

**Object-Oriented Analysis and Design**  
**Prof. Partha Pratim Das**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology – Kharagpur**

**Lecture – 22**  
**Nature of a class: Interface and Implementation**

(Refer Slide Time: 00:25)

**Module 13: Object Oriented Analysis & Design**  
Nature of a Class: Interface and Implementation

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan

This presentation uses diagrams, examples and selected texts from *Object-Oriented Analysis and Design – With Applications* by Grady Booch et. al. (3rd Ed, 2007) with kind permission of the author

Welcome to module 13 of object oriented analysis and design. In the last two modules we have discussed about the nature of objects and their interrelationships between them. In the current module we will start discussing about the nature of classes. What is an interface and what is an implementation will be the major items to talk about so we will try to understand those from the perspective of a class.

(Refer Slide Time: 00:55)

## Module Outline

- Design by Contract
- Interface
  - Stack
- Visibility
- Implementation
  - Stack

Here is the outline which will be available on the left hand side of the presentation all through.  
(Refer Slide Time: 01:06)

### What is a Class?

Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, the "essence" of an object, as it were

- For a class *Faculty*, objects may be:
  - {Partha Pratim Das, Professor, CSE}
  - {Prabir Kumar Biswas, Professor, ECE}
  - {Shyamal Das Mondal, Assistant Professor, CET}
- Class *Faculty* abstracts – *Name*, *Designation*, and *Department*

**A class is a set of objects share a common structure, common behavior, and common semantics**

A single object is simply an instance of a class

Now I am sure most of you have certain strong idea about what is a class? Still we would like more formally define that. We have since we are talking about object-oriented systems so certainly we know that the core of the whole system are objects and we have seen we have defined that what is an object and we have seen the object has state, behavior and identity. So object necessarily is a concrete entity that exists in time and in space we have seen that.

Whereas a class which is closely related to objects is somewhat an abstraction somewhat of an abstract concept which tries to capture the essence of a set of objects which have strong

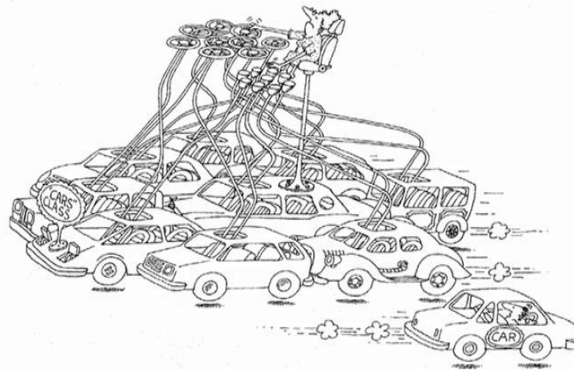
similarities strong commonality in terms of their structure behavior and semantics. So for example if we consider say a class faculty then. We will say that it has let say for example three different objects three different names of professors, their designation and the department that they are associated with.

So each one of this are objects because they have a concrete existence whereas this class faculty abstracts these objects in terms of the name, designation, department and of course a lot of other behavior lot of other operations that will be common to all these objects. So to put in place a class is a set of objects sharing a common structure, common behavior and common semantics. So when we talk about the commonality of the objects in a coherent manner we refer to the class.

So it is a kind of like a template according to which every object of that class have to correspond to have to adhere to have to confirm. So we will say that given any class, an instances of the class is an object okay.

(Refer Slide Time: 03:37)

## What is a Class?



*A class represents a set of objects with a common structure & a common behavior*  
Source: Object-Oriented Analysis and Design – With Applications by Grady Booch et. al. (3rd Ed, 2007)

Now of course this is a very nice cartoonist view of what is a class? You can look at it enjoy that.

(Refer Slide Time: 03:51)

## What is a not Class?

- An object is not a class
- Objects that share no common structure and behavior cannot be grouped in a class because, by definition, they are unrelated except by their general nature as objects
  - Today's newspaper, Bajrangi Bhaijaan movie, State of West Bengal – are all objects that cannot be put together in a class
- While anything can be a class, the attributes of a complex system should be carefully crafted in designing a class:
  - Hierarchic Structure
  - Relative Primitives
  - Separation of Concerns
  - Common Patterns
  - Stable Intermediate Forms

Now if you want to also ask what is not a class? Then certainly we will observe that an object is not a class or the objects a collection objects which share no common structure, common behavior they cannot be grouped in terms of a class. Because the class by definition has to have the common structure and semantics and behavior for example if you just look at the say today's new paper is an object, "Bajrangi Bhaijaan" movie is an object, state of West Bengal is an object.

These all are objects but they cannot be put together in a class. So the class will specifically be in terms of that sharing and beyond that what a class can be is a matter of experience and expertise of the designer who is designing the class. Of course it is possible that if we are talking about an organization as a whole we could theoretically at least represent the whole organization object as one class.

Such a class would be so complex it will have so many different types of interfaces catering to employees, to customers, to vendors, to regulatory authorities, to the management and so on that it will practically become very difficult to deal with to understand to analyze such design of classes. So the point that I would like to remind you about that when we talked about the analysis of complex systems we learnt that a complex system should typically be analyzed to be designed in terms of its attributes.

Five typical attributes are hierarchic structure, relative primitives, and separation of concerns, common patterns and stable intermediate forms. I would just like to remind you and if needed you can go back to the corresponding module and look at that video. Now it is the time that we use these attributes in design of the classes and that will decide what is and what should not be a class.

(Refer Slide Time: 06:22)


## Design by Contract (DbC)

- Design by Contract<sup>1</sup> is a metaphoric approach to design based on how elements of a software system collaborate with each other on the basis of mutual obligations and benefits
- The metaphor comes from business life, where a client and a supplier agree on a contract that defines for example that:
  - The supplier must provide a certain product (obligation) and is entitled to expect that the client has paid its fee (benefit)
  - The client must pay the fee (obligation) and is entitled to get the product (benefit)
  - Both parties must satisfy certain obligations, such as laws and regulations, applying to all contracts
- It has applications in:
  - Software design process
  - Inheritance with specific reference to dynamic binding
  - Exception handling
  - Automatic Software documentation

Source: Design by contract: [https://en.wikipedia.org/wiki/Design\\_by\\_contract](https://en.wikipedia.org/wiki/Design_by_contract)

<sup>1</sup>The term was coined by Bertrand Meyer in connection with his design of the Eiffel programming language and first described in various articles starting in 1986

NPTEL MOOCs Object Oriented Analysis and Design



Now a class typically has an interface and an implementation. I am sure most of you are familiar with these two terms. But to introduce these I would like to take a little tour of what is known as design by contract? Design by contract is a kind of a metaphoric approach where the concepts are borrowed primarily from the business domain where their clients and suppliers have to work together for getting certain products and to be able to work together there they have to agree on a contract where the client has certain obligations.

The supplier has certain obligations and the client gets certain benefits, the suppliers get certain benefits and both of them together comply with the contract and keep within the law of the land. Now the reason I refer to design by contract is a fact that when the methods of object oriented analysis and design were evolving and were getting strong grounds then in the middle of the 80's when design by contract was originally proposed by Bertrand Meyer.

And that has proved to be an every effective approach in solving a wide variety of software design problems or OAD problems and so on. So if we just look at it in terms of the terminology or points that they are between the client and the supply they will be mutual obligations and benefits. And in terms of the contract the supplier must provide a certain product which is the obligation of the supplier to the client.

And it will expect to have a benefit from the client which is the fee that the client fee. Similarly, the client must pay a fee which is the obligation of the client and is entitled to get the product which is the benefit that the supplier provides and both of them have to go by the obligations of the laws and regulations which a part of that contract. So this is the basic concept that design by contract says if this still is completely in terms of the business domain.

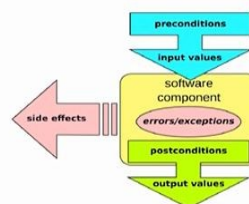
But please note that it designed by a contract or DbC as this is commonly called has formed wide applications in software design, in automated software documentation particularly in designing, inheritance, polymorphism, exceptions and so on. So we will come to those specifies later on.

(Refer Slide Time: 09:19)

## Design by Contract

For a class in OOP with methods:

- **Precondition** to be guaranteed on entry by any client that invokes the message – an obligation for the client, and a benefit for the server (the method itself), as it frees it from having to handle cases outside of the precondition
- **Postcondition** is guaranteed on exit from the routine – an obligation for the server, and obviously a benefit (invoking the message) for the client
- Maintain a certain property, assumed on entry and guaranteed on exit: the class **invariant**



*A Design by Contract scheme*

Source: Design by contract: [https://en.wikipedia.org/wiki/Design\\_by\\_contract](https://en.wikipedia.org/wiki/Design_by_contract)

Let us see how do apply design by contract more directly into a software design or a class design. We say that please take a look in to this diagram here we are talking about a software component which could be thought of as an object. It could also be a simple procedure or a

function so we say that this component this object or this particular method of an object is requested with a message with certain input values.

So what will happen if a message arrives to the object for a particular method then that method will take those input values and start executing so that is what is the software component is trying to do and then it will produce output. Now in the process it can also create some side effects certain other thing could happen like there could be some errors there could be some exceptions thrown and so on.

Now what we are saying in a good object based design the input values must satisfy a set of preconditions that is the client you will remember this our basic model is client-server so the client we sending the message to the server to invoke a message must make sure that the message is good. Let us make sure that the set of preconditions are satisfied by the input values at the same time there is a set of post conditions that must be satisfied for the output value.

So in response to the client's message the output that the server generates or the response message that the server generates must satisfy a set of post conditions. Now these two are guaranteed then certainly the client and the server can work in a lot more safe and segregable manner. So precondition is to be guaranteed at the entry and the client must do that because this is where the message is coming from so it is coming from the client.

So it is an obligation of the client to make sure that the precondition is satisfied. Against this obligation what benefits does the server get? The server gets the benefit that it now knows that the preconditions are satisfied so while it is implementing the software component while this is implementing the method it does not have to think about any situation which is not protected by the preconditions it can limit itself to adjust the preconditions.

Similarly post condition is guaranteed on exit from the routine, and it is the obligation of the server to ensure that and certainly that is the benefit that approve to the client because he has been able to invoke the message and get a successful execution. And in this whole process the



client and the server must maintain that the class has certain invariant certain things should not change should not be violated in the class all those must be maintained.

So if you see the style of specifying this design or building up this frame work of design by contract you will immediately be able to relate it to the typical business scenario of obligations and benefits catering to precondition, post conditions and the invariants in the class. Okay if this is somewhat sounding difficult to you sounding somewhat vague to you we will soon come up with an example and explain this design by contract method.

(Refer Slide Time: 13:24)

## Interface

The **Interface** of a class

- Provides its **Outside View**
- Emphasizes the abstraction while hiding its structure and the secrets of its behavior

The interface applies to all instances of a class and comprises:

- **Methods** – *publicly visible*
- **Contracts** for the Methods

But before that let me introduce what is an interface? We have talked about interfaces in earlier in number of times so you understand that an interface basically provides an outside view. That is whatever an interface of a class and which will by instantiation will become kind of an orifice through which an object can be operated on is necessarily an outsider's view. But someone from outside that object someone from outside the class can see.

And the abstraction and encapsulation will ensure that all the details the structural details and the secrets of the behavior will be hidden behind this interface view. Now interface naturally has to apply to all instances of object all instances of the class. So if I say that a class has certain interface then every object that I create of that class, every instance that I do that must satisfy the interface. So what does the interface has?



Certainly this is what you clearly know that it must have the methods. The methods might be publicly visible we have talked about visibility in terms of objects. We will see what does it specifically mean in terms of classes? But in addition to the methods what is important and this is the concept we are borrowing from design by contract is the interface has contracts or methods has certain precondition post-condition guarantee certain side effects of whatever that interface will do.

So interface will simply be the methods publicly visible methods and the contracts of these methods.

(Refer Slide Time: 15:27)

**Interface:  
Example of Stack**

Consider as Stack class as a LIFO container. The interface may be:

Method	Precondition	Postcondition	Side-effect
Push	Valid object	Object on top	May acquire more memory
Pop	Stack not empty	Top object removed	May release some memory
Top	Stack not empty	Top object returned	
Empty	Nil	State of stack unchanged	

An alternate interface may be:

Method	Precondition	Postcondition	Side-effect
Push	Valid object	Object on top	May acquire more memory
Pop	Nil	Top object removed	Throws exception, if stack is empty
Top	Nil	Top object returned	May release some memory
Empty	Nil	State of stack unchanged	Throws exception, if stack is empty

Yet another interface may be:

Method	Precondition	Postcondition	Side-effect
Push	Valid object	Object on top	May acquire more memory
Pop	Nil	Top object removed	Throws exception, if low on memory
Top	Nil	Top object returned	Throws exception, if stack is empty
Empty	Nil	State of stack unchanged	May release some memory

NPTEL MOOCs Object Oriented Analysis and Design
Partha Pratim Das
1

Let us take an example. Let me consider a stack. I am sure all of you know stack. You know it is LIFO container last in first out. So you can add members to a stack by push, remove members from the stack by pop. If you push, they keep on getting stacked and when you pop it is the last pushed element which we will get extracted out of the stack so the order in which the elements entire and they leave the stack or opposite of each other.

Besides so that gives us the two common methods that stack has push and pop. In addition to that typical we will have another method another part of the interface which is top to get know what is the current top element of the stack and we also have a method empty to check if the stack at

all has anything. So we understand this so given that if we are designing a stack class we will say our interface the method part of the interface at this four methods.

There must be a way to push, pop, top and empty. There could be several other interface functions that could be several other methods also but this is minimum for defining early four. This is minimum for defining a stack. Now let us look at for each one of this let us look at the preconditions and the post conditions and the side effect. So this is the method part and what you see in the rest of the table is a contract part.

This is a contract part. So the contract what we have? In the contract we will lead to a precondition so if we look at push let us say let me pick up another color. If I look at push then what is the precondition that is when this push method will be invoked on the stack what the stack object the particular object can assumed. The stack object would like to is assume that the object that we are trying to push.

If you are doing a push, then you have some object that you want to push for example it could be a stack of integer when you are trying to push an integer. It could be a stack of strings when you are trying to push a string. So of course say a stack of integer object while push is invoked on that we will expect that you give it always an integer object you don't give it a string object or an employee object or something like that.

So the fact that the object will have to be valid is a precondition for the push. Now what is the post condition what must be satisfied when the output is generated by push or when push message actually completes execution natural the object that we have given to put that must exist on the top otherwise the LIFO will not get realized. So it is very critical that this post condition be satisfied. What is the side effect?

Now it may or may not have a side effect but it is possible that whatever way we are internally implementing the stack we will talk about this more when we come to the implementation part of the class but whatever way we do that we need memory to keep the elements so if we repeatedly

keep on doing push at some stage we will need more memory and that requirement will come up to the system so that is the kind of side effect that we can have.

So if we similarly look at pop we need the fact that I can pop an element provided. There is some element in the stack so the precondition of pop could be thought of as the stack not being empty natural post condition is top object has been removed and certainly pop may release some of the memory that used by the last object. Top again will need a precondition that the stack is not empty only then you can get the top most element.

Certainly the post condition is it must return the top object that is what I am trying to find out and the other post condition is the effect of doing a top should not change the state of the top stack object. The state of the stack object must remain unchanged so there are two post conditions and there may not be any side effect.

If you look at empty, we can say that there is not precondition. I can send empty message to any stack whatever it states and the post condition is the state of the stack does not seem there is no side effect. So this is the view of the methods and the contract which together form the interface of the stack class. I would like to warn you particularly those who have already been writing programs in various OOP languages like C++, Java or Python for that matter.

You would tend to often believe or follow that is only the methods only this part is what is the interface? But please understand that if I just know the messages that I can send to the object to invoke these methods then I will not be able to do that task properly unless I am careful about the precondition which is specifying the properties on the input the post condition which specifies the properties on the output and the side effect which can happen during this process.

Just to understand that let us say let us take for the stack itself an alternative interface in this alternate interface again the methods are the same. I am not changing any of the methods but what I have changed is I have changed the contract now I say that well in the contract I do not expect that when pop is invoked when a message is send to pop then client has guaranteed that the stack is not empty. I might ask for doing a pop or a top on a stack which may not be empty.

I will say that there is no precondition so which means that the stack could be empty and I may be asking for a top. So what will happen now certainly if the stack is not empty then the top object is removed so that is same as what I had. If it is not empty, then some memory may get released but what if it is empty naturally the output cannot be generated or the output cannot be satisfied because there no top object so what I will do I will throw an exception.

Throwing an exception is we will explain that more later on. It is kind of telling the system telling the client-server system that well I was asked to process a message which I was not able to do so in turn I send an exception message to some exception or error handler/error controller. So it will since just consider that in the previous interface the precondition said that stack is not empty. So it was necessary to consider a side effect where pop could show an exception of stack being empty.

But here I have changed the precondition and therefore I have a possibility of a side effect where an exception will be thrown for the stack being empty. Similar think will happen for top it could also throw an exception if the stack actually is empty is when you are trying to invoke the top message for that stack. So you can say that just messages being the same do not actually give us the same interface because depending on the contract here and the contract here.

The way the programmer has to implement this method would be very different. You could think of yet another interface so in this interface again we have the same set of methods we have the same kind of preconditions and what we change is we have added in this interface. We have added a specific side effect for push. Here we said that push may create a side effect that I need memory. Push may create a side effect that more memory is needed.

Push here as well will create a side effect that more memory is required. Now stack theoretically is unbounded. I can keep on stacking as long as I want but whatever system I actually use to implement the stack object will finally have a limited memory. So at some point it will run out of memory. It will become low memory. What will happen to that?

So compare to this interface this particular one the later one is a more, well-specified interface because it takes care of a potential situation that can happen in the system when it actually has very low memory to make a push request/push message actually successful. So this example I am sure have shown you how you really apply the design by contract principles in terms of your interface design and how really different interfaces having the same set of methods but having different kinds contract will impact the whole design process.

(Refer Slide Time: 25:50)

## Visibility

We can divide the interface of a class into four parts:

- ① **Public:** a declaration that is accessible to all clients
- ② **Protected:** a declaration that is accessible only to the class itself and its subclasses
- ③ **Private:** a declaration that is accessible only to the class itself
- ④ **Package:** a declaration that is accessible only by classes in the same package

Visibility enforces grades of Encapsulation in a class



The next that we would like to draw your attention to is we mention that methods are publicly visible. I would also remind you of the fact that we have repeatedly been saying that we will enforce encapsulation of the object and certainly since objects are instances of their classes the classes must have a mechanism to encapsulate. Encapsulation basically means hiding certain parts of a class or certain parts of the object that we will get instantiated from other objects from instances of other classes.

So we may note that there are typically four types of visibility or we can say that the interface can be classified divided in to four different parts one that is called public. This is the interface which is accessible to all clients anybody and can use that. And typically when we talk about an interface for use like we were talking about stack this is based on only the public visibility because anybody should be able to use the push, pop, top, empty kind of interface.

Another is a private one which is accessible only to the class itself not to the outsiders not to anyone else. So in the stack example we could have a method which actually checks if the stack has enough memory when push operation is happening. Now this method will not be exposed to the external clients because they do not need to be bothered about whether what the stack class or the stack object should do when there is not enough memory to push another element.

This is an internal matter of the stack classes an internal matter of the stack object but stack object actually by itself need to know whether there is enough memory available or not so I could have a full kind of a method which the stack object will send to itself to invoke and check whether there is enough memory such a method would be a private method because that others should not be able to see this.

In between we will also see that there is a class of visibility factors where a declaration of a method is accessible only to the class itself or to its subclasses. We will come to more of subclasses you know we have talked about subclass in relation to abstraction hierarchy so if something is a subclass of another class then protected visibility is for that subclass. So if something is protected then it is accessible to class but it is not accessible to the outsider but along with the class the subclasses also can access the protected members.

Finally, some of the object systems introduce another few other kinds of visibility like package is one such visibility where all classes that belong to the same package which is kind of saying that if I put the classes together under one folder package is similar in concept to a folder that all the classes but it is not exactly a folder it is somewhat different. It says that if you put all this classes into a single package then the classes would be able to see each other.

The classes would be able to will be visible across each other. So which say that visibility enforces different grades of encapsulation from very a hard encapsulation of private visibility to somewhat limited encapsulation of protected package visibility to complete openness of public visibility in to the full system of objects and classes.