LECTURE 1

January 4, 2014

Karthik.S (IMT2012021)

Miryala Deepa(IMT2012027)

Chandan Yeshwanth(IMT2012010)

Introduction to Object Oriented Programming (Lecture Notes)

Course Information:

Books to be referred to :

Introduction to object oriented programming- by Timothy Budd C++ Primer- by B.Stanley and Lippman

Java How to Program- by Deitel and Deitel

Distribution of marks:

30%-Mid-Sem Examination

20%-Programming Tests and Assignments 15%-Projects

30%-End Sem Examination 5%- Discretion of Professor

Types of Programming Languages

Programming Languages are broadly classified into 3 types based on closeness to underlying hardware- Machine, Assembly and Higher level languages.

Elements of assembly languages:

Contain instructions which are in turn made up of opcode and operands.

Advantages- More access to hardware implies easier exploitation of the same. Also slightly more abstracted than machine level programs. Additionally, have mnemonically instructions which make coding and reading code easier.

Disadvantages- Relatively large amount of coding involved even to achieve simple tasks. It is inherently complex due to lack of modularity and poor readability. Error handling/ debugging is also difficult due to the complex-ity. Additionally there is no portability as it is closely tied to underlying hardware.

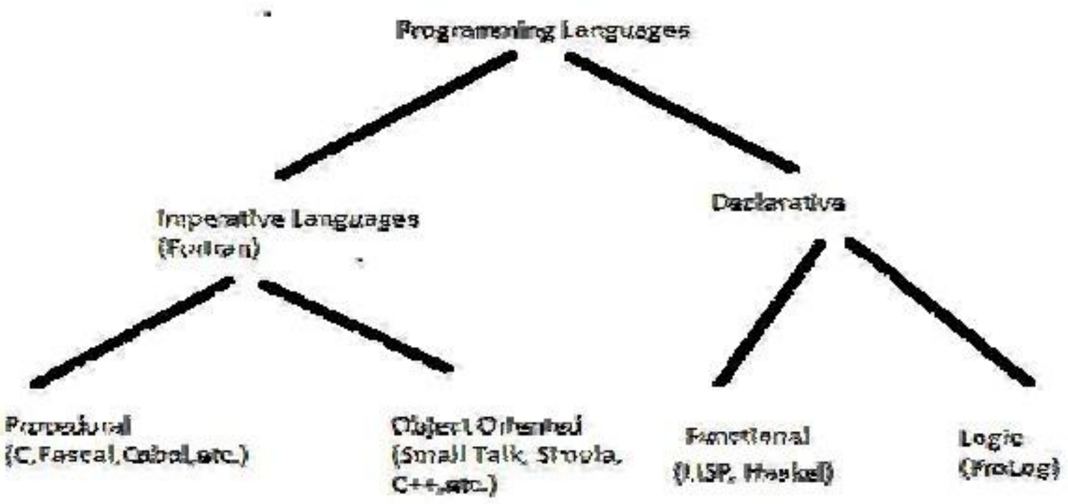
Instruction Set- Defined as the set of all instructions supported at the processor level. Recent advances in technology have enabled processors to have a large, though infinite, instruction sets.

Higher Level Languages

Portability- Defined as the ability of a program written in a given language to run in different architectures.

Higher Level Languages usually provide source code level portability due to their ability to be compiled on different architectures. This in turn implies that compilers for these languages are relatively easy to build irrespective of the architecture.

Classification is as follows.



Imperative Languages were written as one big piece of code without the use of functions. This was found to be cumbersome.

Procedural languages solved this problem by allowing subroutines. This decreased complexity and increased modularity.

Functional- Everything is a function and there are no imperative state-ments involved.

Object Oriented Programming

Object is the main unit of programming as opposed to functions. This is a paradigm shift from other types of programming languages.

There is no data passed between objects, only messages are passed. The data to be operated on is implicitly defined.

While procedural languages have behavioral centric designs, OOP languages have data centric designs.

Eg: while sorting in procedural languages involves passing the array to be sorted to a predefined sort function, in OOP languages, the sort method of a predefined object which contains the array is employed.

OOP has become the norm in software design due to relative easiness of coding.

Elements of OOP

Object: It is a combination of state and behaviour or alternately as and instance of a class. They occupy memory and are analogical to variables.

Class: It is a set containing similar objects or alternately a template for creating an object. They do not occupy memory and are analogical to data types.

State: It is modelled using attributes, members and instance variables.

Behaviour: Characterised by methods, services and responsibilities. Of these methods are the most fundamental.

The message from one object to another usually involves the invocation of methods of the objects.

From a design perspective, the program must be visualized as a collection of agents which collaborate with one another to achieve the goal.

Encapsulation: This essentially means state and behaviour are bound to one another. The only way to manipulate the data/state of an objects is by calling the method of the object. This feature is unique to OOP languages.

Encapsulation: (Unique to OOP) The ability to group state and behaviour

- Only way to manipulate state of the object is through methods.
- We need to pass the message.
- We cannot extract data out of the object and perform operations on it.

Abstraction: It is the ability of a programming language to deal with a concept without going into specifics. E.g.: Int provides an abstraction of an integer which is in turn made up of 4 bytes, thus saving the programmer the trouble of addressing each of these 4 bytes individually. Abstraction is one of the underlying principles of evolution of programming languages.

Abstraction: (not unique to OOP) Inherent property of OOP.

- Deals with many attributes as one variable.
- Evolution of data types is an abstraction.
- Abstraction of behaviour is unique to OOP.

Though abstraction is not unique to OOP languages, they provide relatively better abstraction because of their behavioural support.

This in OOP languages, abstraction and encapsulation go hand in hand to provide a better environment than procedural languages

LECTURE 2

January 11, 2014

Prateek Bansal(IMT2012032)

Pawan Dhananjay (IMT2012014)

Tanmayee Narendra (IMT2012046)

Class is basically a collection of similar objects. But when we are referring to Class we are actually talking about
(1) State - (data members or attributes of class)
(2) Behaviour - (services that Class provide)

But it is not the class that has state and behaviour, it is the object. Until you instantiate an object of class there is no memory allocated for it.

Advantages -

Data protection from accidental damage.
Easier to change something inside i.e. we can modify it anyway we want as long as we have consistent behaviour that is expected.

In some sense C function also support the property of Encapsulation.

1) For instance by defining local variables in the function or by Passing by value we can protect data.
2) As long as the function prototype remains same, function can use either of while-loop or for-loop or use iteration until the behaviour our function is expected to show is same.

Generally data members and methods are different and when we use methods we need to invoke data for the behaviour to be performed.
For eg- A normal function sort would need an array of integers or data to perform the operation on.

This is where Encapsulation is different in Object Oriented Programming. In it we need not give data to the method for the same behaviour.
We just need to do something like - Obj.sort().
As in Object Oriented Programming the data members of class can be directly used inside the methods of the same class so we need not give data to each method of the class explicitly. One advantage of this is we can change data representation and still perform the same behaviour without changing the statement of method call.

Access Specifiers

Private data members or functions - Accessible from only within the mem-ber functions(either private or public) of the class. In C++ by default everything is private.

Public data members or functions - Accessible within the class as well as from outside the class.

```
class Node
{
        int data;
        void modifyData(); public:
        void showNode();
}
```

So now inside the main() following statements would be valid or invalid -

```
Node myNode;
myNode.data = 25;                (INVALID)
cout<_<myNode.data;              (INVALID)
myNode.modifyData();             (INVALID)
```

```
    myNode.showNode();                      (VALID)
```

Constructors and Destructors

Constructor is a special method in class which is automatically called by the compiler every time a new object _comes to life_. It is used to instantiate the state of the object.

Constructors

Special method in a class which is called automatically by the compiler every time a new _object comes to life_ and has a name similar to the class name.

They initialize data members (state of an object) by assigning initial values to data members or call methods as needed. They do not have a return type.

They are defined as className::className(parameters){ body} They are of 2 types -

Built-in compiler constructor which is always called whenever an ob-ject comes to life.

I_ there is a user-defined constructor, then it is called after the built-in constructor.

There can be more than one constructor for a class. The different constructors are overloaded with the only condition that the function prototype must be different.

For class Node,

Node::Node(); and Node::Node(int myData); are 2 different constructors

Node::Node(int myData, char myChar); and Node::Node(char my-Char, int myData) are also two different constructors i.e. the order of parameters matter.

Memory allocation is done in built-in constructor and hence it is not required in user defined constructor.

Since the object is initialized just after creation, there is no question of forgetting to initialize any variable.

In Python, the constructor is the __init__ method which initializes the data members of the python class.

Destructor

Special method in a class that is always invoked by the compiler when _object is about to die_ and has a name similar to class name preceded by a tilde (~).

There can be utmost one user defined destructor per class. The destructor method may include closing opened _les or sockets etc.

They are of two types -

1) Built-in Constructor
2) User-defined Constructor (Optional)

If we don't write a constructor of our own, the compiler will create a constructor on its own internally.

Constructors do not have return type.

Another property of Constructors is that there can be more than one constructor for a class.

```
class Node
{
            int data;
    public:
        Node();                      // Default constructor
        Node(int myData);       // Parameterized constructor
};

Node :: Node()
{
      data = 0;
}

Node :: Node(int myData)
{
      data = myData;
}


main()
{
        Node n1;                  // Default constructor gets invoked.
        Node n2(25);          // Parameterized constructor gets called
}
```

JAVA __

```
public class Demo
{
  String name;
  public Demo()
  {
        name = _ _;
  }                                           // Default Constructor


   public Demo(String myName)
   {
         name = myName;
   }
   public static void main(String args[])
   {
         Demo d1 = new Demo();//Default constructor called
         Demo d2 = new Demo("Ram"); //Parameterized one called
   }

}
```

PYTHON _

```
class Test:

   def __init __(self): print ("Constructor")
   def __del__(self): print ("Destructor")

if __name __ == "__main __": obj = TestClass()
   del obj
```

There can be overloading of Constructors. So to distinguish between over-loaded constructor we should check -

1) Data type of parameters.
2) Number of parameters.
3) Order of parameters.

Destructor is special method in class that is always invoked when a object is _about to die_ . There can be atmost only one destructor for a class.

In C++ the destructor are just the class name preceded by a ~sign. For eg - Node :: ~Node() is a destructor function of class Node.

Overloading is not permitted for destructor.

In C++ structs and classes are almost same except for In struct by default all members are public.
In class by default all members are private.

Both the keywords new and malloc are used for dynamic allocation of memory but the difference between them is that when we use new keyword it will call the constructor of class also along-with allocating memory where as using malloc we can't do that.

LECTURE 3

17 January 2014

Tarun Tater (IMT2012047)

Lijo Johny (IMT2012023)

Ananth Murthy (IMT2012004)

Default Constructor

- They take no arguments and have no return type.
- They are of two types – ‘built in’ and ‘user defined’.
- The mechanism for invocation is same for both.
- Constructor for a class is optional as the compiler would automatically call□ the built-in constructor if no constructor is defined.
- The built-in default constructor gets invoked only when there is no□ constructor in class for the object.

Eg –

```
Class Node{

        Node(); // default constructor

}
```

The two ways to call default constructor -

```
 Node n1,n2; // java – same way, python – no declarations needed.

 Node *nptr = new Node; // not possible in java as no pointers are accepted in java.
```

The main task of the constructor is initialisation.

Initialisation – It means compiler pre-assigns a value before the programmer gets the access to the memory location.

Whereas **assignment** is storing a value(or data) in a pre-defined memory location.

Difference in initialisation and assignment –

```
int age; //declaration +□definition
age = 25; //assignment

int age = 25; //declaration + definition + initialisation // same for java. In python - age = 45;
```

Initialisation can be done in various ways. Three approaches to initialise are -

Approach 1	Approach 2	Approach 3

Class Node{
int data;
Node *next;
public:
Node();
{ data = 0;
next = nullptr; }
show();
};
	Node::Node(): data(0),
	next(nullptr);
	{
	}
		Class Node{
		int data = 0;
		Node *next = nullptr;
		};

'this' keyword – It refers to the current object on which the method is invoked. It is implicitly passed in C++ and java.

In python, 'self' keyword is used for the same functionality. But there, the programmer needs to explicitly pass self.

Copy Constructor

It is used for **initialising object with an object of its own types** that is with an object of the same class.

Copy constructor gets involved -

□ during copy initialisation

Eg - Node n1 = n2; □ □

□ □

□ When arguments are **passed by value.** (not pass by reference) □

Eg –

```
void dosomething(Node n ) //copy constructor is invoked
{
        n.show();
}
Node p1;
dosomething(p1); //when this method is called, a new object 'n' is created which is initialised with p1.
```

Valid and invalid ways to write constructors-

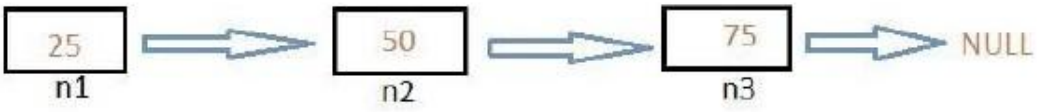
```
class Student{

        string name;

        int age;

        Student(string n, int a);

         //Here, we can even initialise it a default value like int a = 10, then if no value is passed, it
would take initialise the value of integer as 10

        //there is no user-defined default constructor.

};

Student s1("name 01", 20);

Student s2; //wrong as no default constructor

Student s3 = s1; //copy construct
```

Warning – If Student(string n, int a =10) is present and Student(string) is also a constructor, then

Student s1(“name”) will give an error saying it is ambiguous as the compiler cannot decide which constructor to call.

□ □

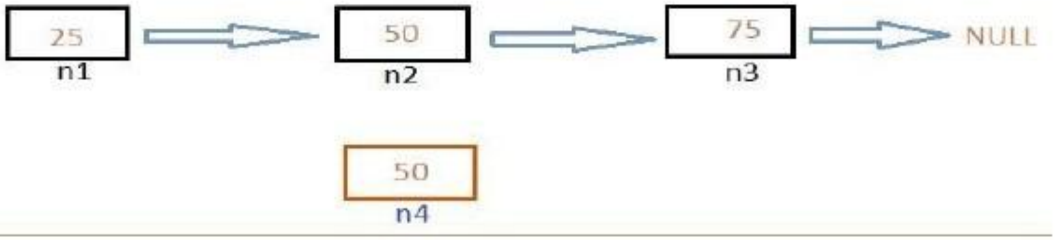
Copy constructor can be used in two types – namely shallow copy and deep□ copy. Eg – In a linklist -



Node n4 = n2;

□ □

Shallow Copy – It would copy the value of n2 into n4 without copying the next pointer.



□ □

□ □

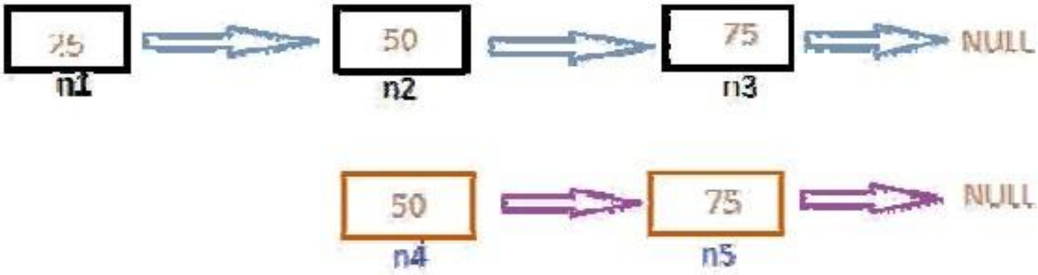
□ copy only direct values □ □

□ □

□ ignore pointers and links □ □

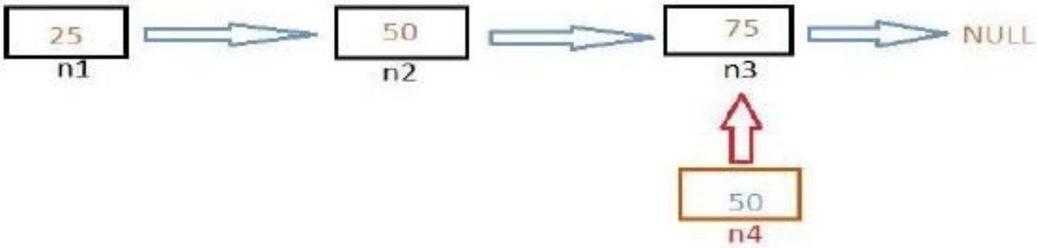
□ □ □

Deep Copy – It would copy the content of n2 into n4 and also **create a copy** of the remaining link list.



□ □

Error Prone Shallow Copy



□ □

- Here, it copies the values and pointers but points to the same element of the previous list and does **not** create a new copy of next element. □ □
- Error prone as if the list element or list is deleted, it would become a 'dangling pointer'. Due to this reason references are generally preferred over pointers as a reference cannot be null. □ □

Parameterized Constructor

- Only this constructor can be overloaded. □ □

□ □

- Invoked when the object takes user-defined parameters. Eg - □ □

```
□ □
Student(string n, int a) : name (n),
age(a){} main() □ □
□ □
{ □ □
□ □
Student s1("John", 20);
Student s2; □ □
□ □
s2 = s1; // assignment. □ □
□ □
} □ □
□ □
//This assignment is equivalent to □ □
     s2.name = s1.name;

     s2.age = s1.age;

     //In java – Student s1 = new Student("John", 20)
```

- The above example is an implicit member-wise assignment. □

□ □

□ For explicit assignment – C++ allows user-defined assignment which can also be overloaded. Eg - □ □

```
□ □
        public: □ □
□ □
        operator = (const Student & s) □ □
□ □
        { □ □
□ □
                //we redefine'=' operator here as per our requirements. □ □
□ □
                // eg - this = &s2;
                //          &s = &s1;
        }
```

Warning -

```
        void Student :: operator = (Student &s1)
        {
                name = s1.name;
                age = s1.age;
        }
```

Here, we cannot write a=b=c as the return type is void, so when b=c is done, there is nothing to return and hence it would be an error.

□ □

While passing an object, it is better to pass it by reference for better performance.

□ □

Copy constructor **cannot** be invoked when arguments are passed by reference.

When destructor acts on the linked list, n3 gets deleted. So n5 points to vacant position, such pointers are

□ □Dangling pointer.

□ □
□ □

Assignment :

```
        Node n1, n2 ;
        n1 = n2 ;
```

□

- □ basically does member wise copy □ □
- □ equivalent to shallow copy (explained later) □ □

The assignment function can be explicitly defined by the user. It can be overloaded. It can also accept objects of 2 different classes as the operands.

In such a case, some data members can be copied.

e.g.

```
        void Student :: operator = (Student & s1)
```

```
                {
                        name = s1.name ; roll_no =
                        s1.roll_no ;
                }
```

Here since the return type is void, a = b = c ; cannot be executed. One can also do :

```
        void Student :: operator = (UniversityRecord & r1)
           {
                   name = r1.name ; roll_no =
                   r1.roll_no ;
           }
```

Where Student is a class with name and roll number as its data members.

LECTURE 4

24 January 2014

Srikrishna S Bhat (IMT2012042)

Shashank Gupta(IMT2012041

Sankalp Johri(IMT2012040)

Object relationships mainly translate t class relationships. The motivation for this cotooncept and object oriented programming is mainly based on making programs look similar to the real world.dw

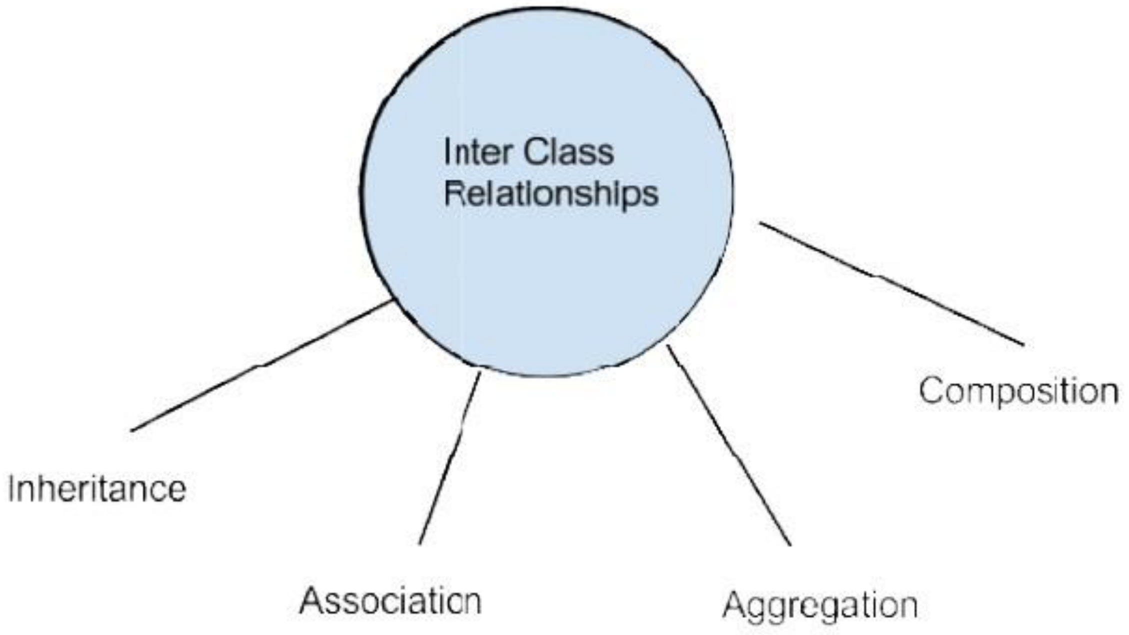
Objects vs Attributes:

i. Attributes make sense when they belong to some object. They cannot be consideredot independent.

ii. Attributes unlike object do not have their own behaviour.ts

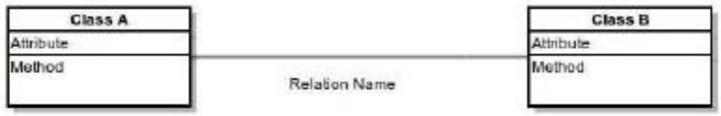
iii. An attribute is simple in its representation whereas the representation of an objectn comprises of many oth attributes.her

Ex: student_name = “VVenugopal Rao” is an attribute whereas an object student_name ofect class Student is represeesented using first_name =”Venugopal”, last_namme=”Rao”.



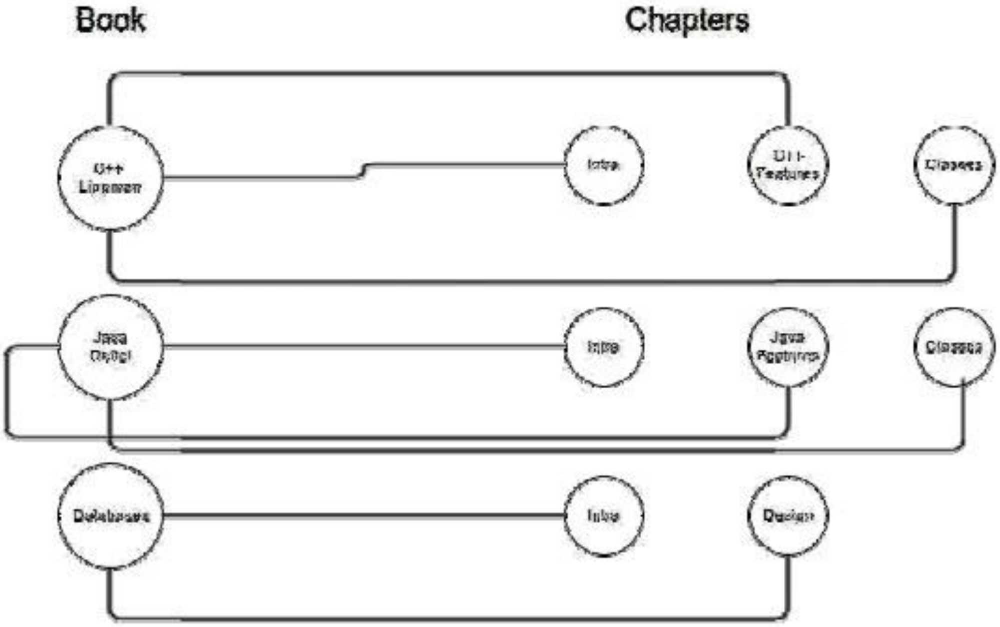
What characterizes a relationship?

1. The type of relationships between the classes and the objects.
2. Cardinality(how many objects of one class are linked to how many objects of other class)



Steps to determine Class Cardinality:-

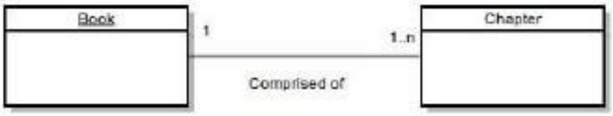
1. Draw out the objects of the class(Object Diagram)



2. Check the number of association of the object of class book to the objects of class chapters.

3. Check the number of associations of the objects of class chapters to the objects of class book.

Fill in the cardinality in the association diagram.



Note:- While _lling out the cardinallity of an association, direction in which the cardinality is taken matters.lling

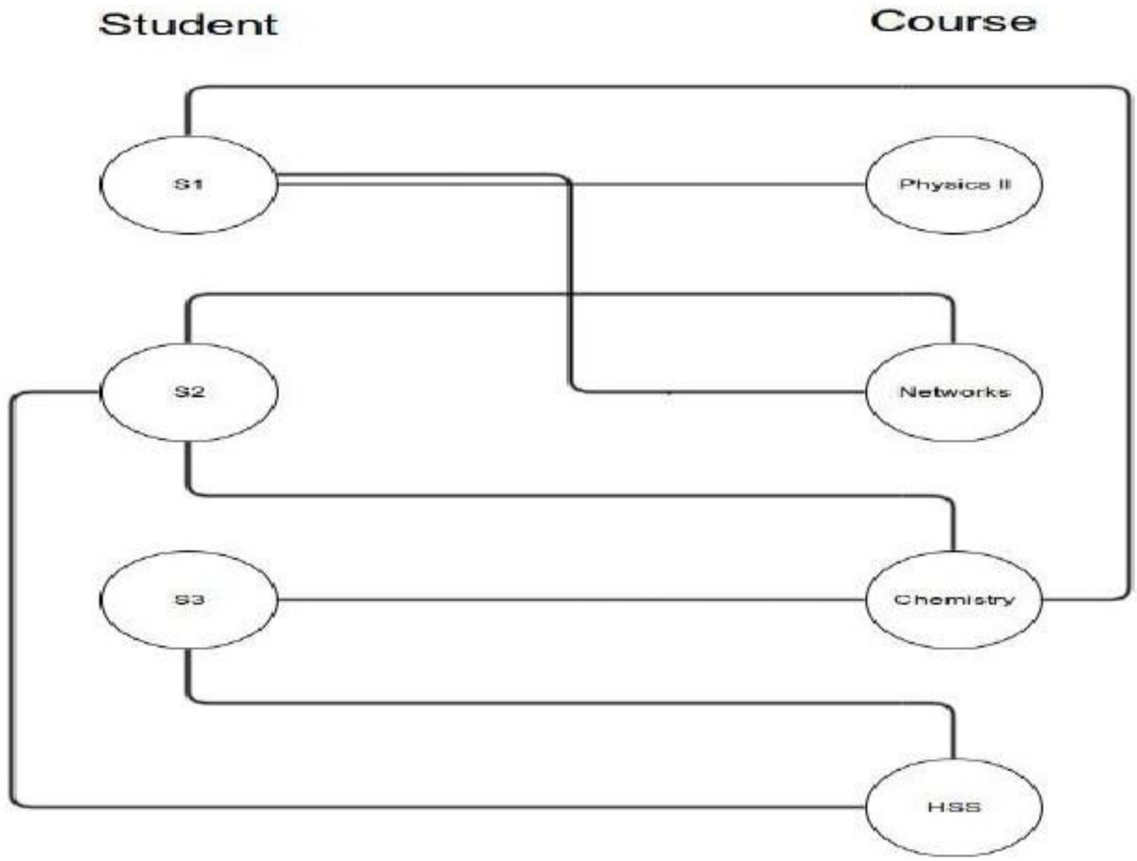
Types Of Associations:-

1. One-Many
   In the example given above when the cardinality of the association is taken as the number of associations from the object of class Book to the objects of class Chapter we get a oneone-many relation.

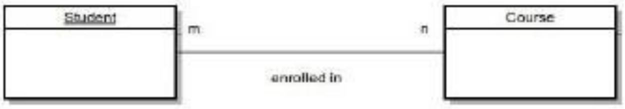
2. Many-One
   In the example given above when the cardinality of the association is taken as the number of associations from the object of class Chapter to the objects of class Book we get a ManyMany-One relation.

3. Many-Many

Many-Many associations can be seen in the example of association be-tween objects of classManystween Student and the objects of class Course.



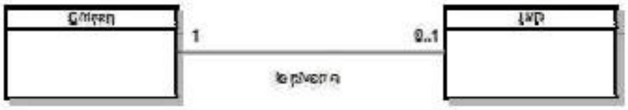
This can be represented as:-



This show a many-many relationship between objects of class Student and object of class Course

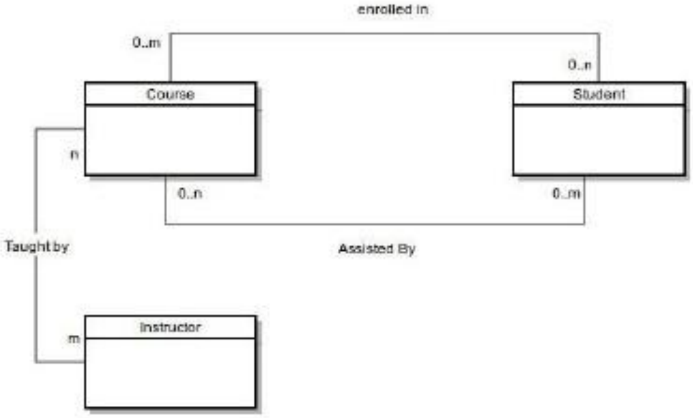
4. One-One(can be considered a special case)One(can

This kind of relationship can be seen between the objects of class Citezen and class UID.



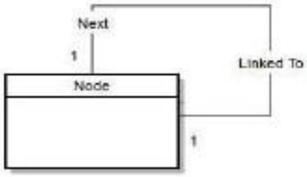
This shows that every citizen is given a unique UID

A Class can be associated with more than one class:



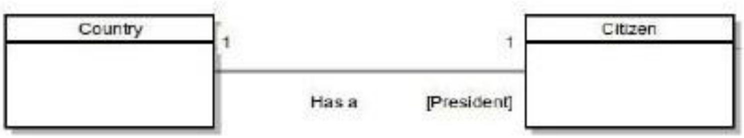
Self Referential Associations(Variation of Association)

This type of association is seen when an obect of a class is associated with another object of the same class.



Role

It is seen that objects of a class can play speci_c roles in association with objects of other classes. For example, in the case of the objects of class Country and the objects of class Citizen, one object of class Citizen plays the role of the president of the country in a one-one association



LECTURE-5

31-1-2014

Reijul Sachdev (IMT2012036)

G.Akhil(IMT2012015)

Abhishek (IMT2012017)

Carrying on from the previous discussion on inter-class relationships, we devote our attention to the latter part of associations and start with inheritance.

1.Associations: They are the most generic of all inter-class relationships. They are completely specified by:

□

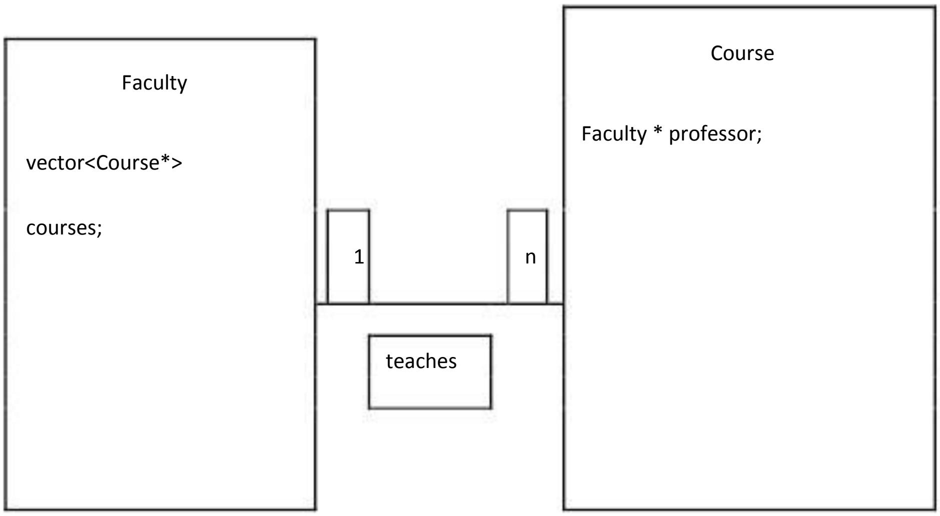
□

- □ Name of the association (and corresponding nature) □
- □ Cardinality (e.g. 1-to-1, m-to-1, m-to-n, etc.) □
- □ Role (provided the associated classes have more than one association between them) □

In addition, associations also possess a direction. They are, by default, bidirectional, however, they may be specially designed to be unidirectional as well.

Now, while UML diagrams provide all of these characteristics (and hence specify completely the association) at a design level, these associations must be mapped to the appropriate representation in any language. In this context, mapping rules are provided to completely specify the association from a programming perspective as well.

As an instance, consider the example given below:



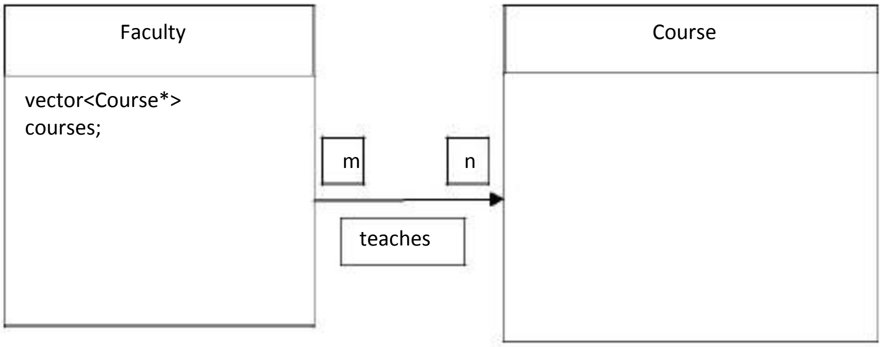
The rules used for mapping to C++ are applied to the above example.

Rules:

- □ For every class in UML, create a C++ class. □
- □ For every attribute, create a data member for the appropriate class. □
- □ If there is an association between two classes, then create a pointer in each class of the other's type. □
- □ If an instance of one class maps to several instances of another, then, use an array (or vector) of pointers of the other class type, rather than a single one. □
- □ If the association is bidirectional, then, an instance of one class should be able to access one (or more) instances of the other class. □

Using associations, we create links between instances of two or more classes. This allows us to traverse a network of instances of various class types using just one particular instance. This is termed navigational access across objects.

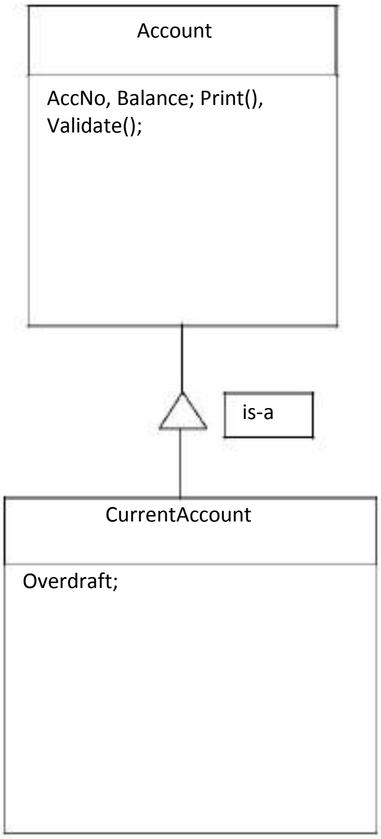
For a unidirectional association, the class pointed to by the arrow does not possess a pointer to an instance of the other class. But, the class from which the arrow points outwards has a pointer and so can access instances of the other class. For instance (pun intended!), consider the example below.



Bidirectional associations have the advantage that they allow greater access between instances and create a more easily traversable network, leading to more modular programming. However, they also require greater maintenance, in that if a link is broken (or established) in one direction, the same must also be done in the other direction.

2.Inheritance: Inheritance is another type of inter-class relationship. It does not involve any connections between objects, but it does allow instances of the subclass to possess all the original data members and methods of its super class.

Note however, that every instance possesses its own copy of these members and that only changes in the super class get reflected in the subclass and not vice-versa. Super class refers to the class from which the current class (the subclass) inherits. They are represented in UML diagrams as below:



In the above case, the members of super class Account, namely AccNo, Balance, Print() and Validate() are implicitly passed to subclass CurrentAccount. Thus, inheritance is a convenient method of reusing attributes and behaviour in different classes implicitly. Thus, the sizeof(CurrentAccount) =sizeof(AccNo)+sizeof(Balance)+sizeof(OVerdraft), even though only Overdraft has been explicitly defined.

Inheritance may also be viewed as a special type of association wherein the nature of the association is always "is-a". Thus, the realtion is read as CurrentAccount "is-a" Account.

Types of inheritance are:

- Single -- Every subclass has one superclass.
- Multiple -- Every subclass may have one or more superclass.

Note that in case of multiple inheritance, problems may arise when both the superclasses possess the same attribute. In this case, separate programming logic is needed to determine which class not to inherit from. The subclass is still, however, an aggregation of the attributes and behaviour of its superclasses.

Finally, since both attributes and behaviour are inherited, so are relationships like association. However, in case of a bidirectional association, the additional members of an instance of the subclass are not visible to the class associated with the superclass. Thus, to this class, its association still appears to be with an instance possessing only the members of the superclass.

JAVA association and inheritance:

Associations are implemented in JAVA using references to the associated classes, as the notion of pointers does not exist in JAVA.

Example:

```
class Course
{...}

class Faculty
{
    private Course courses=new Course;
    ...
}
```

Here, the member courses in Faculty is a reference to an instance of class Course, as required.

On the other hand, inheritance is implemented as follows:

```
class Account{...}

class CurrentAccount extends Account{...}
```

The subclass is said to 'extend' the superclass.

Python association and inheritance:

In Python, associations are implemented in the following manner:

```
class Course(object):

    ...

class Faculty(object):

    def __init__(self):

            self.course = Course();

            ...
```

Here, a reference to class Course is created in class Faculty. Like JAVA, Python too does not support the concept of pointers.

In Python, inheritance is implemented in the following manner:

```
class Account(object):

    def __init__(self):
    ...

class CuurentAccount(Account):

    def __init__(self):

            overdraft=0;

            super(CurrentAccount,self).__init__();

    ...
```

The __init__() functions in Python correspond to the constructors. Thus, the implicitly inherited members are initialized using the inherited constructor as above.

Sl No	INHERITANCE	ASSOCIATION
1.	No pointers required	Pointers required
2.	No navigation between objects	Navigation between objects takes place
3.	Size of sub-class is summation of explicitely declared data memers of all super classes till the subclass including the sub-class.	Size of class is the summation of all the data members declared explicitely in that particular class.
4.	Unidirectional	Can be both uni-directional and bi-directional

Multiple Inheritance:-

It is the phenomena where one sub-class has more than one super class. It is a rarely used form of inheritance because it has many problems and hence is not supported by many programming languages. One of these problems would be the copying of variables even if they have the same name. This leads to scope operations to determine which variable belongs to which super-class. Another problem would be the fact that it can give some issues and challenges while compiling. There is a paper by the makers of JAVA regarding why they chose to stay away from multiple inheritance in JAVA. Although it is supported in C++, it is very rarely required.

LECTURE-6

7-2-2014

N.HANISHA(IMT2012028)

S.Praneeth Kumar (IMT2012043)

Sushravya G M (IMT2012044)

Polymorphism:

The ability(in programming) to present the same interface for differing underlying data types.

It essentially means diferent objects can respond to the same message in different ways, enabling objects to interact with one another without knowing their exact type.

Technically, it is referred to "binding" (dynamic binding or late binding).

For example , integers and floats are implicitly polymorphic since you can add, subtract, multiply and so on, irrespective of the fact that the types are different.

```
Eg: int i =2; float j=1.5; i*j =
    3
    i+j = 3.5
```

Polymorphism in Inherited class relationships in C++:

```
class Product
     {
     price ; /* datamembers */ public:
void getPrice()
     {
     return price;
     }
     };
```

```
class DiscProduct : public Product
/* "public" here represents inherent class specifier */
     {
     discount ; /* datamembers*/ public :
     /* member functions */ void getPrice ()
     {
                  return price- price*discount;
                  }
     };
```

DiscProduct is a subclass of the Product class.

```
//Writing a normal function

void show(Product *p)
     {
     P->getPrice();
     }
```

```
Product *p1 =new Product("Pen",10);
Product *p2 = new DiscProduct("Pin",20,0.20);
```

In this inherited-class relationship between Product and DiscProduct, invoking the function "show" over pointer p2 calls up the DiscProduct::getPrice() instead of usual Product::getPrice() .This phenomenon is called "polymorphism".

Note: Polymorphism in C++ occurs through dynamic binding . Unlike in C++, polymorphism is by default in java. Also, there are no inherent class specifiers in java.

Dynamic Binding: A computer programming mechanism in which the method being called upon an object is looked up by name at runtime.

In other words, the decision is made at run time based upon the type of the actual object.It is also called “late binding”.It is usually seen in Inherited class relationships.

Dynamic Binding in C++, are done only through pointers and references.
The initialization of pointer p2 is done on the basis of dynamic binding. Such type of binding is possible here, because of DiscProduct being the subclass of Product class .

Due to dynamic binding, invoking of “show” function is possible for both above initialized pointers.

show(p1); show(p2);

Note: Such type of invoking is possible in C++ only through pointers and references.

Overriding :

If base class and derived class have member functions with same name and arguments. If we create an object of derived class and write code to access that member function then, the member function in derived class is only invoked, i.e., the member function of derived class overrides the member function of base class. This feature in C++ programming is known as function overriding.

Benefit of overriding:
A subclass can implement a parent class method based on its requirement.

In the above discussed C++ programming example, invoking the show method calls up the getPrice() member function. Due to the dynamic binding for pointer p2 above, invoking the show method , getPrice() function of DiscProduct (subclass) class gets called overrinding the usual getPrice() function of parent class (Product). Thus, DiscProduct::getPrice() overrides Product::getPrice() .

In C++, Overriding can be done by only declaring that particular member function in the superclasses as “virtual”. Also, like overloading, overriding an inherited method is completely optional not compulsory.

Overiding in java:

In java, if a class inherits a method from its super class, then there is a chance to override the method provided that it is not marked final.

Overriding in java is similar to that in C++, except that java doesnot require that virtual declaration for the member functions to be overridden.The methods are virtual by default.

Overloading vs Overriding:

_ Overriding of functions occurs when one class is inherited from another class. Overloading can occur without inheritance.
_ Overloaded functions must differ in function signature i.e. either number of parameters or type of parameters should differ. In overriding, function signatures must be same.
_ Overridden functions are in different scopes; whereas overloaded functions are in same scope.
_ Overriding is needed when derived class function has to do some added or different job than the base class function.
_ Overloading is used to have same name functions which behave differently depending upon parameters passed to them

_ Overloading is a static or compile time binding whereas Overriding is a dynamic or run-time binding.

Constructor and Destructor calls

The compiler automatically call a superclass constructor before executing the subclass constructor. Here, we need to remember that the subclass construcot has no direct access to superclass private data members. Thus, when
For example: class X

```
        {
        public:
        X( ) { cout<<"superclass constructor"<<endl; } ~X( ) { cout<<"superclass
        destructor"<<endl; } };

class Y : public X
        {
        public:
        Y( ) { cout<<"subclass constructor"<<endl; } ~Y( ) { cout<<"subclass
        destructor"<<endl; } };
main( )
        {
        Y obj;
        }
```

OUTPUT:

```
subclass constructor

superclass constructor

superclass destructor

subclass destructor
```

Design strategies of inherited class relationships in C++:

In particular, there are two types of design strategies for inherited class relationships. They are:

1) Specialization (Top – down approach) :
In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems.

Each subsystem is then refined in yet greater detail, sometimes in many additional s subsystem levels, until the entire specification is reduced to base elements.

Top down approach starts with the big picture. It breaks down from there into smaller segments.

2) Generalization (Bottom-up approach) :
This approach is piecing together of systems to give rise to more complex systems, thus making the original

systems sub-systems of the emergent system.

In other words, it checks for the commonality along the “is-a” relationship of these individual systems in forming a complex system.

Thus the systems are created first and super classes are added each time.

Some more significant differences between inheritence of java and C++:

_ In Java, all classes inherit from the Object class directly or indirectly. Therefore, there is always a single inheritance tree of classes in Java, and Object class is root of the tree. In C++, there is forest of classes; when we create a class that doesn’t inherit from anything, we create a new tree in forest.

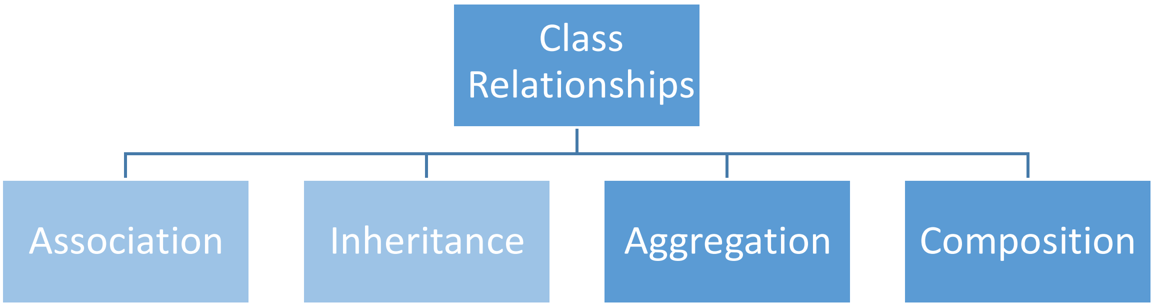
_ Unlike C++, Java doesn’t provide an inheritance specifier like public, protected or private. Therefore, we cannot change the protection level of members of base class in Java, if some data member is public or protected in base class then it remains public or protected in derived class.

_ Unlike C++, Java doesn’t support multiple inheritance. A class cannot inherit from more than one class. A class can implement multiple interfaces though.

Lecture 7

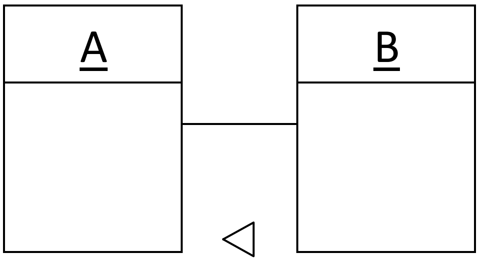
15/02/2014

T Harshavardhan Reddy(IMT2012045)
V Venkata Aditya Sai Kumar Reddy(IMT2012050)
G Keerthana(IMT2012016)

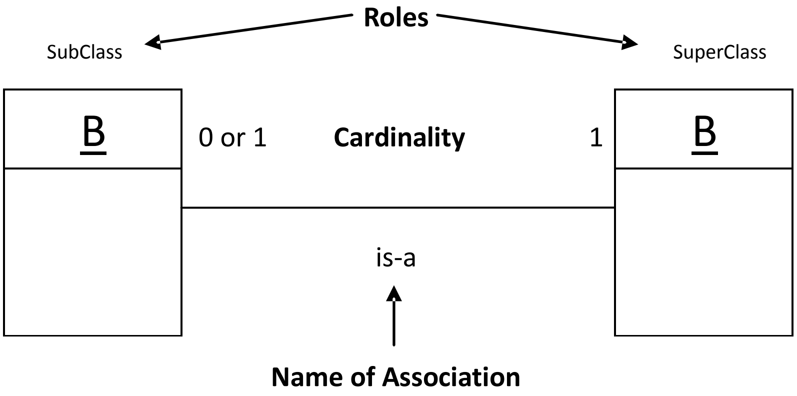


Modeling Inheritance as Association

To model:

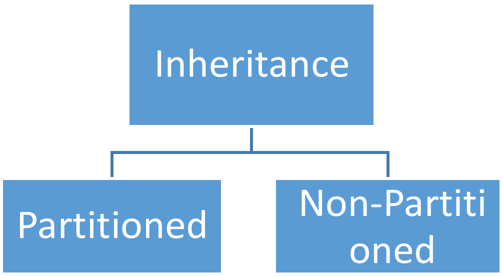


Implemented as Association:



Things to do:

- *Name* the association; in our case, it would be “is-a”
- Find the *cardinality*; it’d be (0 or 1) to (1) in our case
- Describe the *roles* of the Association; it would be super-class and sub-class



Inheritance

Partitioned

Non-Partitioned

Cannot have a stand alone instance of a parent class. Methods of the parent have to be over-ridden with that of the sub-classes.

Implementation: *Pure Virtual Function*

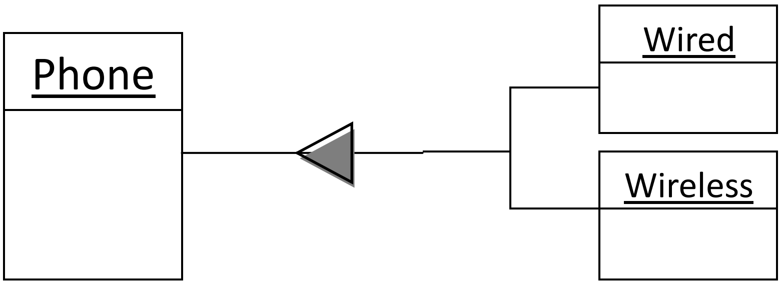
The ones we’re familiar with already. The parent can be instantialized independently.

Pure Virtual Function

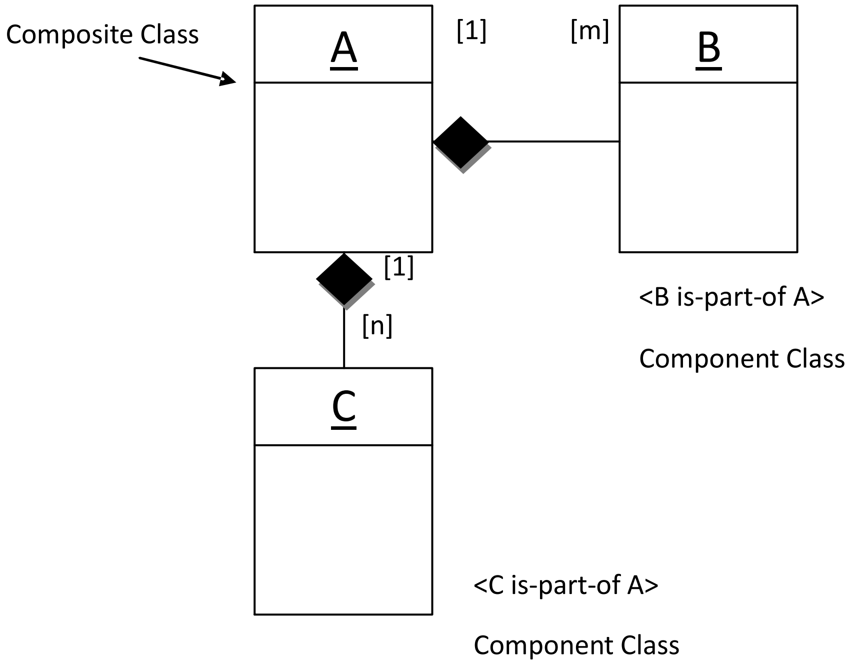
```
class Phone{
. . .
public:
      virtual void dial(int number)=0;
};
```

<must be over-ridden by a sub-class (which again must to be created)>

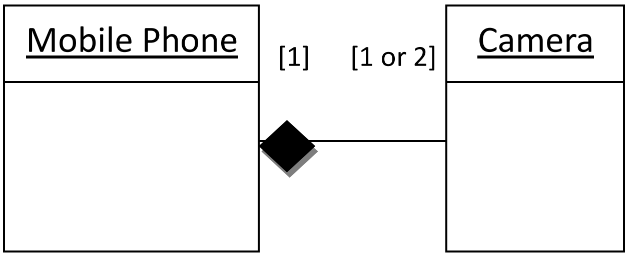
The partitioned Inheritance can be represented by:



Composition <is-part of> or <owns>



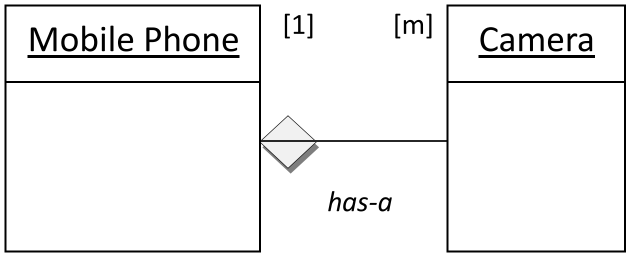
- *Component classes* can be part of only one *Composite class*.
- So component classes must have a composite class to exist. They cannot exist independently.



Implied Semantics

- Cardinality of composite class is *always* one
- Name: *is-part-of*
- Deletion semantics: if A gets deleted, B and C get deleted too (automatically)

Aggregation <has-a>



Implied Semantics

- Cardinality- same as Composition, i.e., *one to many*
- Name- "*has-a*"
- Deletion semantics: *do not hold*

When the Mobile Phone class gets deleted, the camera does not get deleted, rather it can be reused. A good analogy would be the camera being re-used for a new phone when a phone gets damaged (or gets deleted).

Implementations in C++

Aggregation

```
class MobilePhone{

. . .

      Camera* c;

. . .

};


class Camera{

. . .

};
```

A pointer is created so that the instance of camera is not deleted along with MobilePhone class, which is what we want to achieve with *Aggregation*.

Composition

```
class MobilePhone{
. . .
     Camera c;
. . .
};

class Camera{
. . .
};
```

An instance of Camera class is created *inside* the MobilePhone class. So when the MobilePhone instance is deleted, the Camera class data is lost too.

B cannot be part of anything else but **A**

To make **B** totally invisible to everything but **A**,

```
class MobilePhone{
private:
     class Camera{
     . . .
     };
. . .
};
```

A *nested class private declaration* can be used to make it invisible to other classes.

```
class MobilePhone{
private:
     Camera c;
. . .
};

class Camera{
     Camera();
public:
. . .
     friend class MobilePhone;
};
```

Or, the *Camera* class can be declared outside with the **constructor being private**. The *MobilePhone* class can later be made **friend** of the class.

Post Script: Implementing *pure virtual functions* in Java and Python

Java

The closest java gets to the pure virtual function of C++ is the `abstract` modifier. Java lets us define a method without implementing it using the above mentioned modifier. Also the class modified as `abstract` cannot be instantialized as a standalone instance. The methods must be overridden using a subclass.

Python

Python methods are always virtual. They can always be overridden. If a *pure virtual function* is what we desire, we cannot have a compile time error about the method not being overridden, because that's not how Python works because Python is a dynamic language. We can however compromise with the virtual nature of the methods rather than forcing the overriding.

Lecture 8

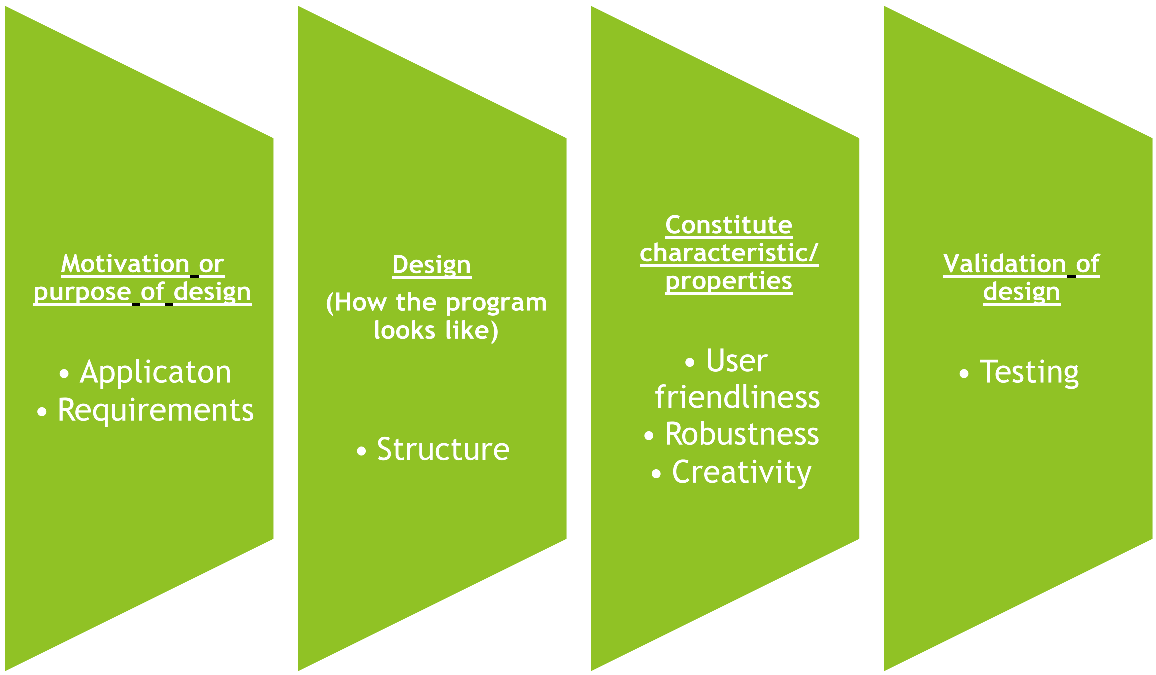
22/02/2014

Lukshya Madan(IMT2012024)
Amit Gupta(IMT2012003)
Deepshree Ravindran(IMT2012013)

Software Design

NOTE: *Software Designing is a generic term. It is not exclusively applicable to OOP.*

What comes to your mind when you hear Software Design?



Motivation or purpose of design

- Applicaton
- Requirements

Design (How the program looks like)

- Structure

Constitute characteristic/ properties

- User friendliness
- Robustness
- Creativity

Validation of design

- Testing

Only **Structure** defines Design. **Application, Requirements, User friendliness, Robustness, Creativity, Testing** only impacts the design.

Software Design Example

We have Scrabble as the largest program written so far. Assume it's a 1000 line program. Most trivial way to do this program would be to have one big program which has only Main function in it. We are still meeting our application requirements here by simply copying and pasting all program code into one place and by declaring all variables as global variables. This is just one way to create the scrabble program and it differs from our current implementation. Therefore we are having two different ways of designing the scrabble program.

One classical way to understand the difference between requirements and design is that, requirement tells us what we want, and design tells how we want to meet those requirements.

Two extremes in Scrabble program: one Big Main design, current implementation of scrabble.

Big Main design: - it is a bad idea:

- Difficult to debug
- Difficult to understand
- Difficult to change
- Reusability

  The way we have written the current code for Scrabble, what is the reused component of the written 1000 line? How much output method have been reused? Is it 0 or likely to be more?
  It is likely to be more, all the Cin methods and functions can be taken into the credit.

- Team development (not possible)

The big main design has all the above problems, so we need a better design which leads us to desirable characteristics of design in most formal sense.

In general, the good characteristics of software design:

1) Modular

Assume we have a C Program, then what is the ideal size of this module? Equating module with functions, we get 30 lines. Modular system started in the days of CRT monitors, when everywhere programs were being developed. It is a good metric to have 30 lines. It is interesting find out how the best coding practices evolved, before the good metric was considered to be 80 lines per page.
In the 1000 line code of Scrabble, the most stupid way to implement would be to write return statement after every 30 lines rather than using call.

```
Main()
{

F1()
30 … F2()
60 … F3()

}
```

Instead we can copy paste 30 lines and move it into functions F1, F2, F3 and declare all variables as global and simply call functions F1, F2, F3. It will work. This way of implementation is Modular and it is a requirement for a good code practice.

2) Cohesion

It is something that holds things together. It divides a program into modules and it makes sure that the program is cohesive. It is required that the statements in module should be directed towards achieving one objective, it is like a deliberately left kind of loop.
Playing scrabble one objective so one big main does not make it cohesive. Objectives are functional in nature and extend of reusability grows if we make the modules cohesive.
We should judge objectives accordingly, so placing a tile is one such objective in the scrabble game and display board a nice cohesive and reusability thing.

In order to achieve cohesion, a C programmer should maintain high cohesion (by writing more cohesive modules), and low coupling.

3) Coupling

It is the dependency of one module on the other module. To increase reusability modules should be loosely coupled. If we take the Display board and take show board functionality and if that functionality is also doing few other things like showing rack, showing score and showing board, so it is doing all these three things, it is not cohesive, as it is doing more than one thing also it is not modular enough because we cannot use each module individually which is the cohesive aspect.

To make the modules independent, we should create coupling between the modules. F2 cannot be used ever without calling F1. We should design the modules such that F2 will not do its job until F3 happens.

From a programming point of view how it happens:

Suppose we use global variables, we have 2 components having 7 data items in the previous functions F1, F2 and F3, we have F3 followed by F2 and then F1. This is our design, where there are three modules, all are cohesive and maintain the sequence F3, F2, F1. We always call these functions in this order. Now we have coupled them.

The regenerated case, where we have bunch of modules where coupling is 0, is that possible? There is a situation where it is very much likely to happen. Where the Cohesive is high, coupling is zero, it is a typical signature of a library (example math.h). In this case where we are having such signature, it won't be termed as application but a library which is highly reusable for building applications. Tus we should minimize coupling. To do so we can use global variables.

Note: *Global variables, high indicators of coupling.*

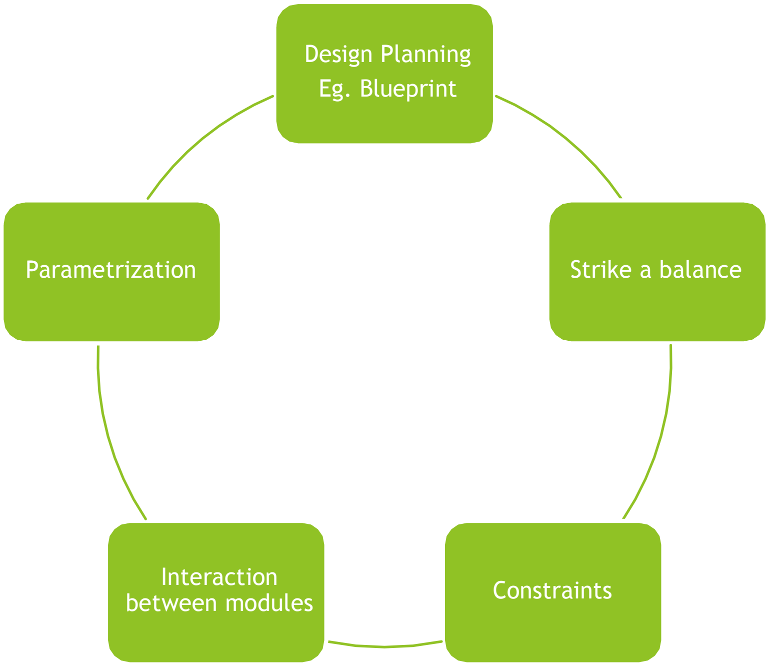
Manifestation of coupling (Effects of coupling)

1) Ordering of invocations across modules.
2) One module cannot be changed without changing some other module.

Software Architecture

What is Architecture?

Software Engineering field can be understood best from Civil Engineering, as there are lots of terms in common while dealing with Software Architecture:



Note: *Software Architecture influences the Software design. It is a precursor to the design.*

Example:

Design Planning, Parameterization, Strike a balance, Constraints: In a typical apartment project, we can create an architecture which says it is very green centric. So such things we will specify at the architectural level. How much greenery we want, we parameterize it. We create a specific design for it.

As part of the architecture we should set aside place for water recycling, trees, solar and then we parameterize it. Now suppose, if we need to design this project in a 2 acre plot, the architecture will be different if we implement same in a 10 acre plot.

We need to achieve many things in this apartment project we need to strike a balance by dividing every module for it into specific timeline for completion with constraints.

Interaction between modules: Architectural aspects: for such interaction we have used function calls so far which is not the only way for modules to interact. If at the architecture level we decide that this interaction should happen, so as one module siting in Delhi and one in Bangalore can interact, or there is a module running inside bank and there is a module running inside IBM can interact, so these decisions will go into the architecture. At the architectural level we will decide how those modules will interact.

Easiest way to remember architecture: Architecture is just collection of principles. Interactions between systems must happen over the internet, which is an architectural decision. Interactions between systems must happen over the Bluetooth, which is an architectural decision.

How architecture will influence (Architecture Principle):

1) **Separation of concerns**

   If there are 3-4 things are to be done, each things are done separately. Classic manifestation from a design point of view: UI vs. Computation should be done separately. So while designing the modules we shouldn't do too much computational activity inside the UI.