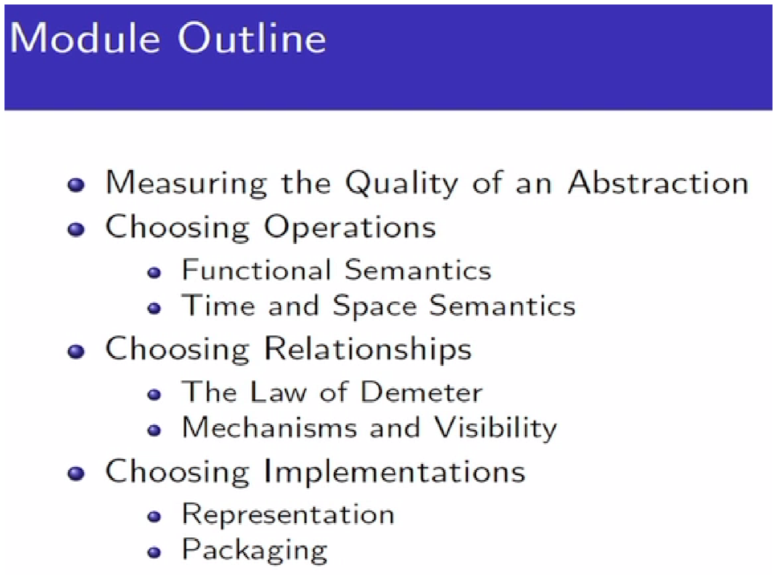
**Object-Oriented Analysis and Design**
**Prof. Partha Pratim Das**
**Department of Computer Science and Engineering**
**Indian Institute of Technology – Kharagpur**

# Lecture - 26
# How to Build Quality Classes and Objects

Welcome to module 15 of object oriented analysis and design. Over the last 4 modules, we have been discussing, introducing about the nature of objects and classes and their relationships. In this module, we will talk about a few basic parameters to decide how to build classes and objects of good quality, that is what defines if my design of classes of objects is good, is moderate, workable or it is pathetic and will cost me lot of pay.

(Refer Slide Time: 01:00)

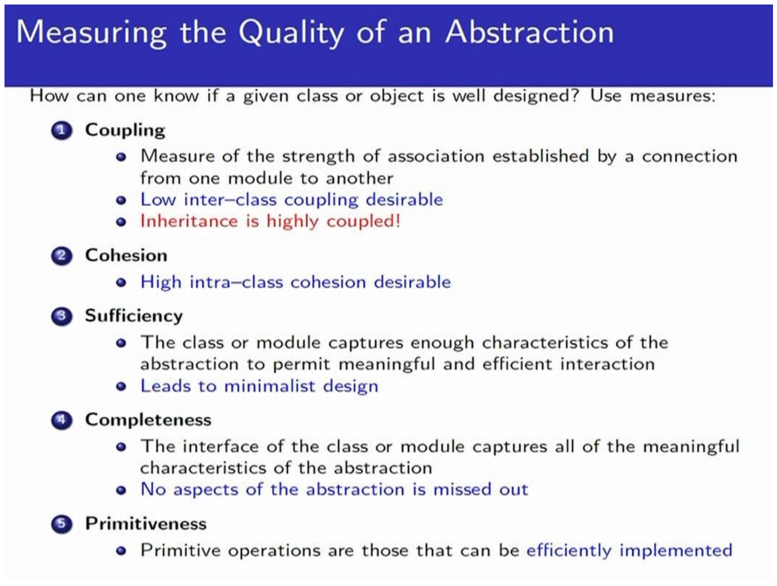


## Module Outline

- Measuring the Quality of an Abstraction
- Choosing Operations
  - Functional Semantics
  - Time and Space Semantics
- Choosing Relationships
  - The Law of Demeter
  - Mechanisms and Visibility
- Choosing Implementations
  - Representation
  - Packaging

So in this module, we will briefly introduce some measures of quality and then on the specific aspects of design like if I am doing a class design then I need to decide on the operations, the relationships and certainly find the implementation. I will highlight the four issues that need to be you need to take care of; you need to be careful about when you do these designs.

(Refer Slide Time: 01:28)



## Measuring the Quality of an Abstraction

How can one know if a given class or object is well designed? Use measures:

1. **Coupling**
   - Measure of the strength of association established by a connection from one module to another
   - Low inter–class coupling desirable
   - Inheritance is highly coupled!
2. **Cohesion**
   - High intra–class cohesion desirable
3. **Sufficiency**
   - The class or module captures enough characteristics of the abstraction to permit meaningful and efficient interaction
   - Leads to minimalist design
4. **Completeness**
   - The interface of the class or module captures all of the meaningful characteristics of the abstraction
   - No aspects of the abstraction is missed out
5. **Primitiveness**
   - Primitive operations are those that can be efficiently implemented

So in terms of the measure of quality a lot of, after a lot of years of research and experience of various projects, successes and failures, the community has more or less agreed to use five major measures to decide whether an abstraction has been captured well or there is scope for improvement. So the first is coupling, coupling basically says that if I have two classes of modules then how couple they are, how closely interrelated they are.

Now, certainly if two classes are; there are two different classes and there is a lot of coupling that this class uses a lot of information from the other classes, that class uses a lot of information from this class and so on, then certainly it may not have been a good idea to make them as separate classes. They should have been in the same module, should have been in the same class.

So that is what we are talking about that if we have the two classes and the coupling that we have which is basically the density of messages that they need to exchange has to be low. So the inter class coupling should be kept at a minimum. Now of course, this will contradict one of the earlier design principles that we have been preaching is inheritance is a good idea because it gives such a good hierarchy.

But certainly if we are using inheritance then we have a very high coupling because any superclass, rather any subclass will use the whole lot of information from the superclass, so there is a lot of coupling in them. But there is always a trade - off in terms of how much you couple or how much you use the inheritance of. The second principle or second measure is cohesion.

That is if I have an object, if I have a class then all the data members and methods, messages that exist whether they should be together. Am I putting too much in to one class that is things which are not related, am I putting them together, we will have to decide based on that. So cohesion must be high, which is intra class all different methods and members that we put in a class, they must be closely related.

So, we had discussed this overall in terms of different aspects of separation have concern and all that these two we had always said that between two entities, between two modules, the traffic that goes across modules must be significantly low, whereas the traffic that goes within a module must be significantly high and that is the basic meaning of coupling and Cohesion. So those turn out to be the measures as well.

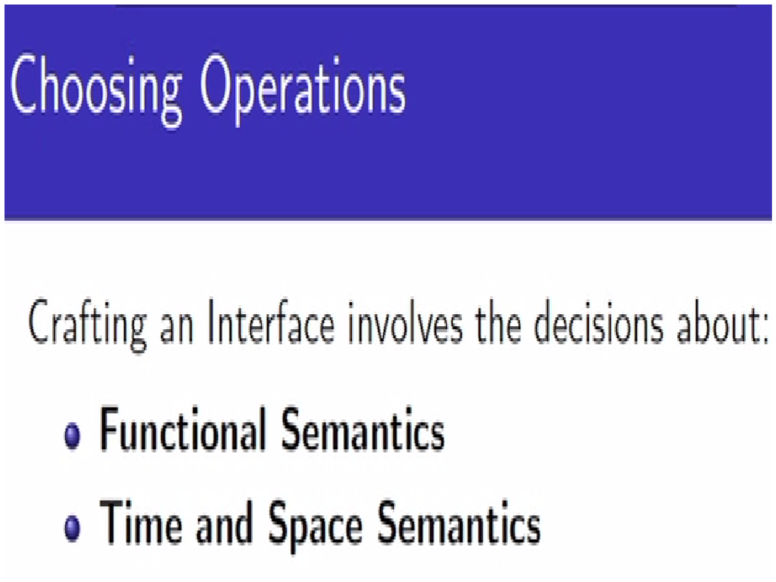
The second next two measures are also related, one says that the design should be sufficient that is you should do a minimalized design, you should not do a design, which has lot of methods in the interface which may not be necessary. For example, you are designing a stack you always talk about designing stacks, there is a push, there is a pop, I could think about a method which does a double push, that is text two elements and pushes them one after other.

I can do that; I mean that way also I can make the semantics of stack correct but there is not something that is a good design because it goes against the minimalist principle of design. It is possible to do with the standard push, so I should not have a double push kind of operations available on my stack. And completeness on the other aspect is, on the other side which says that in terms of the design, how I covered everything that I needed to cover.

For example, if I design a stack which has a push, pop and top and does not have an empty. By analysis, I must be able to reason that this is not a complete design because there will be situations, where you will not know whether you should actually invoke top, send that of message or you should actually invoke the Pop message by adhering to the contractual guarantees that we have talked of.

So in this way, the sufficiency and completeness are two measures that we regularly should apply, whenever we are designing the class and finally, it is the primitiveness, you have to decide in terms of what is the primitive that you build with. Now certainly few primitives have to be given by the OP language that you are using, so they are typically the built-in types, those are the representatives of the classes.

But based on that what you decide to be different classes is based on the primitives that need to be performed. So, this is a kind of a subjective decision, but it will depend significantly on what can we efficiently implement. So these five principles of, measures of coupling, Cohesion, sufficiency, completeness and primitiveness will be necessary for judging any abstractions.
(Refer Slide Time: 07:16)



## Choosing Operations

Crafting an Interface involves the decisions about:

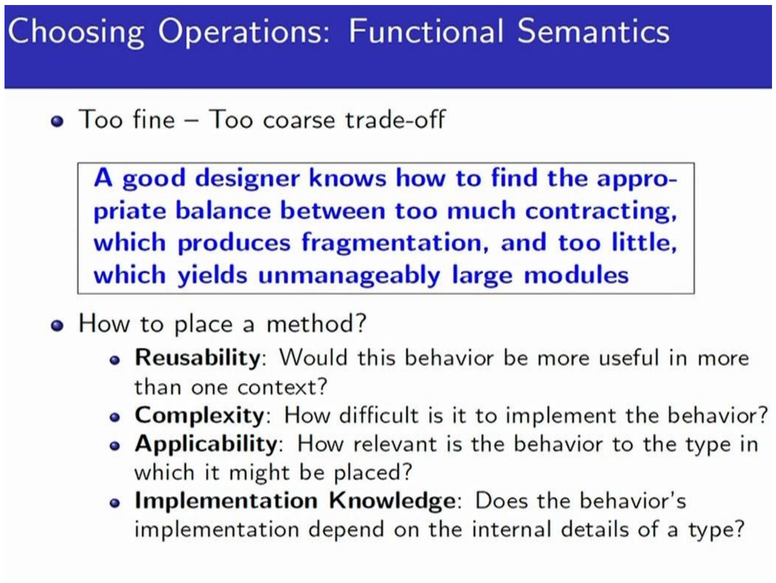
- Functional Semantics
- Time and Space Semantics

So whenever your design is done, you should try to focus on doing these measures and checking out, if you should improve on the design. Next, we will let us talk a little bit about how should you should choose your operations so that is the first thing you need to do, how do you get started with the class design. Certainly we have seen that the whole scenario is a Client - server based, whether there are clients there are servers and messages passing between them.

So the first thing that certainly we need to decide is what could be the set of messages, what could be the operations that need to be supported for a certain class before we can talk about what is its internals or what is its relationships with others. So when you are crafting a interface is not an easy problem at all.

But when you are crafting that then you need to primarily take decisions about these two aspects, one relates more to the overall behavior of the interface and the other relates more to the non-functional implementation related nature of the interface. The functional semantics and the time space semantics, so let us talk about this one by one.

(Refer Slide Time: 08:32)



## Choosing Operations: Functional Semantics

- Too fine – Too coarse trade-off

  **A good designer knows how to find the appropriate balance between too much contracting, which produces fragmentation, and too little, which yields unmanageably large modules**

- How to place a method?
  - **Reusability**: Would this behavior be more useful in more than one context?
  - **Complexity**: How difficult is it to implement the behavior?
  - **Applicability**: How relevant is the behavior to the type in which it might be placed?
  - **Implementation Knowledge**: Does the behavior's implementation depend on the internal details of a type?

In terms of functional semantics that is what should be the different methods that you perform, that you provide to the class, that is a big trade of is, I mean these are standard prescriptions from very successful designers but I will put it in very simple two word terms for you to remember is when you are designing operations, when you are designing methods, you have to always make a trade - off between being too fine and being too coarse which I mean is if you have operations, which every operation which does very small, small things like.

I have an operation which just increments the value, I have another operation which just takes a number and takes out the (()) (09:22) of that number and so on. Probably, I am getting too fine, at the same time if I am designing for a Bank account management, then I say that I have one interface through which the username, user ID and password is taken. It is authenticated the account is identified.

It is displayed and within that the same method itself the account balance is shown. If there is any overdraft default that message is thrown out and so on. So all of that I will do in a single message, in a single operation, then I am getting two codes. So that trade - off is very very important. And certainly that is where object oriented analysis and design becomes somewhat little bit towards the arts feel then being a very concrete science.

Because you cannot exactly ever say us to, what is too fine and what is too coarse. But you will always have to ask yourself this question so whatever operation you design, you think, I mean should I have broken this down into two or more other operations, at the same time, ask for the class that given all these methods, is there are a scope to combine two or more operations and do something better.

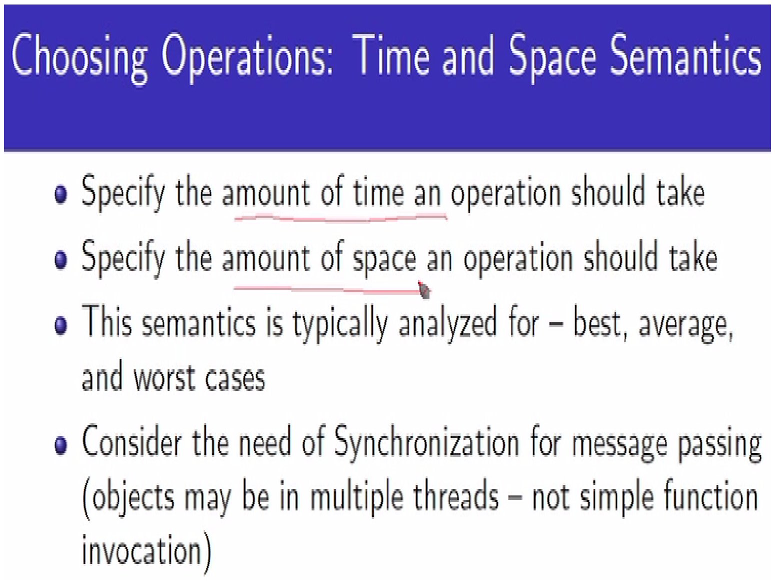
For example, I could talk about stack you will find in many text books, you have an interface for a stack, where the push works in the same way, but pop has a definition that removes that of most element and returns you that element and there is an empty. Now, if you define a pop like that which removes as well as returns you the element then you will have to put a judgment as to, are you making it too coarse, should you split this into two operations, in terms of one which removes the element and one which returns the top element and be strong that you should do the design.

And a quick analysis in this case, will tell you that it is always better to have a separate method to return the top element and a separate method to actually remove the top element, which then you would not return very simply because if you do not have a top, you use pop to do all that then after pop is a only way to check an element, so after checking that if you feel that you did not want to remove this element then you will again have to push this element back.

So that means additional work that means additional time and in some cases, it might mean a very different semantics. So that those kind of choices are what is important and to place a method that if you find this is ok that this is the operation I can see then I need to find out an appropriate class for doing that, decide whether there should be a class for this, these are the factors that you should keep in mind that certainly you should put it in a class which can be reused.

You will have to look out what is the complexity of its implementation, how often it can be used and of course, it will depend on the knowledge that you have, the designer has in terms of the behavior's implementation, the platform dependence and so on. But, these are some of the factors that we will need to keep in mind in deciding how we close on the functional semantics of an operation.

(Refer Slide Time: 13:07)



## Choosing Operations: Time and Space Semantics

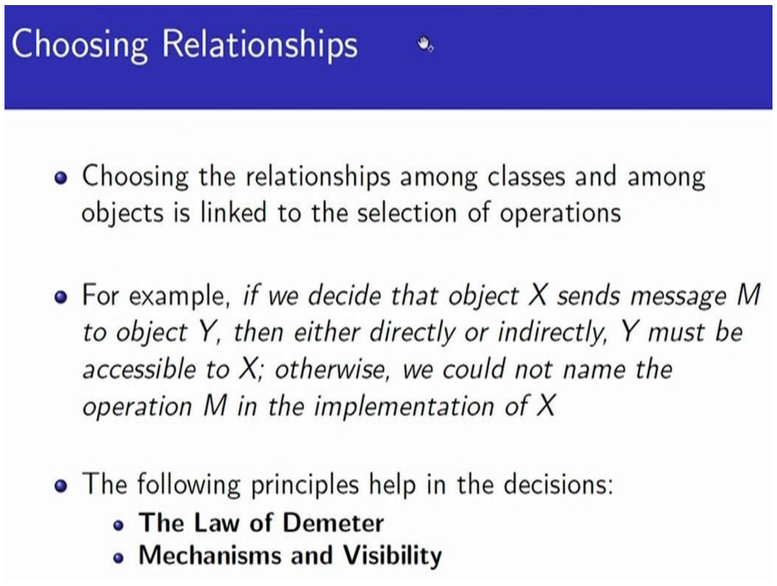
- Specify the amount of time an operation should take
- Specify the amount of space an operation should take
- This semantics is typically analyzed for – best, average, and worst cases
- Consider the need of Synchronization for message passing (objects may be in multiple threads – not simple function invocation)

Now once we have the functional semantics then there are factors relating to more details of implementation and some of the questions that need to be answered is, in terms of amount of time and amount of space. Certainly you will need to have some idea as to what you want, for example if you are very (()) (13:32) example will be about finding elements from a large pool of existing data elements.

So, if it is important that you need to find the elements very quickly, then you will possibly design the operations in a way, so the time is optimized. But you may be able to afford some space but it could be other ways, if you really do not care about how much time it takes, but you are trying to implement the functionality of this search on a mobile phone, where space is a bigger premium than time.

So those are the time, space semantics that need to be decided, you could do that based on a best case, average case, worst case and so on. And specifically we call that whenever you have operations, the operations or messages acting on objects, objects are in a concurrent situation. We have explained the different situation of synchronization between them. So, the time space semantics also will have to look into whether there is a need for synchronizing, whether there is a need for mutual exclusion that we need to put to in the design.

So your overall choice of operations should be guided by this requirements of the functional semantics and the time space semantics.
(Refer Slide Time: 14:58)



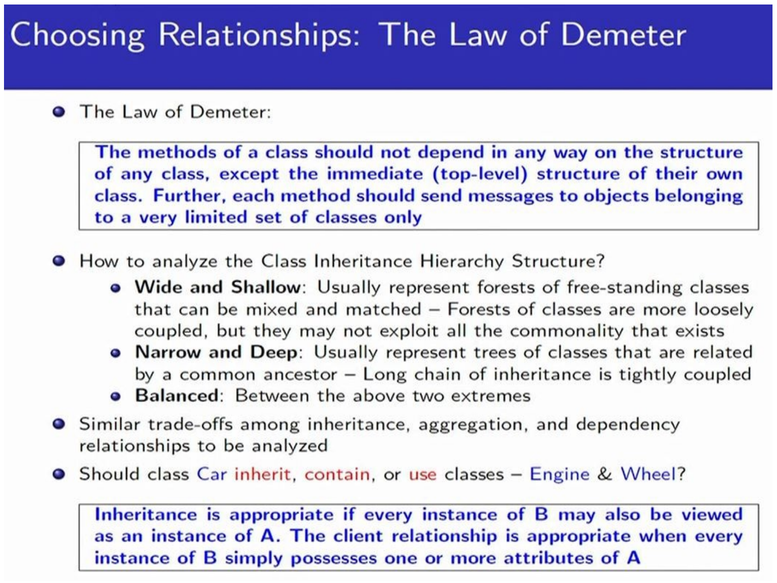
## Choosing Relationships

- Choosing the relationships among classes and among objects is linked to the selection of operations
- For example, *if we decide that object X sends message M to object Y, then either directly or indirectly, Y must be accessible to X; otherwise, we could not name the operation M in the implementation of X*
- The following principles help in the decisions:
  - **The Law of Demeter**
  - **Mechanisms and Visibility**

Moving on once you have chosen the operations, the next would be to choose the relationships amongst classes. What it means that suppose I have say, I have object X and that sends message M to object Y. Now, certainly this message M is what I have decided on, because I have chosen my operation, so I know the message M. Now, once we have that then naturally if this message has to be send either directly or indirectly, the necessity would be that Y must be accessible to X.

That is in terms of different encapsulation visibility restrictions and so on, unless is Y accessible to X this message design, the operation design will not work. So, the relationships have to be chosen keeping in mind the different choice of operations and the way you have started doing the design.

(Refer Slide Time: 16:05)



## Choosing Relationships: The Law of Demeter

- The Law of Demeter:

  **The methods of a class should not depend in any way on the structure of any class, except the immediate (top-level) structure of their own class. Further, each method should send messages to objects belonging to a very limited set of classes only**

- How to analyze the Class Inheritance Hierarchy Structure?
  - **Wide and Shallow**: Usually represent forests of free-standing classes that can be mixed and matched – Forests of classes are more loosely coupled, but they may not exploit all the commonality that exists
  - **Narrow and Deep**: Usually represent trees of classes that are related by a common ancestor – Long chain of inheritance is tightly coupled
  - **Balanced**: Between the above two extremes
- Similar trade-offs among inheritance, aggregation, and dependency relationships to be analyzed
- Should class Car inherit, contain, or use classes – Engine & Wheel?

  **Inheritance is appropriate if every instance of B may also be viewed as an instance of A. The client relationship is appropriate when every instance of B simply possesses one or more attributes of A**

And their two principles which people follow very frequently. First is the law of Demeter, the law of Demeter says that in terms of choosing the relationship, you have to make a lot of trade - off judgment, which is first, I am just trying to use one inheritance hierarchy to explain this that I can have a hierarchy that which is more like say, like this, 1 base class, one superclass everything else is a leaf child class, is a one kind of Hierarchy or maybe not only that but I have this, lot of here.

So these kind of we will say wide and shallow that is where the depth of the hierarchy is less, but the breadth is very high. So if the breadth is very high then you have a large number of leaf classes, so it will give you a forest. Now, certainly it is much less like a tree, it is more like a forest, so all these leaf classes are independent. So if they are independent they hardly share the properties that could have been shared, that is a basic purpose of inheritance to share properties across classes.

So if the hierarchy is very wide and shallow then you are not exploding the commonality that exists and you are missing out on key design clue that may happen. On the other hand, on the extreme if your hierarchy is very narrow and deep suppose you have this, that is this width is low but this depth is high. Then, also it will not be a good design because you will have long Chains of inheritance where you have just every class is trying to get something from the parent and so on, possibly (()) (18:12) is so much that you do not have a good manageable design.

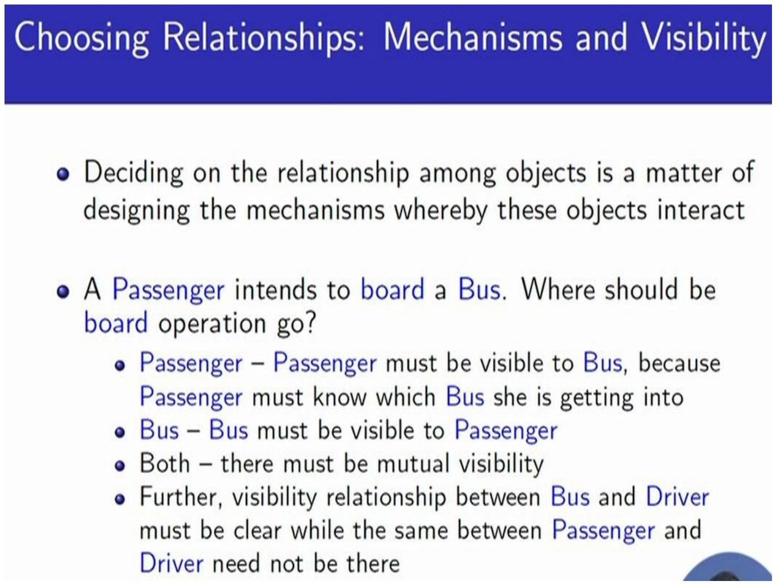
So you recall the kind of vehicle design we discussed about in the earlier module, the module 14 those kind of are known as the balanced design, where you have kind of two extremes that your depth is not very high, at the same time, the width is also not very high so there is a fair amount of commonality that is exploited and at the same time in terms of exploiting that you have not made the whole design very (()) (18:48) and decide on that.

The similar principles can be used amongst single inheritance in cases of aggregation that is how much to compose, how many different parts should I break it on to (()) (19:06) to represent a car, Do I take the car and then break it down in terms of and nuts and bolts and wires and connectors and seat covers and all that or I do hierarchical aggregation there, what is a dependency relationships and so on.

So on the same principle of trade - off what is very fine and what is very coarse this relationship designs must be done and also there should be design choices made in terms of how do you relate that, for example you have car and engine. Certainly, I started saying that this is a case of aggregation but who said this has to be a case of aggregation, aggregation is one way of looking at itself. Car contains engine is one view.

Now could I say that car inherits from engine is a specialization of engine, will have to see whether that might sense. But you can certainly say that car uses an engine, I think of car as a separate entity, engine as a separate entity, car uses an engine. So there is certain decisions to be made in all these cases as to, which one is the right way to relate different classes and accordingly you can you should make that choice and that is what the law of Demeter talks about.

(Refer Slide Time: 20:28)



## Choosing Relationships: Mechanisms and Visibility

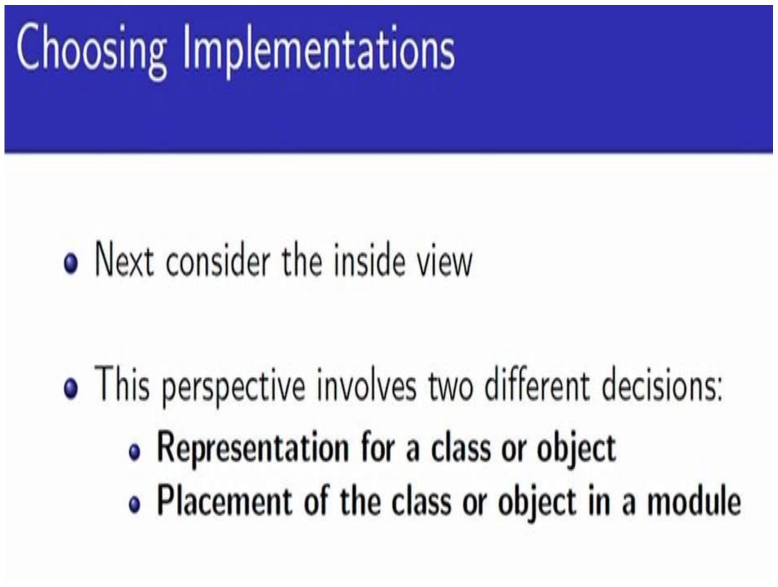
- Deciding on the relationship among objects is a matter of designing the mechanisms whereby these objects interact
- A Passenger intends to board a Bus. Where should be board operation go?
    - Passenger – Passenger must be visible to Bus, because Passenger must know which Bus she is getting into
    - Bus – Bus must be visible to Passenger
    - Both – there must be mutual visibility
    - Further, visibility relationship between Bus and Driver must be clear while the same between Passenger and Driver need not be there

Besides that, you will have to look into the mechanisms, the visibility requirement. Suppose a passenger intends to board a bus, so there is a passenger class, there is a bus class and you need to place the board method and for that board method you need to have the visibility between passenger and bus.

Now there are certainly 3 choices you can have a board method in passenger and let the bus goes as a parameter to that, so that where you send that or the bus can have a board method with a passenger sense it and goes a parameter or both could have board methods which collaborate between themselves. Now depending on which class has a board method, the visibility will have to be accordingly. For example, if bus has a board method then bus must be visible to the passenger.

So the passenger can send the message to it. So these are the different choices in terms of mechanism and visibility that you will need to apply to decide on the relationship.

(Refer Slide Time: 21:30)

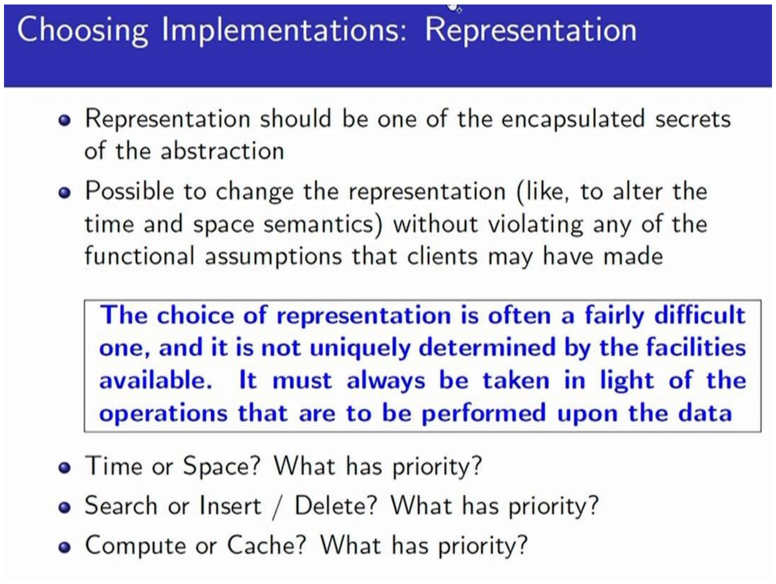


## Choosing Implementations

- Next consider the inside view
- This perspective involves two different decisions:
  - **Representation for a class or object**
  - **Placement of the class or object in a module**

The third aspect in terms of the quality of design will depend on the implementation which is often would mean a lot of skill, lot of understanding of your vehicle of implementation that is the implementation OP language as well as the platform. I would just like to highlight that two factors that need to be critically looked at regularly is the representation that you are using for the class and the packaging or the placement of the class that is in a large system.

There are large number of classes, objects, modules, objects occurring, now you need to certainly group them into certain sub systems, certain modules. So the representation has to be appropriate as well as the grouping has to be appropriate. So that again the traffic across modules can be minimized and they can be more cohesive in that form.
(Refer Slide Time: 22:25)



## Choosing Implementations: Representation

- Representation should be one of the encapsulated secrets of the abstraction
- Possible to change the representation (like, to alter the time and space semantics) without violating any of the functional assumptions that clients may have made

**The choice of representation is often a fairly difficult one, and it is not uniquely determined by the facilities available. It must always be taken in light of the operations that are to be performed upon the data**

- Time or Space? What has priority?
- Search or Insert / Delete? What has priority?
- Compute or Cache? What has priority?

So representation should be done in a way so that you remember the basic principle of the implementation, that implementation must hide, it must encapsulate all the secret, all the details of the implementation. Couple of modules back while discussing about the nature of classes I talked about this aspect with an example of a stack and I show that how for same interface we could have multiple people different implementations of a stack, which could change irrespective of, which interface the customer is using and without the knowledge of the customer.

So that is a very basic requirement that any implementation must satisfy that, any change in the implementation usually should not impact the interface contract with the customer and that is what you should try to follow but there are several trade - offs to be made, several choices to be made resemble, for example some of the common ones are should I optimize for space or should I optimize for time. The new dimension being added in many places, should I optimize for power?

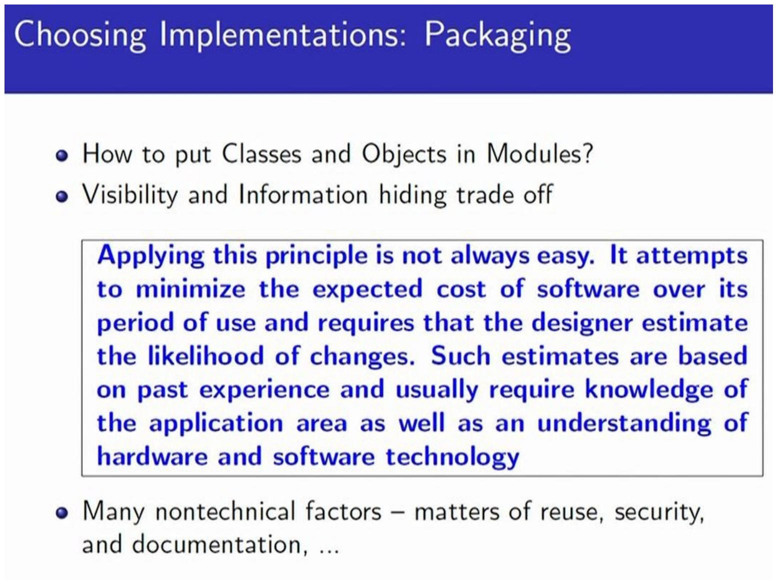
Similarly, in terms of any application that needs to do some kind of a search somewhere there is always a choice between do I optimize on search, or do I optimize on insert, delete. We all know that if we use some kind of array structure, we will have very optimal search but very inefficient insert, delete. If you do a list, insert, delete is very quick but the search gets very slow. If for that we want to do a binary search tree, then possibly these two get balanced.

But I need more time rather I need more space, because now the binary search tree has different pointers to manage and so on. So there are different choices, another very common choice in terms of the class design that we will need to do is compute or cache. That is there are several values which you could compute on the fly or compute once and keep it stored.

I will just give an example, suppose you regularly need the age of an employee now what you could do is, naturally age is dynamic so it does not remain the same so what you do you keep the date of birth, so what you could do whenever the age is required you could compute the age and return that.

Typically, you do not need the age in the granularity of the days, you need it in terms of years. So one we could be that every time you do this that is basically compute your choice, the other way could be that ok, you have an age field and which you compute on 1st January for all employees. Because your logic is that within that year the age will not change or whatever changes will happen that is just for one year which does not impact or on the date of birth of that employee you compute the age and store it.

So what you are doing, you are caching the computation. So there is a tradeoff between what, which one should be used and these different factors will decide what kind of representation, what kind of implementation you will choose for.
(Refer Slide Time: 25:51)


## Choosing Implementations: Packaging

- How to put Classes and Objects in Modules?
- Visibility and Information hiding trade off

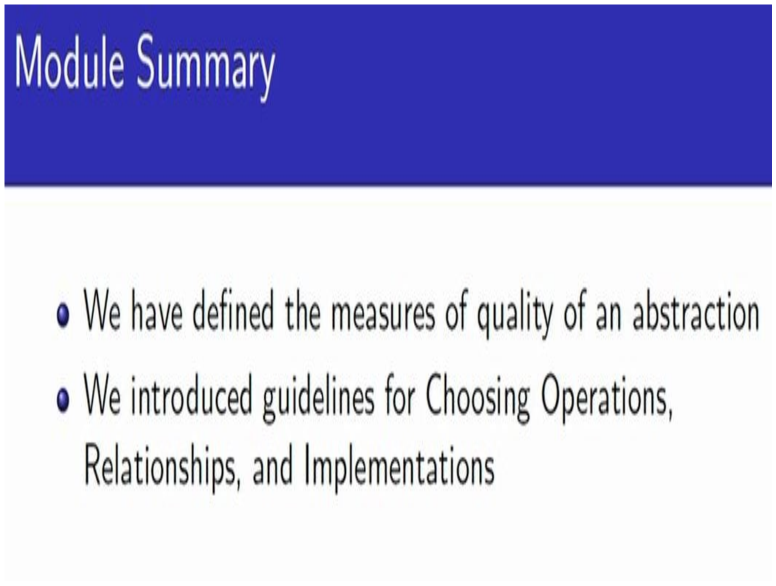
**Applying this principle is not always easy. It attempts to minimize the expected cost of software over its period of use and requires that the designer estimate the likelihood of changes. Such estimates are based on past experience and usually require knowledge of the application area as well as an understanding of hardware and software technology**

- Many nontechnical factors – matters of reuse, security, and documentation, ...

Finally, in terms of packaging you have to put the classes and objects in modules. If you are using a platform OP language like Java or C sharp then you will, these are actual physical decisions because you have to decide, which java package some classes going and the other classes will possibly go in some other package and so on. If you are doing in C sharp, you have to decide on the projects, if you are doing in C++, you will need to decide on your own packaging in terms of different source files, header assemblies and so on.

But this is a very critical decision to make because this will directly have an impact on the visibility and information hiding the trade-off between visibility and information hiding because if you put certain classes together in to one module then it is much easier to make them mutually visible and hide all of their visibility from someone, who is not in the same module and so on and so forth.

So those are the factors in packaging besides that there could be lot of non-technical factors like how you want to reuse, if you want to reuse something across variety of different modules you would not like to put them hard into one module but may be have a separate module for it. The security could be the matter, the documentations several other factors which could impact the choice of packaging in implementation.
(Refer Slide Time: 27:32)



## Module Summary

- We have defined the measures of quality of an abstraction
- We introduced guidelines for Choosing Operations, Relationships, and Implementations

So to sum up in this module, we try to introduce you to the concept of having certain quality measures for designing the abstraction and for designing the classes and objects in terms of coupling, cohesion, sufficiency, completeness and primitiveness and then for the three major aspects of the design dealing with the choice of operations or methods, the choice of relationships between classes and finally the choice of implementation we have tried to talk about some of the pinpoints that can happen in the whole design, some of the parameters that you need to keep in mind while you go ahead with the design.

We will take this forward in the modules for the next week, where we will expose you with different semi-structured techniques of identifying objects and classes and relationships and as you do that all these principles that we have talked about in this module you should bear in mind. More and more you can put them into practice, you will become a better designer of the object system that you are given to work with.