs



6

Relative Costs



- Requirement 2%
- Specification 4%
- Planning 1%
- Design 6%
- Module Coding 5%
- Module Testing 7%
- Integration 8%
- Maintenance 67%

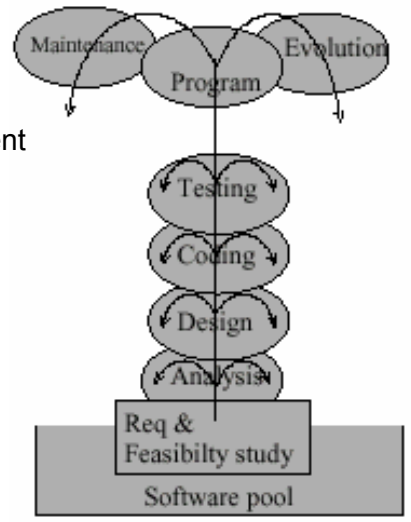
Approximate relative costs of
the phases of the software life cycle.

Maintenance Factors: 70% of System Cost

- Changes in User Requirements: 41.8%
- Changes in Data Formats: 17.4%
- Emergency Fixes: 12.4%
- Routine Debugging: 9%
- Hardware Changes: 6.2%
- Documentation: 5.5%
- Efficiency Improvements: 4%

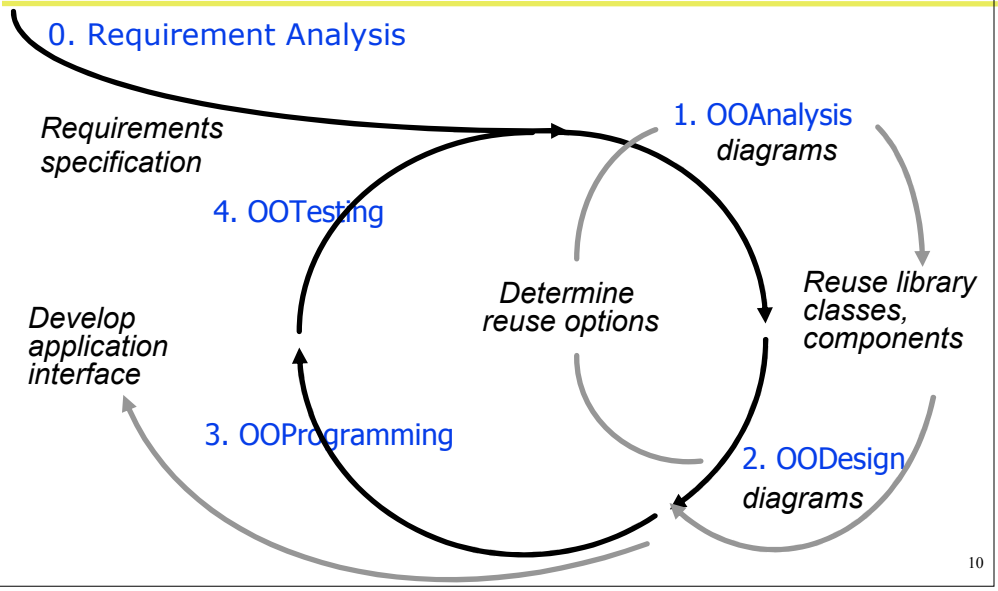
The Object Oriented Approach

- Software Development involves
  - Analysing a problem
  - Developing a solution
- Issues common to any form of development
  - Identifying boundaries of a system
  - Setting user's requirements
- OO approach does not alter these
- Characteristics which make it attractive
  - An object view of the problem domain
  - Seamlessness of development
  - Resilience to change
  - Emphasis on reusability



9

One Cycle of the Object-Oriented Development

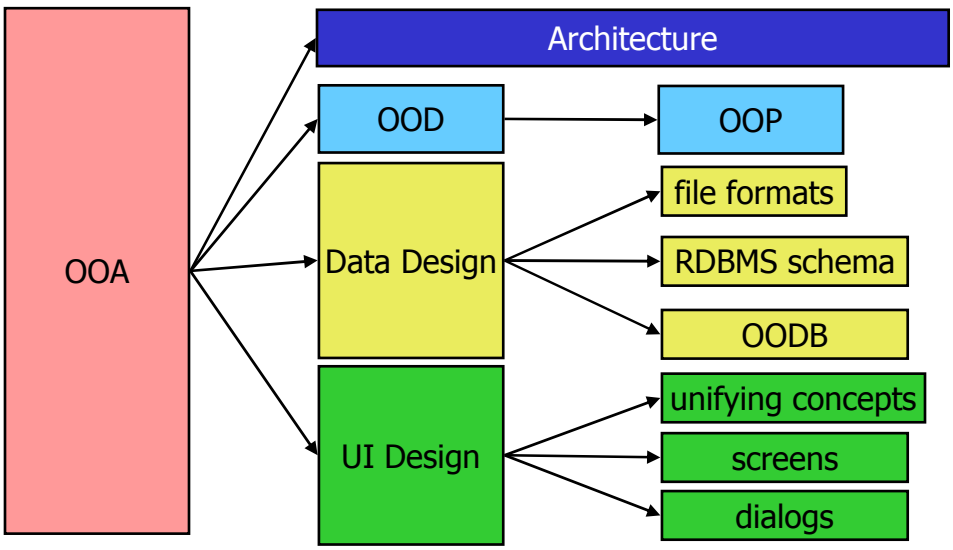


10

OOA, OOD, OOP

- OOA is concerned with developing an object model that captures the requirement
  - examines the requirements of a problem through the classes and objects found in the vocabulary of the problem domain
- OOD is concerned with translating the OOA model into a specific model that can be implemented by software
  - leads to Object Oriented decomposition and uses logical and physical notations to represent the static and dynamic aspects of the system
- OOP is concerned with realising the OOD model using an OO programming language such as Java or C++
  - uses objects not algorithms: each object is an instance of a class which is related to another via inheritance relationships

OOA is a Prerequisite to many Design Tasks



O-O Analysis and Design Deliverables

- OOAnalysis
  - Use Cases
  - Use Case Diagrams
  - Static Structure
  - Diagrams (Class Diagrams)
  - State Diagrams
- OODesign
  - Revised Analysis
  - Sequence Diagrams
  - Collaboration Diagrams
  - Component Diagrams
  - Implementation Diagrams
- User Interface Designs
  - Screens
  - Menus
  - Storyboards
  - Reports
- Architectural Models
- Network Models
- System Models

Why is the Word "Model" Important?

- Model
  - an abstraction of something for the purpose of understanding it before building it
  - omits nonessential details
- An abstraction or view is a model
  - For example, a class is an abstraction of a real-world entity or concept
  - There are model types in building a system
- UML Models capture
  - the structural, or static, features of systems
  - the behavioral, or dynamic, features of systems.
- UML Models have several independent dimensions
  - Each emphasize particular qualities of a model
  - Each dimension has a diagram type

UML has 9 kinds of Diagrams

1. Class Diagram
2. Object Diagram
3. Component Diagram
4. Deployment Diagram
5. Use Case Diagram
6. Sequence Diagram
7. Collaboration Diagram
8. Statechart Diagram
9. Activity Diagram

Structural Diagrams

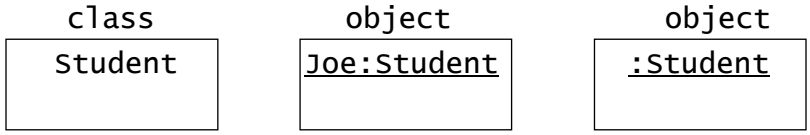
Behavioral Diagrams

A 3-step Design Process

- Identify the classes that need to be part of the system
- Assign responsibilities to each class
- Identify the relationships between various classes

Classes and Objects

- Class: A rectangle including the name of the class
- Object: A rectangle including the name of the object and the name of the class separated by colon



- Classes and objects are described by nouns

Sample problem

- Design a program that allocates students to tutorials and lectures
- The program is given two files
  - one lists the students and the subjects they are enrolled in
  - the other is a list of tutorials and lectures for a given subject
- Generate a report of tutorials each student is allocated to

- Invalid classes:
  - program
  - files
  - report
- Valid classes:
  - Student
  - Tutorial
  - Lecture
  - Subject

Determine the Responsibilities of each Class

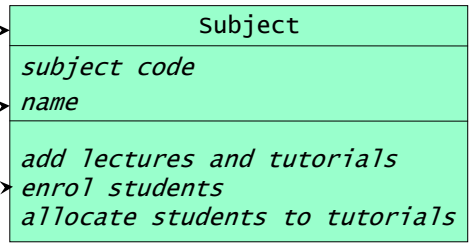
- The responsibility or role of a class is the duties it is responsible for
- For example, each subject is responsible for allocating enrolled students into tutorials
- The roles of each class of objects also become the methods of each class
- Roles can be identified by listing the verbs

Sample problem

- Design a program that allocates students to tutorials and lectures
- The program is given two files
  - one lists the students and the subjects they are enrolled in
  - the other is a list of tutorials and lectures for a given subject
- Generate a report of tutorials each student is allocated to
- Roles can then be assigned to classes
- Subject
  - allocate students to tutorials
- Student
  - generate report of tutorials for each student
- Other roles may also be identified to achieve the goals of the program
- Subject
  - add lectures and tutorials
  - enrol students
- Tutorials
  - add students

Classes

- Represents sets of objects
- Defines
  - Name
  - Attributes
  - Operations



UML syntax for attributes

visibility name [multiplicity] : type = initial value {property}

+ ports [2…*] : Port = null {addOnly}

- visibility: public (+), protected (#), private (-)
- name: string
- multiplicity: any valid multiplicity (see next)
- type: language-dependent specification

- property
  - changeable: default,freely modifiable
  - addOnly: may add new values; no changes allowed
  - frozen: the value may not change after the object is initialized

Examples

- name : String='Unknown'
- birthDate : Date
- radius : Integer = 25 {readonly}{radius > 0}
- -counter : Integer
- time : DateTime::Time
- dynamArray[*]
- name[1] : String
- firstName[0..1] : String
- firstNames[1..5] : String

UML syntax for operations

visibility name (parameter list) : return-type-expression {property}

+ assignAgent (a : Agent) : Boolean

- visibility: public (+), protected (#), private (-)
- name: string
- parameter list: arguments (name : type = default value)
- return-type-expression: language-dependent specification

- property
  - isQuery: does not change the state of the system
  - sequential: does not protect against multiple threads
  - guarded: does protect against multiple threads
  - concurrent: multiple threads can execute it at the same time

Examples

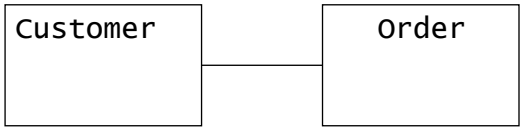
- position(x,y)
- position(x : Integer, y : Integer)
- resize( )
- resize : GeomFigure( )
- resize(byFactor : Real) : GeomFigure
- dataHasChaged( )
- checkDataChange( )

Identify the Relationships between various Classes

- UML class and object diagrams
- Either class diagram or object diagram. We cannot mix classes with objects
- Shows the static structure of the system
  - Static relationships
    - Association
      (e.g.: a company has many employees)
    - Generalization (subtypes)
      (e.g.: an employee is a kind of person)
    - Dependencies
      (e.g.: a company is using trucks to ship products)

Associations

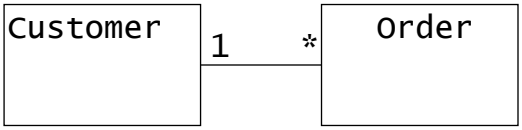
- Associations represent relationships between objects (class instances)
- In the simplest case association can be drawn as a line



- This is called binary association because there is a relationship between two classes
  - In the general case n-ary associations are allowed

Associations

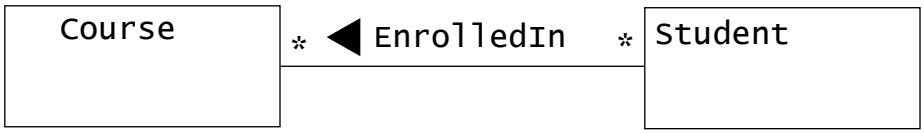
- From a conceptual perspective associations represent semantic relations between classes (e.g. an Order comes from a single Customer but a Customer may launch several Orders)
- Each end of an association has a multiplicity: how many object can participate in the relationship: *(many), 1(exactly one), 0(none), 0..1(none or one), 1..*(at least one)



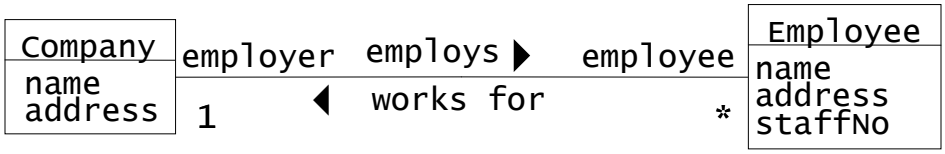
- Other valid multiplicities:
  - 0..4
  - 3, 7
  - 0..* (default)
  - 0..3, 7, 9..*
  - e.t.c

Associations

- Sometimes an association is given a name to make its meaning explicit
- Example: A Student is enrolled in a course



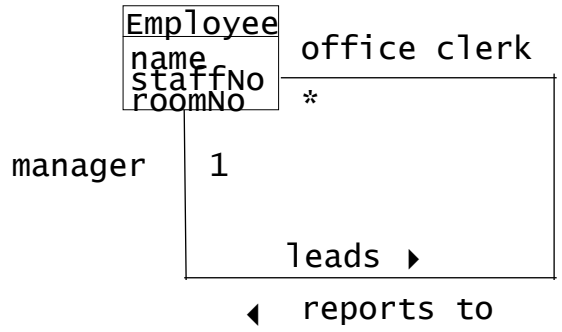
- Sometimes the classes of the objects that participate in the relationship are assigned roles explicitly



29

Recursive associations

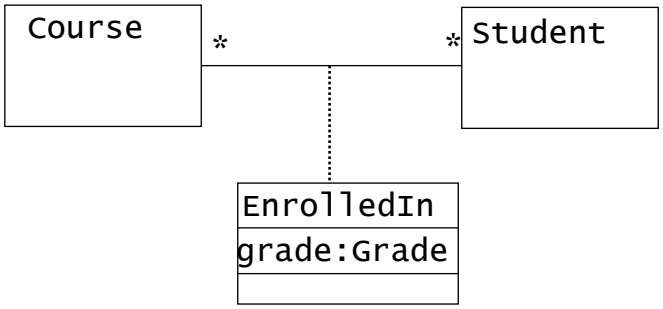
- Recursive associations are associations in which only one class is involved
- Assigning roles explicitly is important in recursive association



30

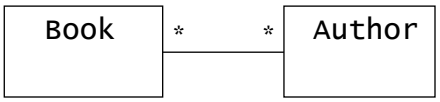
Attributed association

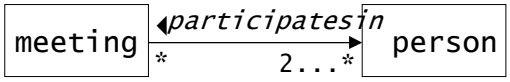
- Sometimes there are one or more attributes that are of the association itself rather than of the participating objects
- In this case we have an association class rather than just a name



Directed association

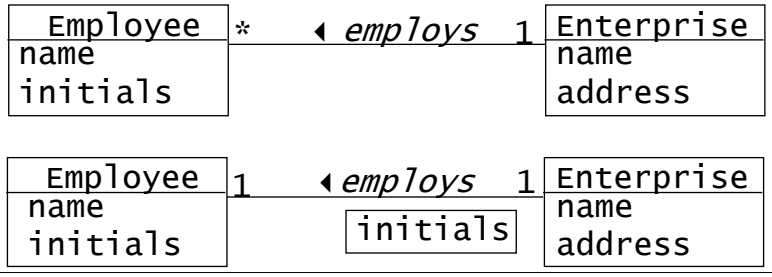
- By default association is bidirectional
  - We want to be able to go from a book to its authors and from an author to the books he has written
- Directed association is an association in which you can directly navigate from one of the involved association roles to the other, but not vice versa.
  - From a meeting you can navigate to all meeting persons but from a person you cannot find all the meetings he participates in





Qualified associations

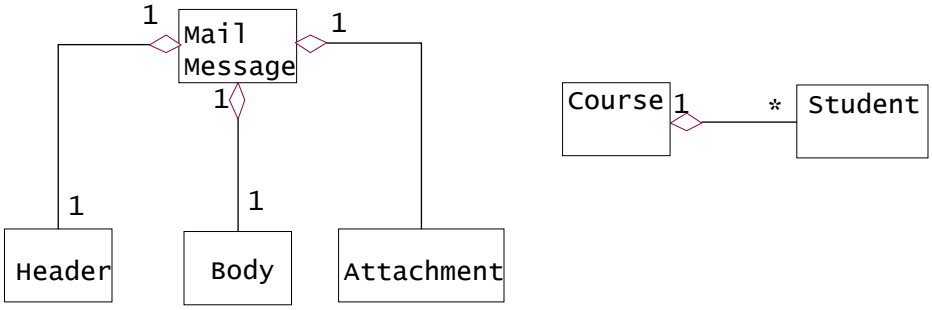
- A qualified association is an association in which qualifying attributes are used to subdivide the referenced set of objects into partitions, where each partition may occur only once
- A class has an attribute that acts as a key to the objects (instances of the class) that are created
- Each instance of the class with a certain value for the "key attribute" may occur only once



Aggregation

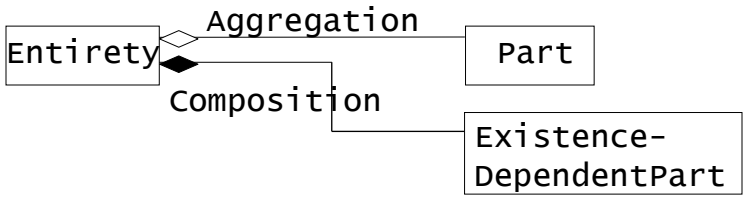
- A special case of association in which the involved classes represent an whole-part hierarchy.
- A class is formed as a collection of other classes. i.e the relationship is between a whole and its constituent parts
- It is meaningful to use phrase "is part of" to describe the relationship of the "part" with the "whole"
- It is meaningful to use phrase "has A" to describe the relationship of the "whole" with a "part"

Aggregation

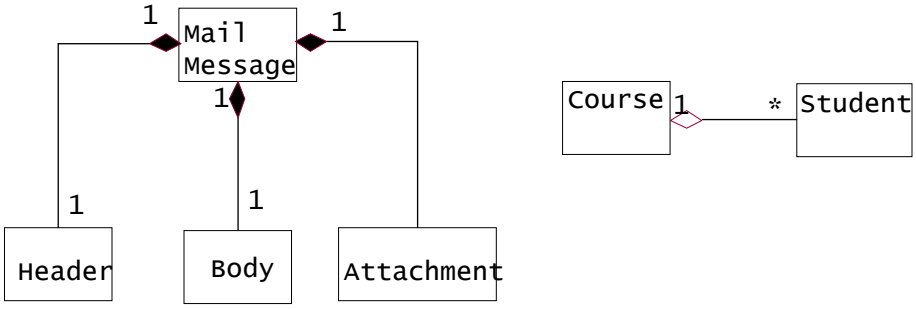


Composition

- It is a special case of aggregation in which the “parts” have no existence independent of the “whole”
  - Each “part” can belong to only one “whole”
  - The “whole” is responsible for creating and destroying the “parts”



Composition

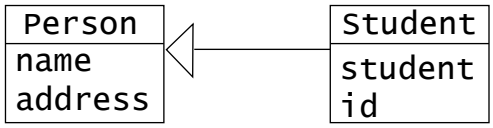


When is association used?

- An object of one class *uses* the services of an object of another object, or they mutually use each others services
- Associations can be used to represent aggregation or composition - where objects of one class are wholes that are composed of objects of the other class as parts
- Associations can also be used to represent a situation in which objects are related, even though they don't exchange messages. This typically happens when at least one of the objects is basically used to store information
  - E.g. an object of type AddressBook stores objects of type Person without sending them any message

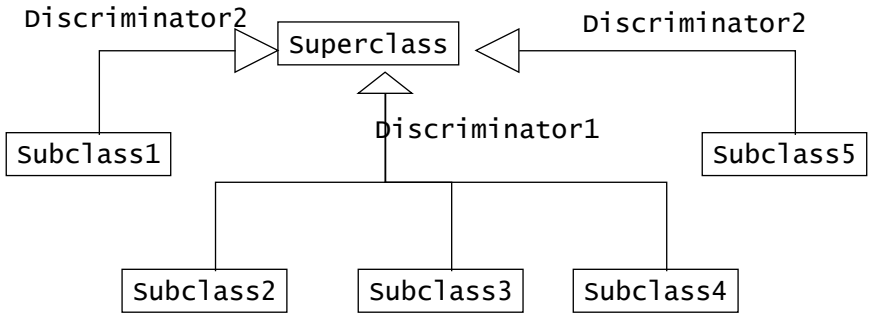
Generalization, Specialization

- Is a relationship between classes
- Inheritance is a relation between superclasses and subclasses which enables attributes and operations of a superclass to become accessible to its subclasses.
- Generalization or specialization according to the point of view
- Superclass/ Subclass
- If class B inherits from class A it is legitimate to say that "B isA A"
- Example: class Student extends class Person with an attribute "student id"

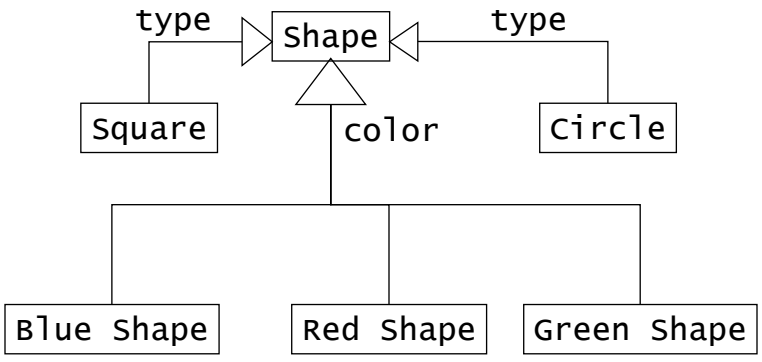


Generalization, Specialization

- *Discriminator* denotes the aspect relevant for hierarchical structuring of the properties
- *Partition* is an set of subclasses based on the same discriminator

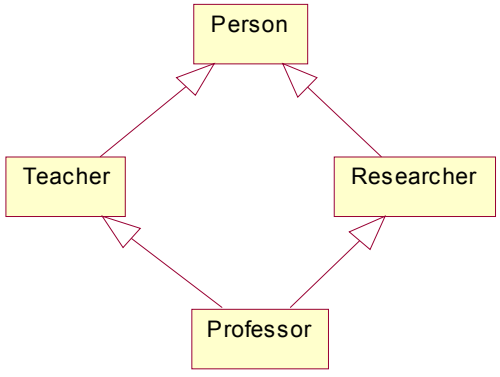


Example



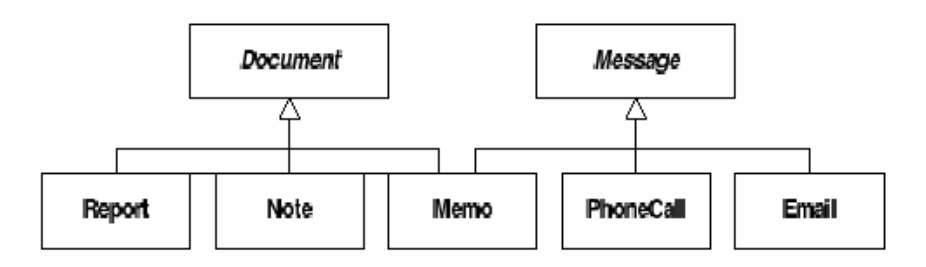
Multiple Inheritance

- A subclass may have more than one superclasses
- OO Languages like Java do not support multiple inheritance



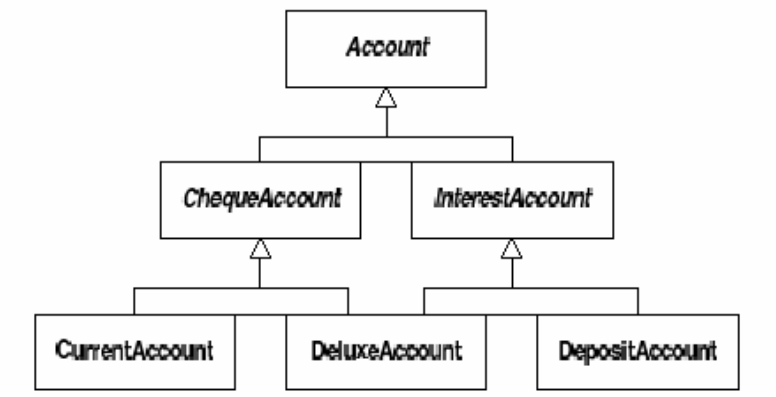
Multiple Inheritance

- Name collision: two or more different superclasses use the same name for some of their properties
  - Resolve this ambiguity: Use different names
  - Example: If both class Document and class Message have an attribute "name" they should change it to "document_name" and "message_name" respectively



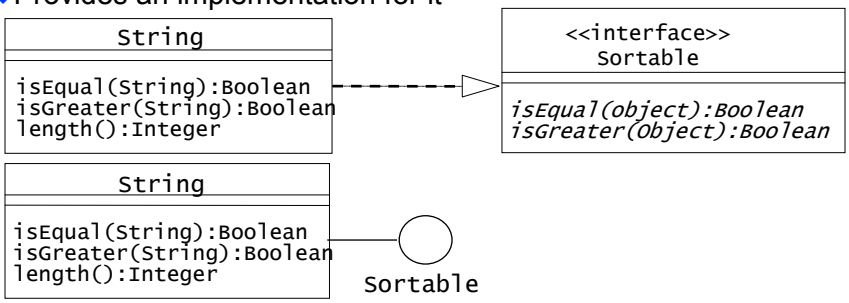
Multiple Inheritance

- Repeated inheritance
  - Only one copy of the attributes and operations of the common superclass



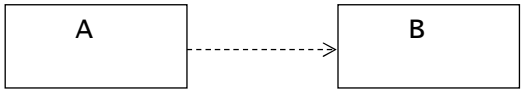
Realization

- It is similar to inheritance
- When B inherits from A it inherits
  - Interface (specification)
  - A's implementation
- When B realizes A it
  - Inherits the interface
  - Provides an implementation for it



Dependency

- It is a relationship between classes
- Dependency is a relation between two model elements which shows that a change in one element requires a change in the other
- Less formally: We say that a class A depends on a class B if a change to class B's interface could necessitate a change to class A
- Example: Class A may be using a method of class B that no longer exists. Class A depends on class B



Dependency types

- bind:specifies that the source instantiates the target template using the given actual parameters
- derive: specifies that the source may be computed from the target
- friend: specifies that the source is given special visibility into the target
- instance-of: specifies that the source is an instance of the target
- instantiate: specifies that the source creates instances of the target
- powertype: specifies that the target is a powertype (all children of a given parent)
- refine: specifies that the source is at a finer degree of abstraction than the target
- use: specifies that the semantics of the source depends on the semantics of the public part of the target

47

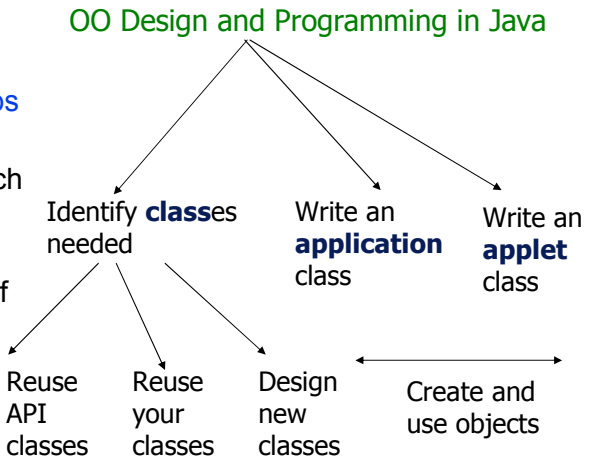
Dependency relations occur when…

- The dependent class has a method that takes an object of the class it depends on as a parameter, and uses that object in some way in implementing the method
- The dependent class has a method that returns as its value an object of the class it depends on
- The dependent class creates an object of the class it depends on, but only uses it within one method (doesn't keep a reference to it as an instance variable - if it did, we would have an association)
- In Java, usage dependencies typically show up in the signatures of methods - as the type of a parameter or a return value - but the object in question is not stored as an instance variable

48

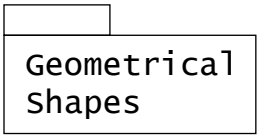
Object-oriented Design and Programming: UML and Java

- Designing an object-oriented program requires 5 steps:
  - Identify the classes
  - Determine the relationships between classes
  - Determine the roles of each class (what each class is responsible for)
  - Determine the attributes of each class
  - Determine superclasses and subclasses
  - Refine object design



Packages

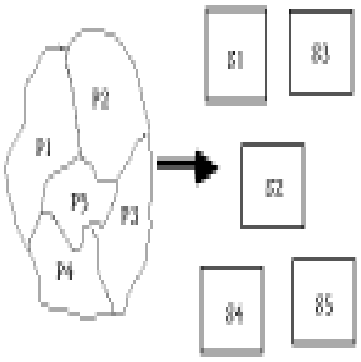
- Divide and conquer
- Break a large system into smaller and more manageable pieces
- UML does this using packages
- An UML package may include
  - Classes
  - Interfaces
  - Use Cases
  - Interactions
  - Diagrams
  - Other packages



- An UML package dependency diagram shows the various packages and the dependencies between them
- A package B depends on some package A when some entity in package B depends on some entity in package A

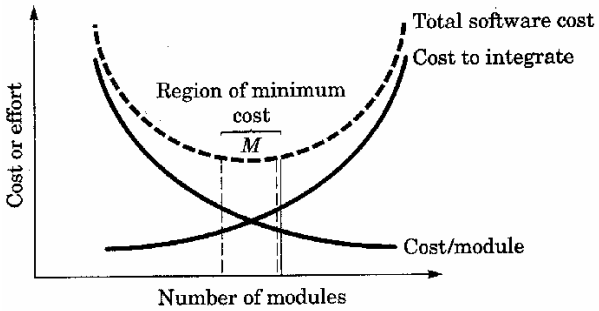
More on Modularity

- Modularity is based on what we understand from research on problem solving
  - The idea is to take a large problem and decompose it into sub-problems
  - The hope is the sub-problems are easier to deal with than the original problem
- Modules should be characterized by design decisions that each hides from all others
  - Modules should be specified and designed so that information within a module is inaccessible to other modules that have no need for such information



More on Modularity

- Partitioning a program into individual components which can be compiled separately, but which have connections with other modules
  - Fundamental to the issue of modularity is what is contained in a module and what type/kind/ quantity of connections
- Modules are intended to package program functionality
  - Modules should localize program behavior



Objectives of Modularity

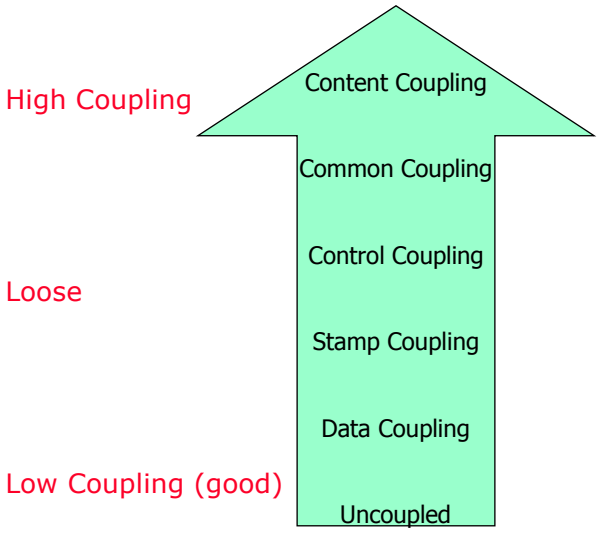
- Modular Decomposability
  - Provide a systematic mechanism to decompose a problem into sub problems
- Modular Composability
  - Enable reuse of existing components
- Modular Understandability
  - Can the module be understood as a stand alone unit? Then it is easier to understand and change
- Modular Continuity
  - If small changes to the system requirements result in changes to individual modules, rather than system-wide changes, the impact of the side effects is reduced
- Modular Protection
  - If there is an error in the module, then those errors are localized and not spread to other modules

Measuring Modularity

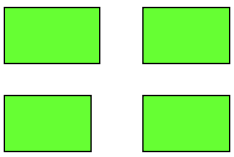
- Cohesion
  - Modules are characterized as being highly cohesive, which implies all the items in the module are there to serve a single purpose (the purpose of the module)
- Coupling
  - Modules are loosely coupled to the outside world, which means there are few connections

Coupling

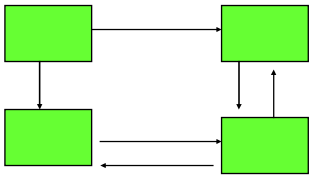
- A measure of the degree of independence between modules
- When there is little interaction between two modules, the modules are described as loosely coupled
- When there is a high degree of interaction the modules are described as tightly coupled



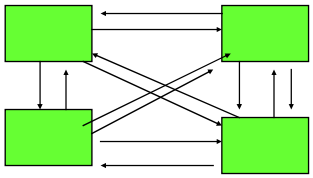
Coupling and Dependency



Uncoupled



Loosely Couple: Some Dependencies



Highly Couple: Many Dependencies

Types of Coupling

1. Content Coupling: Two modules are content coupled if one module refers to or changes the internals of another module

2. Common Coupling: Two modules are common coupled if they share the same global data areas

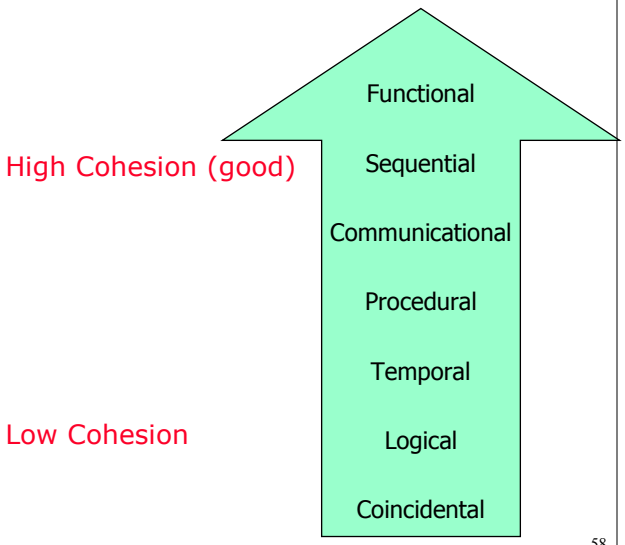
3. Control Coupling: Two modules are control coupled if data from one is used to direct the order of instruction execution in the other

4. Stamp Coupling: Two modules are stamp coupled if they communicate via a composite data item

5. Data Coupling: Two modules are data coupled if they communicate via a variable or array that is passed directly as a parameter between the two modules. The data is used in problem related processing, not for program control purposes

Cohesion

- A measures of how strongly the elements within a module are related
- The stronger the better



Types of Cohesion

1. Functional: Each element in a module is a necessary and essential part of one and only one function
2. Sequential: The elements of a module are related by performing different parts of a sequence of operations where the output of one operation is the input to the next
3. Communicational: The elements of a module all operate on the same data
4. Procedural: The elements of a module are all part of a procedure
5. Temporal: The elements of a module are related by time but need not occur in a certain order or operate on the same data
6. Logical: The elements of a module are all oriented toward performing a certain class of operations
7. Coincidental: The elements of a module are essentially unrelated by any common function, procedure, data, or anything

Separation of Concerns

Separation of concerns is at the core of software engineering. In its most general form, it refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular concept, goal, or purpose. Concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts. The prevalent kind of concern in object-oriented programming is data or class; each concern in this dimension is a data type defined and encapsulated by a class. Features like printing, persistence, and display capabilities, are also common concerns, as are aspects, like concurrency control and distribution, roles, viewpoints, variants, and configurations. Achieving a “clean” separation of concerns has been hypothesized to reduce software complexity and improve comprehensibility; promote trace-ability within and across artifacts and throughout the software lifecycle; limit the impact of change, facilitating evolution and non-invasive adaptation and customization; facilitate reuse; and simplify component integration. Modern languages and methodologies, however, suffer from a problem we have termed the “tyranny of the dominant decomposition”: they permit the separation and encapsulation of only one kind of concern at a time. Examples of tyrant decompositions are classes (in object-oriented languages), functions (in functional languages), and rules (in rule-based systems). It is, therefore, impossible to encapsulate and manipulate, for example, features in the object-oriented paradigm, or objects in rule - based systems. Thus, it is impossible to obtain the benefits of different decomposition dimensions throughout the software lifecycle.

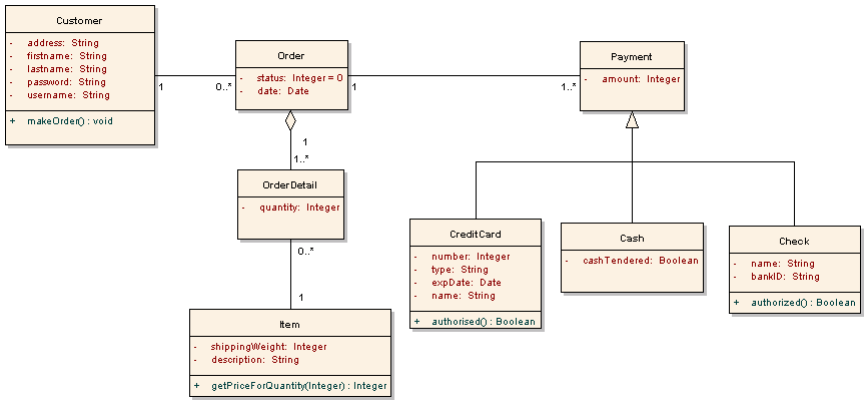
Example “e-shop” (1/3)

- Θέλουμε να μοντελοποιήσουμε μ’ένα class diagram το τμήμα πωλήσεων ενός ηλεκτρονικού καταστήματος.
  - Ποιες κλάσεις ?
  - Μεταβλητές (ιδιότητες) κλάσεων ?
  - Μέθοδοι κλάσεων ?
  - Σχέσεις μεταξύ των κλάσεων ?

Example “e-shop” (2/3)

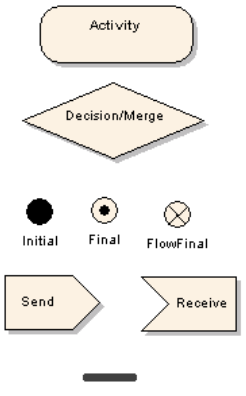
- Classes: customer, order, orderDetail(part of order), item, payment
  - Customer: lastname,firstname,address,username,password - makeOrder
  - Order: status, date
  - OrderDetair: quantity
  - Payment: amount
    - Credit: number,type,expDate,name - authorized
    - Cash: cashTendered
    - Check: name,bankID - authorized
  - Item: shippingWeight, description - getPriceForQuantity

Example “e-shop” (3/3)



Activity Diagrams

- Περιγράφει την ακολουθία των δραστηριοτήτων και υποστηρίζει συνθήκες και παραλληλία (φροντ. HY351)
- Δομικά στοιχεία ενός διαγράμματος δραστηριότητας
  - Activity
  - Partition
  - Decision/Merge
  - States
    - Initial
    - Final
    - Flow Final
  - Send/Receive
  - Region
  - Fork/Join



Example “ATM”

- Ακολουθία δραστηριοτήτων για την ανάληψη και κατάθεση χρημάτων στο ATM.